Universita' degli Studi di Messina
Dipartimento di Matematica e Informatica

Programming II project

NetLayer
11 June 2015

Authors:

Vittorio Romeo

Professors:

Massimo Villari

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-scratchdevelopment.

- 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 more functional interface 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.

9

- **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.
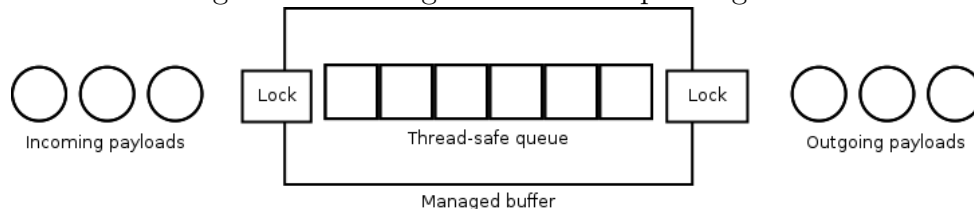
4..2.2 Data serialization

- **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.
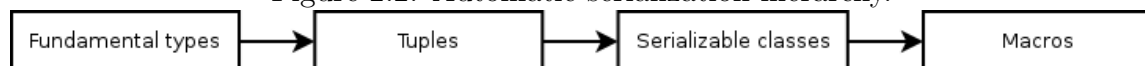
4..2.3   Tunnel management

- **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.
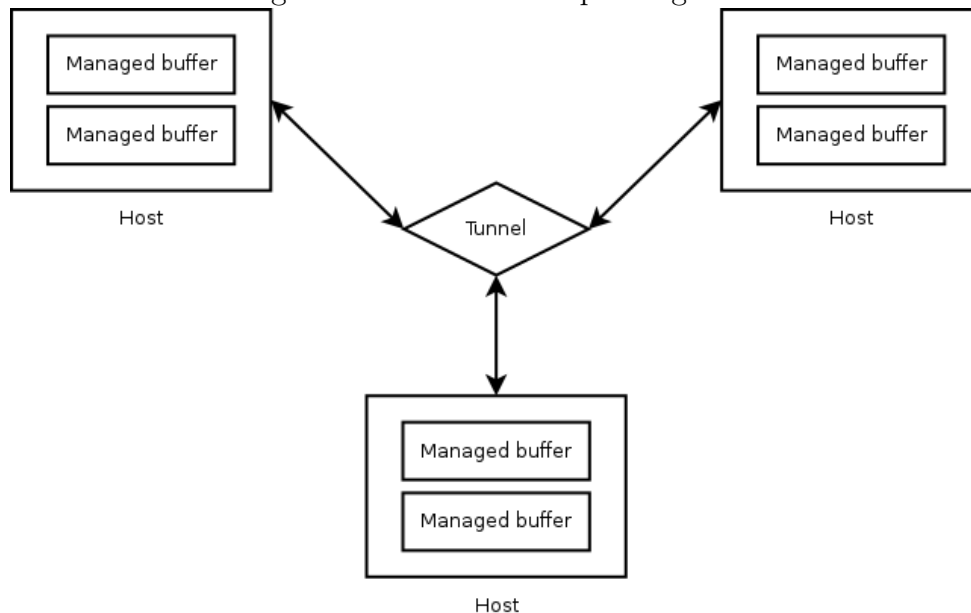
4..2.4   Context binding

- **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.
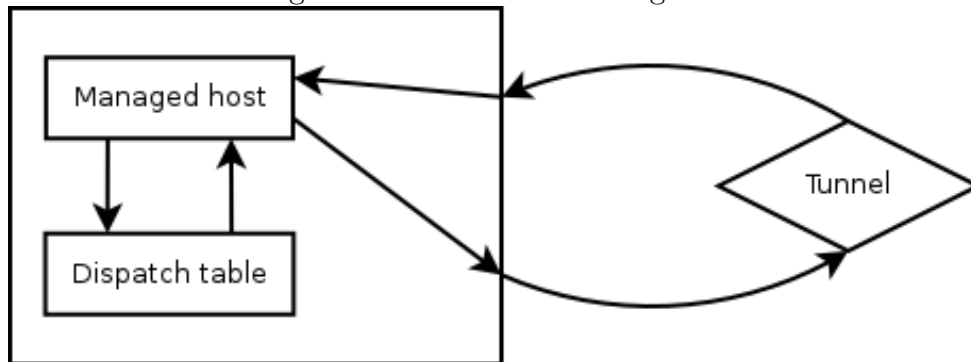
4..3   Example use cases

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.
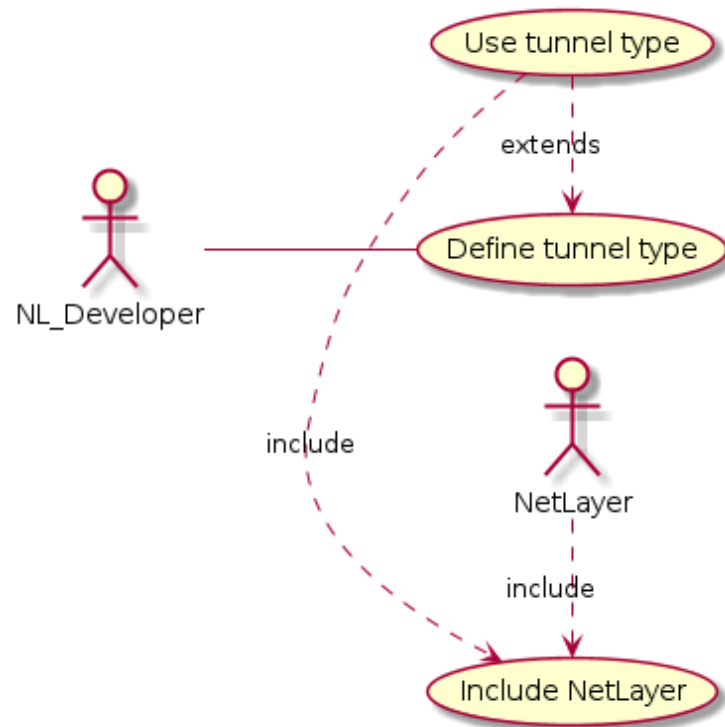
4..3.2   Defining serializable types

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.
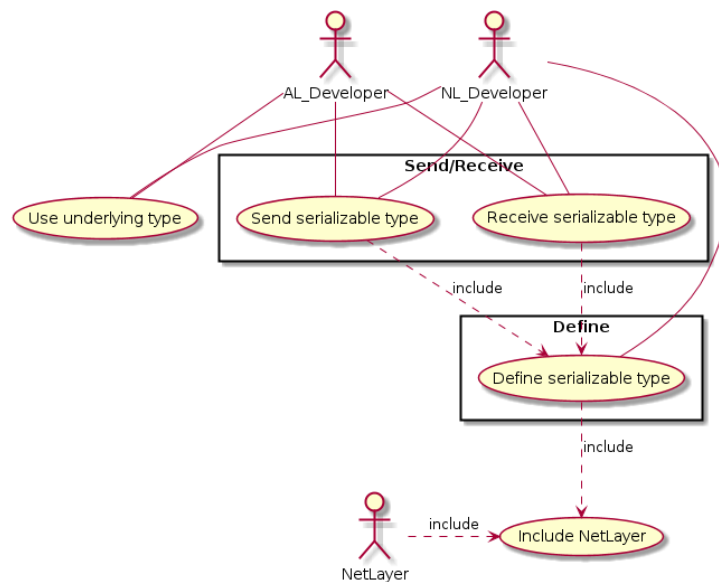
4..3.3 Binding types to dispatch table

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.
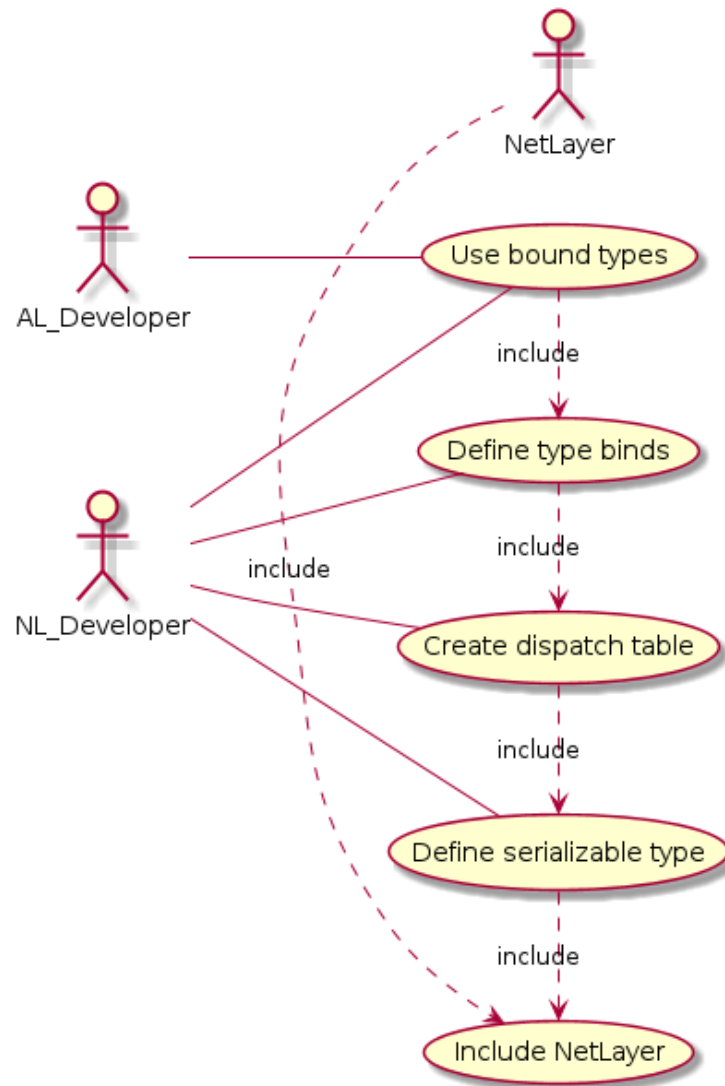
4..3.4 Defining a context host

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.
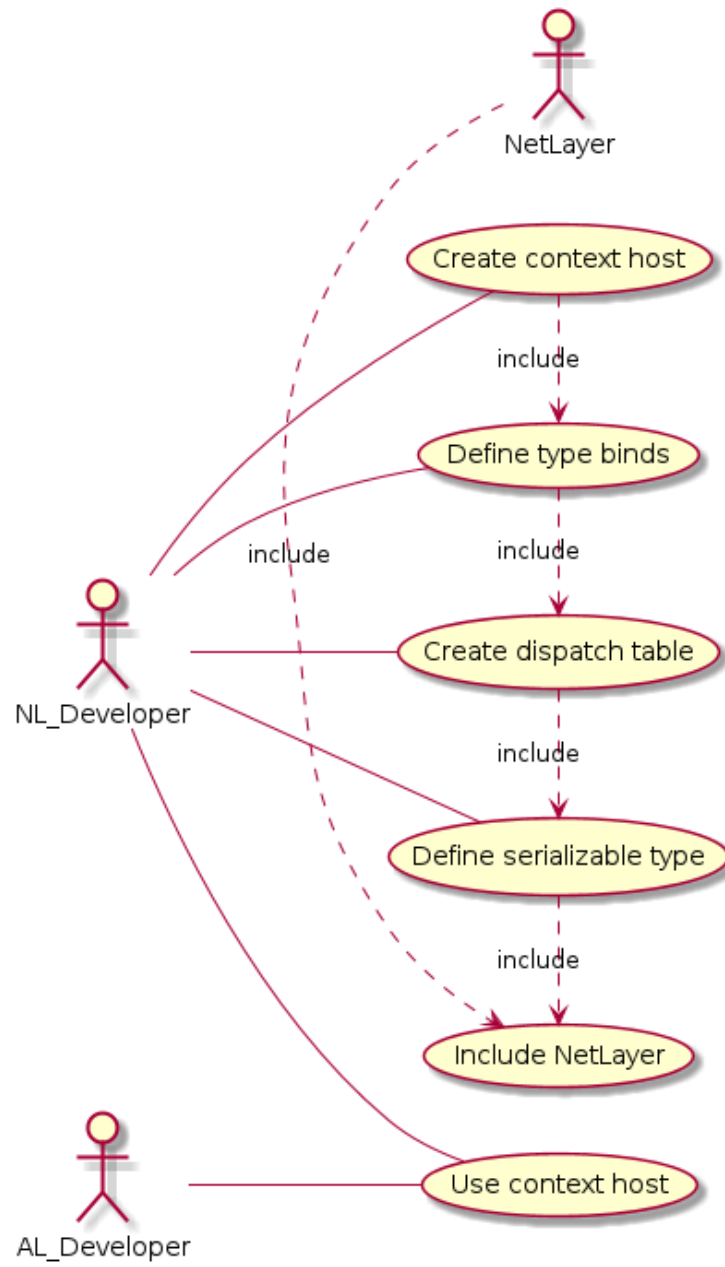
4..3.5 Handling incoming payloads

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.
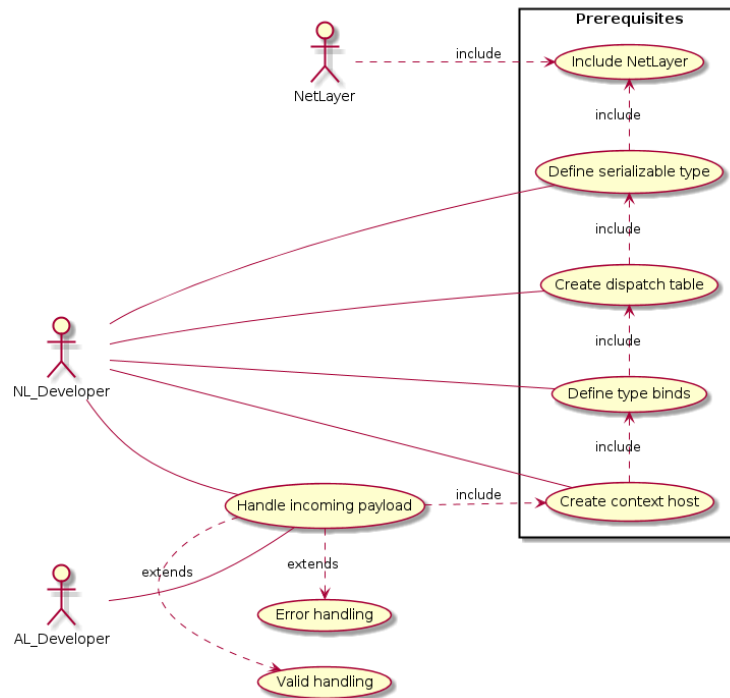
4..3.6   Handling outgoing payloads

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.
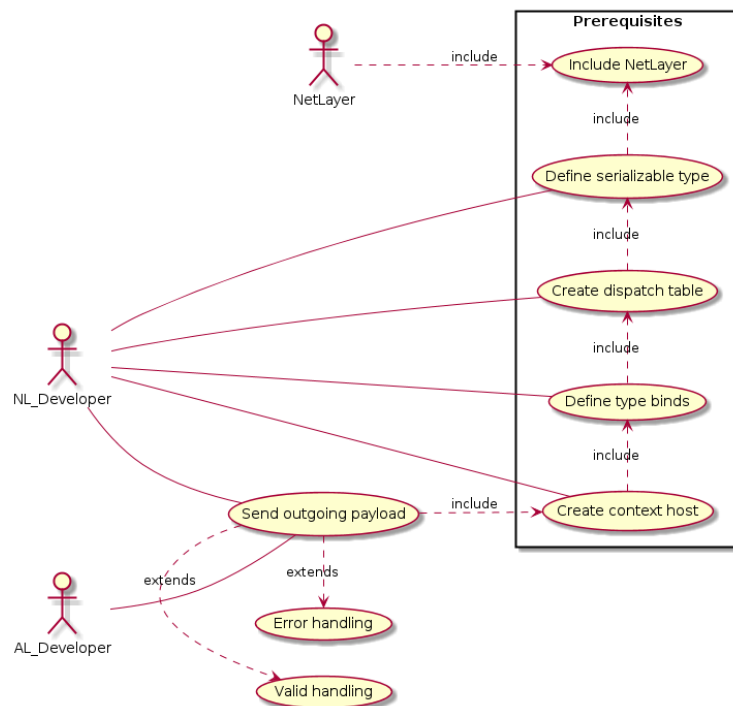
4..4 Non-functional requirements

