Universita' degli Studi di Messina
Dipartimento di Matematica e Informatica

Programming II project

NetLayer

11 June 2015

Authors:
Vittorio Romeo

Professors:
Massimo Vilari

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** multithreaded 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.

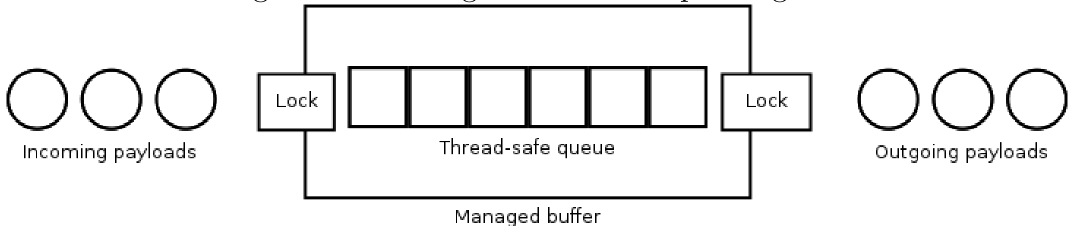
Functional requirements may be **calculations**, **technical details**, **data manipulation and processing** and other specific functionality that define what a system is supposed to accomplish.

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.

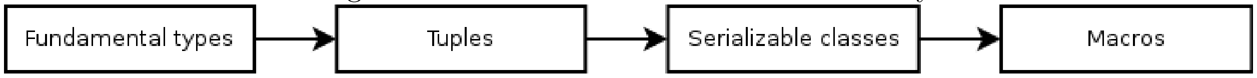
Figure 2.1: Managed buffer example diagram.



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.

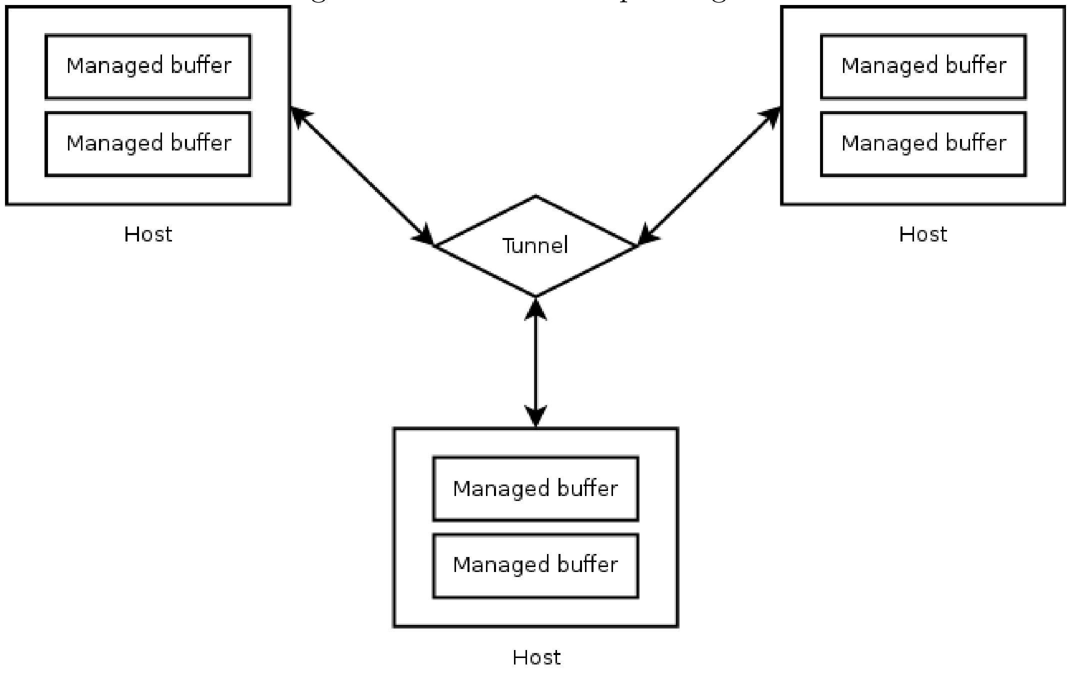
Figure 2.2: Automatic serialization hierarchy.



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.

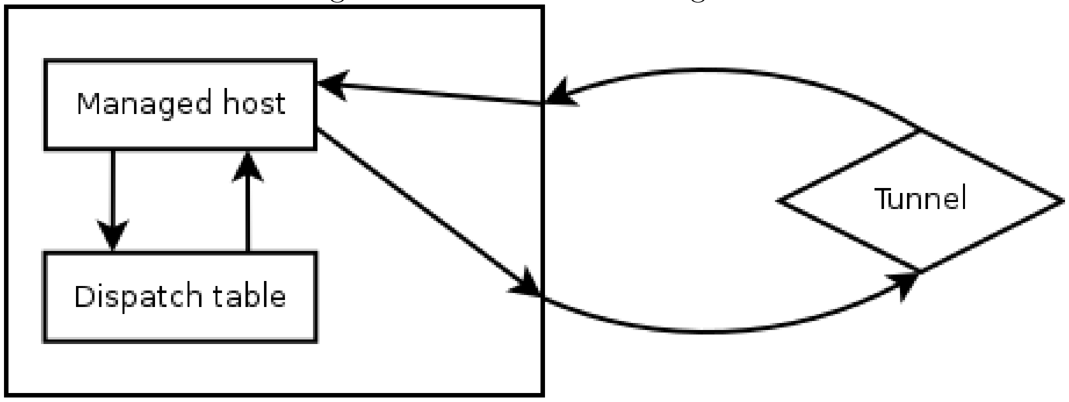


4..2.4 Context binding

- **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.

