Universita' degli Studi di Messina
Dipartimento di Matematica e Informatica

Programming II project

NetLayer

11 June 2015

Authors:
Vittorio Romeo

Professors:
Massimo Villari

http://vittorioromeo.info

http://unime.it

Contents

Part I

Project specifications

The following part of the document describes the project and its design/development process without exploring its implementation details.

The part begins with a synthesis of the **client request**. After a careful analysis of the request, a **Software Requirements Specification** (SRS) was written.

Writing a correct and informative SRS is of utmost importance to achieve an high-quality final product and ensuring the development process goes smoothly.

The SRS will cover the following points in depth:

- **Scope and purpose**.
- **Feature and functions**.
- **External interface requirements**.
- **Functional requirements**.
- **Example use cases**.
- **Non-functional requirements**.
- **Analysis models**.

Chapter 1

Client request

The client requests the design and implementation of an **open-source multi-purpose C++14 networking library**.

The library must allow the client to develop its own **server-client** architectures and applications with ease, while still being performant and allowing low-level operations if required.

The client intends to use the library as a basis for the networking layer in applications belonging to different domains, ranging from **chat web applications** to **real-time games and simulations**.

The library must fulfill the following requirements:

- The library must be written in **modern C++14**, making use of the latest features to improve performance, readability and flexibility.
- The library must target **UNIX** systems, **Windows** and **MacOS**.
- The library must have a **layered architecture**, allowing developers using it to go as low-level/high-level they desire.
- The library must deal with **byte serialization** of native and user-defined classes. Nested serializable data structures must be supported.
- The library must provide a generic **tunnel abstraction** that represents a network entity providing and receiving data. A UDP socket tunnel implementation must be provided with the library.

- The library must provide an high-level abstraction for **server-client** multi-threaded architectures, allowing applications to asynchronously interact with any number of sockets and conveniently handle received packets via function dispatching.
- The library must provide metaprogramming facilities to generate and bind packet types at compile-time, allowing performant code generation for serialization/deserialization and communication.
- The library must be released under an **open-source** license and promote collaboration and external contributions.

The client intends using the requested library **to build platforms** for various projects, both for internal company usage and public usage.

It is imperative for the library to be easily integrable with existing legacy system, such as architectures depending on relational databases.

For ease of development and deployment, the client requested the library to be optionally usable in **header-only** mode and compatibility with the **CMake** build system.

The abstraction provided by the library must work asynchronously by default, but an option to use blocking IO must be present.

Chapter 2

Software Requirements Specification

1. Introduction

1..1 Software engineering

Software engineering is the study and an application of engineering to the design, development, and maintenance of software.

The Bureau of Labor Statistics' definition is Research, design, develop, and test operating systems-level software, compilers, and network distribution software for medical, industrial, military, communications, aerospace, business, scientific, and general computing applications.

Typical formal definitions of software engineering are:

- The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.
- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- An engineering discipline that is concerned with all aspects of software production.

- The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.

1..1.1 Background

The term **software engineering** goes back to the '60s, when more complex programs started to be developed by teams composed by experts.

There was a radical transformation of software: from **artisan product** to **industrial product**.

A software engineer needs to be a good programmer, an algorithm and data structures expert with good knowledge of one or more programming languages.

He needs to know various design processes, must have the ability to convert generic requirements in well-detailed and accurate specifications, and needs to be able to communicate with the end-user in a language comprehensible to him comprehensible.

Software engineering, is, however, a discipline that's still evolving. There still are no definitive standards for the software development process.

Compared to traditional engineering, which is based upon mathematics and solid methods and where well-defined standards need to be followed, software engineering is greatly dependent on personal experience rather than mathematical tools.

Here's a brief history of software engineering:

- **1950s**: Computers start to be used extensively in business applications.
- **1960s**: The first software product is marketed.

  IBM announces its unbundling in June 1969.
- **1970s**: Software products are now regularly bought by normal users.

  The software development industry grows rapidly despite the lack of financing.

  The first software houses begin to emerge.

1..1.2 Differences with programming

- A programmer writes a complete program.
- A software engineer writes a software component that will be combined with components written by other software engineers to build a system.

- Programming is primarily a personal activity.
- Software engineering is essentially a team activity.
- Programming is just one aspect of software development.
- Large software systems must be developed similar to other engineering practices.

1..2 SRS

This **Software Requirements Specification** (SRS) chapter contains all the information needed by software engineers and project managers to design and implement the requested forum creation/management framework.

The SRS was written following the **Institute of Electrical and Electronics Engineers** (IEEE) guidelines on SRS creation.

1..3 Purpose

The SRS chapter is contained in the **non-technical** part of the thesis.

Its purpose is providing a **comprehensive description** of the objective and environment for the software under development.

The SRS fully describes **what the software will do** and **how it will be expected to perform**.

1..4 Scope

1..4.1 Identity

The software that will be designed and produced will be called **NetLayer**.

1..4.2 Feature extents

The complete product will:

- Provide a library for the **development of multi-purpose network applications and architectures**.

- Provide abstractions for all the major **operating systems' networking layer**.
- Provide an extensible and flexible **data serialization** module for primitive and user-defined classes.

NetLayer, however, will not be a complete framework for the development of applications. Every part of an application that does not deal with networking issues will not be covered by the product.

1..4.3 Benefits and objectives

Development using NetLayer will give companies and individuals several benefits over from-scratch development.

- Usage of NetLayer will provide access to an **easy-to-integrate** and **easy-to-use** networking library.
- Development and testing time will be **significantly reduced**.
- Code making use of the library will be **modern, efficient and readable** thanks to C++14 features and abstractions.

2. General description

2..1 Product perspective and functions

The product shares many basic aspects and features with existing networking libraries, improving upon them in the following ways:

- A layer-based architecture allows developers to make use of both low-level constructs and operations and high-level abstractions in the same application.
- The library will optionally allow developers to use a programming style similar to **functional programming**, making use of callbacks and first-class functions to deal with packet management and function dispatching.
-

2..2 User characteristics

NetLayer is targeted towards modern C++ developers experienced with C++11 and C++14 features. The library makes heavy use of modern metaprogramming paradigms and techniques - unfamiliar users will not be able to make full use of the library.

A m̈ore functionalïnterface is provided where possible, allowing users to use convenient abstractions for data serialization functions and networking functions.

Familiarity with multithreading and synchronous computation is also required to use the library.

3. Glossary

The following list contains all the main elements that compose the architecture of NetLayer.

- **Packet Buffer**: dynamically resizable buffer that can store and provide serialized generic data.
- **Address**: union of an IP address and a port.
- **Payload**: abstraction consisting of an Address and PcktBuf. It can be sent to and received by Tunnel instances.
- **Tunnel**: abstraction of a Payload provider/receiver. The default Tunnel is an UDP socket.
- **Thread Safe Queue**: a lock-based thread safe queue that supports concurrent enqueueing and dequeueing.
- **Managed Packet Buffer**: abstraction consisting of a Thread Safe Queue and a reference to a Tunnel. It can be either a **Managed Receive Buffer**, which enqueues received data from the tunnel, or a **Managed Send Buffer**, which enqueues data that will be sent through the tunnel.
- **Managed Host**: union of an Address, a Tunnel, a Managed Receive Buffer and a Managed Send Buffer. Represents a network entity capable of sending and receiving data through a tunnel.

- **Serializable**: abstraction over a tuple of generic types that automatically allows the user to serialize and deserialize data. Serializable packets can also be nested and contain dynamically-resizable data structures.
- **Packet Bind**: compile-time bind of a Serializable to a Packet type. Used to generate a dispatch table.
- **Dispatch Table**: compile-time function table that binds a function to specific packet binds. Used to handle received packets.
- **Context Managed Host**: union of a managed host and a dispatch table.

4. Specific requirements

4..1 External interface requirements

External interface requirements identify and document the interfaces to other systems and external entities within the project scope.

4..1.1 User interfaces

The product will not provide any graphical user interface. The users of the library will be able to access its functions and types using C++14.

4..1.2 Software interfaces

The **open-source policy** of NetLayer will allow its users to expand or improve existing functionality and to interact with other existing technologies.

4..2 Functional requirements

In software engineering, a **functional requirement** defines a function of a system and its components.

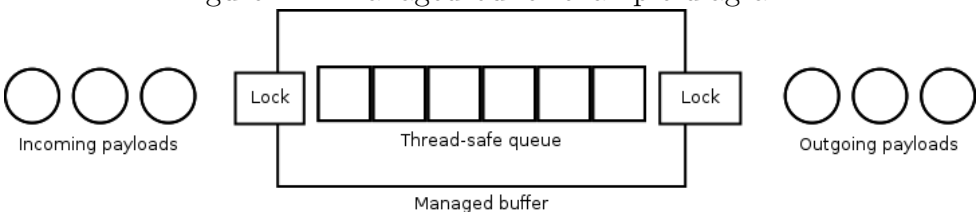
Functional requirements may be **calculations**, **technical details**, **data manipulation and processing** and other specific functionality that define what a system is supposed to accomplish.

Behavioral requirements describing all the cases where the system uses the functional requirements are captured in **use cases**.

4..2.1 Packet management

- **Payloads and thread-safe queue**: an abstraction consisting of an address and data is provided, along with a thread-safe queue that is used for sending/receiving data to/from the network in managed buffers.
- **Managed buffers**: payloads will be enqueued and dequeued in managed buffers, that allow to asynchronously access the contents of their queue.

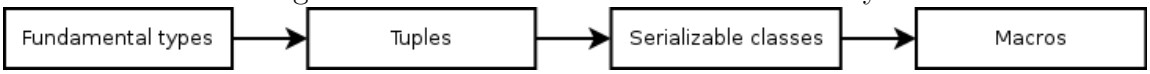
Figure 2.1: Managed buffer example diagram.



4..2.2 Data serialization

- **Fundamental types**: fundamental C++ type serialization will automatically be provided by the library.
- **Common C++ classes**: serialization for commonly used C++ classes, such as `std::vector` and `std::array`, is provided by default.
- **Extensible serialization**: library user will be able to extend the serialization system with their own types, using simple class inheritance or macros for convenience.

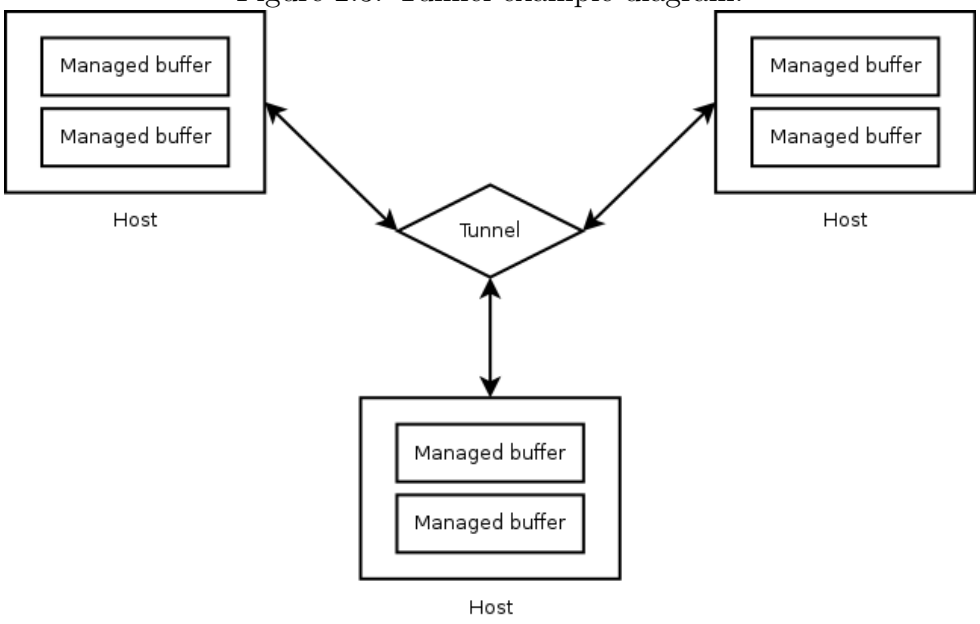
Figure 2.2: Automatic serialization hierarchy.



4..2.3 Tunnel management

- **Default tunnel: UDP**: a tunnel implementation, wrapping an UDP socket, is provided by default.
- **Default tunnel: mock**: a mock tunnel, for unit-testing purposes, is provided by default.
- **Tunnel interface**: an extensible tunnel interface is provided, allowing the users of the library to implement their own network tunnels.

Figure 2.3: Tunnel example diagram.

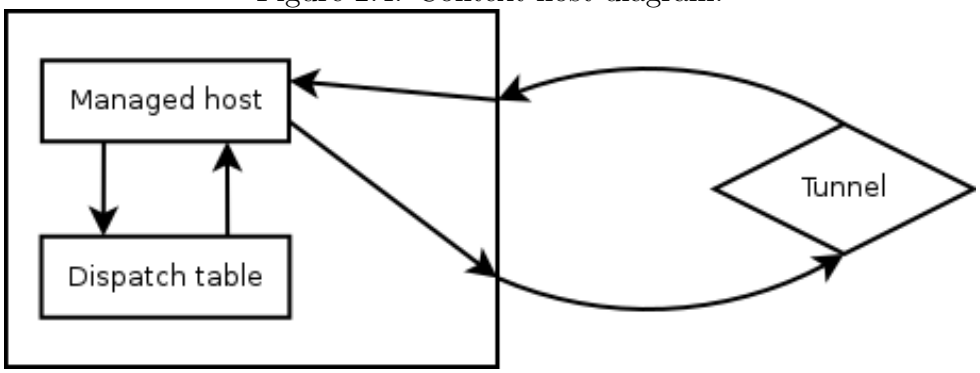


4..2.4 Context binding

- **Managed hosts**: abstraction consisting of a send managed buffer and a receive managed buffer. Allows the user to add processing threads and to poll the buffers for received data.
- **Dispatch table**: an extensible table, configurable at compile-time, is provided to allow the user to define functions which will automatically handle specific packet types.