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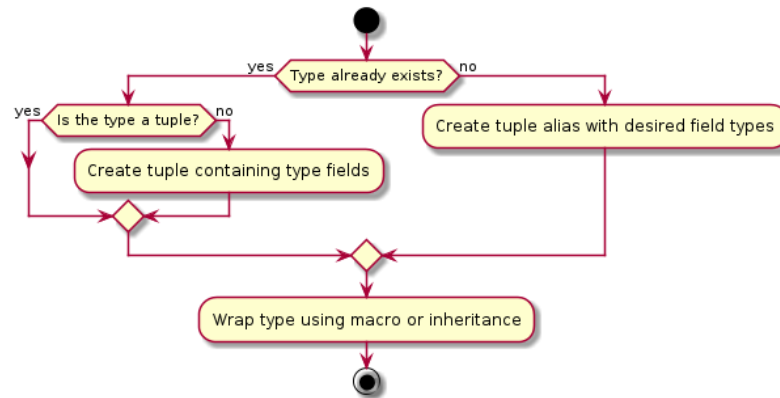
The following diagram shows the required steps to serialize an object into a payload.
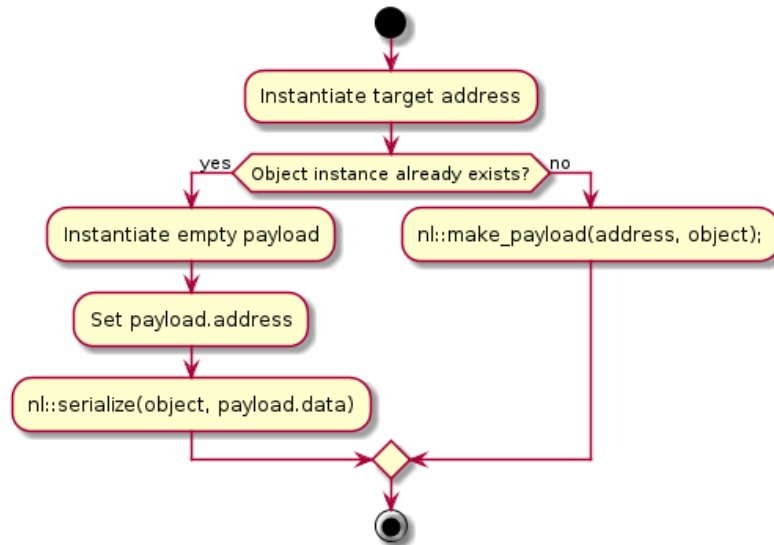
The following diagram shows the required steps to deserialize a payload into an object.
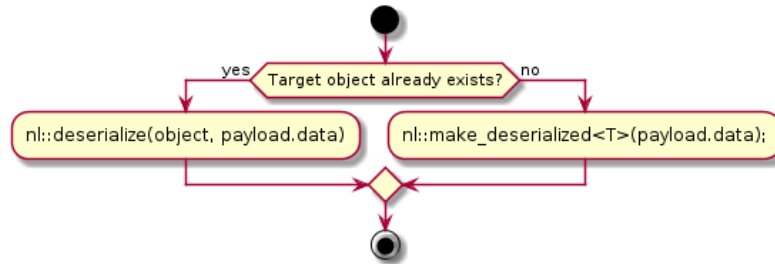
The following diagram shows the required steps to define a tunnel.

Create a custom tunnel class T

Define_methods

Define T::receive_payload(Payload&);

Define T::send_payload(Payload&);

Define T::bind();

Use T as a template parameter for managed hosts
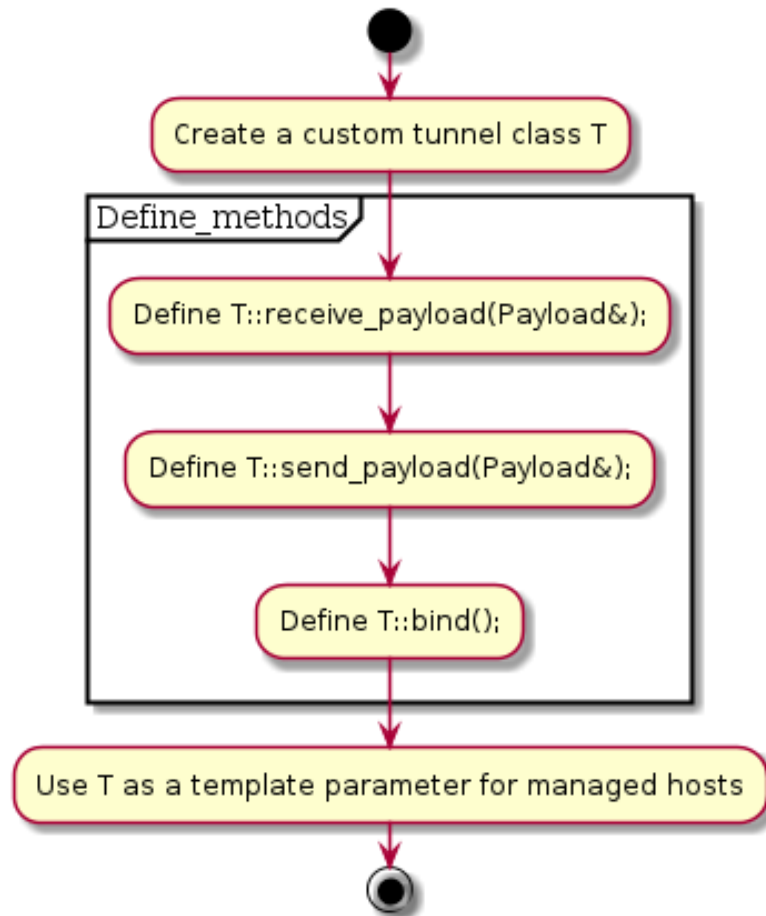
The following diagram shows the required steps to create a managed host.
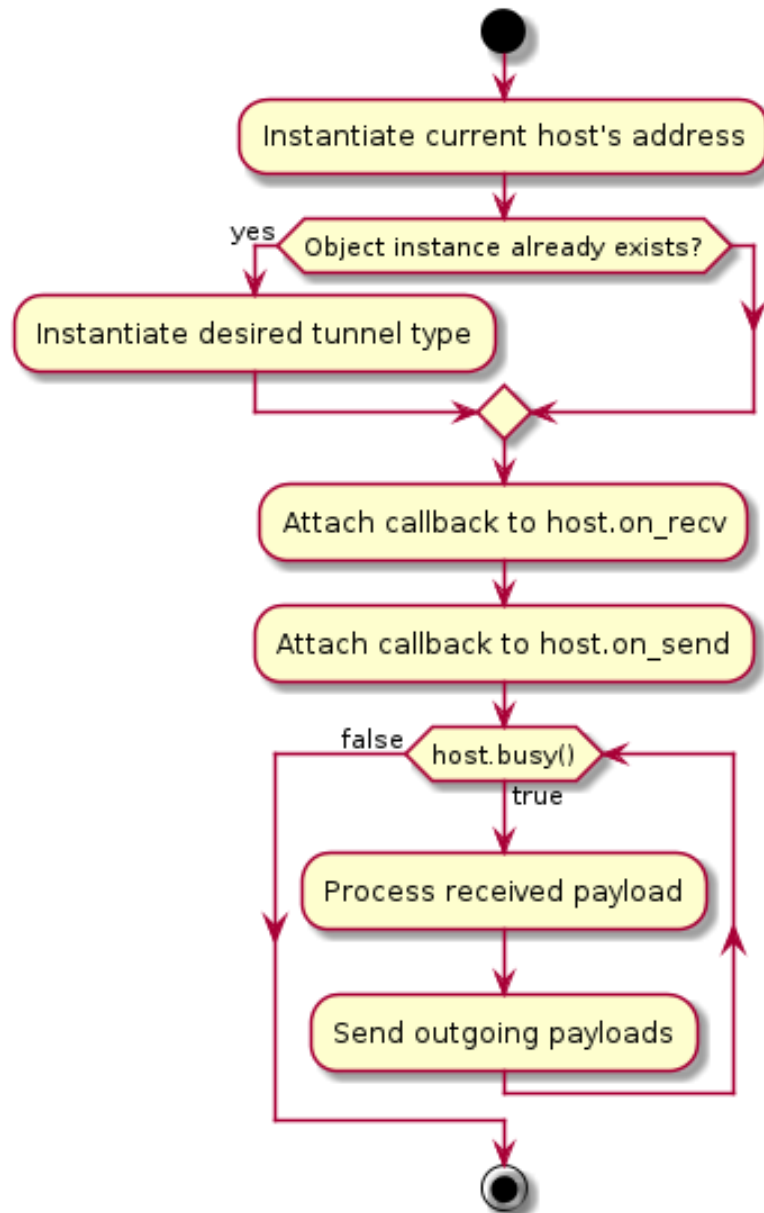
The following diagram shows the required steps to create and bind a dispatch table.
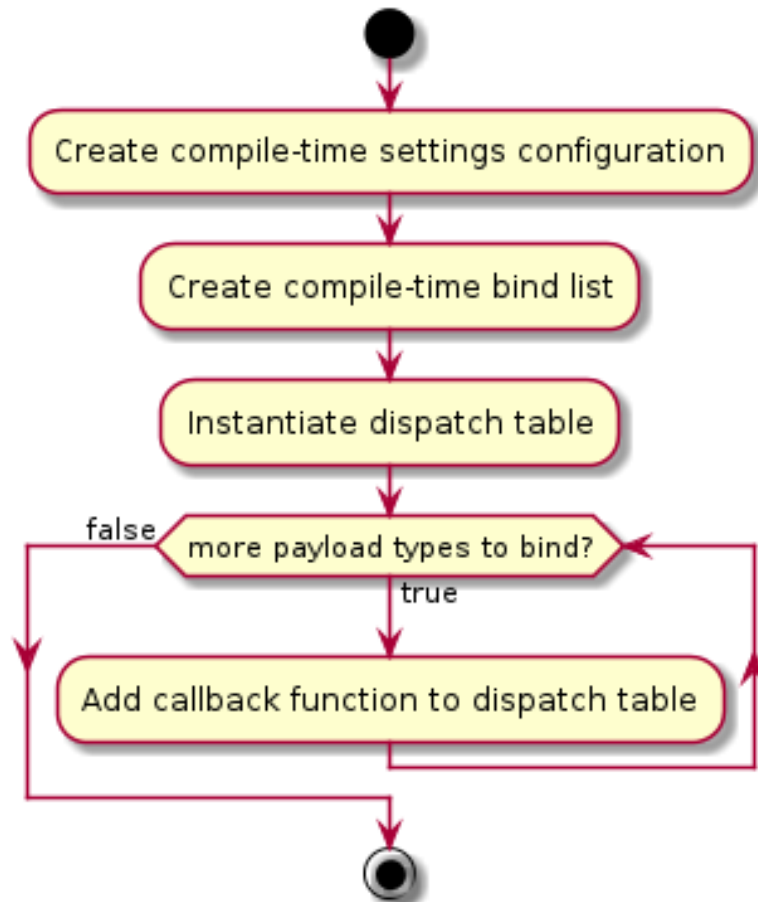
The following diagram shows the required steps to create a context host.
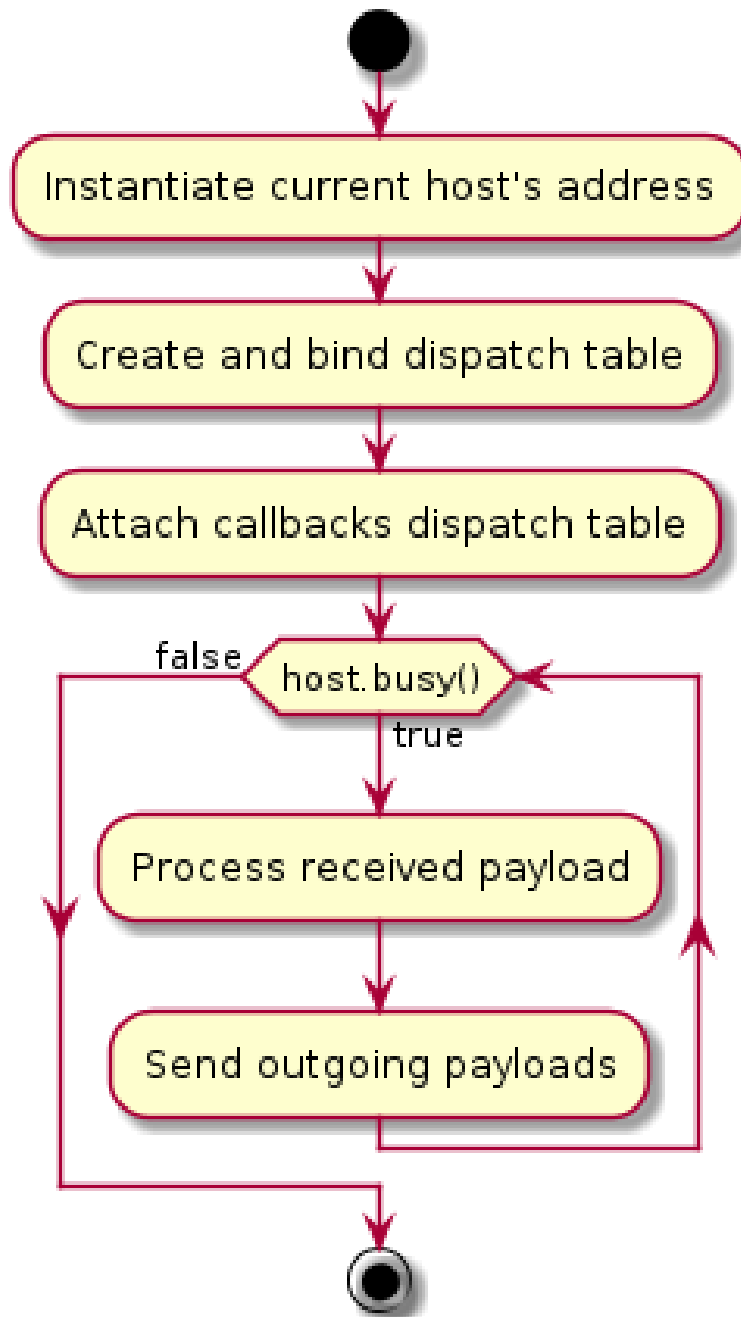
The following diagram shows the required steps to receive a bound payload.
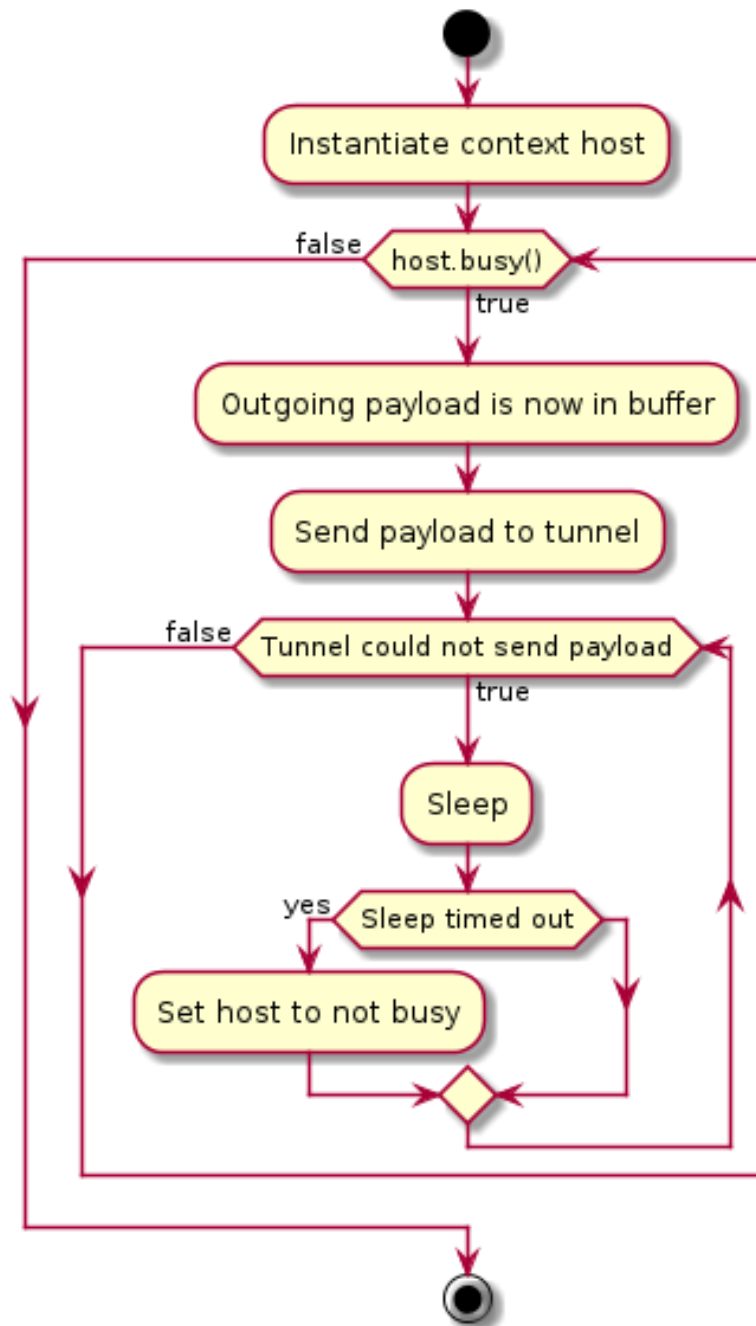
The following diagram shows the required steps to send a bound payload.
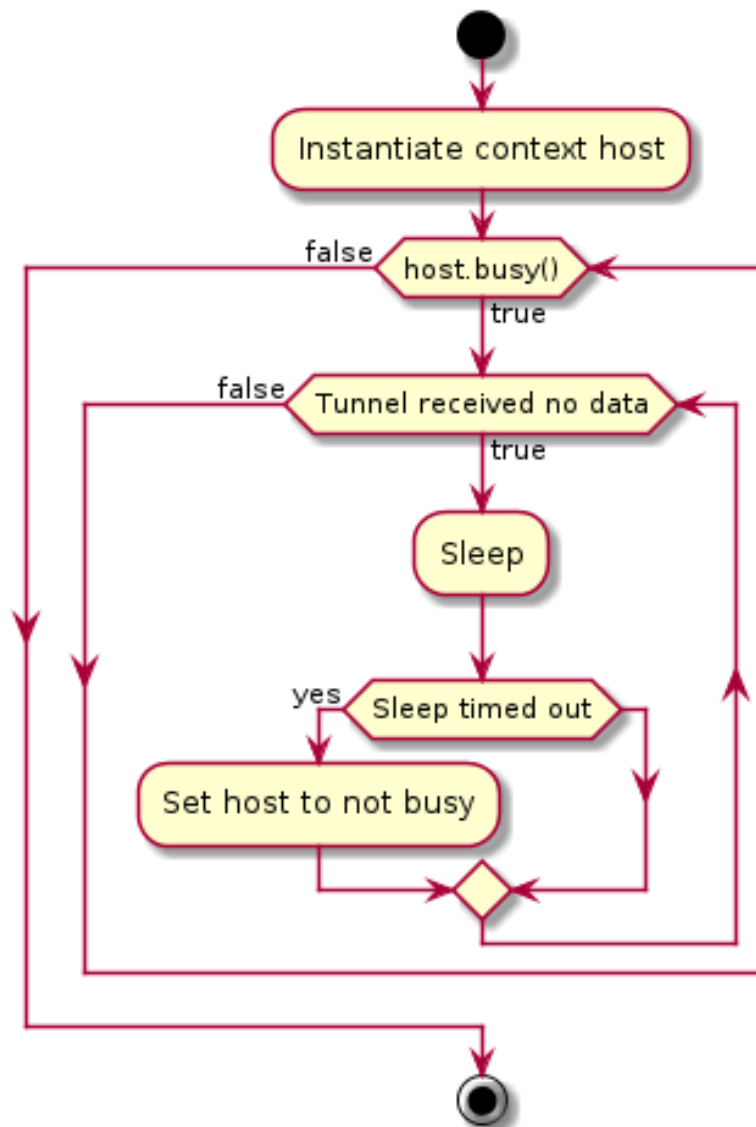
5..2 Class diagrams