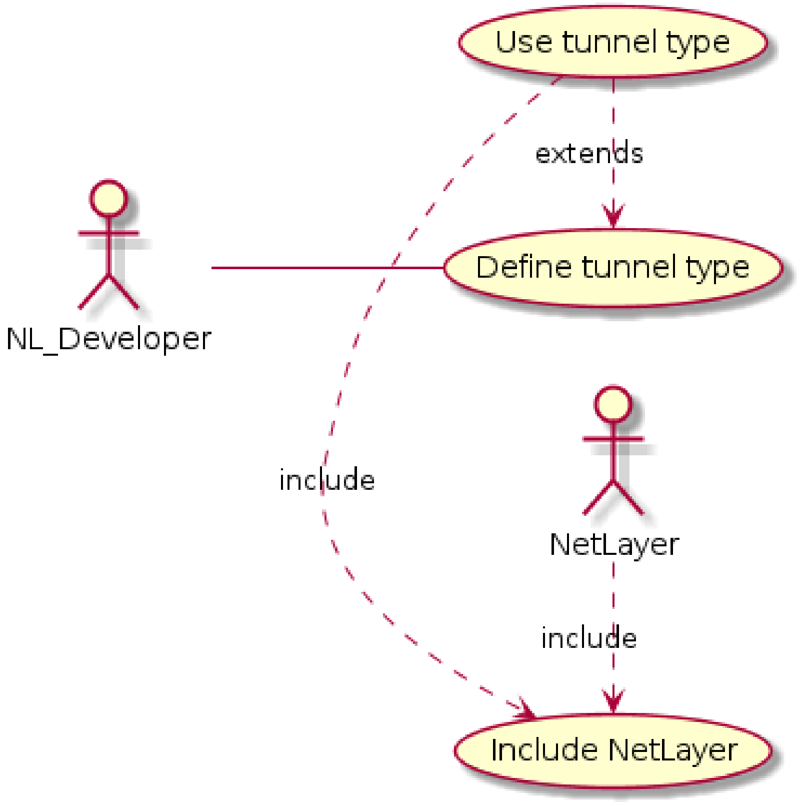
4..3.1.2 Pre-conditions

- 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.

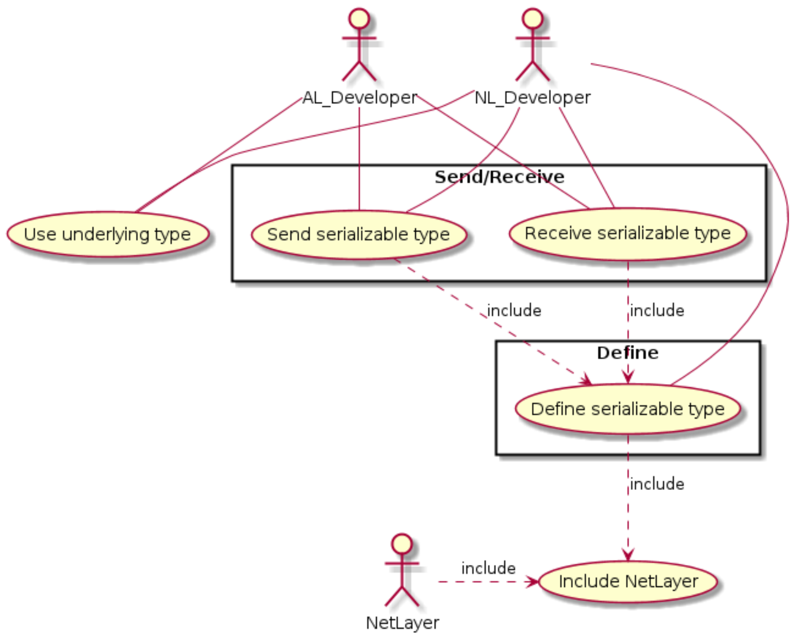
4..3.2.2 Pre-conditions

- 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.