- **Context host**: union of a managed host and a dispatch table. Provides a convenient interface to quickly develop a server/client architecture capable of sending and receiving payloads.

Figure 2.4: Context host diagram.



4..3 Example use cases

In software and systems engineering, a **use case** is a list of steps, typically defining interactions between one or more actors and a system, to achieve a goal.

In the following examples, we'll cover possible use cases for two different developer types using NetLayer:

- **Networking layer developer**: developer managing packet types, tunnel types and their bindings.
- **Application layer developer**: developer managing application logic, making use of existing NetLayer bindings.

4..3.1 Defining tunnel types

Defining tunnel types is a low-level operation done by networking layer developers. It allows NetLayer users to implement their own protocols or mock payload providers/receivers.

4..3.1.1 Actors

- Networking layer developer.
- NetLayer library.

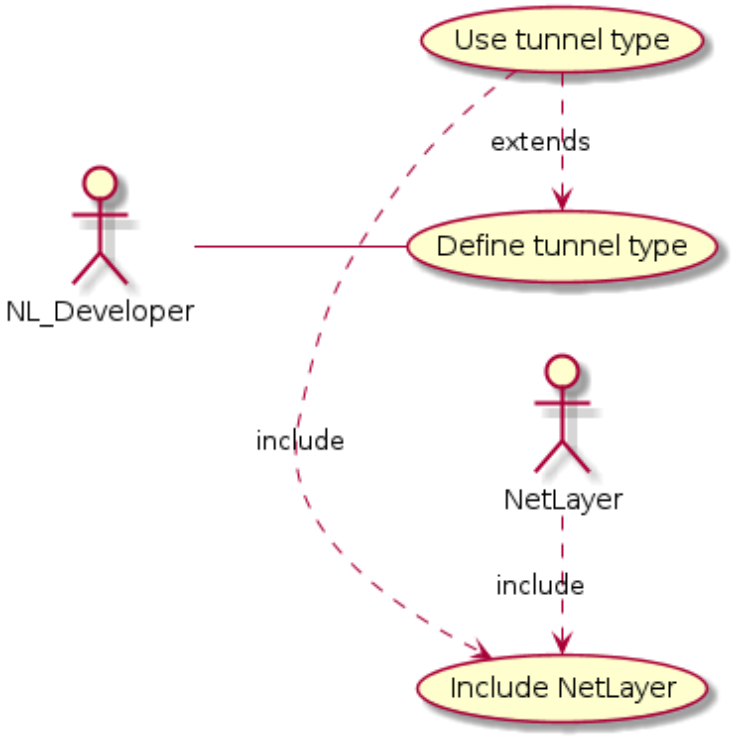
4..3.1.2 Pre-conditions

- NetLayer was correctly included.

4..3.1.3 Post-conditions

- An usable tunnel type for context hosts or manual payload management was defined.

Figure 2.5: Defining tunnel types - use case diagram.



4..3.2 Defining serializable types

Serializable types will be usually defined by networking layer developers and used by both types of developers. Fundamental types are automatically serializable. Tuples of fundamental types can be marked as serializable and wrapped in custom interfaces using either compile-time inheritance or helper preprocessor macros.

4..3.2.1 Actors

- Networking layer developer.
- Application layer developer.
- NetLayer library.

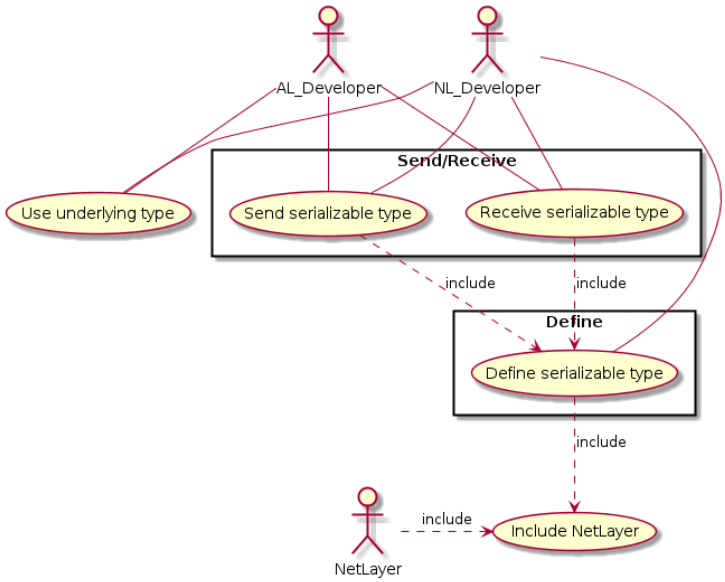
4..3.2.2 Pre-conditions

- NetLayer was correctly included.

4..3.2.3 Post-conditions

- Any number of serializable types were defined.
- Defined types can be sent/received through tunnels.
- Defined types can be used as normal C++ types.

Figure 2.6: Defining serializable types - use case diagram.



4..3.3 Binding types to dispatch table

The creation and management of a dispatch table is usually handled by networking layer developers. They will define and bind all packet types that can be received and sent by the application.

4..3.3.1 Actors

- Networking layer developer.
- NetLayer library.

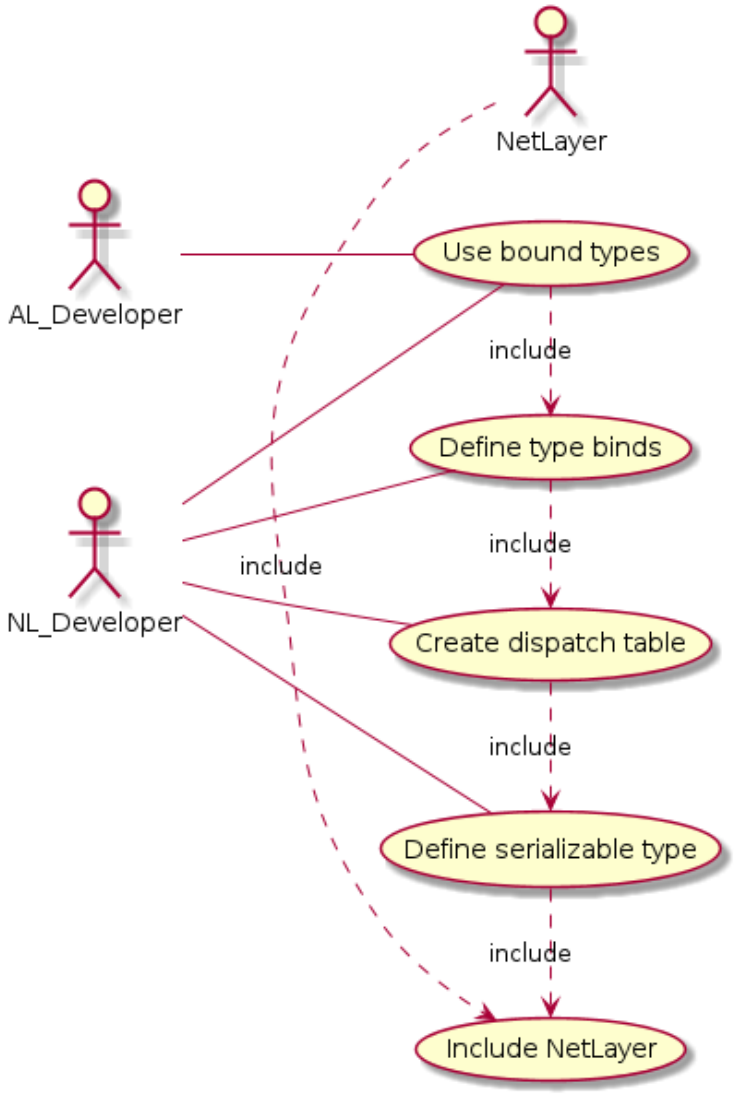
4..3.3.2 Pre-conditions

- NetLayer was correctly included.

4..3.3.3 Post-conditions

- A dispatch table was defined.
- Functions can now be assigned to every inbound payload type.

Figure 2.7: Binding types to dispatch table - use case diagram.



4..3.4 Defining a context host

After the creation of a dispatch table, the definition of a context host is required. A context host is the union of a dispatch table and a managed host.

4..3.4.1 Actors

- Networking layer developer.
- NetLayer library.

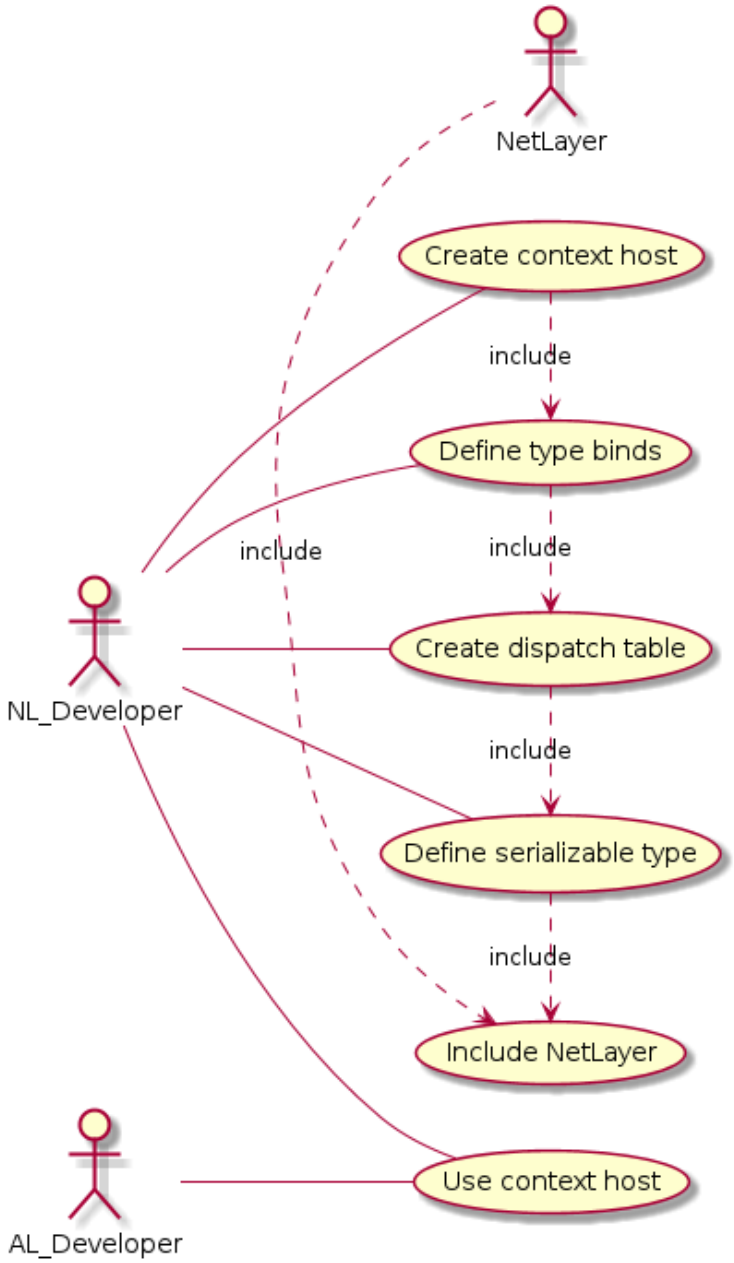
4..3.4.2 Pre-conditions

- NetLayer was correctly included.
- A dispatch table was created.
- A tunnel type was chosen.

4..3.4.3 Post-conditions

- An usable context host was defined.
- A previously defined dispatch table is now bound to the context host.
- The context host can now send/receive bound payload types.

Figure 2.8: Defining a context host - use case diagram.



4..3.5 Handling incoming payloads

Incoming payloads can be either handled by managed hosts, bypassing the dispatch table, or by context hosts. Context hosts automatically call a bound function depending on the type of the received payload.

4..3.5.1 Actors

- Networking layer developer.
- Application layer developer.
- NetLayer library.

4..3.5.2 Pre-conditions

- NetLayer was correctly included.
- A context host was defined.
- Payload types were bound to dispatch table.

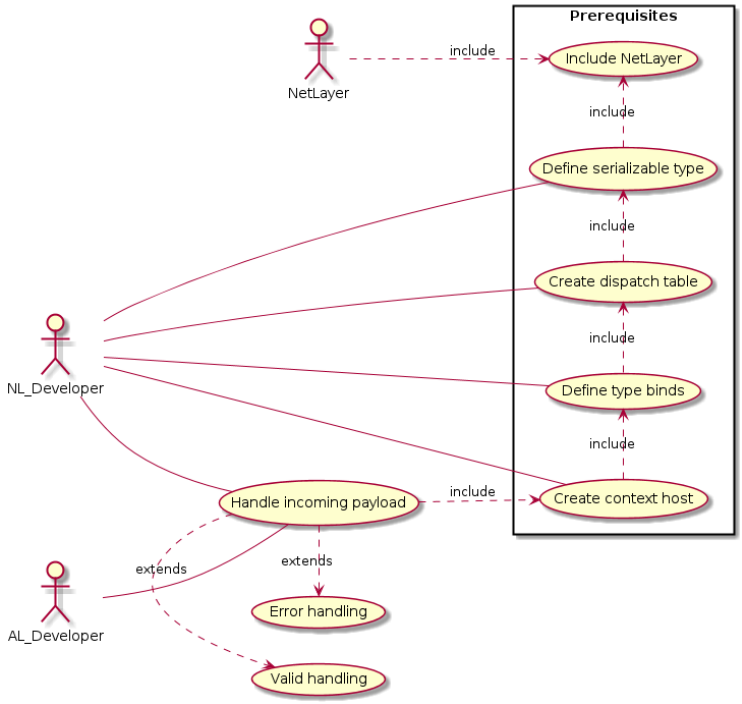
4..3.5.3 Flow of events

-

4..3.5.4 Post-conditions

- Incoming payloads were handled or an error occurred.