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.
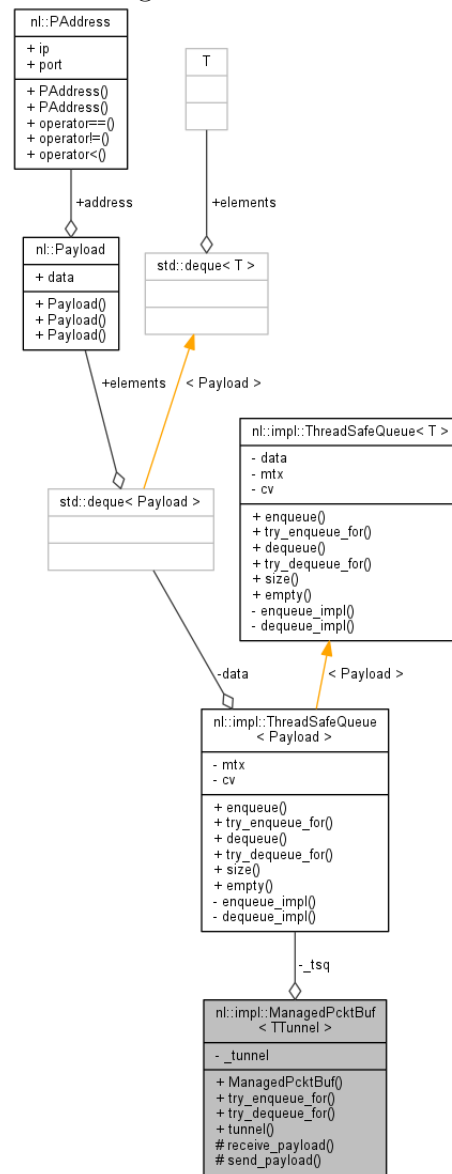
Figure 2.13: Managed host collaboration diagram.
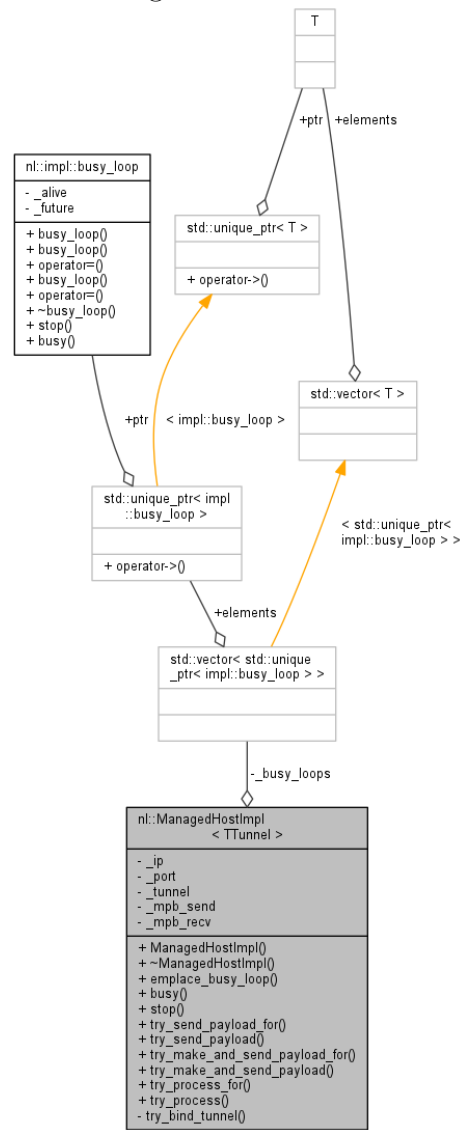
Figure 2.14: Context host collaboration diagram.
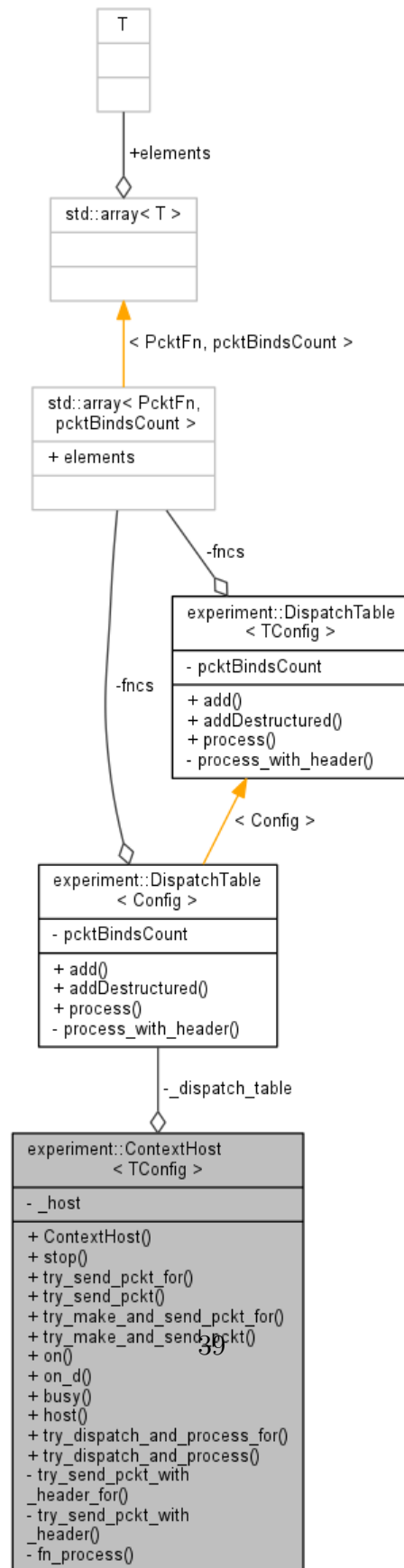
Figure 2.15: Dispatch table collaboration diagram.
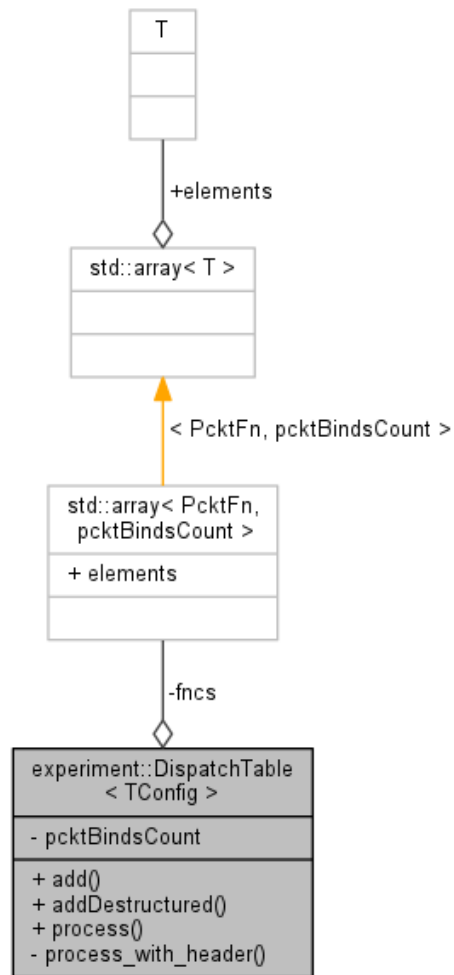
Part II

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 shown for all modules of the library.

The library is divided in multiple modules:

- **Common**: contains type aliases and dependencies shared by all modules. Also manages API macros and debugging functions.

- **Serialization**: provides a generic interface to serialize and deserialize C++ objects. Defines serializaion/deserialization functions for commonly used standard types.

- **Payload**: contains the payload abstraction, composed by an address and generic serializable data.

- **Tunnel**: contains all the default tunnel abstractions. They are: UDP socket, TCP socket, TCP listener, mock tunnel.

- **Architecture**: provides types required to build a network application architecture. Contains managed packet buffers, hosts and auxiliary thread-safe data structures.

- **Pckt**: provides utilities to automatically define serialization/deserialization functions for user-defined types. Also contains convenient functions to initialize, send and receive those structures.

- **Bound**: metaprogramming module containing classes and functions to allow compile-time binding and definitions of context hosts.

Chapter 3

Development process

1.   Environment and tools

All modules of NetLayer 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.

Qt Creator was used as the main IDE for the project - it seamlessly integrates **CMake**, the de-facto standard build system for modern C++ projects, speeding up development time significantly.

To write shell scripts, 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 used for the example projects bundled with NetLayer that require a web server or a database.1

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. Test-driven development

The **test-driven development** (TDD) paradigm was used to develop NetLayer and example applications.

TDD is a software development process that relies on the repetition of a very short development cycle: first the developer writes an automated test case that defines a

desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

5.   Technologies used

C++14, which is the latest official C++ standard, released in 2 March 2014 **(paper N3936)**, is the language of choice for NetLayer.

Like C++11, this newer standard is a huge step forward for the language. **Smarter memory managament**, **automatic type deduction**, and countless new programming and metaprogramming features allow developers to write much safer and powerful code. C++11 and C++14 features are what make NetLayer possible and its syntax cleaner and easier to use.

On top of C++14 and its standard library, the **SSV framework** is being used as well. This framework was written completely from scratch by me, **Vittorio Romeo**, and is available under the open-source AFL3.0 license on GitHub.

The **SSVUtils** library, used throughout the whole program, features many heterogeneous self-contained modules: an efficient and modern handwritten JSON parser, a preprocessor metaprogramming module, a template metaprogramming module, automatic console formatted output for containers and user-defined types, efficient data structures (bimap, handle vector, growable arrays), advanced memory management facilities, type-safe variadic unions, handwritten templating system, filesystem management, easy benchmarking of portions of code, and much more.

An additional dependencies for NetLayer is the **SFML** library, which offers lightweight abstraction over sockets and packets.

6.   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.