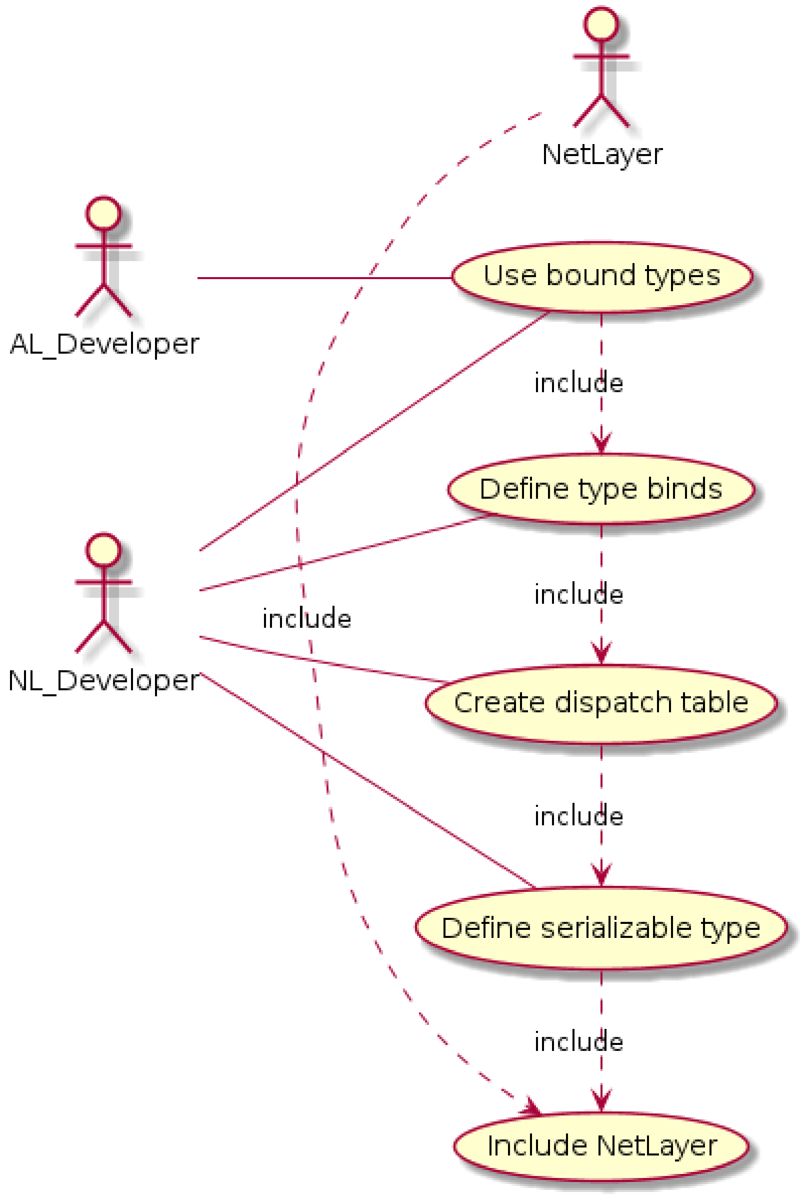
4..3.3.2 Pre-conditions

- 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.

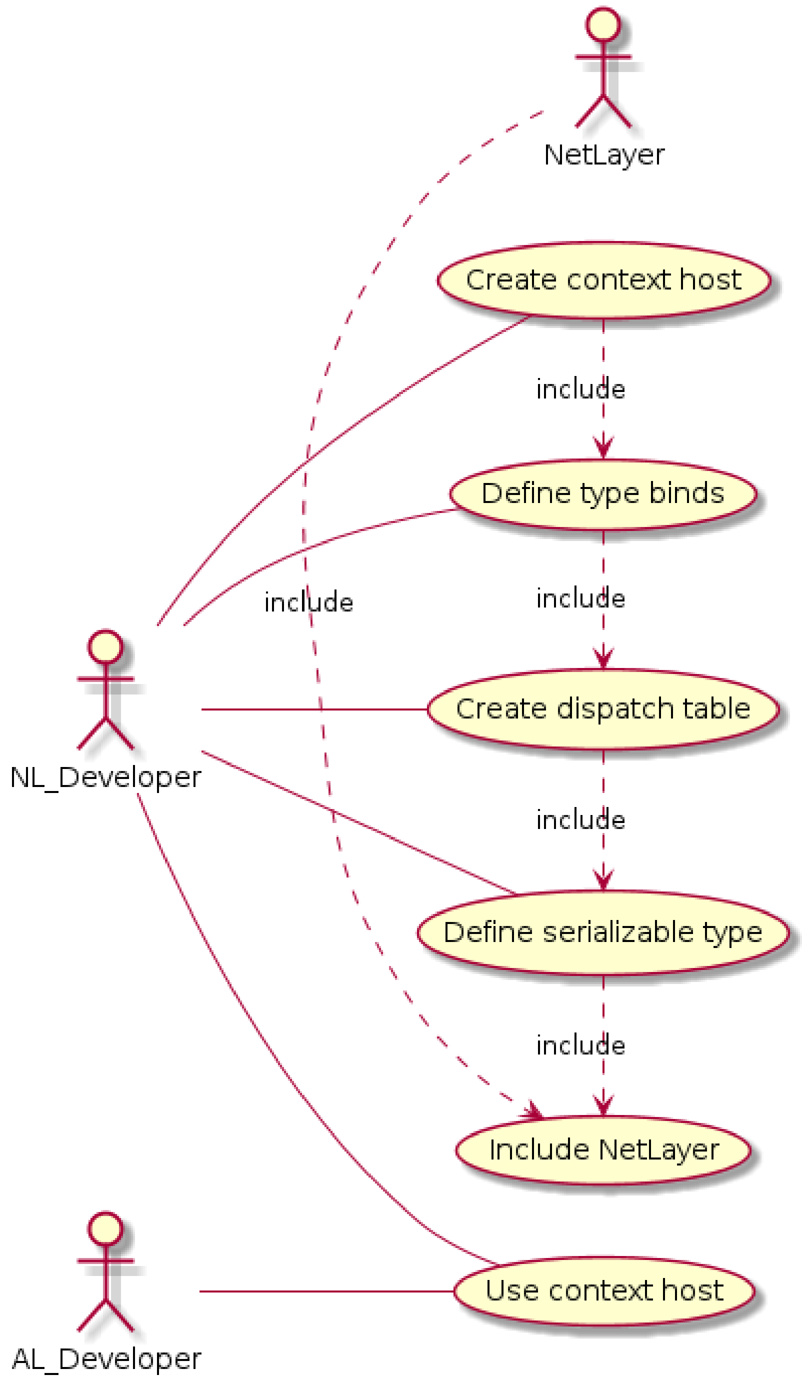
4..3.4.2 Pre-conditions

- 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.