Figure 2.9: Handling incoming payloads - use case diagram.



4..3.6 Handling outgoing payloads

Outgoing packets can be sent both by managed hosts and context hosts. Context hosts provide functionality to mark the type of the packet, so that receivers can handle it thanks to dispatch tables.

4..3.6.1 Actors

- Networking layer developer.
- Application layer developer.
- NetLayer library.

4..3.6.2 Pre-conditions

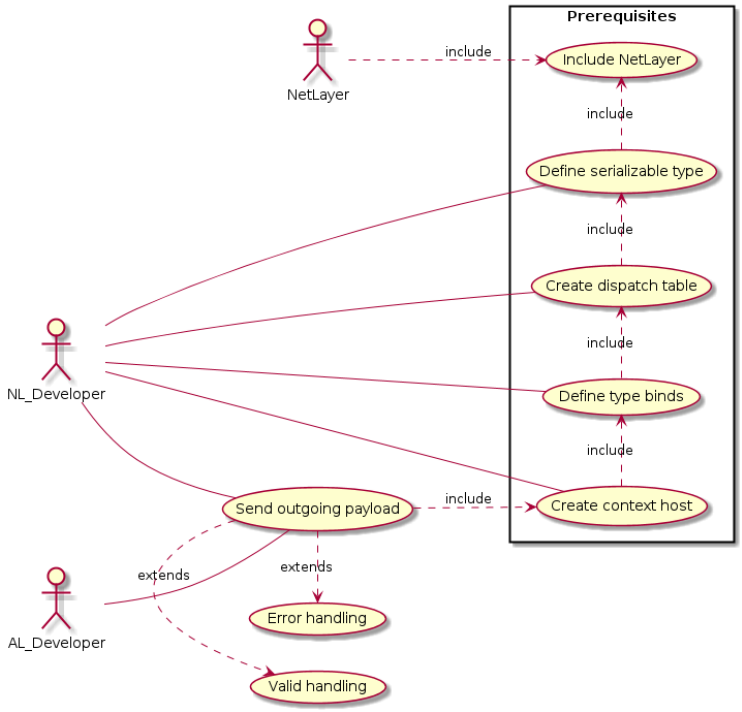
- NetLayer was correctly included.

- A context host was defined.
- Payload types were bound to dispatch table.

4..3.6.3 Post-conditions

- Outgoing payloads were sent or an error occurred.

Figure 2.10: Handling outgoing payloads - use case diagram.



4..4 Non-functional requirements

Functional requirements are supported by **non-functional requirements** (also known as quality requirements), which impose constraints on the design or implementation (such as performance requirements, security, or reliability).

4..4.1 Performance

The system will be designed from the ground-up with emphasis on performance.

The layered architecture will allow users to follow the **pay only what you use** principle, which is one of C++'s biggest selling points.

NetLayer tries to do as much as possible during compilation, avoiding unnecessary run-time polymorphism overhead.

4..4.2 Reliability

The system will have to be reliable and keep working in case of errors.

Since NetLayer is a general-purpose library, intended for use in multiple domains, **exceptions are not used** throughout the library. Real-time simulation and game development industries prefer avoiding using exceptions because they unfortunately always bring a small amount of runtime overhead, even when rarely used throughout the program.

NetLayer will allow users to define and use their preferred error handling systems.

4..4.3 Security

Encryption and other security features are out of NetLayer's scope - users can implement them on top of the library if necessary.

4..4.4 Maintainability and portability

Being an open-source project, **maintainability**, **extensibility** and **portability** are key.

The code layer will be carefully designed and organized to allow easy maintenance, bugfixing and feature addition.

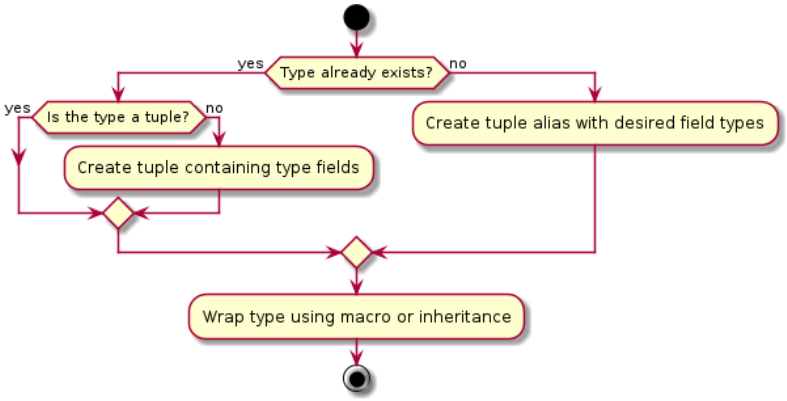
To ensure maximum portability, the product will be designed to work on the most popular **GNU/Linux** distributions and will be thoroughly tested on different platforms.

5. Analysis models

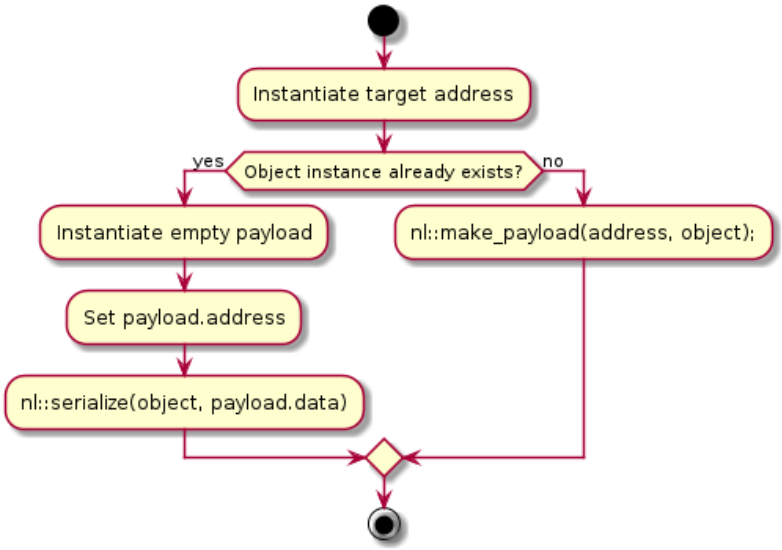
5..1 Activity diagrams

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organisational processes (i.e. workflows). Activity diagrams show the overall flow of control.

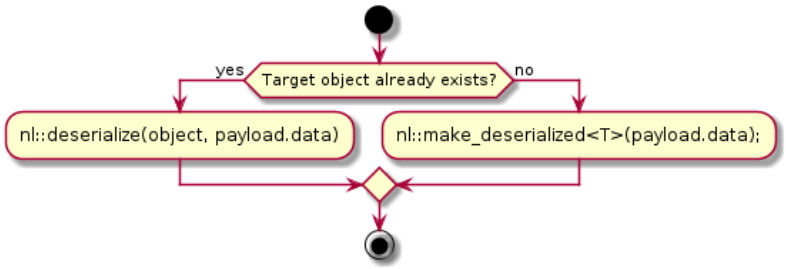
The following diagram shows the required steps to define a serializable type.



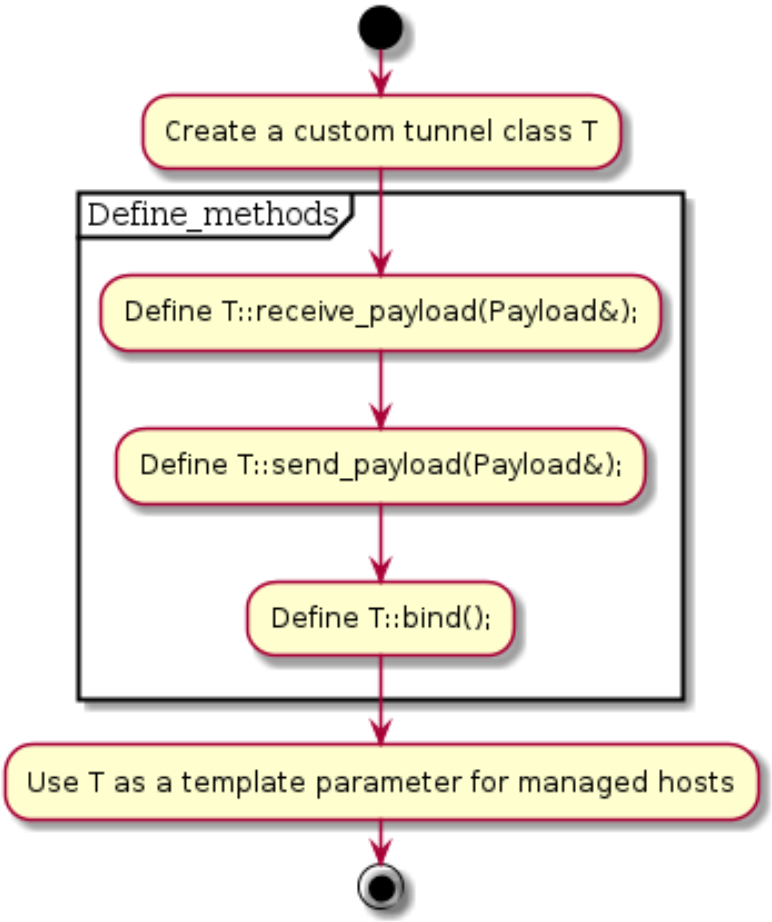
The following diagram shows the required steps to serialize an object into a payload.



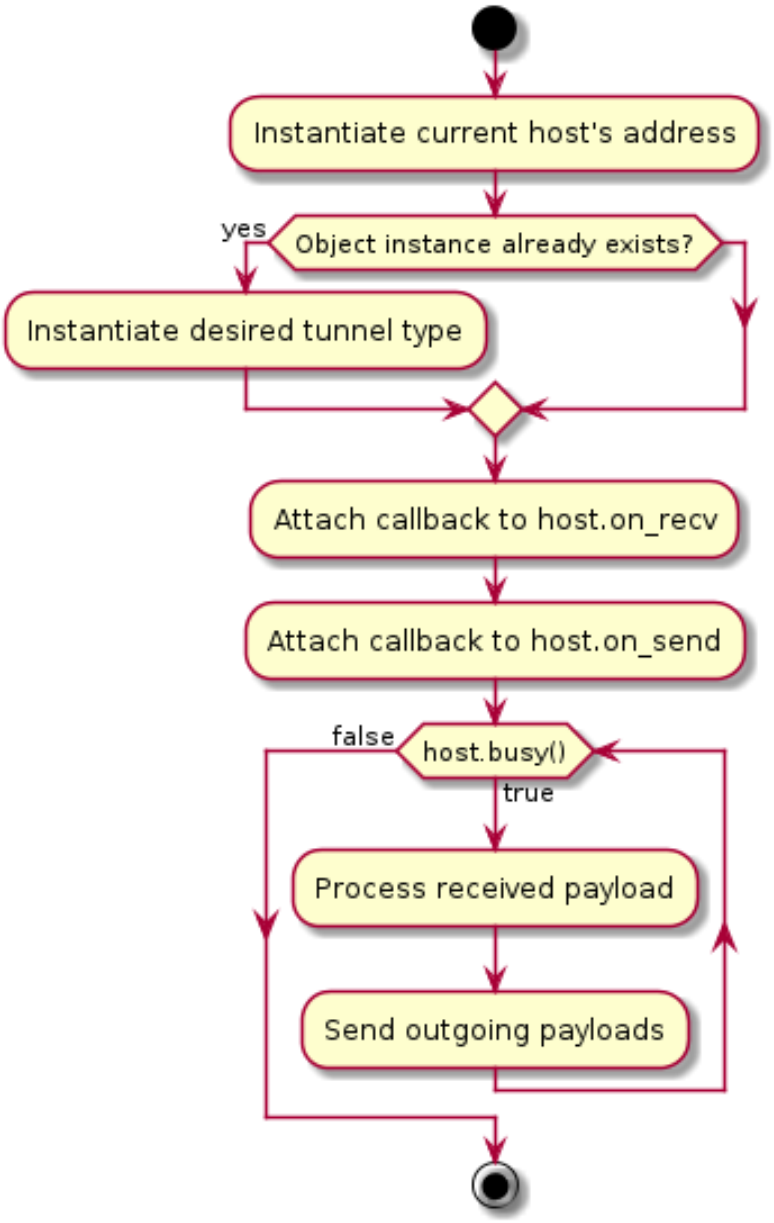
The following diagram shows the required steps to deserialize a payload into an object.