6..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.

```bash
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/latexpp`

6..2 PlantUML

PlantUML is a software that allows easy creation of UML diagrams from a simple language. It was used to create activity diagrams and use case diagrams.

6..3 Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources. It automatically generates well-formatted and complete documentation pages for classes, namespaces and functions. Thanks to **Graphviz** integration, it also automatically generates class and collaboration diagrams.

Chapter 4

Library structure

Figure 4.1: NetLayer modules.
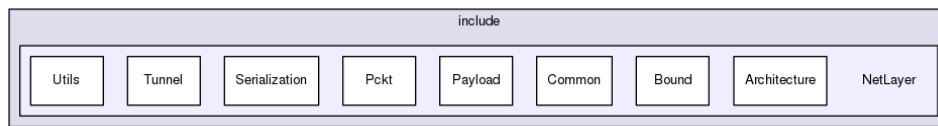
Figure 4.2: NetLayer module dependencies.
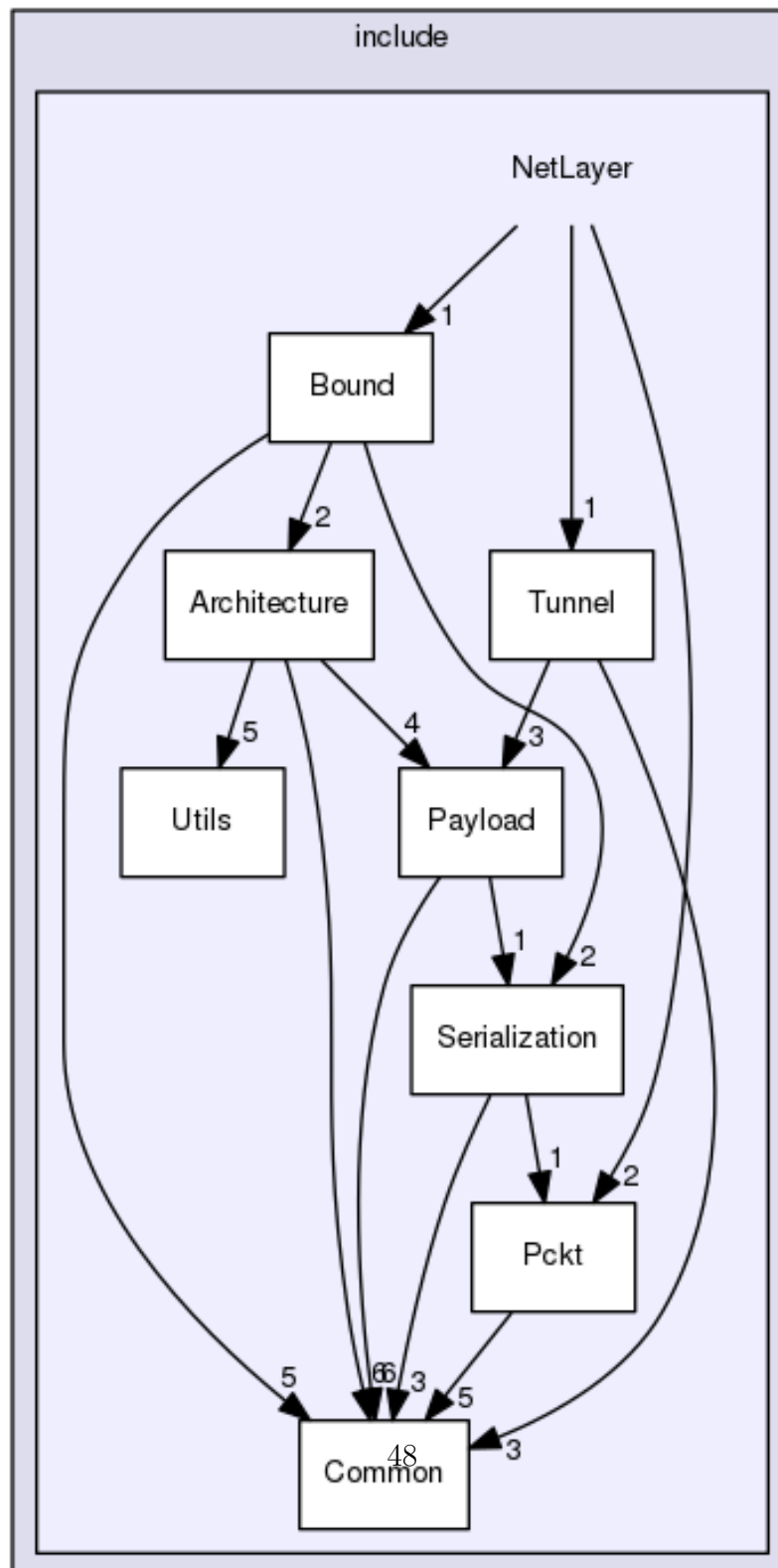
Figure 4.3: Serialization dependency graph.
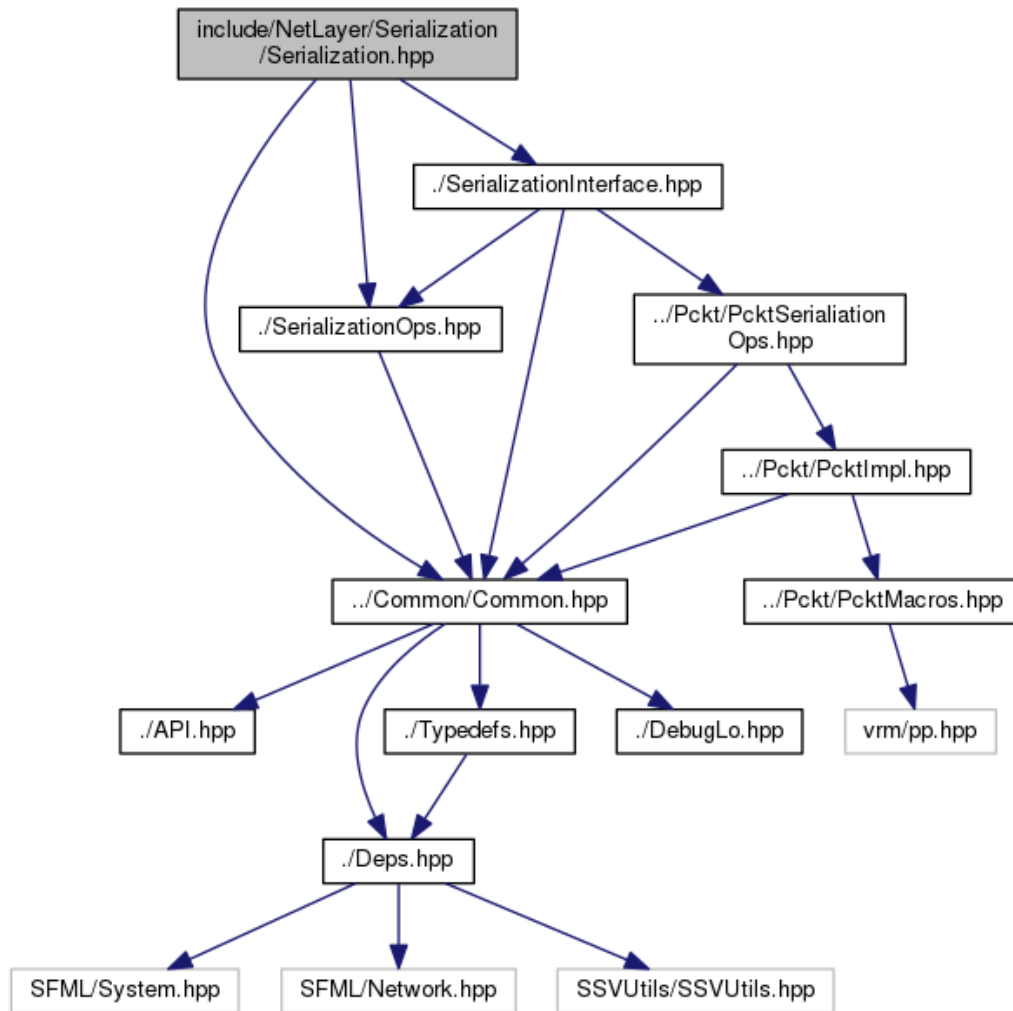
Figure 4.4: Tunnel dependency graph.
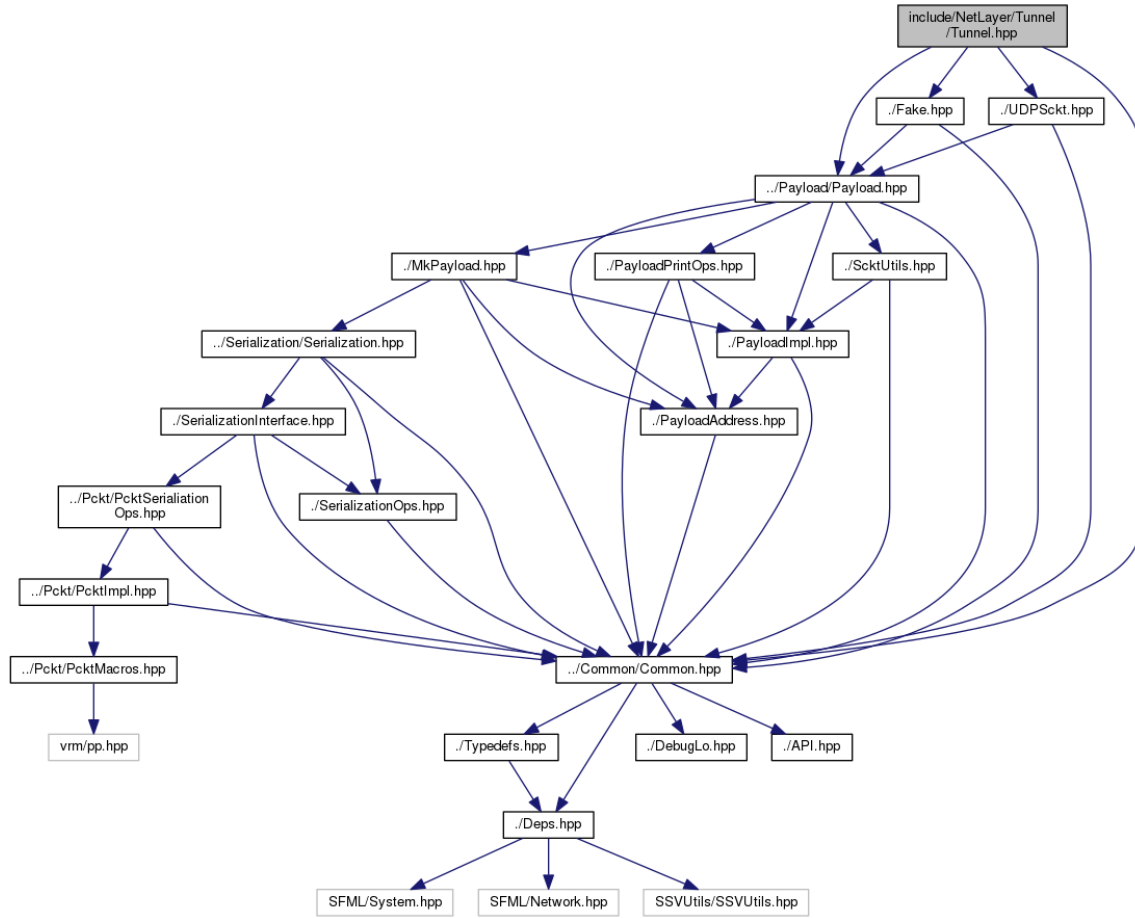
Figure 4.5: Payload dependency graph.
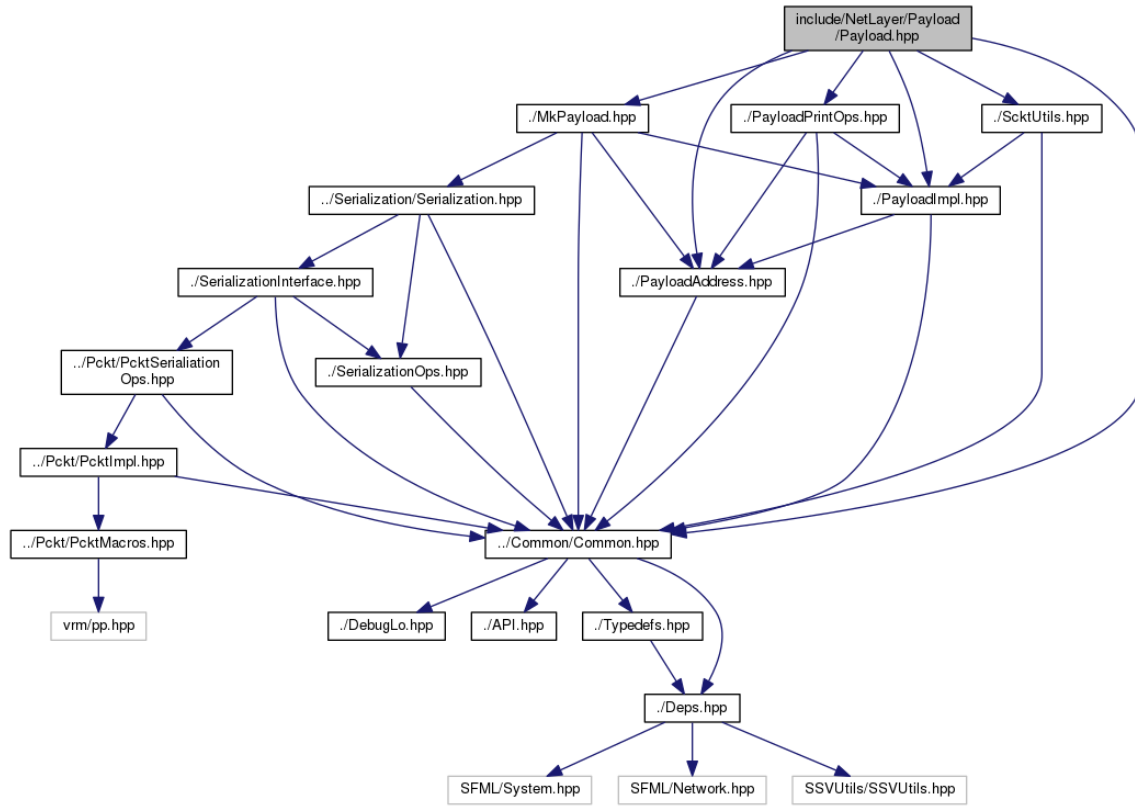
Figure 4.6: Packet types dependency graph.
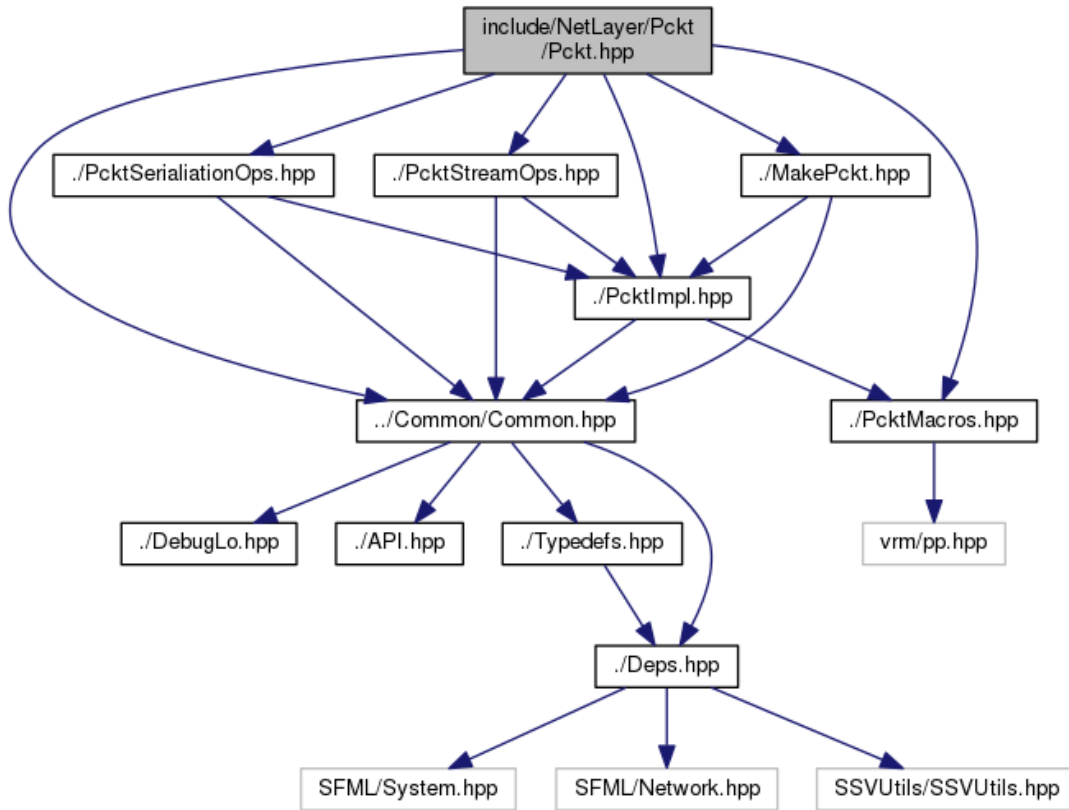
Figure 4.7: Architecture dependency graph.
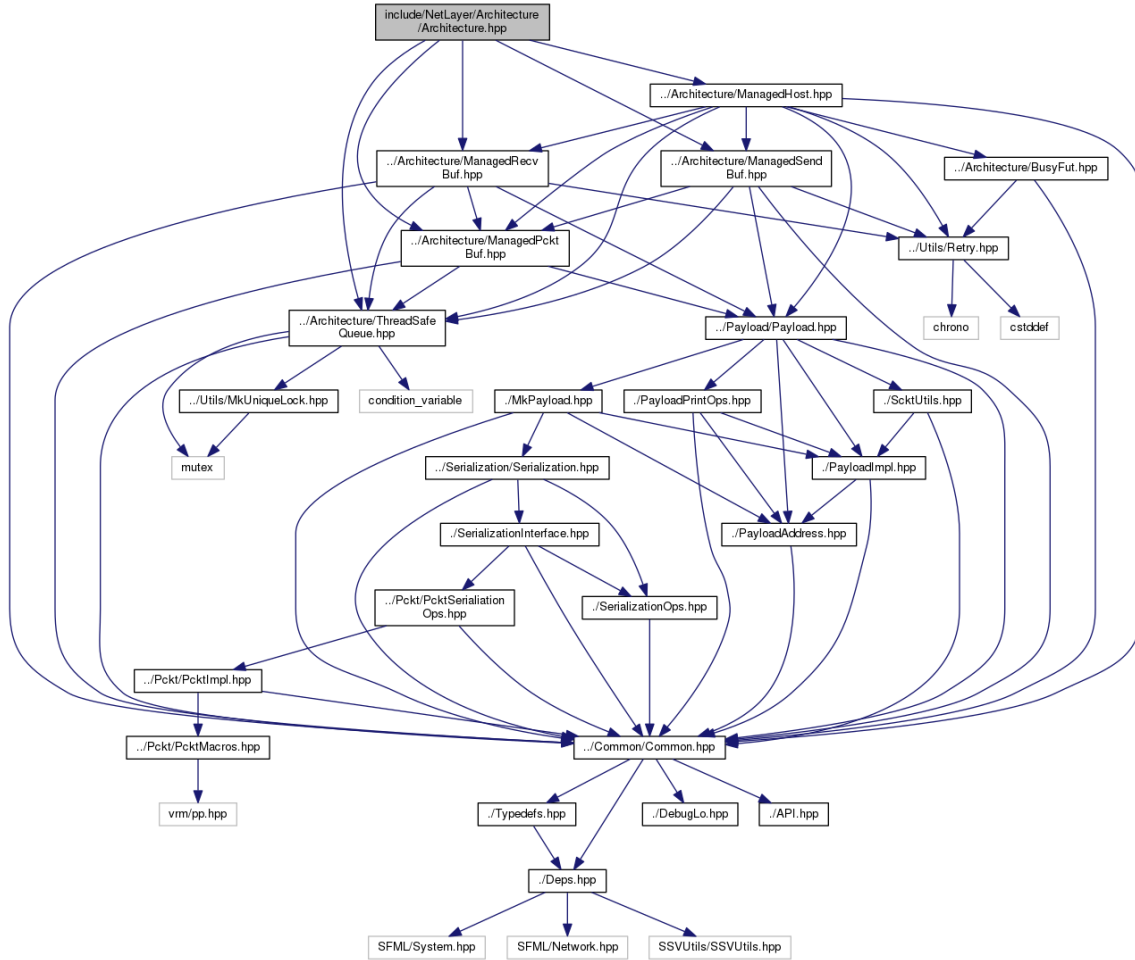
Chapter 5

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:

nl::impl::busy_loop
+ busy_loop() + busy_loop() + operator=() + busy_loop() + operator=() + ~busy_loop() + stop() + busy()
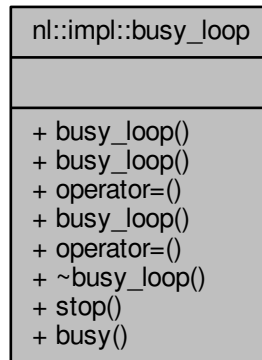
Public Member Functions

- 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 >:

```
┌─────────────────────────────────────┐
│ experiment::ContextHost             │
│         < TConfig >                 │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ + ContextHost()                     │
│ + stop()                            │
│ + try_send_pckt_for()               │
│ + try_send_pckt()                   │
│ + try_make_and_send_pckt_for()      │
│ + try_make_and_send_pckt()          │
│ + on()                              │
│ + on_d()                            │
│ + busy()                            │
│ + host()                            │
│ + try_dispatch_and_process_for()    │
│ + try_dispatch_and_process()        │
└─────────────────────────────────────┘
```
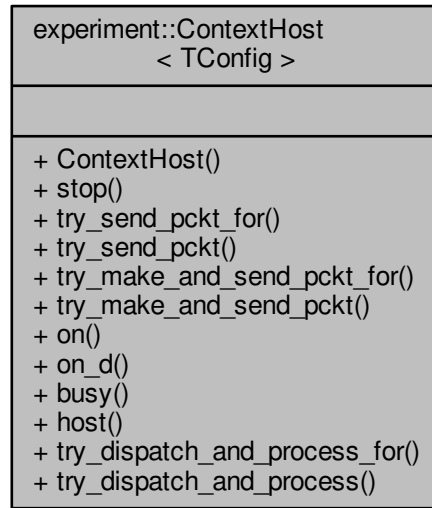
Public Member Functions

- **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 >:
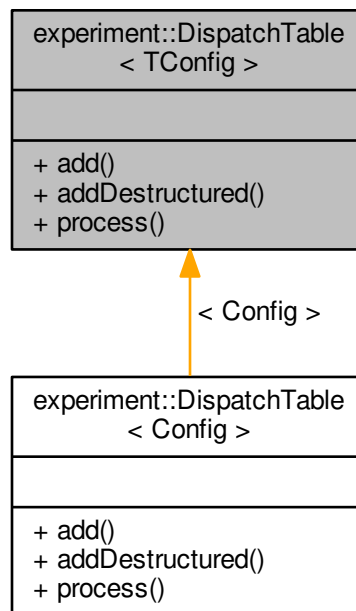
Collaboration diagram for experiment::DispatchTable< TConfig >:

```
┌─────────────────────────────┐
│ experiment::DispatchTable   │
│       < TConfig >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + add()                     │
│ + addDestructured()         │
│ + process()                 │
└─────────────────────────────┘
```
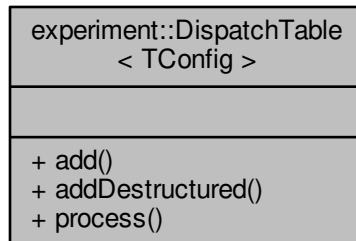
Public Member Functions

- 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 >:

```
┌─────────────────────────────────────┐
│ nl::ManagedHostImpl                  │
│         < TTunnel >                  │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ + ManagedHostImpl()                 │
│ + ~ManagedHostImpl()                │
│ + emplace_busy_loop()               │
│ + busy()                            │
│ + stop()                            │
│ + try_send_payload_for()            │
│ + try_send_payload()                │
│ + try_make_and_send_payload_for()   │
│ + try_make_and_send_payload()       │
│ + try_process_for()                 │
│ + try_process()                     │
└─────────────────────────────────────┘
```
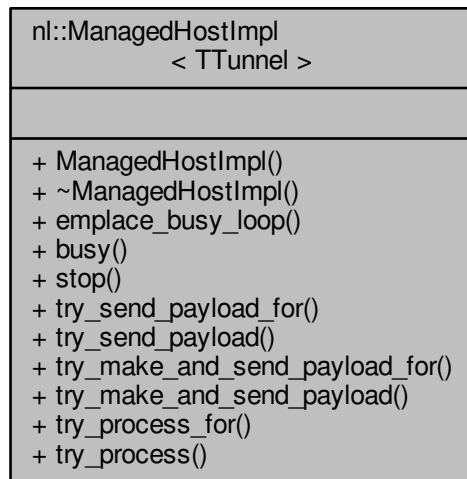
Public Member Functions

- 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 >:

```
┌─────────────────────────────┐
│  nl::impl::ManagedPcktBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
├─────────────────────────────┤
│ + ManagedPcktBuf()          │
│ + try_enqueue_for()         │
│ + try_dequeue_for()         │
│ + tunnel()                  │
│ # receive_payload()         │
│ # send_payload()            │
└─────────────────────────────┘
```

```
┌─────────────────────────────┐      ┌─────────────────────────────┐
│ nl::impl::ManagedRecvBuf     │      │ nl::impl::ManagedSendBuf     │
│        < TTunnel >           │      │        < TTunnel >           │
├─────────────────────────────┤      ├─────────────────────────────┤
├─────────────────────────────┤      ├─────────────────────────────┤
│ + ~ManagedRecvBuf()         │      │ + ~ManagedSendBuf()         │
│ + recv_step()               │      │ + send_step()               │
└─────────────────────────────┘      └─────────────────────────────┘
```

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

```
┌─────────────────────────────┐
│ nl::impl::ManagedPcktBuf     │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ManagedPcktBuf()          │
│ + try_enqueue_for()         │
│ + try_dequeue_for()         │
│ + tunnel()                  │
│ # receive_payload()         │
│ # send_payload()            │
└─────────────────────────────┘
```
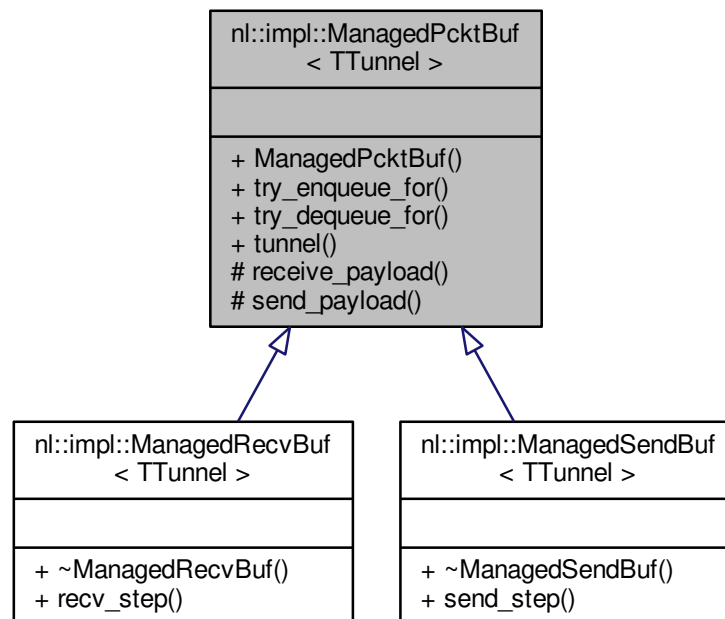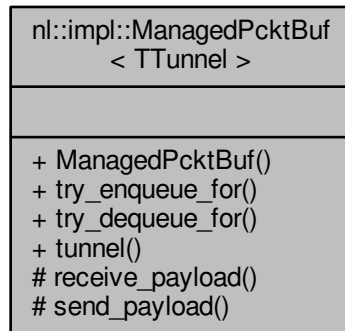
Public Member Functions

- **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 >:

```
┌─────────────────────────────────┐
│   nl::impl::ManagedPcktBuf       │
│        < TTunnel >               │
├─────────────────────────────────┤
│                                 │
├─────────────────────────────────┤
│ + ManagedPcktBuf()              │
│ + try_enqueue_for()             │
│ + try_dequeue_for()             │
│ + tunnel()                      │
│ # receive_payload()             │
│ # send_payload()                │
└─────────────────────────────────┘
                 △
                 │
┌─────────────────────────────────┐
│   nl::impl::ManagedRecvBuf       │
│        < TTunnel >               │
├─────────────────────────────────┤
│                                 │
├─────────────────────────────────┤
│ + ~ManagedRecvBuf()             │
│ + recv_step()                   │
└─────────────────────────────────┘
```

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

```
┌─────────────────────────────┐
│  nl::impl::ManagedPcktBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ManagedPcktBuf()          │
│ + try_enqueue_for()         │
│ + try_dequeue_for()         │
│ + tunnel()                  │
│ # receive_payload()         │
│ # send_payload()            │
└─────────────────────────────┘
              △
              │
┌─────────────────────────────┐
│  nl::impl::ManagedRecvBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ~ManagedRecvBuf()         │
│ + recv_step()               │
└─────────────────────────────┘
```
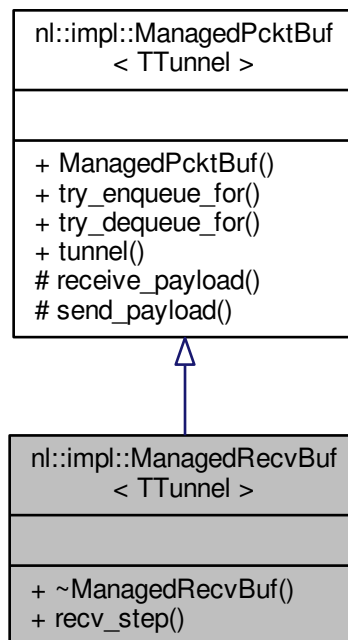
Public Member Functions

- 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 >:

```
┌─────────────────────────────┐
│  nl::impl::ManagedPcktBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ManagedPcktBuf()          │
│ + try_enqueue_for()         │
│ + try_dequeue_for()         │
│ + tunnel()                  │
│ # receive_payload()         │
│ # send_payload()            │
└─────────────────────────────┘
              △
              │
┌─────────────────────────────┐
│  nl::impl::ManagedSendBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ~ManagedSendBuf()         │
│ + send_step()               │
└─────────────────────────────┘
```

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

```
┌─────────────────────────────┐
│  nl::impl::ManagedPcktBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ManagedPcktBuf()          │
│ + try_enqueue_for()         │
│ + try_dequeue_for()         │
│ + tunnel()                  │
│ # receive_payload()         │
│ # send_payload()            │
└─────────────────────────────┘
                △
                │
┌─────────────────────────────┐
│  nl::impl::ManagedSendBuf    │
│        < TTunnel >           │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ + ~ManagedSendBuf()         │
│ + send_step()               │
└─────────────────────────────┘
```

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
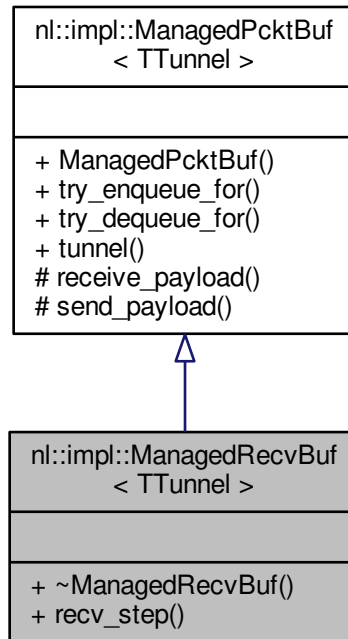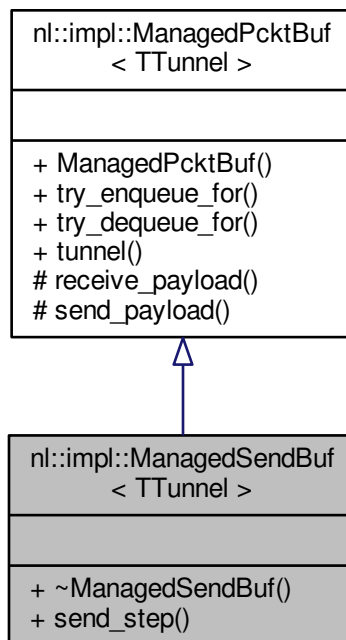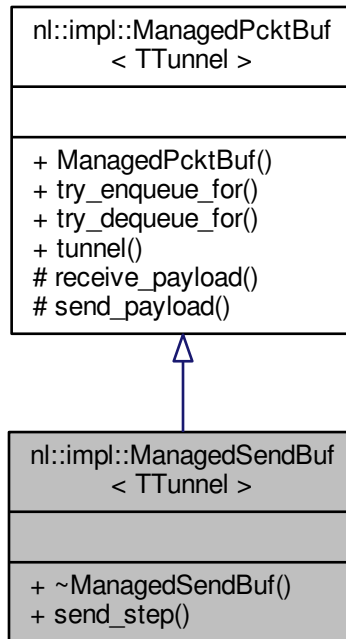
Collaboration diagram for nl::PAddress:

```
┌─────────────────────┐
│    nl::PAddress      │
├─────────────────────┤
│ + ip                │
│ + port              │
├─────────────────────┤
│ + PAddress()        │
│ + PAddress()        │
│ + operator==()      │
│ + operator!=()      │
│ + operator<()       │
└─────────────────────┘
```
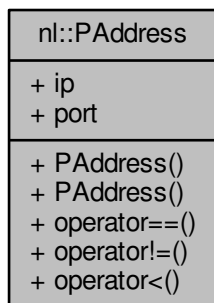
Public Member Functions

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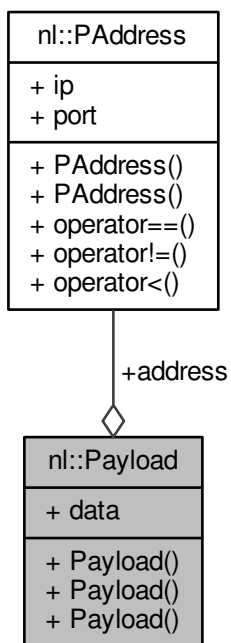
The documentation for this struct was generated from the following file:

- include/NetLayer/Payload/PayloadAddress.hpp

9. nl::Payload Struct Reference

Collaboration diagram for nl::Payload:

```
                    ┌─────────────────────┐
                    │    nl::PAddress     │
                    ├─────────────────────┤
                    │ + ip                │
                    │ + port              │
                    ├─────────────────────┤
                    │ + PAddress()        │
                    │ + PAddress()        │
                    │ + operator==()      │
                    │ + operator!=()      │
                    │ + operator<()       │
                    └─────────────────────┘
                              │
                              │ +address
                              ◇
                    ┌─────────────────────┐
                    │    nl::Payload      │
                    ├─────────────────────┤
                    │ + data              │
                    ├─────────────────────┤
                    │ + Payload()         │
                    │ + Payload()         │
                    │ + Payload()         │
                    └─────────────────────┘
```

Public Member Functions

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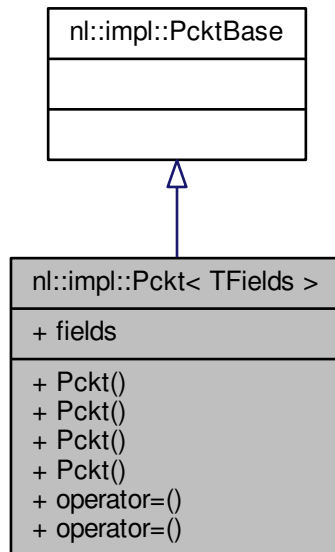
Public Attributes

- PAddress **address**

- PcktBuf **data**

The documentation for this struct was generated from the following file:
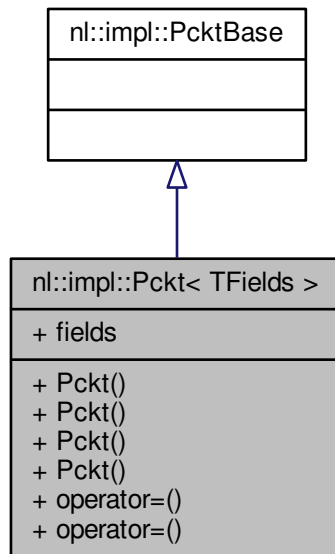
- include/NetLayer/Payload/PayloadImpl.hpp

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)
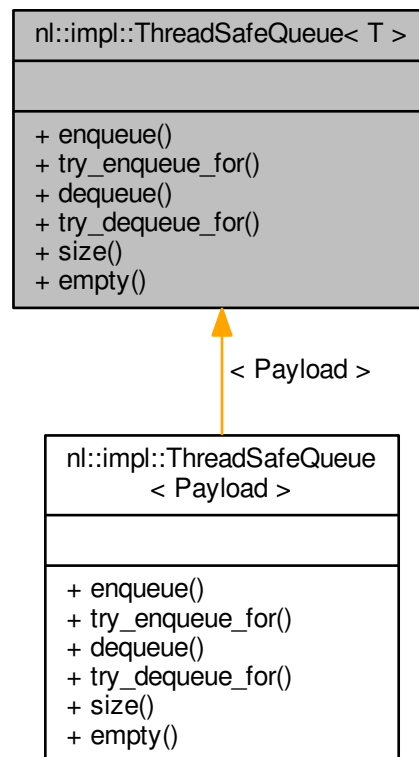
Public Attributes

- TplType **fields**

The documentation for this struct was generated from the following file:
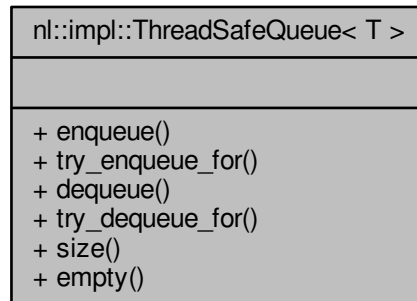
- include/NetLayer/Pckt/PcktImpl.hpp

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

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

```
┌─────────────────────────────────────┐
│  nl::impl::ThreadSafeQueue< T >      │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│  + enqueue()                        │
│  + try_enqueue_for()                │
│  + dequeue()                        │
│  + try_dequeue_for()                │
│  + size()                           │
│  + empty()                          │
└─────────────────────────────────────┘
                  ▲
                  │  < Payload >
                  │
┌─────────────────────────────────────┐
│  nl::impl::ThreadSafeQueue          │
│         < Payload >                 │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│  + enqueue()                        │
│  + try_enqueue_for()                │
│  + dequeue()                        │
│  + try_dequeue_for()                │
│  + size()                           │
│  + empty()                          │
└─────────────────────────────────────┘
```

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