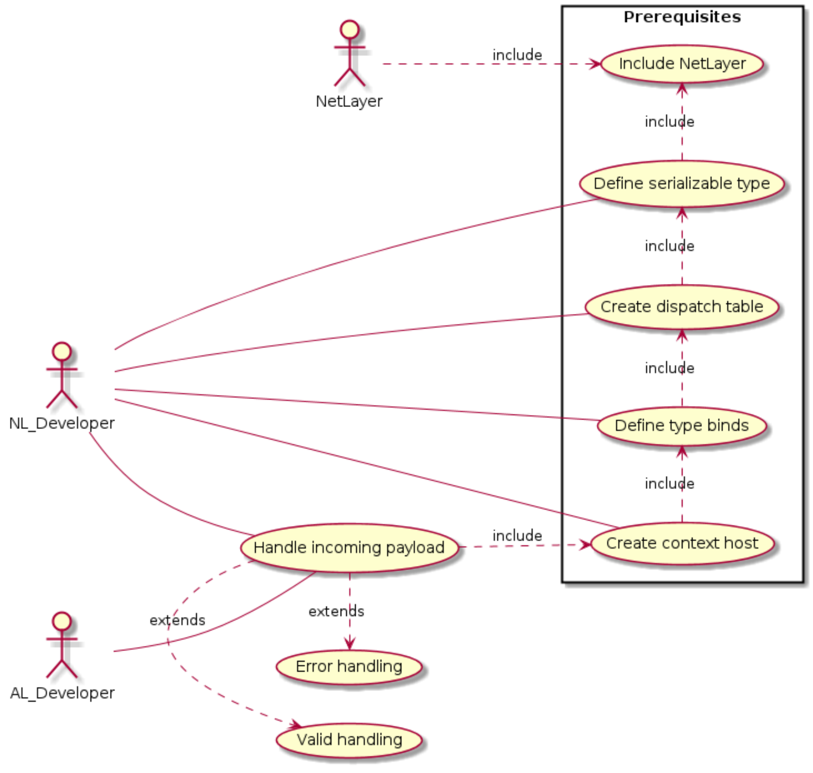
4..3.5.3 Flow of events

-

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.

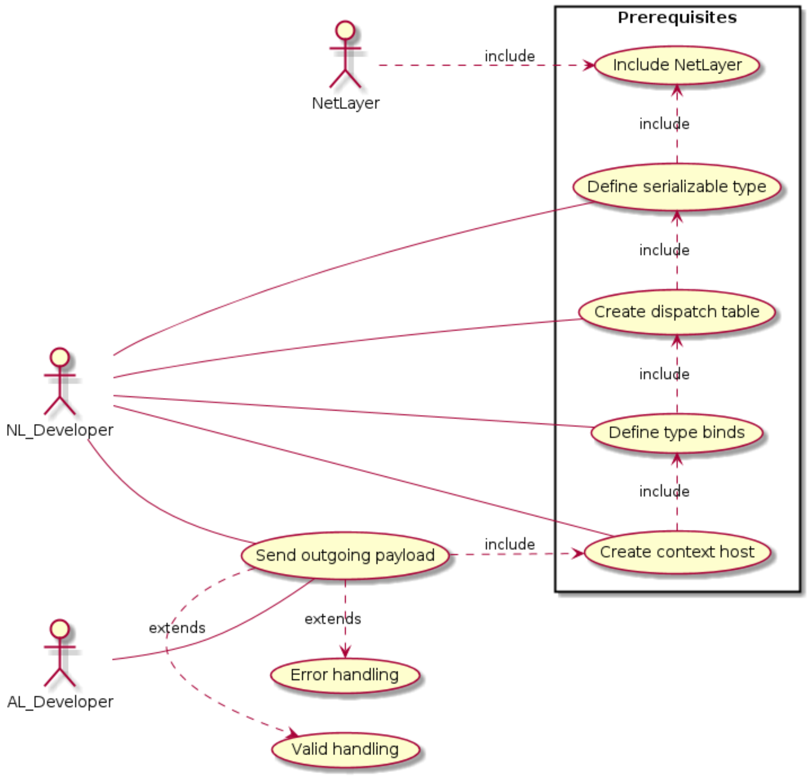
4..3.6.2 Pre-conditions

- 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, bug-fixing and feature addition.

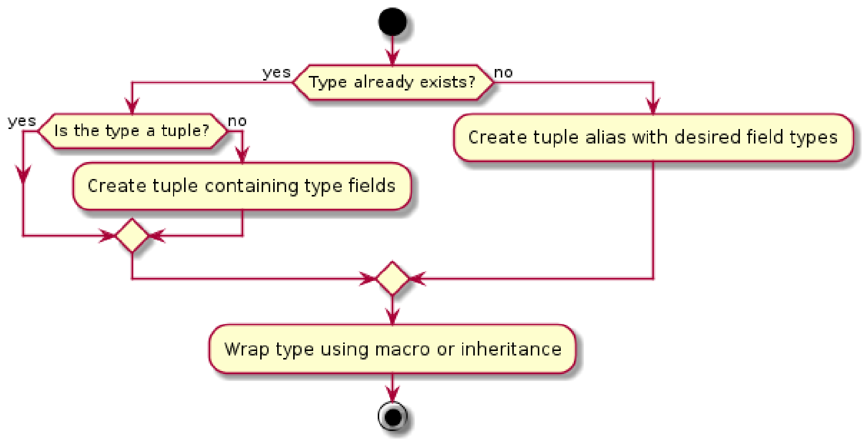
To ensure maximum portability, the product will be designed to work on the most popular **GNU/Linux** distributions and will be thoroughly tested on different platforms.