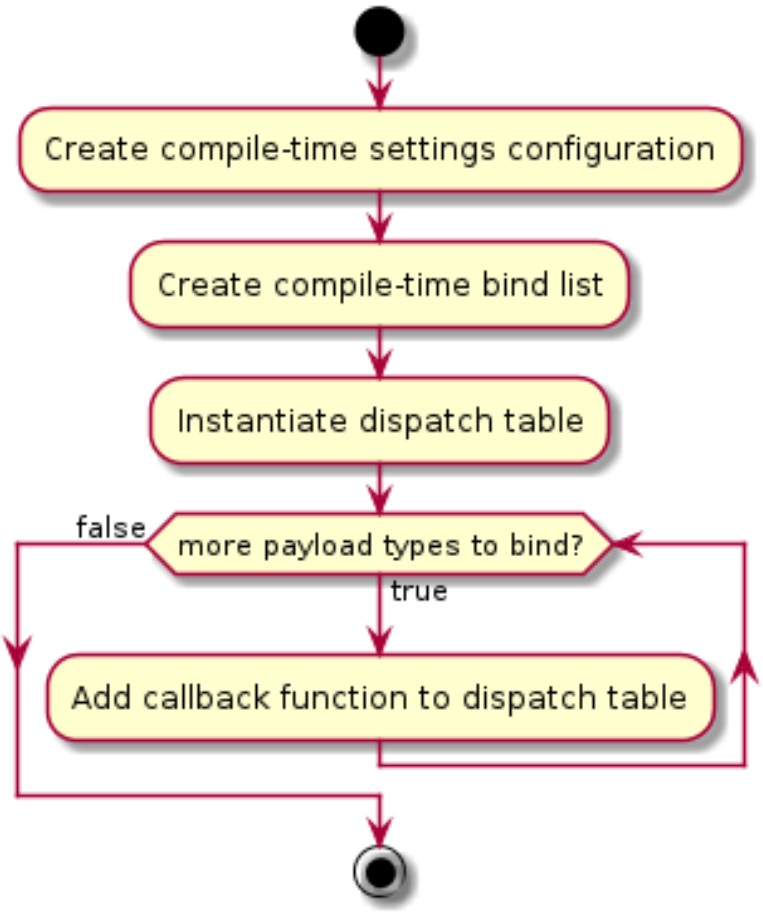
The following diagram shows the required steps to define a tunnel.



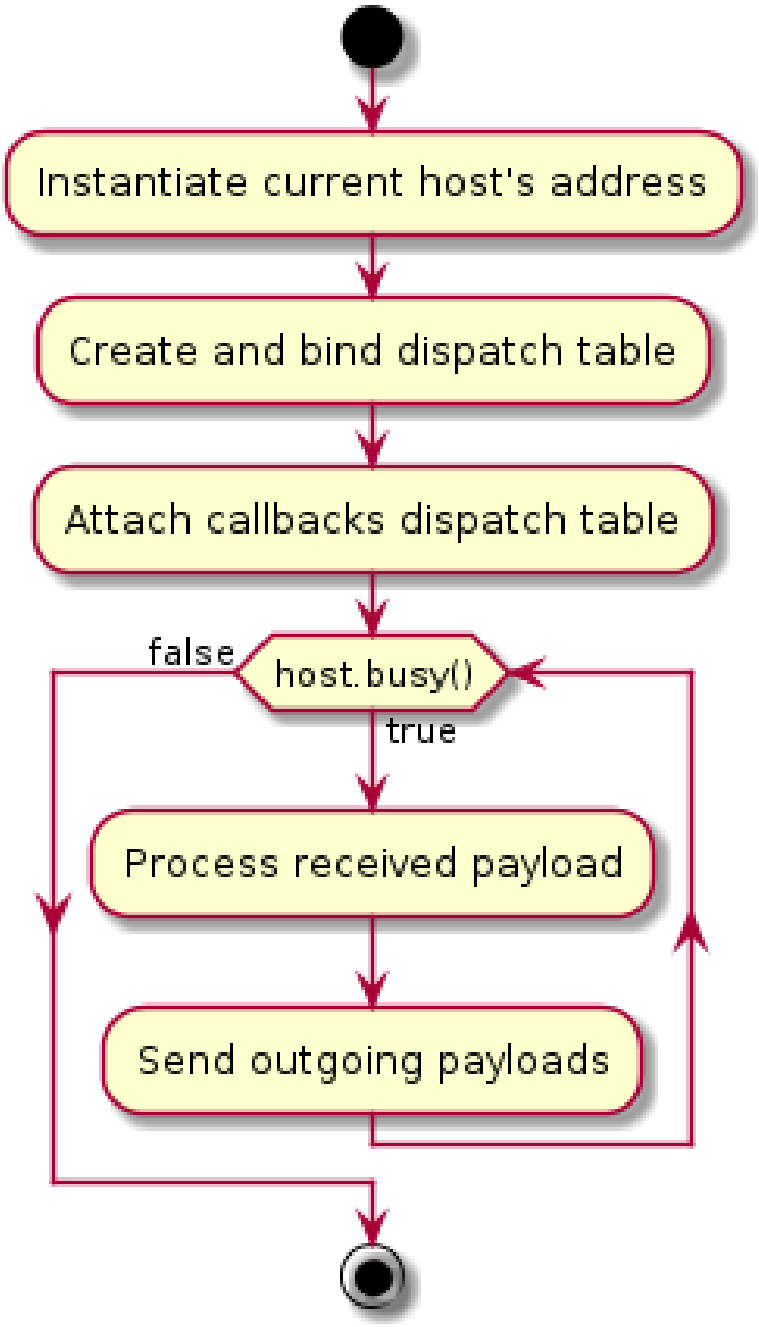
The following diagram shows the required steps to create a managed host.



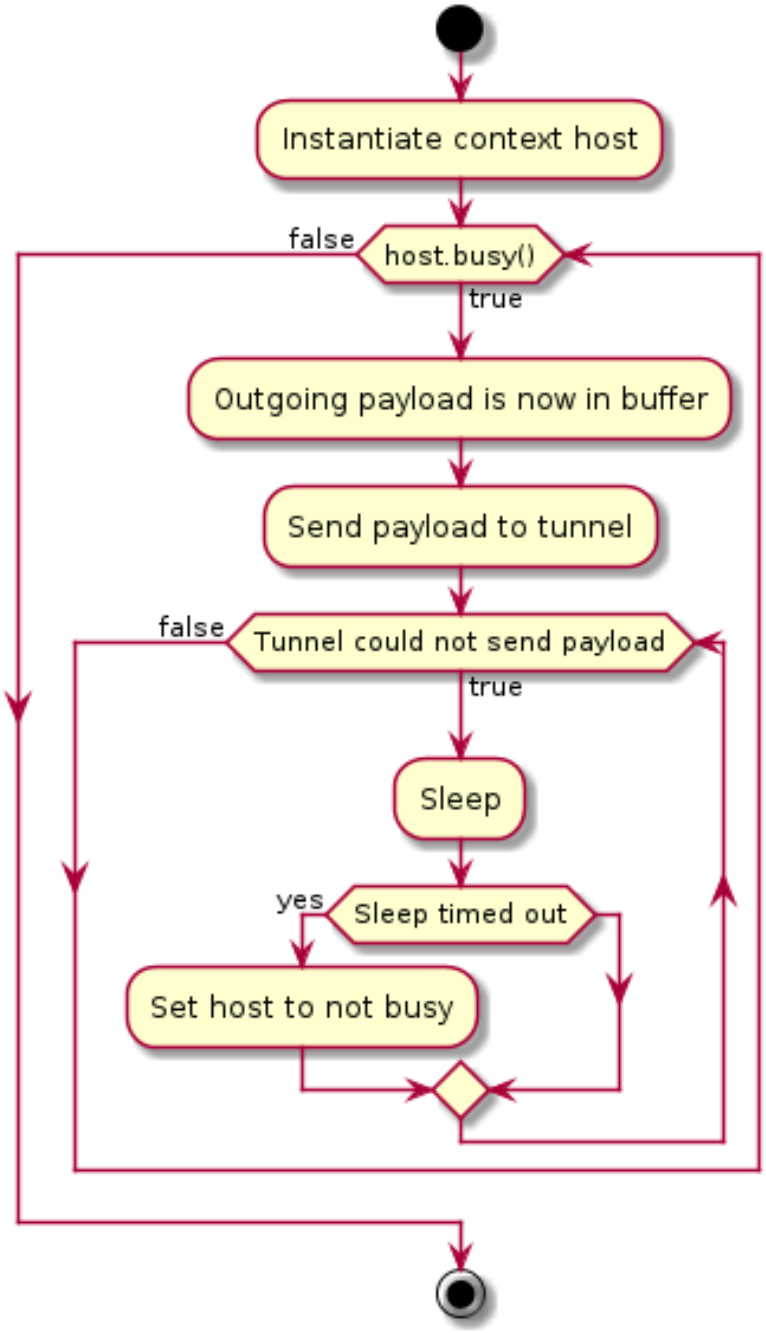
The following diagram shows the required steps to create and bind a dispatch table.



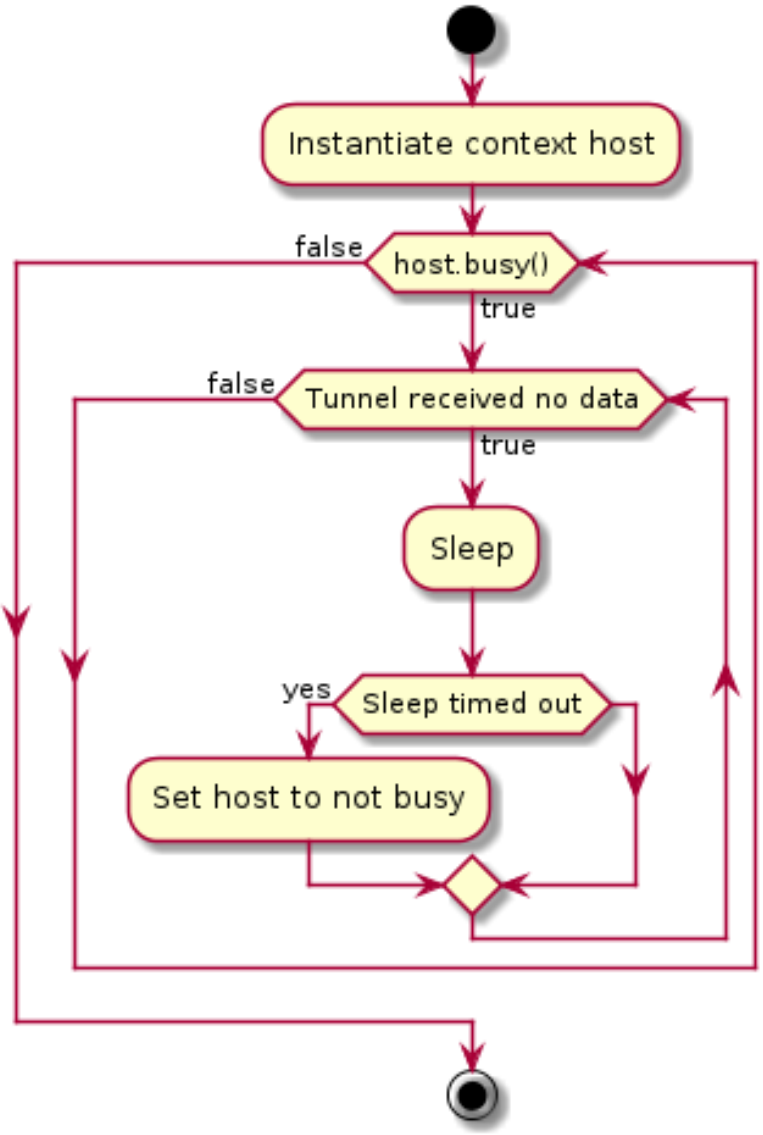
The following diagram shows the required steps to create a context host.



The following diagram shows the required steps to receive a bound payload.



The following diagram shows the required steps to send a bound payload.



5..2 Class diagrams

Class diagrams are created using UML.

The **Unified Modeling Language** (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system.

It offers a way to visualize a system's architectural blueprints in a diagram, including elements such as:

- Any activities (jobs).
- Individual components of the system.
- And how they can interact with other software components.
- How the system will run.
- How entities interact with others (components and interfaces).
- External user interface.

Figure 2.11: NetLayer complete class diagram.