```
┌─────────────────────────────────┐
│ nl::impl::ThreadSafeQueue< T >   │
├─────────────────────────────────┤
│                                 │
├─────────────────────────────────┤
│ + enqueue()                     │
│ + try_enqueue_for()             │
│ + dequeue()                     │
│ + try_dequeue_for()             │
│ + size()                        │
│ + empty()                       │
└─────────────────────────────────┘
```

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:

```
┌─────────────────────────────────┐
│  nl::impl::Tunnel::UDPSckt       │
├─────────────────────────────────┤
│                                 │
├─────────────────────────────────┤
│ + UDPSckt()                     │
│ + operator=()                   │
│ + UDPSckt()                     │
│ + operator=()                   │
│ + UDPSckt()                     │
│ + ~UDPSckt()                    │
│ + bind()                        │
│ + receive_payload()             │
│ + send_payload()                │
└─────────────────────────────────┘
```
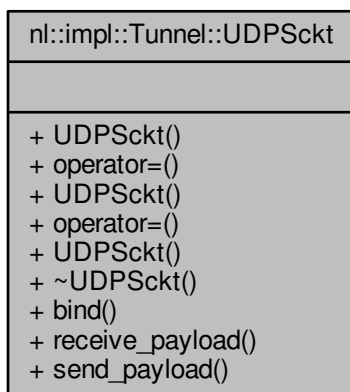
Public Member Functions

- **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

Part III

Conclusion

Chapter 6

Learning experience

Creating the **veeForum** framework was an incredible learning experience. Many new technologies had to be used and integrated in the project to achieve a satisfactory final result.

- **Planning and design**: using Software Engineering guidelines and design processes was extremely educational and contributed to the quality of the project.

- **GNU/Linux containers**: using Docker containers for the database and the web application was beneficial in understanding the advantages of containers and allowed a decoupled and highly portable final result.

- **LaTeX and LatexPP**: using LaTeX and writing a simple preprocessor for it was a very education thesis-writing and typesetting experience and definitely resulted in a high-quality document.

- **Complex database design**: having to deal with 20+ tables, 30+ triggers and 15+ stored procedures for the correct functioning of the system was a very educational experience in complex SQL coding and database design practices. Making everything work toghether, safely and efficiently, required learning complex SQL constructs such as **cursors** and **imperative loops** and spending a lot of time on table/relationship design beforehand.

- **Complex PHP framework**: having two feature-rich PHP modules that needed to conveniently expose functionality to communicate with the database

to the developer was a great learning experience in **object-oriented development** and **library design**. Generating HTML using polymorphic PHP class hierarchies was also extremely beneficial in the understanding of dynamic HTML page generation processes and tree-like syntactic structures.

Chapter 7

Future

veeForum can be greatly improved, and will probably be expanded upon and used as the forum framework for some future projects.

Here are some possible improvements:

- **Theming and customization**: add theming capabilities and user-friendly graphical customization options to the default web application.

- **Private messaging**: add synchronous and asynchronous private messaging capabilities between users (and/or groups), along with new permissions and privileges.

- **RSS feed**: add the possibility to subscribe to sections/threads/users and receive an RSS feed.

- **Public JSON API**: add a public RESTful JSON API that can be used to retrieve the forum contents and display them in custom clients.

- **Personal user profiles**: add personal user profile customization options, such as avatars, biographies, interests, contact information, etc...

- **Email notifications**: allow users to receive notification alerts as emails.

- **Dashboard**: add a dashboard displaying the most popular threads and users in the main page.

- **Real-time statistics**: show the real-time number of users viewing a particular section or thread directly in the web interface.

- **Mailing list support**: add a mailing bot that can be used to interact with the forum contents directly from the user's favorite email client.

Part IV

Example application: NLBroadcast

Chapter 8

Project specifications

1.   Client request

The client requests a command-line persistent chat application, allowing users to register, login, create channels and broadcast messages. The application should be able to run both in server mode and client mode.

The messages must be saved and users must be able to retrieve them by specifying a date range.

Users registered to the system can subscribe to any number of channels. When users are logged in, and a message is broadcasted to one of the channels they're subscribed to, they must receive a notification with the broadcast content.

2.   Scope

2..1   Identity

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

2..2   Feature extents

The complete product will:

- Allow users to deploy and run a server with the following capabilities:

- Manage user credentials.
- Listen to incoming broadcasts and send notifications to users.

- Allow users to connect to an existing server to perform the following actions:

  - Registration to the server.
  - Login and logout.
  - Create and delete channels.
  - Subscribe and unsubscribe from channels.
  - Send broadcasts to channels.
  - Retrieve broadcast from channels in a specific date range.

3.   Glossary

- **User**: registered user to a NLBroadcast server. Has an associated username and password.

- **Channel**: possible target for broadcast messages. Can be created and removed by users. Users can subscribe and unsubscribe to channels.

- **Broadcast**: text-based message that can target a specific channel. To send a broadcast to a specific channel, users must be subscribed to it.

Chapter 9

Technical analysis

1. Dependencies

The completed product will depend on **NetLayer** and **sqlpp11**. The sqlpp11 library is an open-source project that allows developers to use a type-safe embedded C++ domain specific language for SQL queries.

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.

To connect **sqlpp11** to **MariaDB**, the additional small **sqlpp11-connector-mysql** library will be used.

2.   Namespace Documentation

3.   db_actions Namespace Reference

Namespaces

- impl

Functions

- template<typename TF >
  void for_channels (TF &&f)
- template<typename TF >
  bool message_by_id (int id, TF &&f)
- template<typename TF >
  bool user_by_id (int id, TF &&f)
- template<typename TF >
  bool user_by_username (const std::string &username, TF &&f)
- bool has_user_by_id (int id)
- bool has_user_by_username (const std::string &username)
- template<typename TF >
  bool channel_by_id (int id, TF &&f)
- template<typename TF >
  bool channel_by_name (const std::string &name, TF &&f)
- bool has_channel_by_id (int id)
- bool has_channel_by_name (const std::string &name)
- auto create_user (const std::string &user, const std::string &pass)
- auto create_channel (int user_id, const std::string &name)
- auto delete_channel (int id)
- auto add_user_to_channel (int user_id, int channel_id)
- auto remove_user_from_channel (int user_id, int channel_id)
- auto create_message (int user_id, int channel_id, const std::string &contents)
- auto is_user_in_channel (int user_id, int channel_id)

- template<typename TF >
  void for_users_subscribed_to (int channel_id, TF &&f)

3..1 Function Documentation

3..1.1 auto db_actions::add_user_to_channel (int *user_id,* int *channel_id*)

Here is the call graph for this function:

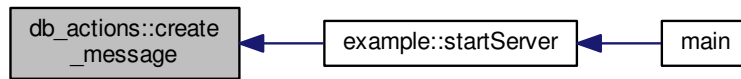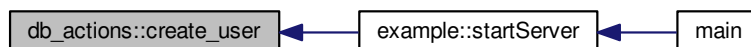Here is the caller graph for this function:

86
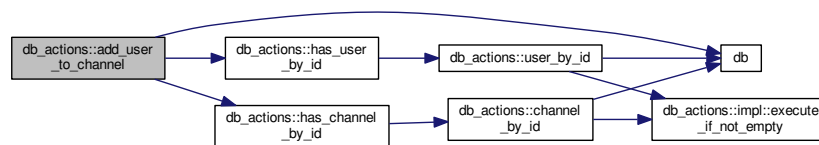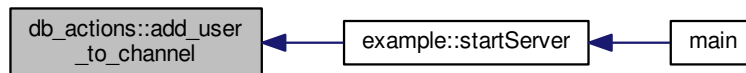
3..1.2 template<typename TF > bool db_actions::channel_by_id (int id, TF && f)

Here is the call graph for this function:
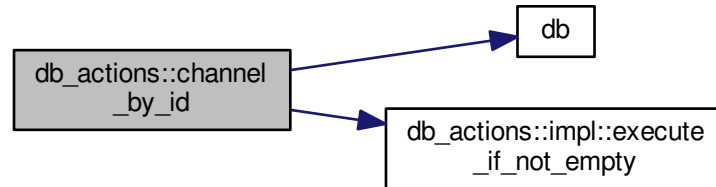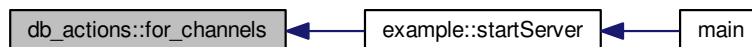

Here is the caller graph for this function:
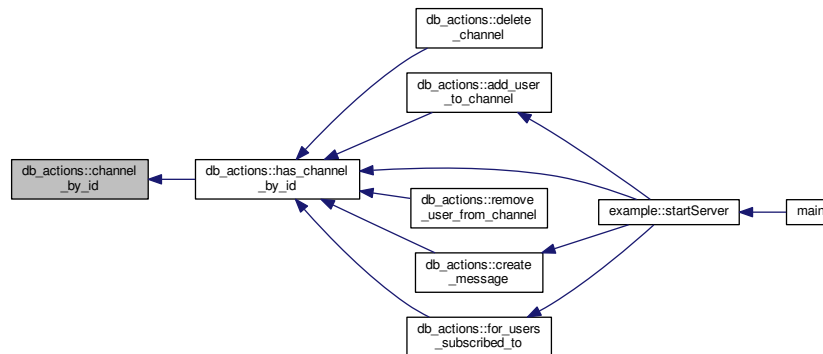
3..1.3 template<typename TF > bool db_actions::channel_by_name (const std::string & *name,* TF && *f*)

Here is the call graph for this function:

```
db_actions::channel          db
   _by_name
                    db_actions::impl::execute
                         _if_not_empty
```

Here is the caller graph for this function:

```
db_actions::channel  ◄─  db_actions::has_channel  ◄─  example::startServer  ◄─  main
   _by_name                   _by_name
```
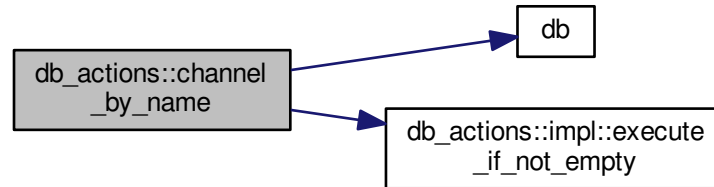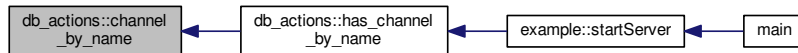
3..1.4   auto db_actions::create_channel (int *user_id,* const std::string & *name*)

Here is the call graph for this function:
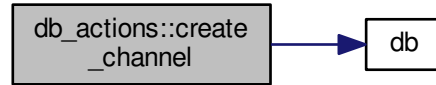
Here is the caller graph for this function:
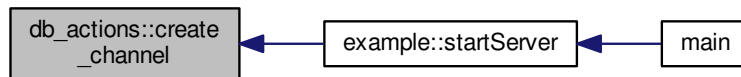
3..1.5   auto db_actions::create_message (int *user_id,* int *channel_id,* const std::string & *contents*)

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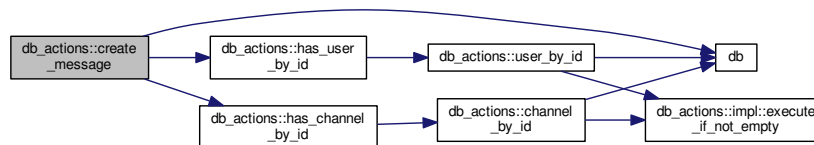
Here is the caller graph for this function:

3..1.6 auto db_actions::create_user (const std::string & *user,* const std::string & *pass*)

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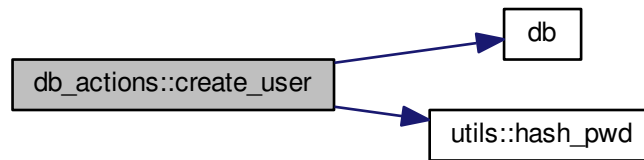
Here is the caller graph for this function:

3..1.7 auto db_actions::delete_channel (int *id*)

Here is the call graph for this function:
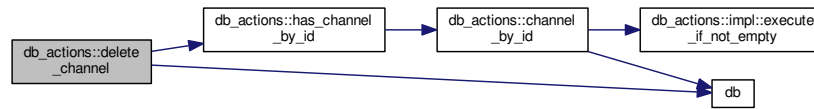
3..1.8 template<typename TF > void db_actions::for_channels (TF && *f*)

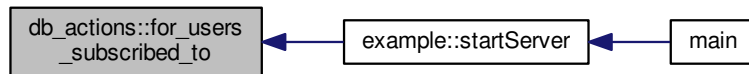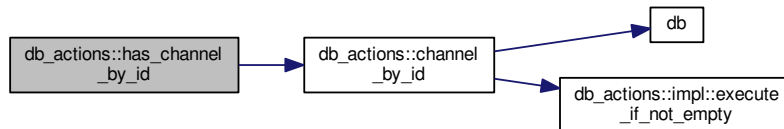Here is the call graph for this function:
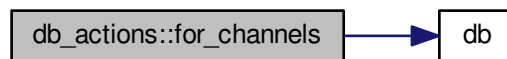
Here is the caller graph for this function:

3..1.9 template<typename TF > void db_actions::for_↩ users_subscribed_to (int *channel_id,* TF && *f*)

Here is the call graph for this function:
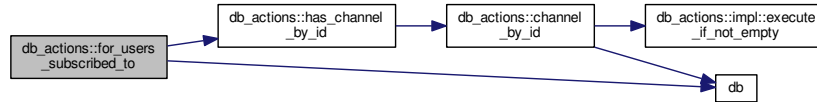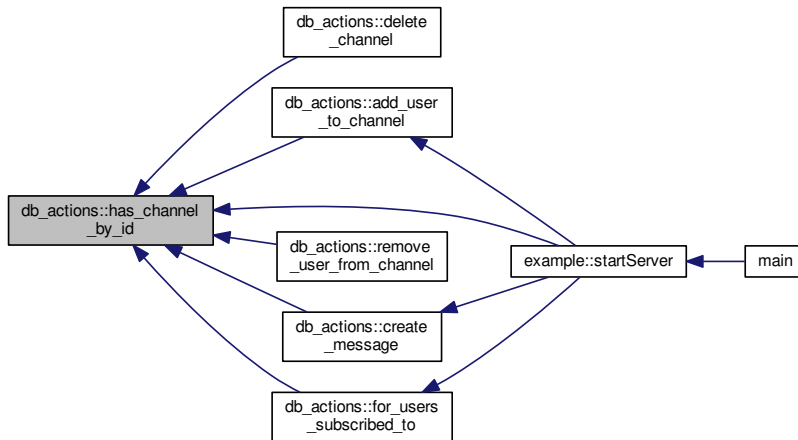

Here is the caller graph for this function:

3..1.10 bool db_actions::has_channel_by_id (int *id*)
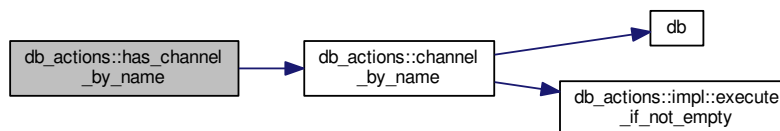
Here is the call graph for this function:
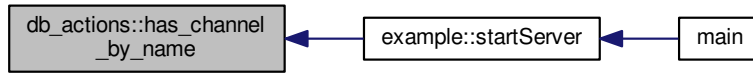
Here is the caller graph for this function:

3..1.11   bool db_actions::has_channel_by_name (const std::string & name   )
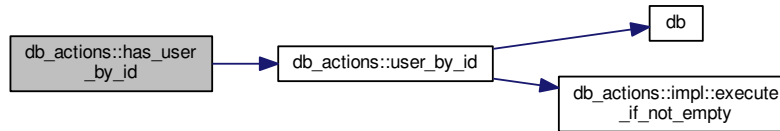
Here is the call graph for this function:
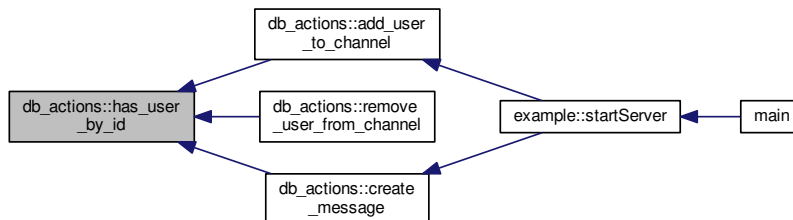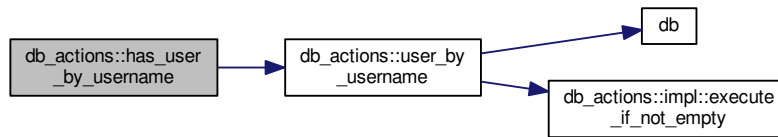
Here is the caller graph for this function:

```
db_actions::has_channel
    _by_name          ◄──────  example::startServer  ◄──────  main
```

3..1.12 bool db_actions::has_user_by_id (int *id*)

Here is the call graph for this function:

```
                                                              ┌─────┐
                                                              │ db  │
                                                              └─────┘
                                                             ▲
db_actions::has_user              db_actions::user_by_id ───┤
    _by_id          ──────►                                  │
                                                             ▼
                                                  db_actions::impl::execute
                                                       _if_not_empty
```

Here is the caller graph for this function:

```
                              db_actions::add_user
                                   _to_channel

db_actions::has_user      db_actions::remove
    _by_id          ◄──   _user_from_channel        example::startServer  ◄──  main

                              db_actions::create
                                   _message
```
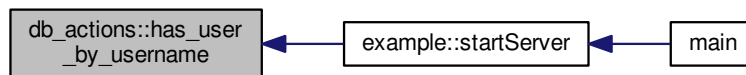
94

3..1.13 bool db_actions::has_user_by_username (const std::string & username)

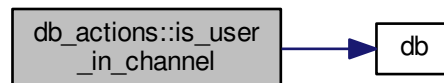Here is the call graph for this function:
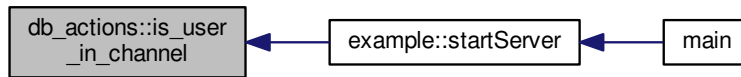

Here is the caller graph for this function:

3..1.14 auto db_actions::is_user_in_channel (int *user_id,* int *channel_id*)
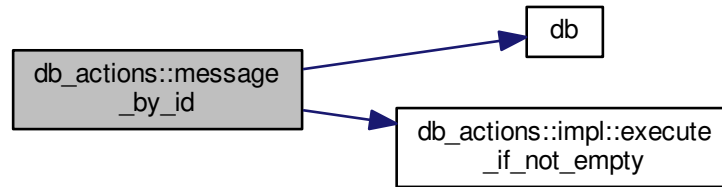
Here is the call graph for this function:

Here is the caller graph for this function:

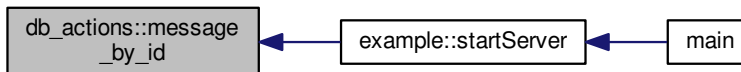### 3..1.15 template<typename TF > bool db_actions::message_by_id (int id, TF && f)
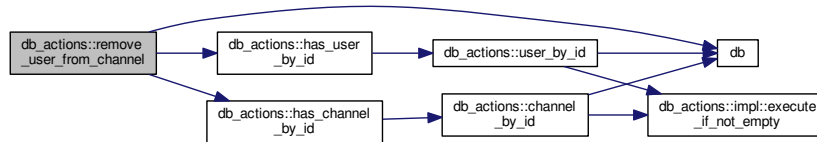
Here is the call graph for this function:
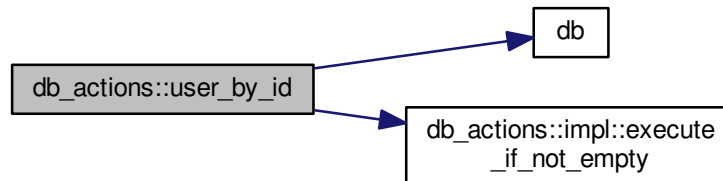

Here is the caller graph for this function:

3..1.16 auto db_actions::remove_user_from_channel (int *user_id*, int *channel_id*)

Here is the call graph for this function:
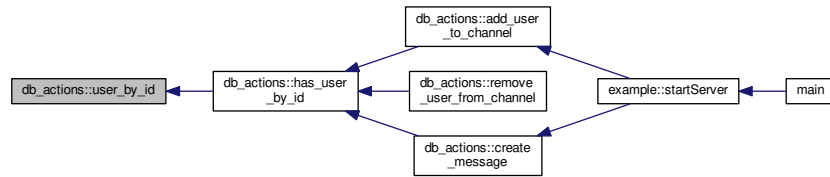

3..1.17 template<typename TF > bool db_actions::user_by_id (int *id*, TF && *f*)

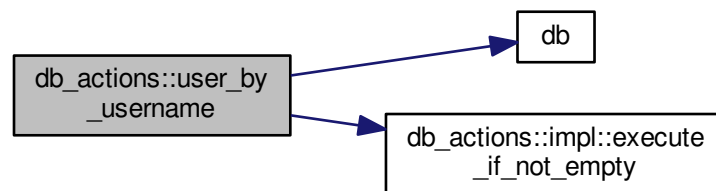Here is the call graph for this function:

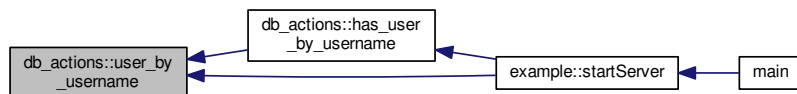Here is the caller graph for this function:

3..1.18 template<typename TF > bool db_actions::user_by_username (const std::string & *username*, TF && *f*)

Here is the call graph for this function:

Here is the caller graph for this function:

4. db_actions::impl Namespace Reference

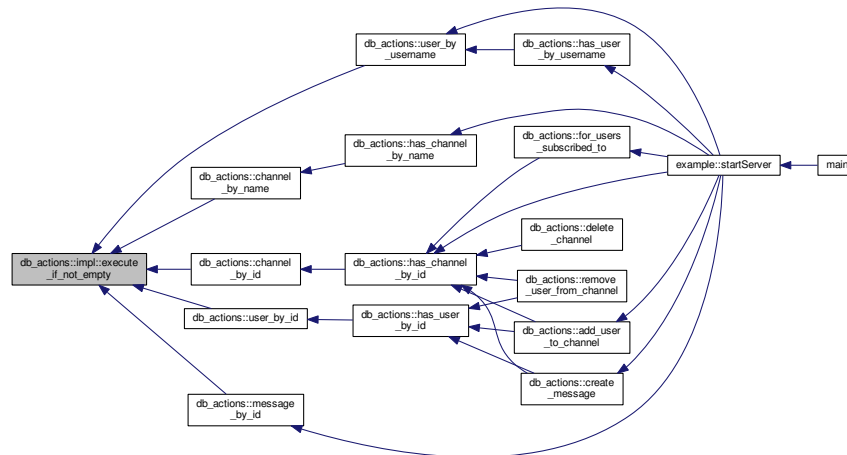## Functions

- template<typename T , typename TF >
  bool execute_if_not_empty (T &&r, TF &&f)

4..1 Function Documentation

4..1.1 template<typename T , typename TF > bool db_actions::impl::execute_if_not_empty (T && *r,* TF && *f*)

Here is the caller graph for this function:

5. example Namespace Reference

Namespaces

- to_c
- to_s

Classes

- struct client_state
- class connection_state
- class server_state

Typedefs

- using MySettings = nle::Settings< nl::UInt32 >
- using MyServerConfig = decltype(my_server_config)
- using MyClientConfig = decltype(my_client_config)
- using MyCtxServer = nle::ContextHost< MyServerConfig >
- using MyCtxClient = nle::ContextHost< MyClientConfig >

Enumerations

- enum cs {
  cs::unlogged, cs::logged, cs::awaiting_channel_list, cs::selecting_channel,
  cs::awaiting_login_response, cs::awaiting_create_channel_response, cs::awaiting↩
  _registration_response, cs::awaiting_create_message_response,
  cs::awaiting_subscribe_response }

Functions

- constexpr auto my_pckt_binds (nle::pckt_binds< to_s::Registration, to_s::Login,
  to_s::CreateChannel, to_s::DeleteChannel, to_s::SendMessage, to_s::GetMessages,
  to_s::ChannelList, to_s::Subscribe, to_s::Logout, to_c::Outcome, to_c::Messages,
  to_c::Notify, to_c::Channels, to_c::TimedOut >())
- constexpr auto my_server_tunnel (nle::tunnel_type< nl::Tunnel::TCPListener
  >{})
- constexpr auto my_client_tunnel (nle::tunnel_type< nl::Tunnel::TCPSckt >{})
- constexpr auto my_server_config (nle::make_config< MySettings >(my_pckt_↩
  binds, my_server_tunnel))
- constexpr auto my_client_config (nle::make_config< MySettings >(my_pckt_↩
  binds, my_client_tunnel))

- void startServer ()
- void startClient (nl::Port port)

5..1 Typedef Documentation

5..1.1 using example::MyClientConfig = typedef decltype(my_client_config)

5..1.2 using example::MyCtxClient = typedef nle::ContextHost<My↩ClientConfig>

5..1.3 using example::MyCtxServer = typedef nle::ContextHost<My↩ServerConfig>

5..1.4 using example::MyServerConfig = typedef decltype(my_server_config)

5..1.5 using example::MySettings = typedef nle::Settings<nl::UInt32>

5..2 Enumeration Type Documentation

5..2.1 enum example::cs [strong]

Enumerator

unlogged

logged

awaiting_channel_list

selecting_channel

awaiting_login_response

awaiting_create_channel_response

awaiting_registration_response

awaiting_create_message_response

awaiting_subscribe_response

5..3 Function Documentation

5..3.1 constexpr auto example::my_client_config (nle::make_config< MySettings > *my_pckt_binds, my_client_tunnel*)

5..3.2 constexpr auto example::my_client_tunnel (nle::tunnel_type< nl::Tunnel::TCPSckt >{})

5..3.3 constexpr auto example::my_pckt_binds (nle::pckt_binds< to_s::Registration, to_s::Login, to_s::CreateChannel, to_s::DeleteChannel, to_s::SendMessage, to_s::GetMessages, to_s::ChannelList, to_s::Subscribe, to_s::Logout, to_c::Outcome, to_c::Messages, to_c::Notify, to_c::Channels, to_c::TimedOut > *()*)

5..3.4 constexpr auto example::my_server_config (nle::make_config< MySettings > *my_pckt_binds, my_server_tunnel*)

5..3.5 constexpr auto example::my_server_tunnel (nle::tunnel_type< nl::Tunnel::TCPListener >{})

5..3.6 void example::startClient (nl::Port *port*)
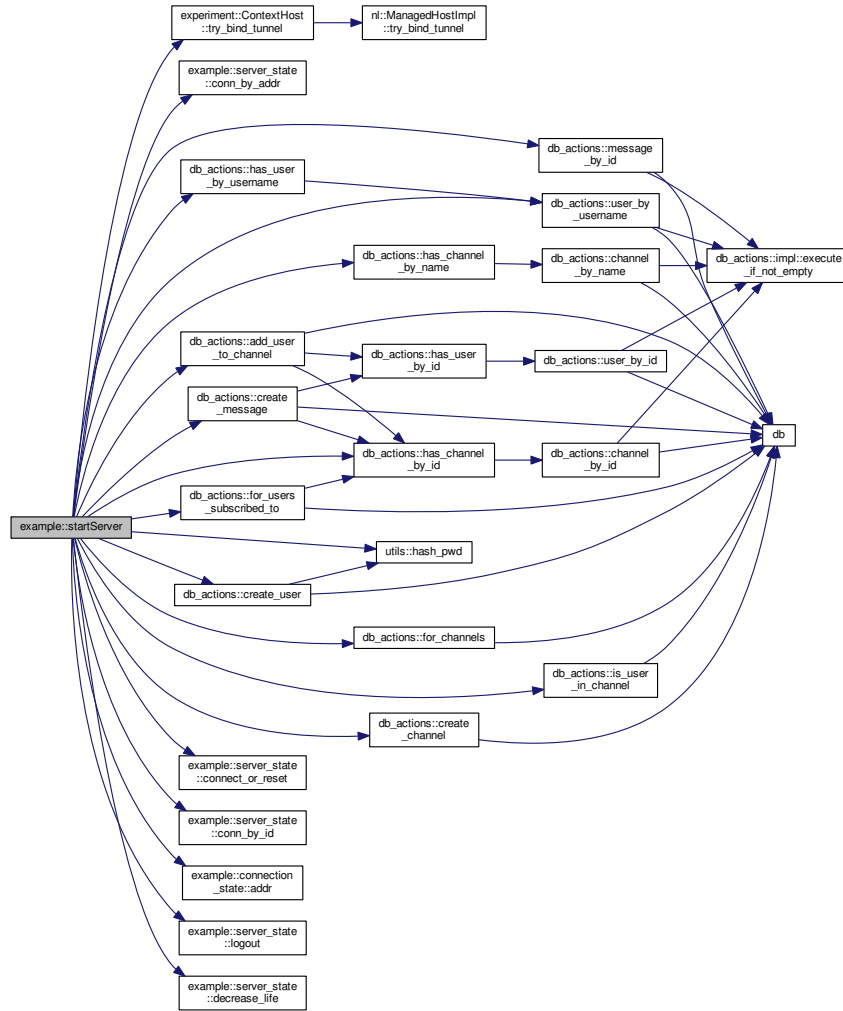
Here is the call graph for this function:
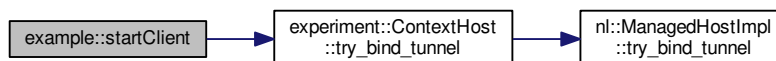
Here is the caller graph for this function:

5..3.7 void example::startServer (   )

Here is the call graph for this function:

Here is the caller graph for this function:

6. example::to_c Namespace Reference

Functions

- NL_DEFINE_PCKT (Outcome,(((bool), valid), ((int), type)))
- NL_DEFINE_PCKT_1 (Messages,((std::vector< std::string >), messages))
- NL_DEFINE_PCKT_1 (Channels,((std::vector< std::string >), channels))
- NL_DEFINE_PCKT (Notify,(((int), channel_id), ((std::string), msg)))
- NL_DEFINE_PCKT_0 (TimedOut)

Variables

- constexpr int ot_login = 0
- constexpr int ot_registration = 1
- constexpr int ot_create_channel = 2
- constexpr int ot_create_message = 3
- constexpr int ot_subscribe = 4

6..1 Function Documentation

6..1.1 example::to_c::NL_DEFINE_PCKT (Outcome , (((bool), valid), ((int), type)))

6..1.2 example::to_c::NL_DEFINE_PCKT (Notify , (((int), channel_id), ((std::string), msg)))

6..1.3 example::to_c::NL_DEFINE_PCKT_0 (TimedOut)

6..1.4 example::to_c::NL_DEFINE_PCKT_1 (Messages , ((std::vector< std::string >), messages))

6..1.5 example::to_c::NL_DEFINE_PCKT_1 (Channels , ((std::vector< std::string >), channels))

6..2 Variable Documentation

6..2.1 constexpr int example::to_c::ot_create_channel = 2

6..2.2 constexpr int example::to_c::ot_create_message = 3

6..2.3 constexpr int example::to_c::ot_login = 0

6..2.4 constexpr int example::to_c::ot_registration = 1

6..2.5 constexpr int example::to_c::ot_subscribe = 4

7. example::to_s Namespace Reference

Functions

- NL_DEFINE_PCKT (Registration,(((std::string), user), ((std::string), pass)))
- NL_DEFINE_PCKT (Login,(((std::string), user), ((std::string), pass)))
- NL_DEFINE_PCKT_1 (CreateChannel,((std::string), name))
- NL_DEFINE_PCKT_1 (DeleteChannel,((int), id))
- NL_DEFINE_PCKT_1 (Subscribe,((int), id))
- NL_DEFINE_PCKT (SendMessage,(((int), channel_id), ((std::string), contents)))
- NL_DEFINE_PCKT (GetMessages,(((int), channel_id), ((int), count)))

- NL_DEFINE_PCKT_0 (ChannelList)
- NL_DEFINE_PCKT_0 (Logout)

7..1 Function Documentation

7..1.1 example::to_s::NL_DEFINE_PCKT (Registration , (((std::string), user), ((std::string), pass)))

7..1.2 example::to_s::NL_DEFINE_PCKT (Login , (((std::string), user), ((std::string), pass)))

7..1.3 example::to_s::NL_DEFINE_PCKT (SendMessage , (((int), channel_id), ((std::string), contents)))

7..1.4 example::to_s::NL_DEFINE_PCKT (GetMessages , (((int), channel_id), ((int), count)))

7..1.5 example::to_s::NL_DEFINE_PCKT_0 (ChannelList)

7..1.6 example::to_s::NL_DEFINE_PCKT_0 (Logout)

7..1.7 example::to_s::NL_DEFINE_PCKT_1 (CreateChannel , ((std::string), name))

7..1.8 example::to_s::NL_DEFINE_PCKT_1 (DeleteChannel , ((int), id))

7..1.9 example::to_s::NL_DEFINE_PCKT_1 (Subscribe , ((int), id))

8. Class Documentation

9. example::client_state Struct Reference

Collaboration diagram for example::client_state:

example::client_state
+ s
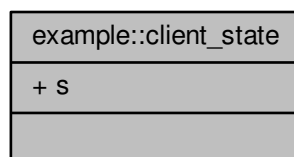
Public Attributes

- cs s {cs::unlogged}

9..1 Member Data Documentation

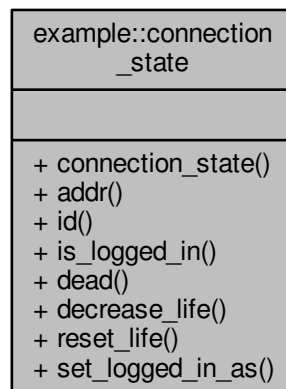### 9..1.1 cs example::client_state::s {cs::unlogged}

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

- src/main.cpp

10. example::connection_state Class Reference

Collaboration diagram for example::connection_state:

Public Member Functions

- connection_state (const nl::PAddress &addr)
- const auto & addr ()

- const auto & id ()
- auto is_logged_in ()
- auto dead ()
- void decrease_life ()
- void reset_life ()
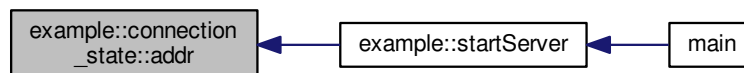- void set_logged_in_as (int id)

Friends

- class server_state

10..1 Constructor & Destructor Documentation

10..1.1 example::connection_state::connection_state (const nl::PAddress & *addr*)

10..2 Member Function Documentation

10..2.1 const auto& example::connection_state::addr ( )

Here is the caller graph for this function:

10..2.2   auto example::connection state::dead (   )

10..2.3   void example::connection state::decrease life (   )

10..2.4   const auto& example::connection state::id (   )

10..2.5   auto example::connection state::is logged in (   )

10..2.6   void example::connection state::reset life (   )

10..2.7   void example::connection state::set logged in as (int *id*   )

10..3   Friends And Related Function Documentation

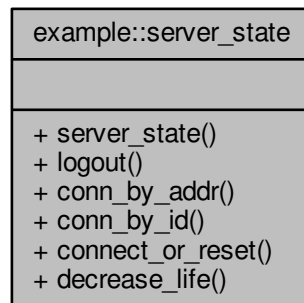### 10..3.1   friend class server state [friend]

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

- src/main.cpp

11.   example::server state Class Reference

Collaboration diagram for example::server state:

```
+---------------------------+
|  example::server_state    |
+---------------------------+
|                           |
+---------------------------+
| + server_state()          |
| + logout()                |
| + conn_by_addr()          |
| + conn_by_id()            |
| + connect_or_reset()      |
| + decrease_life()         |
+---------------------------+
```

Public Member Functions

- server_state ()=default
- bool logout (const nl::PAddress &x)
- connection_state ∗ conn_by_addr (const nl::PAddress &x)
- connection_state ∗ conn_by_id (int id)
- void connect_or_reset (int id, const nl::PAddress &x)
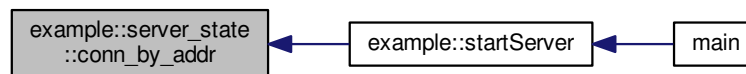- template<typename TF >
  void decrease_life (TF &&f)

11..1 Constructor & Destructor Documentation

11..1.1 example::server_state::server_state ( ) [default]

11..2 Member Function Documentation

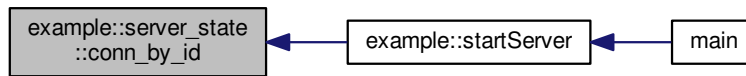### 11..2.1 connection_state∗ example::server_state::conn_by_addr (const nl::PAddress & *x*)
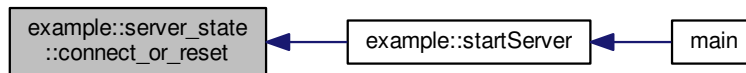
Here is the caller graph for this function:

11..2.2 connection_state∗ example::server_state::conn_by_id (int *id*)
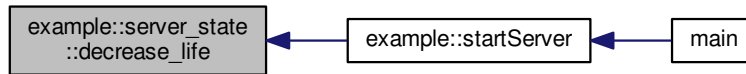
Here is the caller graph for this function:

11..2.3 void example::server_state::connect_or_reset (int *id,* const nl::PAddress & *x*)

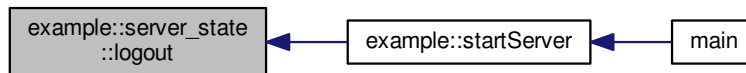Here is the caller graph for this function:

11..2.4 template<typename TF > void example::server_state::decrease↩ _life (TF && *f*)

Here is the caller graph for this function:

11..2.5 bool example::server_state::logout (const nl::PAddress & *x*)

Here is the caller graph for this function:

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

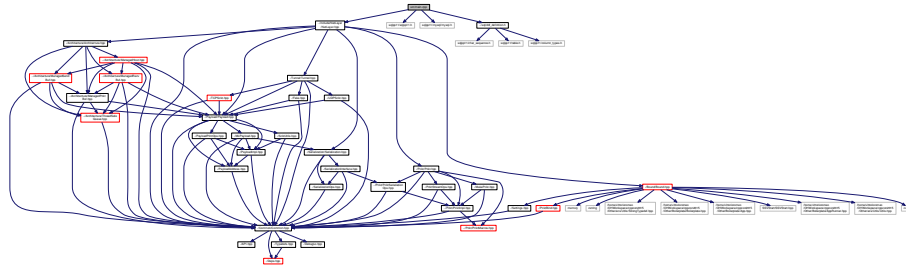- src/main.cpp

12. File Documentation

13. src/main.cpp File Reference

```
#include "../include/NetLayer/NetLayer.hpp"
```

```
#include <sqlpp11/sqlpp11.h>
#include <sqlpp11/mysql/mysql.h>
#include "../sql/ddl_definition.h"
```
Include dependency graph for main.cpp:

Classes

- class example::connection_state
- class example::server_state
- struct example::client_state

Namespaces

- example
- example::to_s
- example::to_c
- utils
- db_actions
- db_actions::impl

Macros

- #define EXAMPLE_USE_UDP 0

Typedefs

- using example::MySettings = nle::Settings< nl::UInt32 >
- using example::MyServerConfig = decltype(my_server_config)
- using example::MyClientConfig = decltype(my_client_config)
- using example::MyCtxServer = nle::ContextHost< MyServerConfig >
- using example::MyCtxClient = nle::ContextHost< MyClientConfig >

Enumerations

- enum example::cs {
  example::cs::unlogged, example::cs::logged, example::cs::awaiting_channel_list,
  example::cs::selecting_channel,
  example::cs::awaiting_login_response, example::cs::awaiting_create_channel_response,
  example::cs::awaiting_registration_response, example::cs::awaiting_create_message↩
  _response,
  example::cs::awaiting_subscribe_response }

Functions

- template<typename T >
  auto getInput (const std::string &title)
- auto getInputLine (const std::string &title)
- auto & db ()
- void initialize_db_connection ()
- example::to_s::NL_DEFINE_PCKT (Registration,(((std::string), user), ((std↩
  ::string), pass)))
- example::to_s::NL_DEFINE_PCKT (Login,(((std::string), user), ((std::string),
  pass)))
- example::to_s::NL_DEFINE_PCKT_1 (CreateChannel,((std::string), name))
- example::to_s::NL_DEFINE_PCKT_1 (DeleteChannel,((int), id))
- example::to_s::NL_DEFINE_PCKT_1 (Subscribe,((int), id))
- example::to_s::NL_DEFINE_PCKT (SendMessage,(((int), channel_id), ((std↩
  ::string), contents)))

- example::to_s::NL_DEFINE_PCKT (GetMessages,(((int), channel_id), ((int), count)))
- example::to_s::NL_DEFINE_PCKT_0 (ChannelList)
- example::to_s::NL_DEFINE_PCKT_0 (Logout)
- example::to_c::NL_DEFINE_PCKT (Outcome,(((bool), valid), ((int), type)))
- example::to_c::NL_DEFINE_PCKT_1 (Messages,((std::vector< std::string >), messages))
- example::to_c::NL_DEFINE_PCKT_1 (Channels,((std::vector< std::string >), channels))
- example::to_c::NL_DEFINE_PCKT (Notify,(((int), channel_id), ((std::string), msg)))
- example::to_c::NL_DEFINE_PCKT_0 (TimedOut)
- auto utils::hash_pwd (const std::string &x)
- template<typename T , typename TF >
  bool db_actions::impl::execute_if_not_empty (T &&r, TF &&f)
- template<typename TF >
  void db_actions::for_channels (TF &&f)
- template<typename TF >
  bool db_actions::message_by_id (int id, TF &&f)
- template<typename TF >
  bool db_actions::user_by_id (int id, TF &&f)
- template<typename TF >
  bool db_actions::user_by_username (const std::string &username, TF &&f)
- bool db_actions::has_user_by_id (int id)
- bool db_actions::has_user_by_username (const std::string &username)
- template<typename TF >
  bool db_actions::channel_by_id (int id, TF &&f)
- template<typename TF >
  bool db_actions::channel_by_name (const std::string &name, TF &&f)
- bool db_actions::has_channel_by_id (int id)
- bool db_actions::has_channel_by_name (const std::string &name)
- auto db_actions::create_user (const std::string &user, const std::string &pass)

- auto db_actions::create_channel (int user_id, const std::string &name)
- auto db_actions::delete_channel (int id)
- auto db_actions::add_user_to_channel (int user_id, int channel_id)
- auto db_actions::remove_user_from_channel (int user_id, int channel_id)
- auto db_actions::create_message (int user_id, int channel_id, const std::string &contents)
- auto db_actions::is_user_in_channel (int user_id, int channel_id)
- template<typename TF >
  void db_actions::for_users_subscribed_to (int channel_id, TF &&f)
- constexpr auto example::my_pckt_binds (nle::pckt_binds< to_s::Registration, to_s::Login, to_s::CreateChannel, to_s::DeleteChannel, to_s::SendMessage, to_↩s::GetMessages, to_s::ChannelList, to_s::Subscribe, to_s::Logout, to_c::Outcome, to_c::Messages, to_c::Notify, to_c::Channels, to_c::TimedOut >())
- constexpr auto example::my_server_tunnel (nle::tunnel_type< nl::Tunnel::TC↩PListener >{})
- constexpr auto example::my_client_tunnel (nle::tunnel_type< nl::Tunnel::TC↩PSckt >{})
- constexpr auto example::my_server_config (nle::make_config< MySettings >(my↩_pckt_binds, my_server_tunnel))
- constexpr auto example::my_client_config (nle::make_config< MySettings >(my↩_pckt_binds, my_client_tunnel))
- void example::startServer ()
- void example::startClient (nl::Port port)
- int main ()

Variables

- std::unique_ptr< mysql::connection > _db
- example_ddl::TblUser tbl_user
- example_ddl::TblChannel tbl_channel
- example_ddl::TblMessage tbl_message
- example_ddl::TblUserChannel tbl_user_channel
- constexpr int example::to_c::ot_login = 0

- constexpr int example::to_c::ot_registration = 1
- constexpr int example::to_c::ot_create_channel = 2
- constexpr int example::to_c::ot_create_message = 3
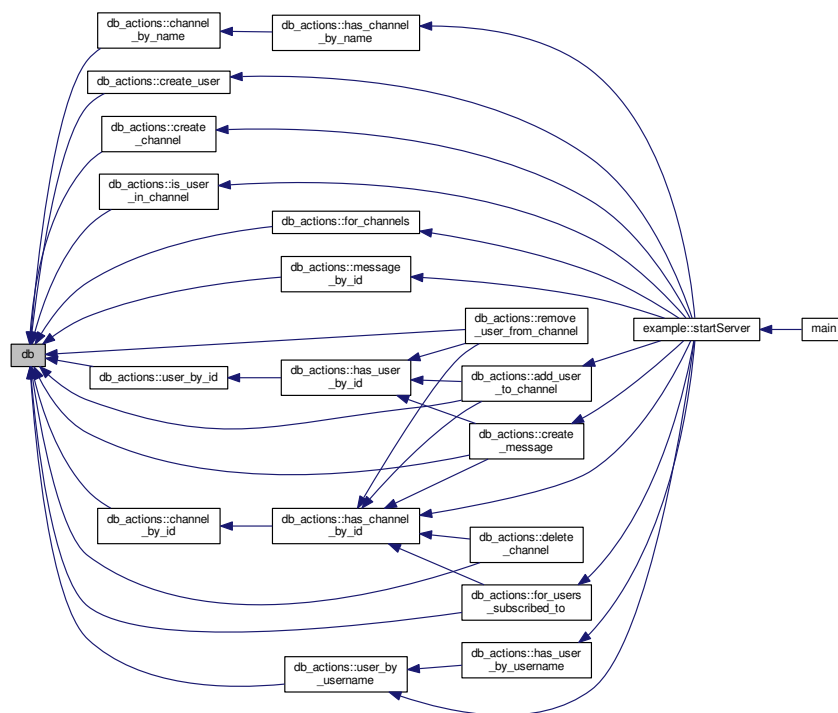- constexpr int example::to_c::ot_subscribe = 4

13..1 Macro Definition Documentation

13..1.1 #define EXAMPLE_USE_UDP 0

13..2 Function Documentation

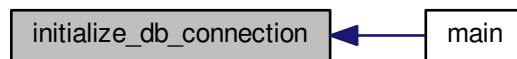### 13..2.1 auto& db ( )

Here is the caller graph for this function:

13..2.2   template<typename T > auto getInput (const std::string &
title)

13..2.3   auto getInputLine (const std::string & *title* **)**

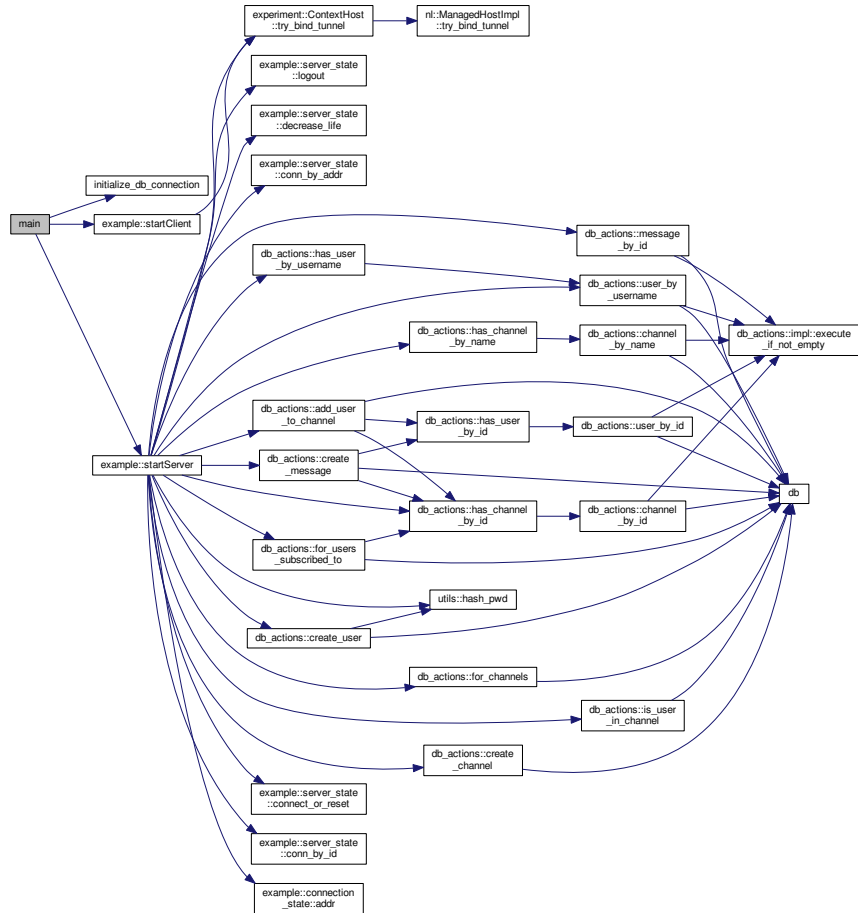**13..2.4   void initialize_db_connection (   )**

Here is the caller graph for this function:

13..2.5 int main ( )

Here is the call graph for this function:

13..3   Variable Documentation

13..3.1   std::unique_ptr<mysql::connection> _db

13..3.2   example_ddl::TblChannel tbl_channel

13..3.3   example_ddl::TblMessage tbl_message

13..3.4   example_ddl::TblUser tbl_user

13..3.5   example_ddl::TblUserChannel tbl_user_channel

14.   Screenshots

Part V

References

Listed below are the references used during the development of **veeForum** and the writing of this thesis.

- veeForum GitHub repository: `https://github.com/SuperV1234/veeForum`

- LatexPP GitHub repository: `https://github.com/SuperV1234/Experiments`

- UNIME website: `http://unime.it`

- My personal website: `https://vittorioromeo.info`

- ShareLaTeX learn: `https://www.sharelatex.com/learn`

- Wikipedia - Software engineering: `http://en.wikipedia.org/wiki/Software_engineering`

- Software engineering vs Programming: `http://www.ics.uci.edu/~ziv/ooad/intro_to_se/tsld008.htm`

- IEEE SRS guidelines: `home.agh.edu.pl/~jsw/io/IEEE830.doc`

- MariaDB documentation: `https://mariadb.org/docs/`

- PHP documentation: `http://php.net/docs.php`

- Git documentation: `https://git-scm.com/documentation`

- Docker documentation: `https://docs.docker.com/`

- GitHub: `https://github.com/`

- Arch Linux wiki: `https://wiki.archlinux.org/`