5. Analysis models

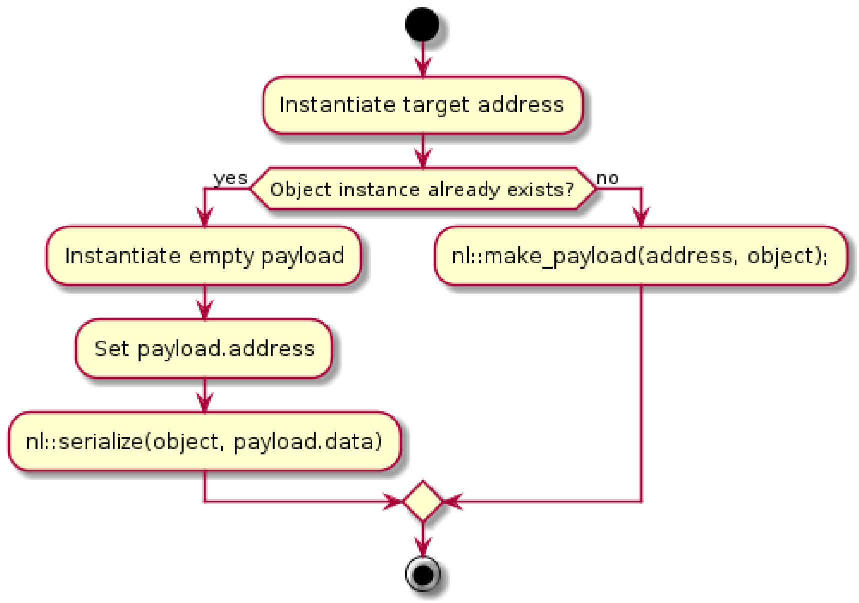
5..1 Activity diagrams

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.

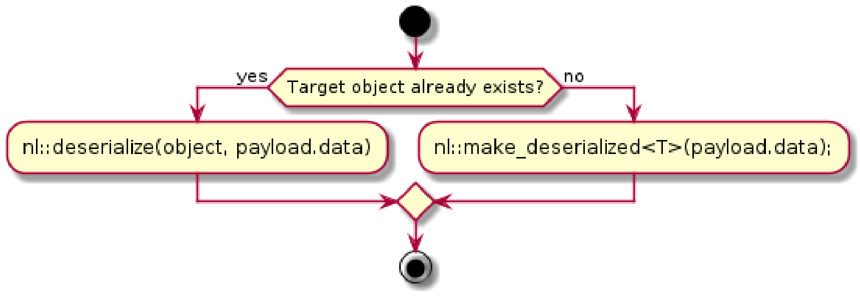

Type already exists?
yes
no
Is the type a tuple?
yes
no
Create tuple containing type fields
Create tuple alias with desired field types
Wrap type using macro or inheritance