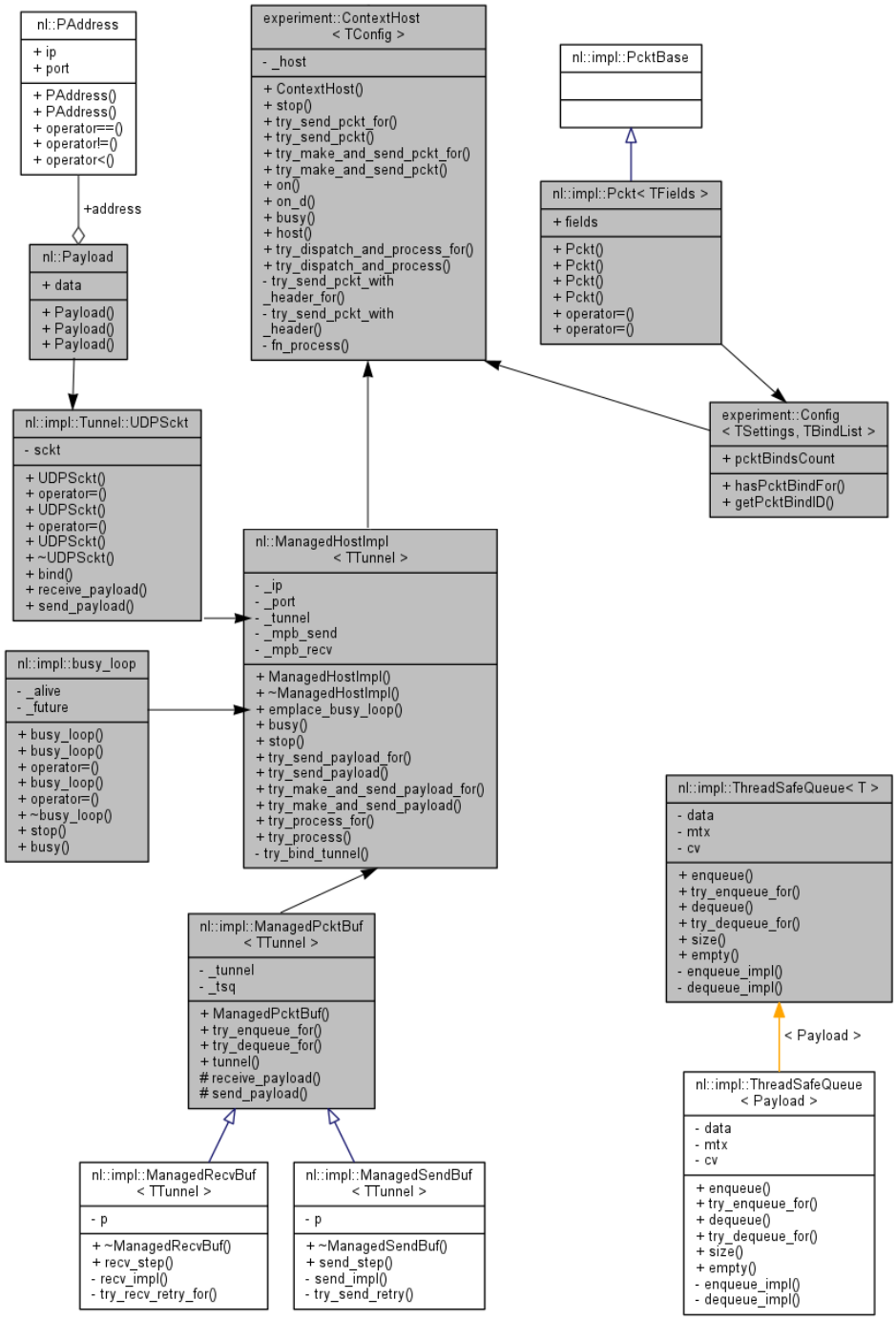


Figure 2.12: Managed buffer collaboration diagram.

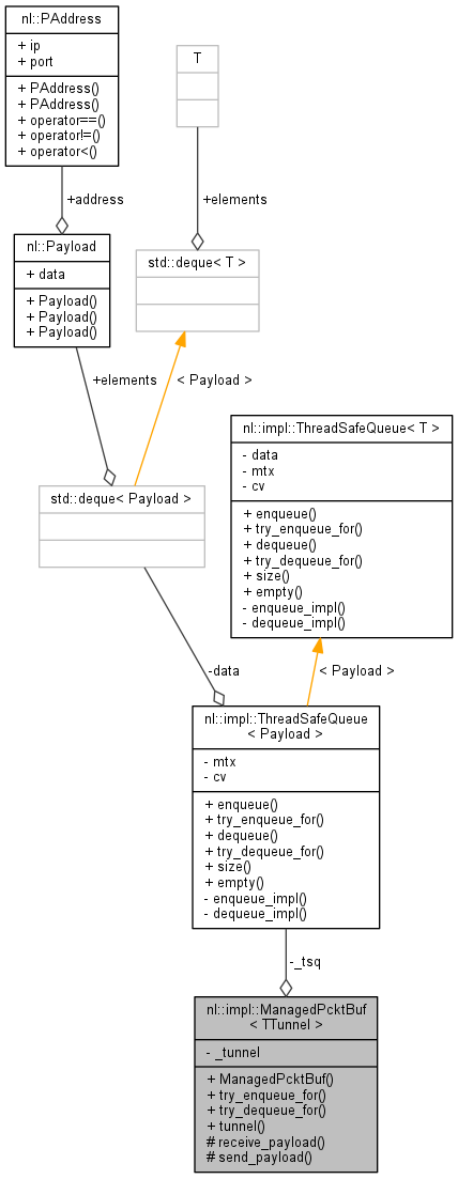


Figure 2.13: Managed host collaboration diagram.

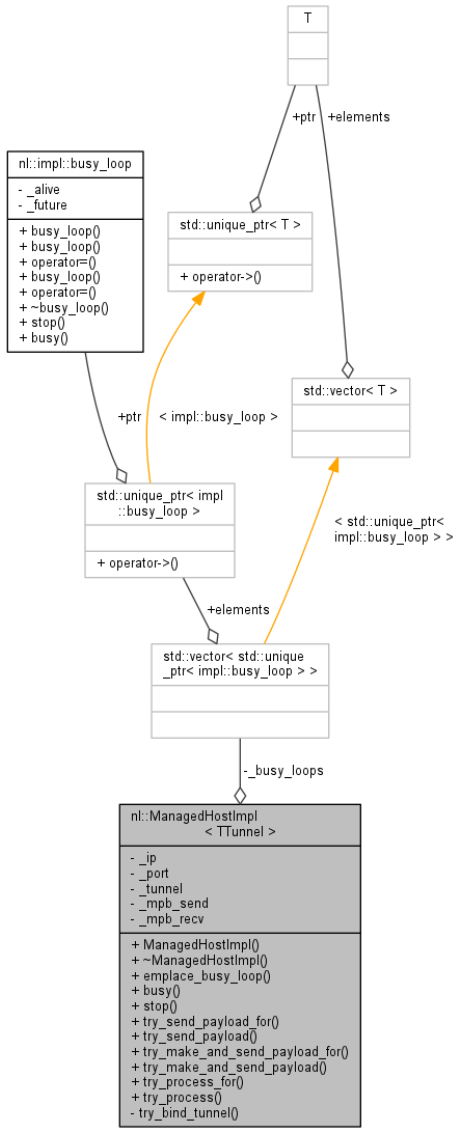


Figure 2.14: Context host collaboration diagram.

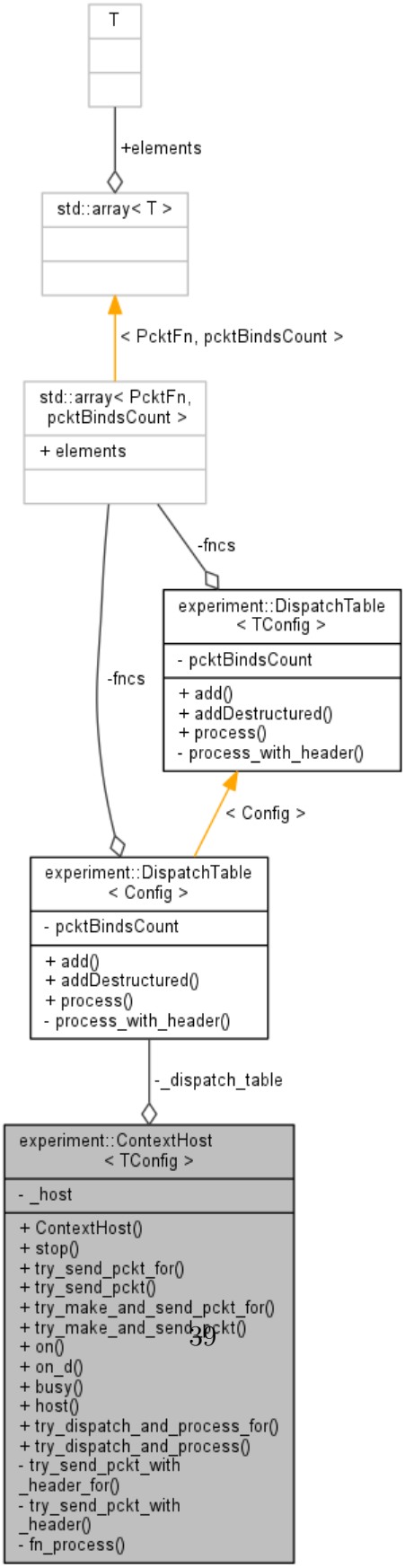
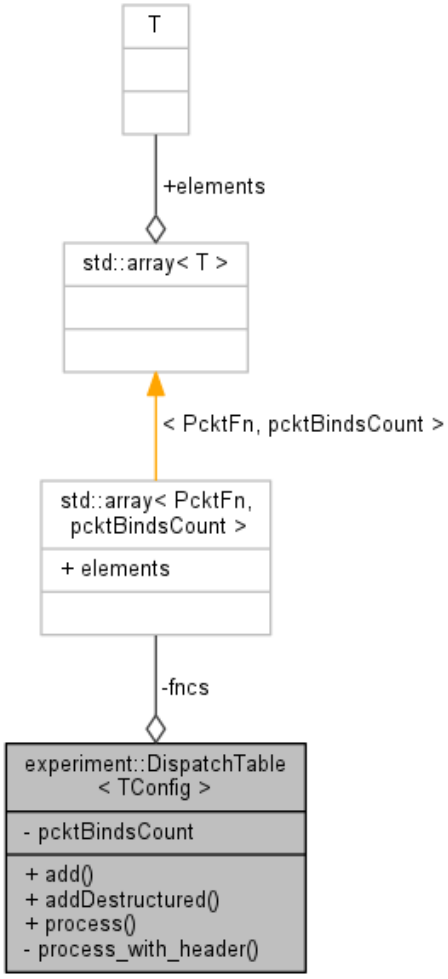


Figure 2.15: Dispatch table collaboration diagram.



Part II

Technical analysis

The following part of the thesis will cover all implementation choices and details for NetLayer in depth.

Firstly, the **development environment and tools** and **chosen technologies** will be described and motivated.

Afterwards, the technical details, including code examples and APIs, will be described for the two modules of the application: the **database** and the **web application**.

Every **table** of the database will be analyzed in detail, directly showing commented **DDL** code. The database also contains important **stored procedures** and **triggers** that are core part of the system's logic and that need to be explained in depth - the related **DML** code will be shown and commented.

The web application itself is divided in multiple modules:

- A **database interface backend module**, that interfaces with the database and wraps its tables and stored procedures.
- A **HTML5 generation module**, that greatly simplifies the creation of dynamic forum web pages by wrapping HTML5 controls in **object-oriented wrappers** that can be easily bound to callbacks and database events.
- A **modern responsive AJAX frontend** that allows users and interact with the backend module from multiple device, limiting postbacks and page refreshes.

Chapter 3

Development process

1. Environment and tools

All modules of veeForum have been developed on **Arch Linux x64**, a lightweight GNU/Linux distribution.

Arch is installed as a minimal base system, configured by the user upon which their own ideal environment is assembled by installing only what is required or desired for their unique purposes. GUI configuration utilities are not officially provided, and most system configuration is performed from the shell and a text editor. Based on a rolling-release model, Arch strives to stay bleeding edge, and typically offers the latest stable versions of most software.

No particular integrated development environments (IDEs) were used during the development - a modern graphical text editor, **Sublime Text 3**, was used instead.

2. Docker

Docker is an open-source project that **automates the deployment of applications** inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux.

Docker uses resource isolation features of the Linux kernel such as **cgroups** and **kernel namespaces** to allow independent containers to run within a single Linux instance.

This technology has been used since the beginning of the development process to

separate veeForum data and packages from the host system and to dramatically increase **portability** and **ease of testing**.

Docker is also used for the installation of the product on target systems - with a single command it is possible to **retrieve all required dependencies**, correctly **configure the system** and **automatically install veeForum**.

3. Version control system

Version control systems (VCSs) allow the **management of changes** to documents, computer programs, large web sites, and other collections of information.

Nowadays, a version control system is **essential** for the development of any project. Being able to track changes, develop features in separate **branches**, have multiple programmers work on the same code base without conflicts and much more is extremely important for projects of any scope and size.

The chosen VCS is **Git**, a distributed revision control system with an emphasis on **speed**, **data integrity**, and support for **distributed, non-linear workflows**.

Git is widely appreciated in the private and open-source programming communities - it was initially designed and developed by **Linus Torvalds** for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development.

The veeForum project is **open-source** and **appreciates feedback and contributions**. It is hosted on **GitHub**, a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git, while adding **additional features** that make collaboration and public contributions easy and accessible.

4. LAMP stack

The server and web application run on a **LAMP stack**, on a GNU/Linux machine.

A LAMP stack is composed by the following technologies:

- **L**: GNU/Linux machine.
- **A**: Apache HTTP server.

The Apache HTTP server is the world's most widely used web server software.

Apache has been under open-source development for about 20 years - it supports all modern server-side technologies and programming languages, and also is **extremely reliable** and **secure**.

- **M**: Stands for MySQL server, but **MariaDB**, a modern drop-in replacement for MySQL is used as the DBMS.

  MariaDB is fully compliant with the MySQL standard and language, but it is more performant and has additional features. It is the default DBMS in the Arch Linux distribution.

  By default, MariaDB uses the **XtraDB** storage engine, a performance enhanced fork of the InnoDB storage engine.

  Percona XtraDB includes all of InnoDB's robust, reliable ACID-compliant design and advanced MVCC architecture, and builds on that solid foundation with more features, more tunability, more metrics, and more scalability. In particular, it is designed to scale better on many cores, to use memory more efficiently, and to be more convenient and useful.

- **P**: PHP5, the server backend language.

  HTML5, PHP5 and JavaScript conformant to the 5.1 ECMAScript specification (along with the JQuery library) are used for the development of the web application.

  The **AJAX** (Asynchronous JavaScript and XML) paradigm will be used to ensure that the application feels responsive and that user interaction is immediately reflected on the web application.

5. Thesis

The current document was written using LaTeX, an high-quality typesetting system; it includes features designed for the production of **technical and scientific documentation**.

LaTeX was chosen for the current document because of the visually pleasant typography, its extensibility features and its abilities to include and highlight source code.

5..1 LatexPP

A small **C++14** LaTeX preprocessor named **LatexPP** was developed for the composition of this thesis.

LatexPP allows to use an intuitive syntax that avoids markup repetition for code highlighting and macros.

Preprocessing and compiling a LaTeX document using LatexPP is simple and can be automated using a simple **bash** script.

```
1  #!/bin/bash
2
3  latexpp ./thesis.lpp > ./thesis.tex
4  pdflatex -shell-escape ./thesis.tex && chromium ./thesis.pdf
```

LatexPP is available as an open-source project on GitHub:
`https://github.com/SuperV1234/Experiments/Random`

Chapter 4

Project structure

The project folder and file structure is organized as such:

- **./doc/**

  Folder containing the documentation of the project.

  - **./latex/**
    LatexPP and LaTeX source and output files.

- **./sql/**

  Folder containing the SQL DDL scripts.

  - **./scripts/**
    Contains all the parts that make up the complete SQL initialization script.
  - **./mkScript.sh**
    Builds the complete SQL initialization scripts from the files in ./scripts/.
  - **./script.sql**
    Complete SQL initialization scripts that sets up a database suitable veeForum.

- **./exe/**

  Folder containing executable scripts to setup the system.

  - **./docker/**

    Docker-related scripts.

    * **./start.sh**
      Starts a Docker instance containing veeForum.
    * **./cleanup.sh**
      Cleans any running veeForum Docker instance.
    * **./shell.sh**
      Starts a Docker instance containing veeForum, controlling an instance of bash inside it.
    * **./httpdLog.sh**
      Prints the Apache error log of the current running veeForum Docker instance.

- **./www/**

  Folder containing web application data.

  - **./css/**

    CSS3 stylesheets.

  - **./js/**

    ECMAScript 5 script files.

  - **./json/**

    Non-relational data storage files, in JSON format.

  - **./php/**

    PHP backend code.

    * **./lib/**
      Backend to database interface library and HTML5 generation library.
    * **./core/**
      PHP frontend files that generate the responsive HTML5 web application user interface.

Chapter 5

Library structure

Figure 5.1: NetLayer modules.

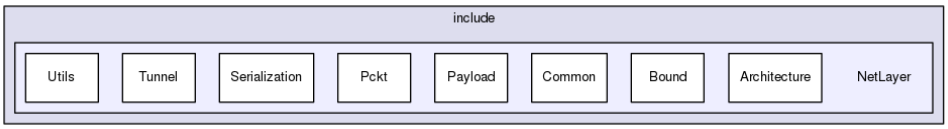


Figure 5.2: NetLayer module dependencies.

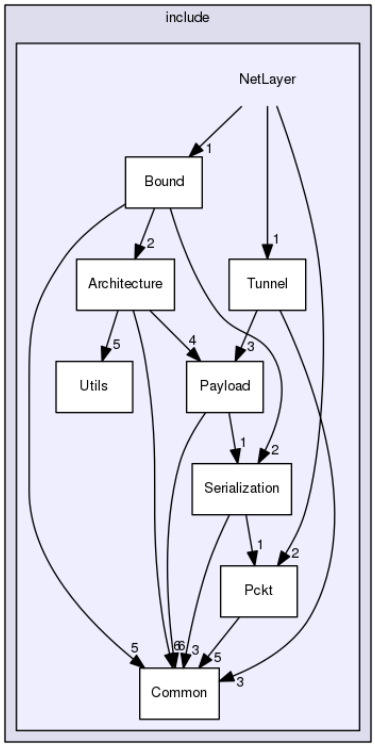


Figure 5.3: Serialization dependency graph.

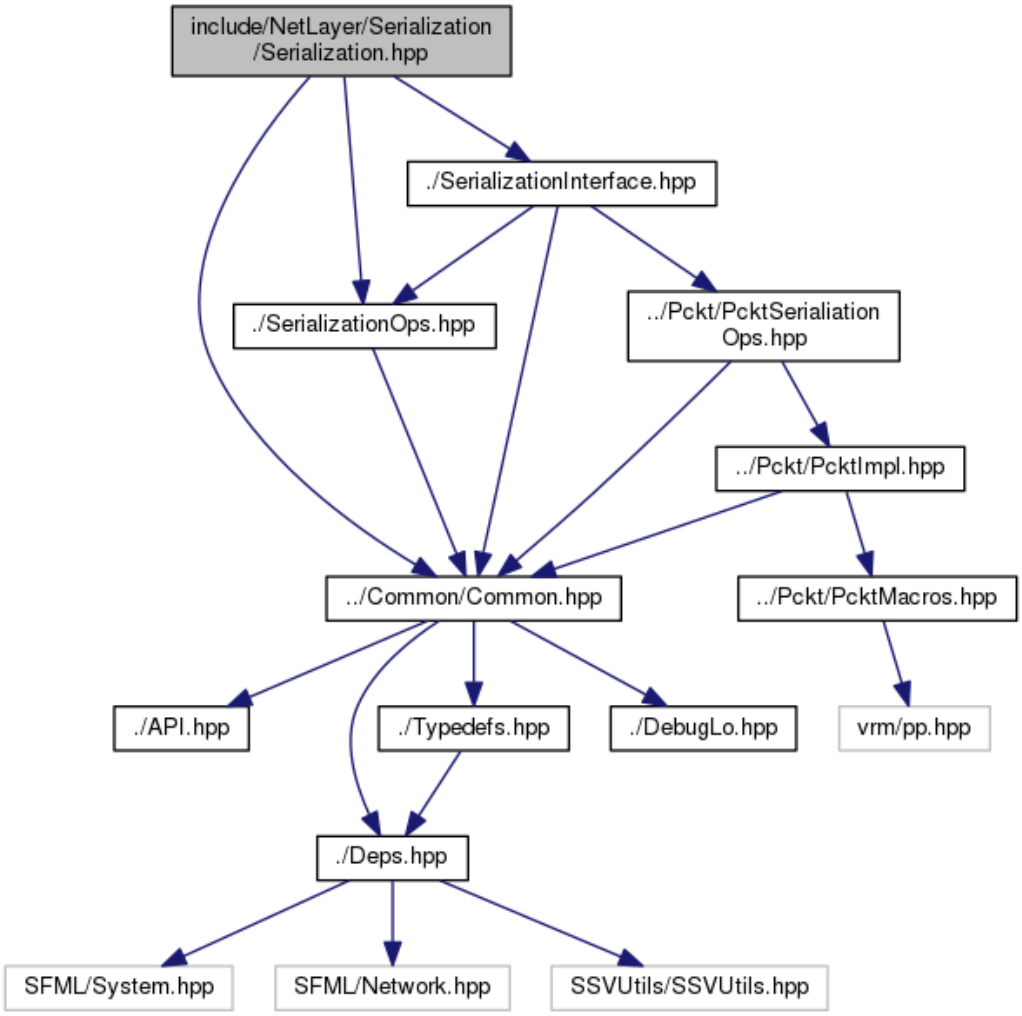


Figure 5.4: Tunnel dependency graph.

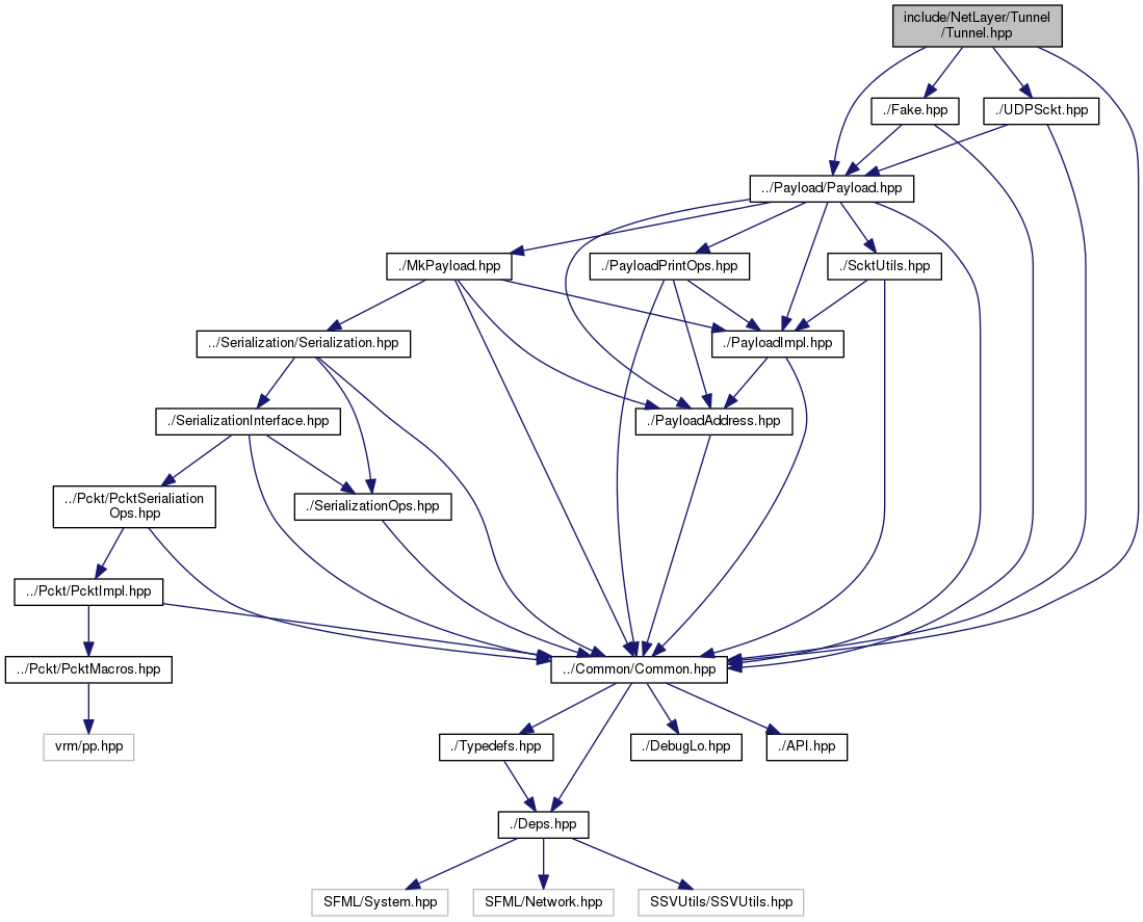


Figure 5.5: Payload dependency graph.

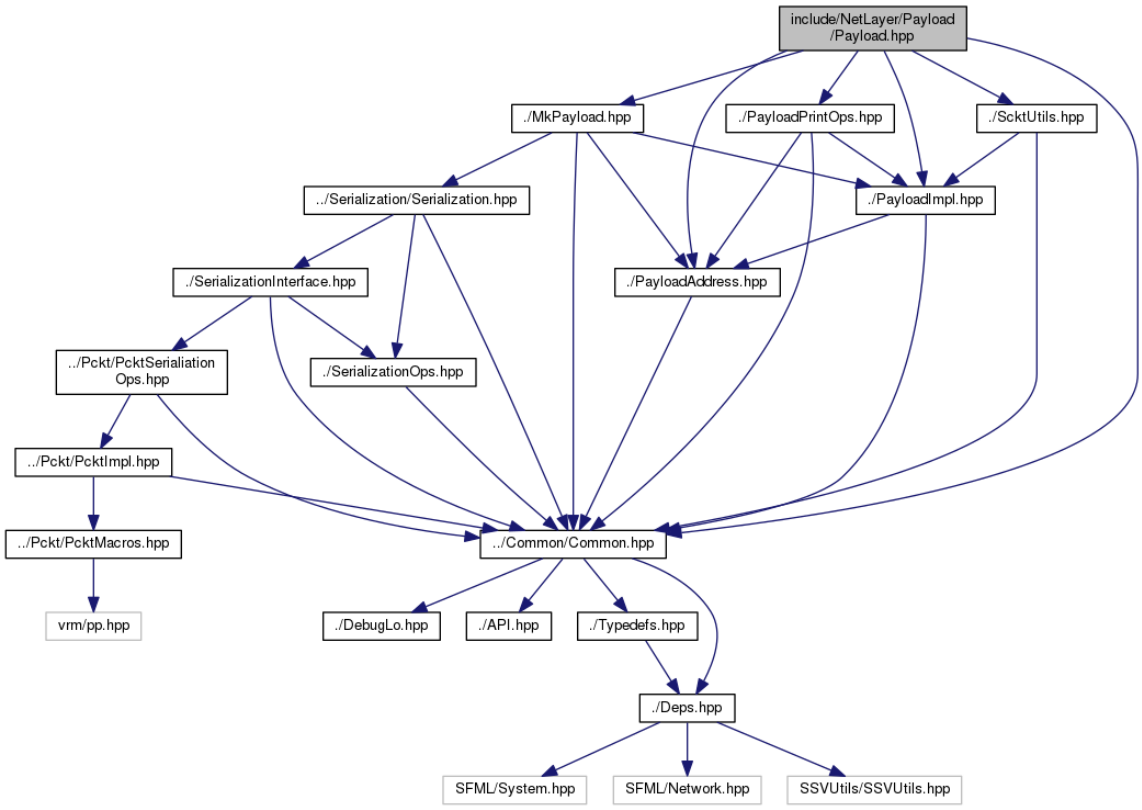


Figure 5.6: Packet types dependency graph.