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

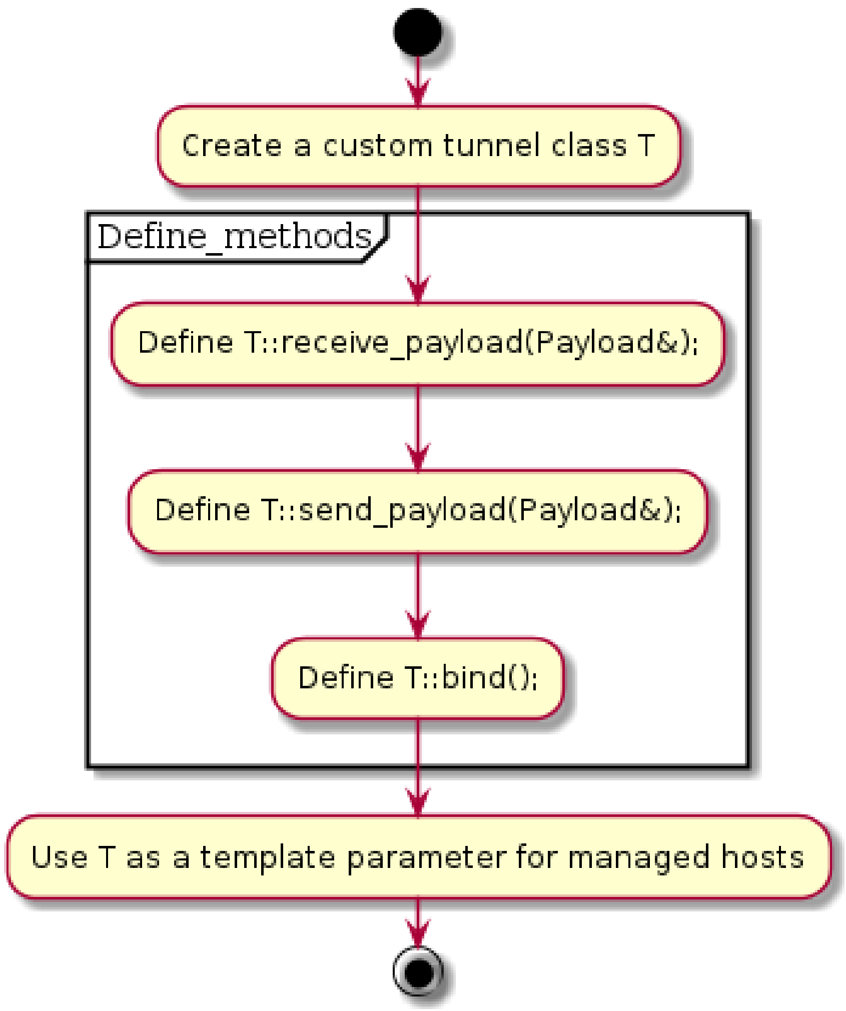


Instantiate target address
yes
Object instance already exists?
no
Instantiate empty payload
nl::make_payload(address, object);
Set payload.address
nl::serialize(object, payload.data)

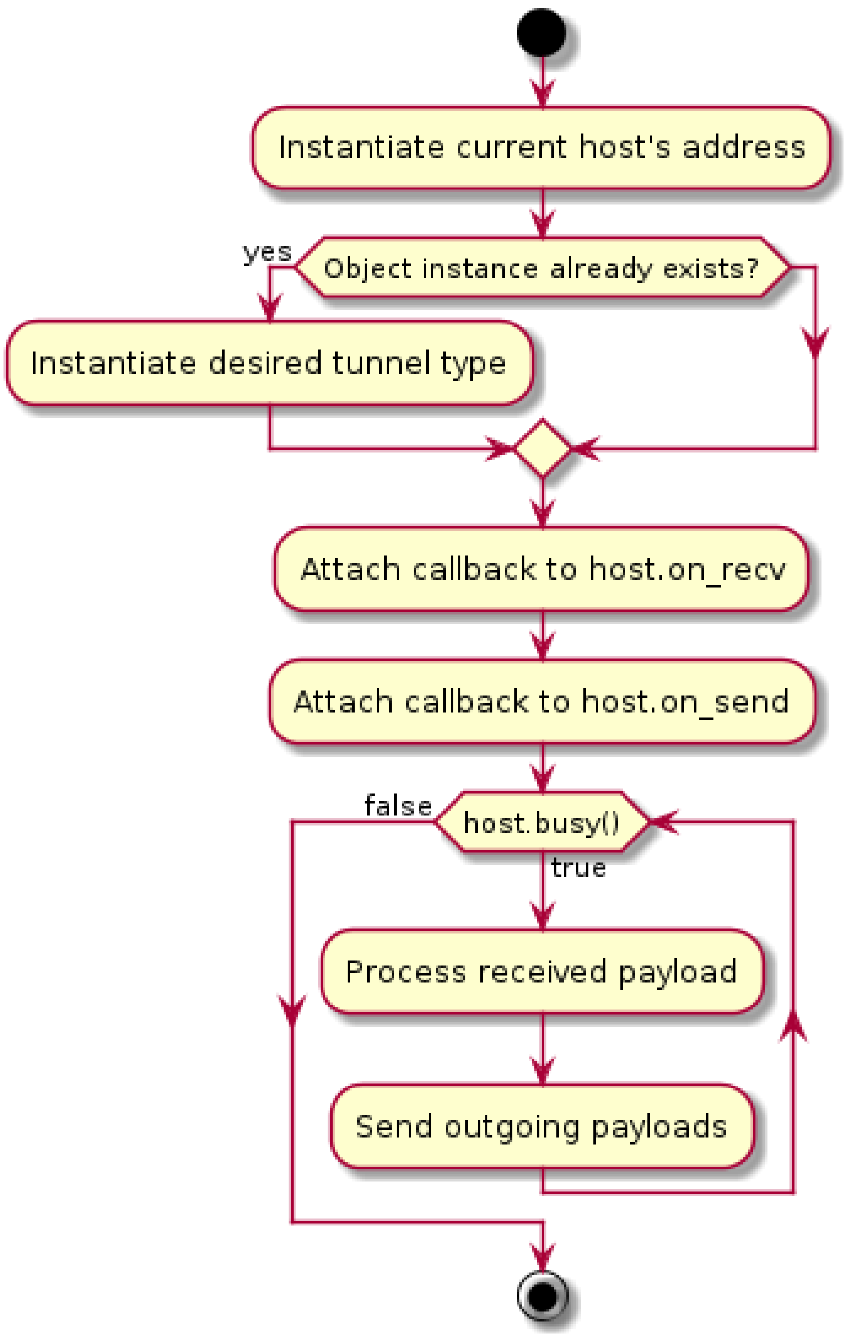
The following diagram shows the required steps to deserialize a payload into an object.


yes
Target object already exists?
no
nl::deserialize(object, payload.data)
nl::make_deserialized<T>(payload.data);