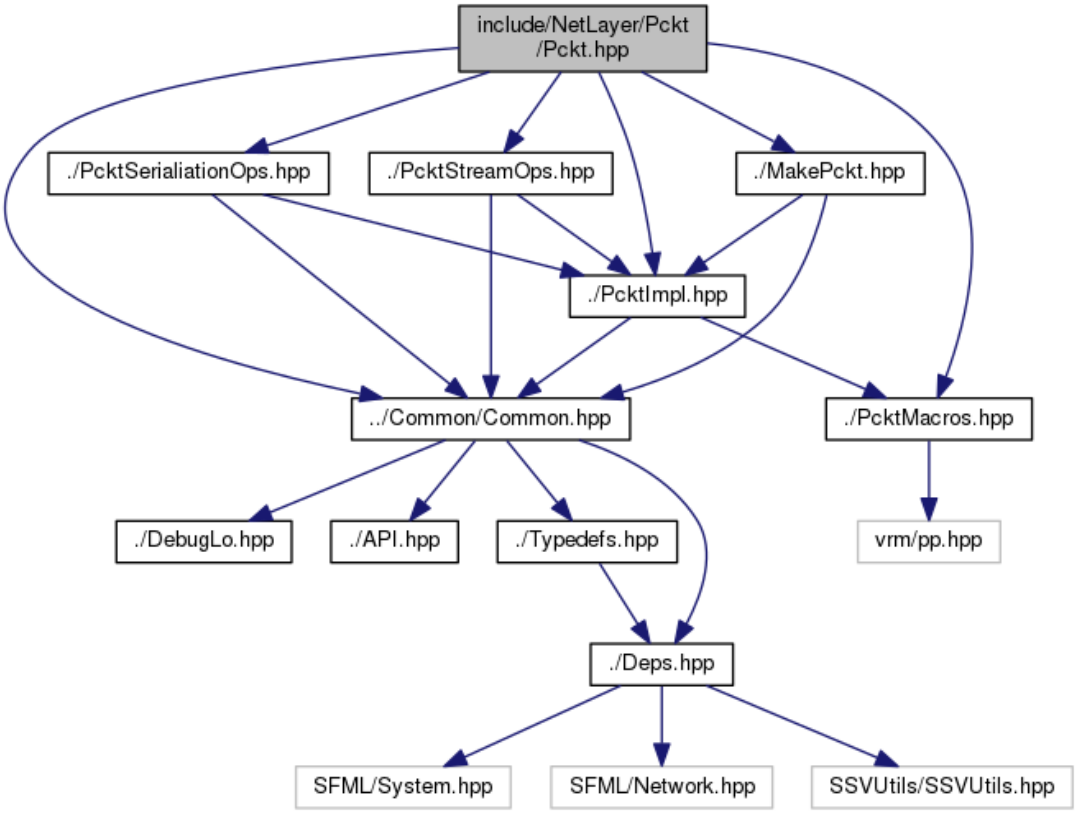
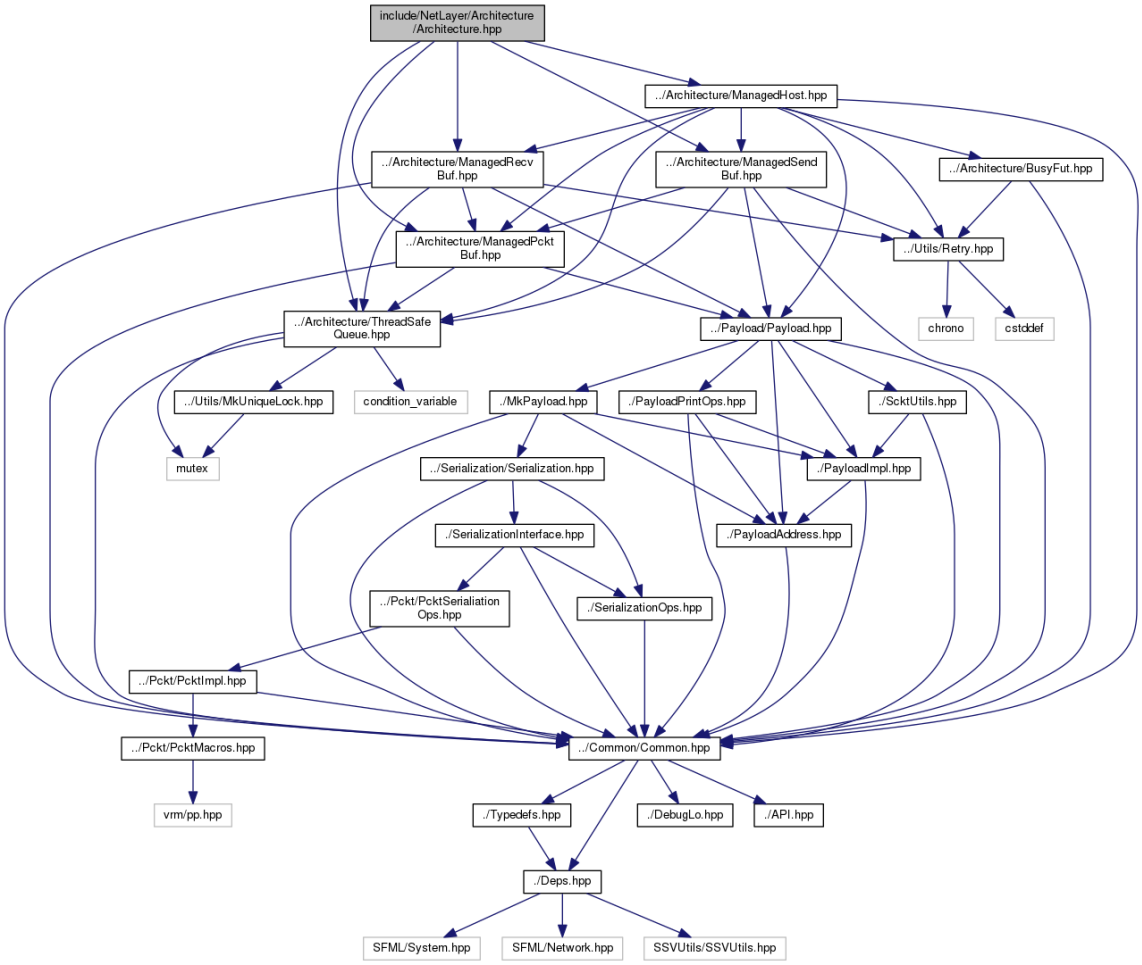


Figure 5.7: Architecture dependency graph.



Chapter 6

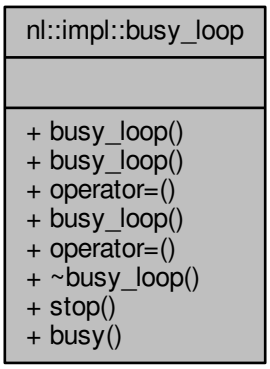
Class documentation

1. nl::impl::busy_loop Class Reference

Wrapper around an `std::future<void>` that runs a busy loop until explicitly stopped by the user.

`#include <BusyFut.hpp>`

Collaboration diagram for nl::impl::busy_loop:



Public Member Functions

- template<typename TF >
  busy_loop (TF &&f)
- **busy_loop** (const busy_loop &)=delete
- busy_loop & **operator=** (const busy_loop &)=delete
- **busy_loop** (busy_loop &&)=default
- busy_loop & **operator=** (busy_loop &&)=default
- void **stop** () noexcept
- bool **busy** () const noexcept

1..1 Detailed Description

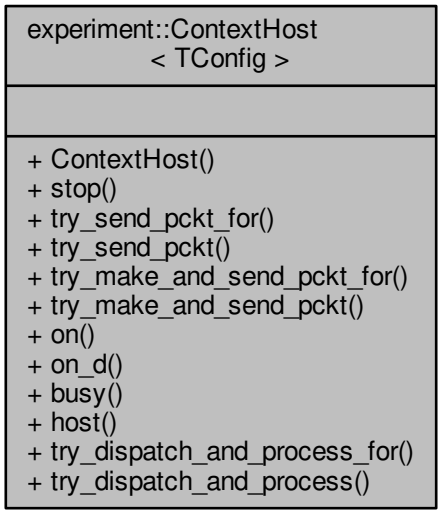
Wrapper around an `std::future<void>` that runs a busy loop until explicitly stopped by the user.

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

- include/NetLayer/Architecture/BusyFut.hpp

2. experiment::ContextHost< TConfig > Class Template Reference

Collaboration diagram for experiment::ContextHost< TConfig >:



Public Member Functions

- **ContextHost** (nl::Port port)
- void **stop** ()
- template<typename T , typename TDuration >
  auto **try_send_pckt_for** (const PAddress &pa, const TDuration &d, T &&pckt)
- template<typename T >
  auto **try_send_pckt** (const PAddress &pa, T &&pckt)
- template<typename T , typename TDuration , typename... Ts>
  auto **try_make_and_send_pckt_for** (const PAddress &pa, const TDuration &d, Ts &&...xs)

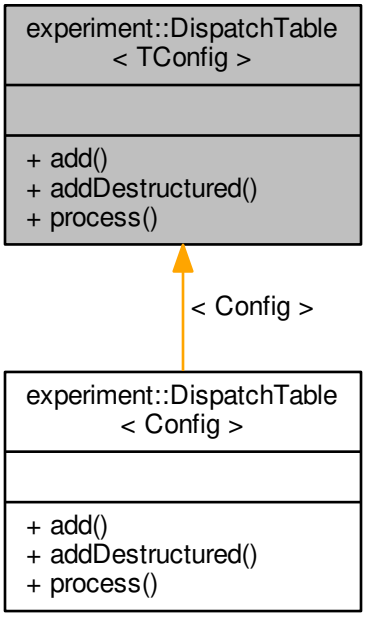
- template<typename T , typename... Ts>
  auto **try_make_and_send_pckt** (const PAddress &pa, Ts &&...xs)
- template<typename TPckt , typename TF >
  void **on** (TF &&fn)
- template<typename TPckt , typename TF >
  void **on_d** (TF &&fn)
- bool **busy** () const noexcept
- auto & **host** () noexcept
- template<typename TDuration >
  auto **try_dispatch_and_process_for** (const TDuration &d)
- auto **try_dispatch_and_process** ()

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

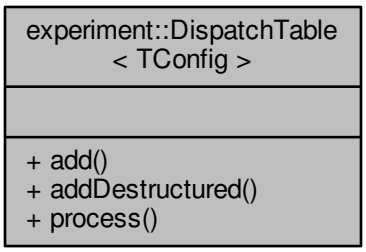
- include/NetLayer/Bound/ContextHost.hpp

3. experiment::DispatchTable< TConfig > Class Template Reference

Inheritance diagram for experiment::DispatchTable< TConfig >:



Collaboration diagram for experiment::DispatchTable< TConfig >:



Public Member Functions

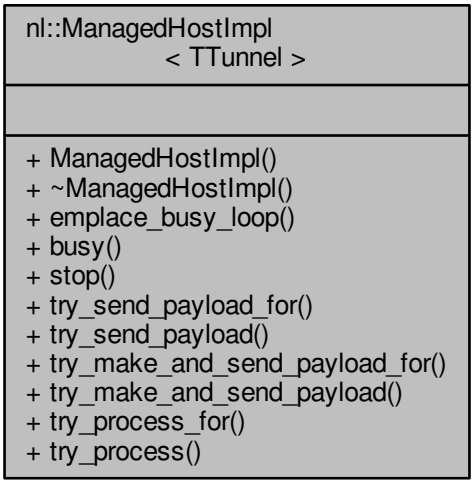
- template<typename T , typename TF >
  void **add** (TF &&fnToCall)
- template<typename T , typename TF >
  void **addDestructured** (TF &&fnToCall)
- void **process** (const PAddress &sender, PcktBuf &p)

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

- include/NetLayer/Bound/DispatchTable.hpp

4. nl::ManagedHostImpl< TTunnel > Class Template Reference

Collaboration diagram for nl::ManagedHostImpl< TTunnel >:



Public Member Functions

- template<typename... TTunnelArgs>
  ManagedHostImpl (Port port, TTunnelArgs &&...ts)
- template<typename TF >
  auto & **emplace_busy_loop** (TF &&f)
- bool **busy** () const noexcept
- void **stop** ()
- template<typename TDuration >
  auto **try_send_payload_for** (Payload &p, const TDuration &d)
- auto **try_send_payload** (Payload &p)