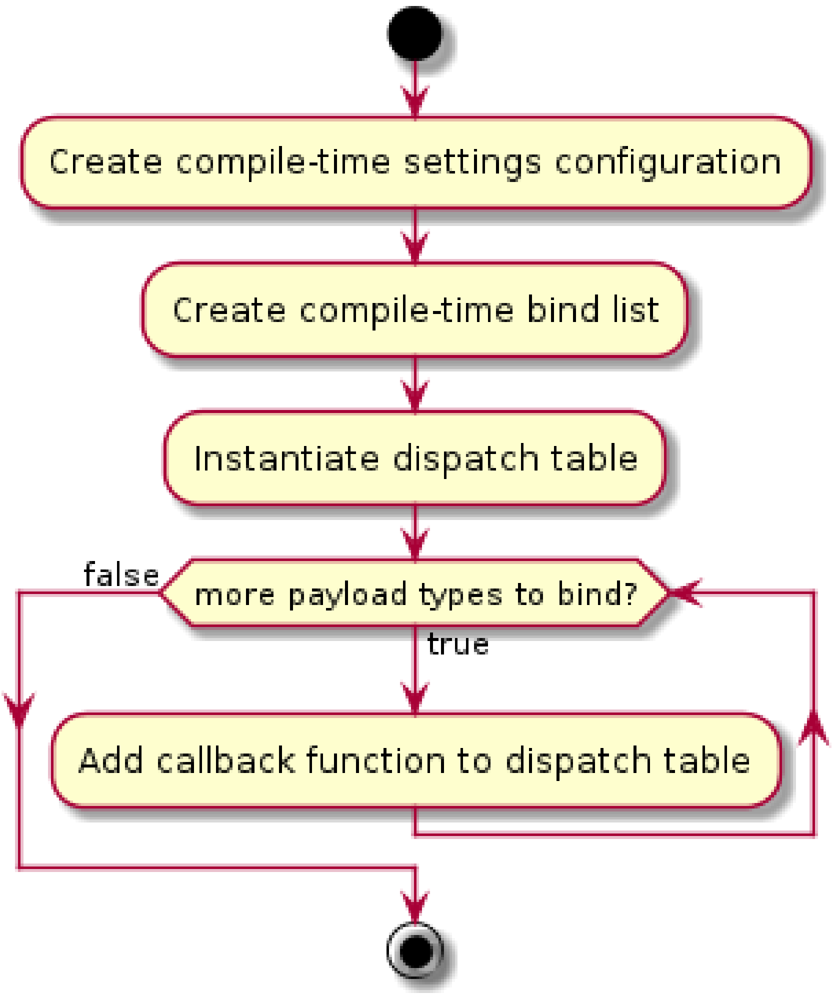
The following diagram shows the required steps to define a tunnel.


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

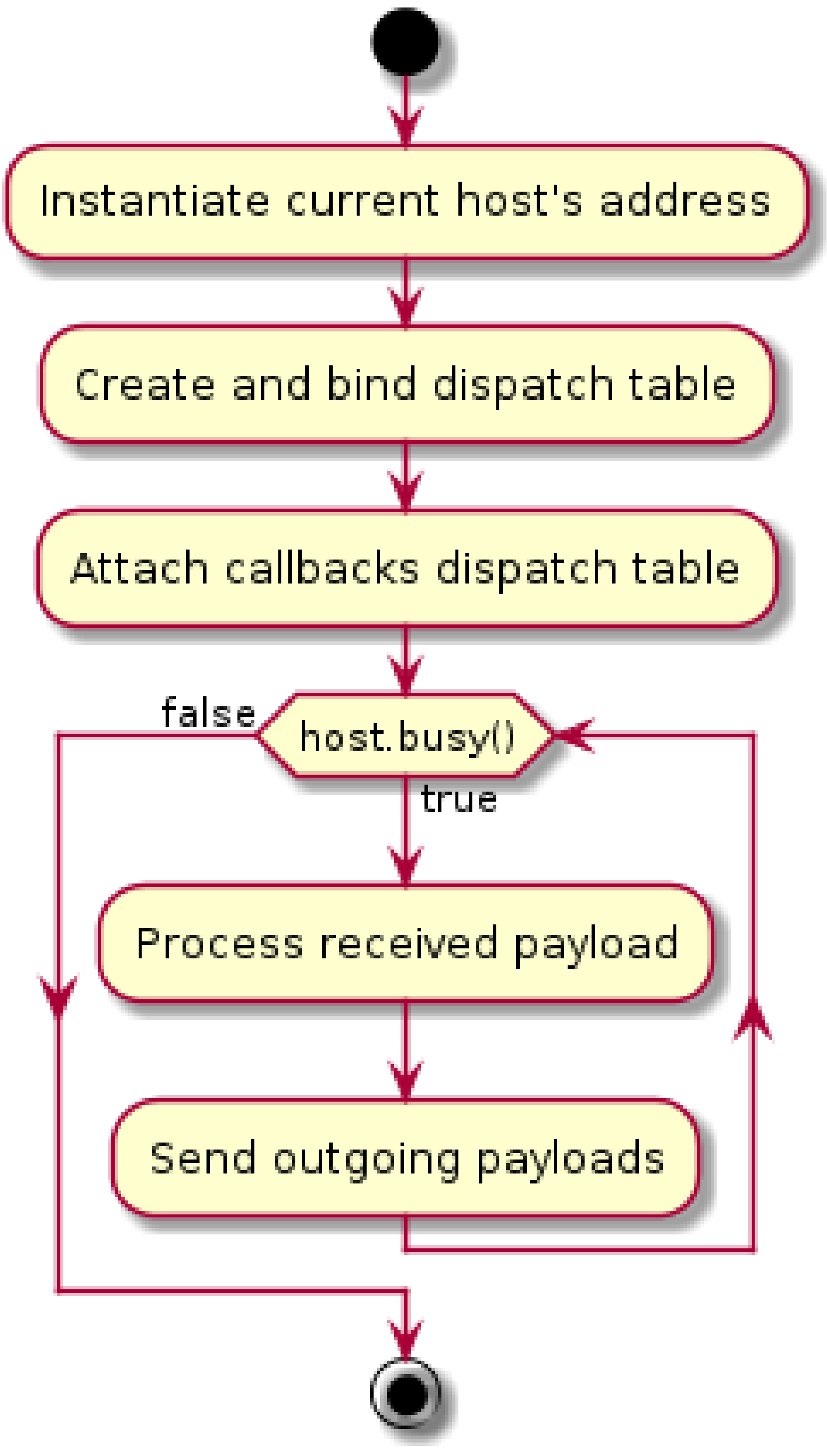
The following diagram shows the required steps to create a managed host.