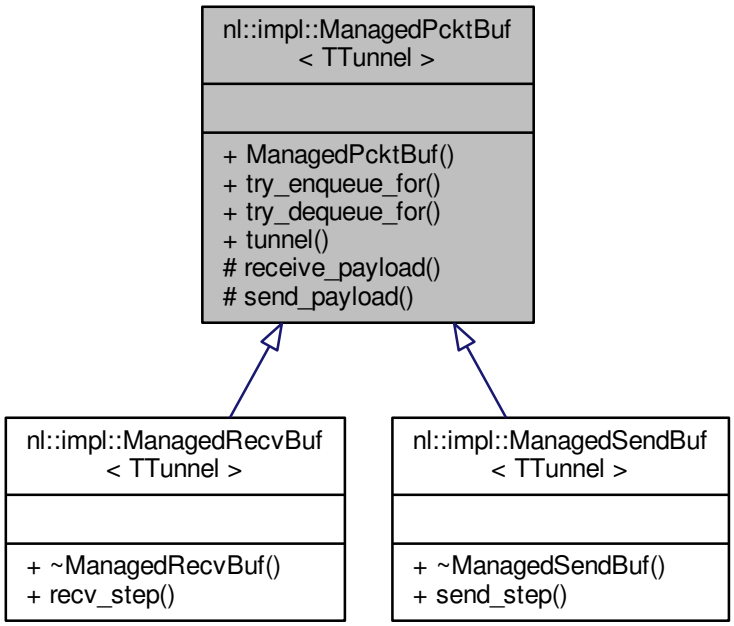
- template<typename TDuration , typename... Ts>
  auto **try_make_and_send_payload_for** (const PAddress &pa, const TDuration &d, Ts &&...xs)
- template<typename... Ts>
  auto **try_make_and_send_payload** (const PAddress &pa, Ts &&...xs)
- template<typename TF , typename TDuration >
  bool **try_process_for** (const TDuration &d, TF &&f)
- template<typename TF >
  bool **try_process** (TF &&f)

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

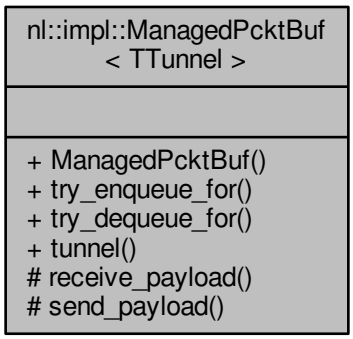
- include/NetLayer/Architecture/ManagedHost.hpp

5. nl::impl::ManagedPcktBuf< TTunnel > Class Template Reference

Inheritance diagram for nl::impl::ManagedPcktBuf< TTunnel >:



Collaboration diagram for nl::impl::ManagedPcktBuf< TTunnel >:



Public Member Functions

- **ManagedPcktBuf** (TTunnel &t)
- template<typename TDuration , typename... Ts>
  bool **try_enqueue_for** (const TDuration &d, Ts &&...xs)
- template<typename TDuration >
  bool **try_dequeue_for** (const TDuration &d, Payload &p)
- auto & **tunnel** () noexcept

Protected Member Functions

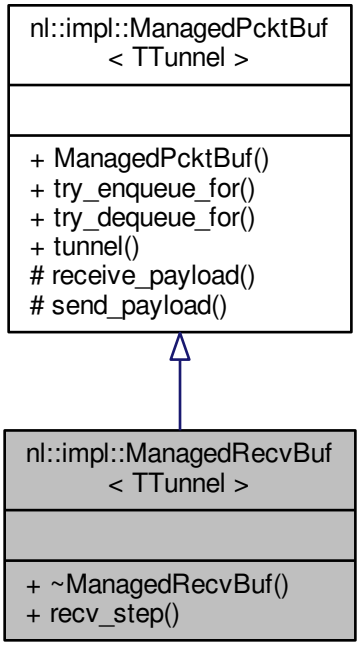
- auto **receive_payload** (Payload &p)
- auto **send_payload** (Payload &p)

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

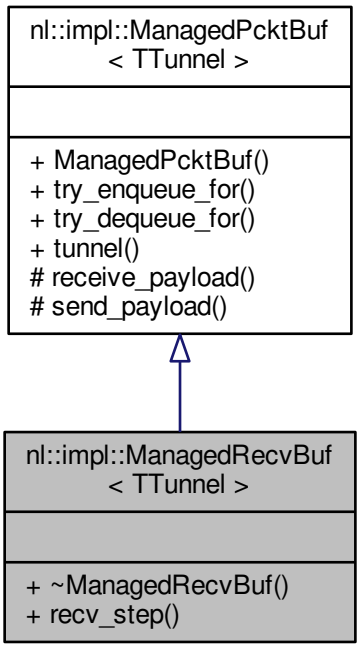
- include/NetLayer/Architecture/ManagedPcktBuf.hpp

6. nl::impl::ManagedRecvBuf< TTunnel > Class Template Reference

Inheritance diagram for nl::impl::ManagedRecvBuf< TTunnel >:



Collaboration diagram for nl::impl::ManagedRecvBuf< TTunnel >:



Public Member Functions

- auto **recv_step** ()

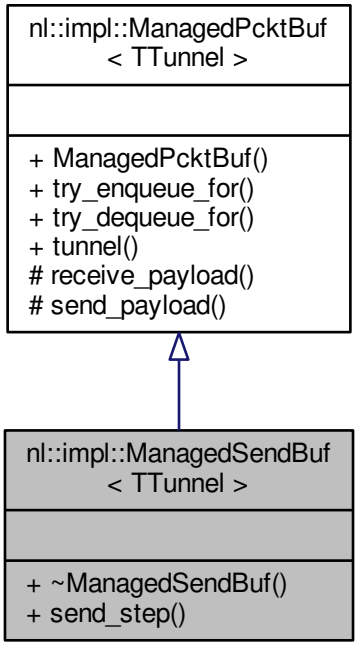
Additional Inherited Members

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

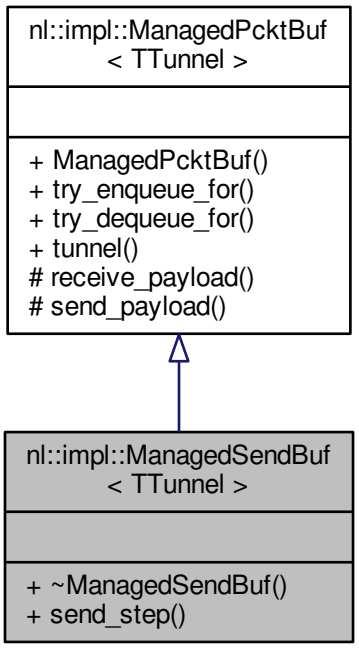
- include/NetLayer/Architecture/ManagedRecvBuf.hpp

7. nl::impl::ManagedSendBuf< TTunnel > Class Template Reference

Inheritance diagram for nl::impl::ManagedSendBuf< TTunnel >:



Collaboration diagram for nl::impl::ManagedSendBuf< TTunnel >:



Public Member Functions

- auto **send_step** ()

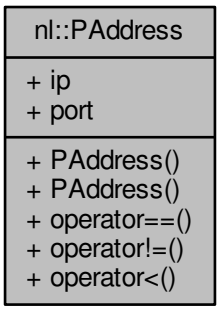
Additional Inherited Members

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

- include/NetLayer/Architecture/ManagedSendBuf.hpp

8. nl::PAddress Struct Reference

Collaboration diagram for nl::PAddress:



Public Member Functions

- **PAddress** (const IpAddr &mIp, Port mPort) noexcept
- bool **operator==** (const PAddress &rhs) const noexcept
- bool **operator!=** (const PAddress &rhs) const noexcept
- bool **operator<** (const PAddress &rhs) const noexcept

Public Attributes

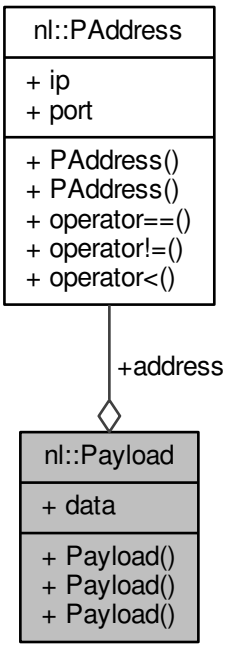
- IpAddr **ip**
- Port **port**

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

- include/NetLayer/Payload/PayloadAddress.hpp

9. nl::Payload Struct Reference

Collaboration diagram for nl::Payload:



Public Member Functions

- **Payload** (const PAddress &mAddress) noexcept
- template<typename TData >
  Payload (const PAddress &mAddress, TData &&mData)

Public Attributes

- PAddress **address**

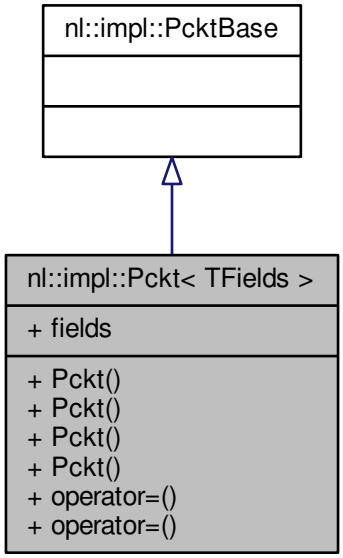
- PcktBuf **data**

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

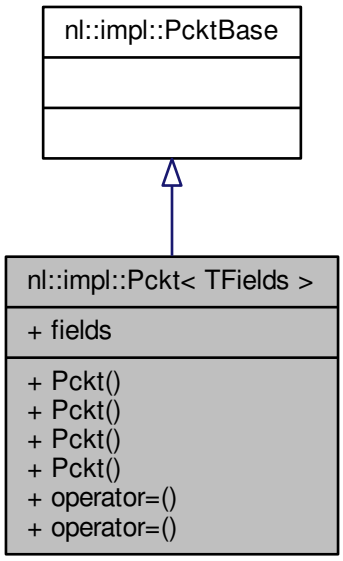
- include/NetLayer/Payload/PayloadImpl.hpp

10. nl::impl::Pckt< TFields > Struct Template Reference

Inheritance diagram for nl::impl::Pckt< TFields >:



Collaboration diagram for nl::impl::Pckt< TFields >:



Public Types

- using **TplType** = std::tuple< TFields... >

Public Member Functions

- template<typename... Ts>
  Pckt (nl::init_fields, Ts &&...mX)
- **Pckt** (const Pckt &mX)
- **Pckt** (Pckt &&mX)
- Pckt & **operator=** (const Pckt &mX)
- Pckt & **operator=** (Pckt &&mX)

Public Attributes

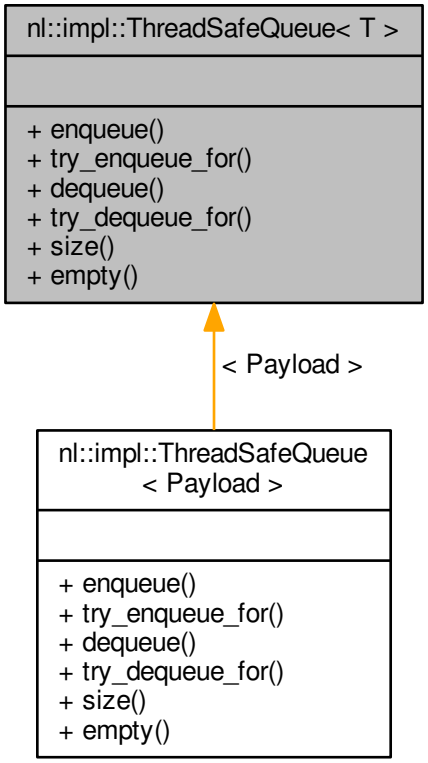
- TplType **fields**

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

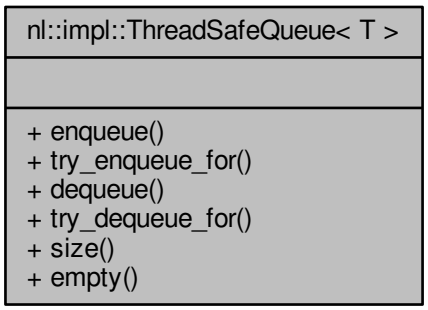
- include/NetLayer/Pckt/PcktImpl.hpp

11. nl::impl::ThreadSafeQueue< T > Class Template Reference

Inheritance diagram for nl::impl::ThreadSafeQueue< T >:



Collaboration diagram for nl::impl::ThreadSafeQueue< T >:



Public Member Functions

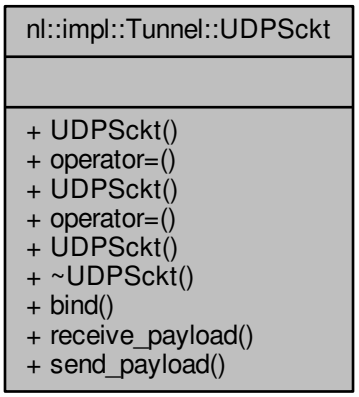
- template<typename... TArgs>
  void **enqueue** (TArgs &&...mArgs)
- template<typename TDuration , typename... TArgs>
  bool **try_enqueue_for** (const TDuration &mDuration, TArgs &&...mArgs)
- T **dequeue** ()
- template<typename TDuration >
  bool **try_dequeue_for** (const TDuration &mDuration, T &mOut)
- auto **size** () const
- auto **empty** () const

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

- include/NetLayer/Architecture/ThreadSafeQueue.hpp

12. nl::impl::Tunnel::UDPSckt Class Reference

Collaboration diagram for nl::impl::Tunnel::UDPSckt:



Public Member Functions

- **UDPSckt** (const UDPSckt &)=delete
- UDPSckt & **operator=** (const UDPSckt &)=delete
- **UDPSckt** (UDPSckt &&)=default
- UDPSckt & **operator=** (UDPSckt &&)=default
- template<typename... Ts>
  UDPSckt (Ts &&...xs)
- bool **bind** (Port x)
- auto **receive_payload** (Payload &p)
- auto **send_payload** (Payload &p)

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

- include/NetLayer/Tunnel/UDPSckt.hpp