Instantiate current host's address
Object instance already exists?
yes
Instantiate desired tunnel type
Attach callback to host.on_recv
Attach callback to host.on_send
host.busy()
false
true
Process received payload
Send outgoing payloads

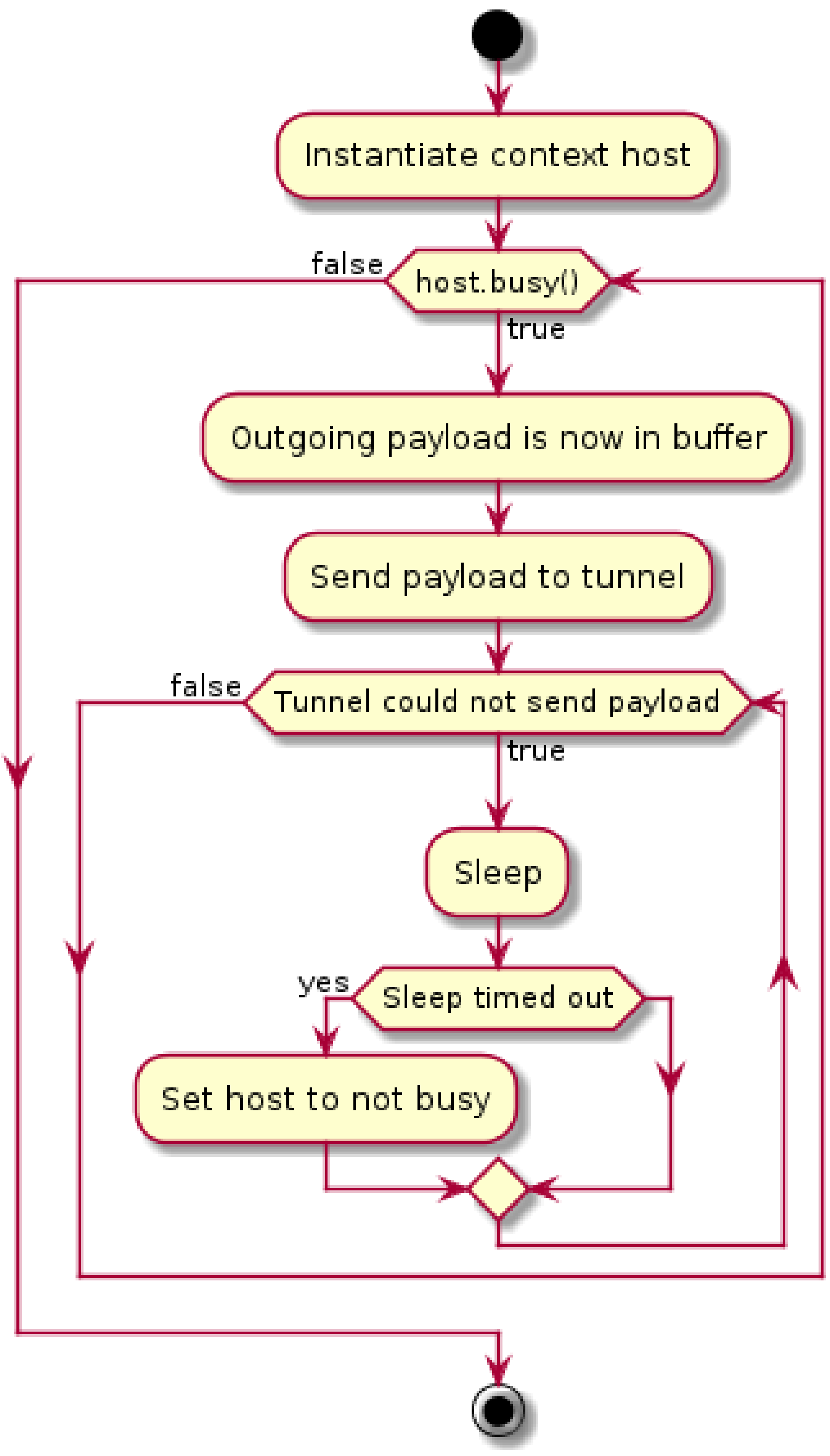
The following diagram shows the required steps to create and bind a dispatch table.


Create compile-time settings configuration
Create compile-time bind list
Instantiate dispatch table
more payload types to bind?
false
true
Add callback function to dispatch table

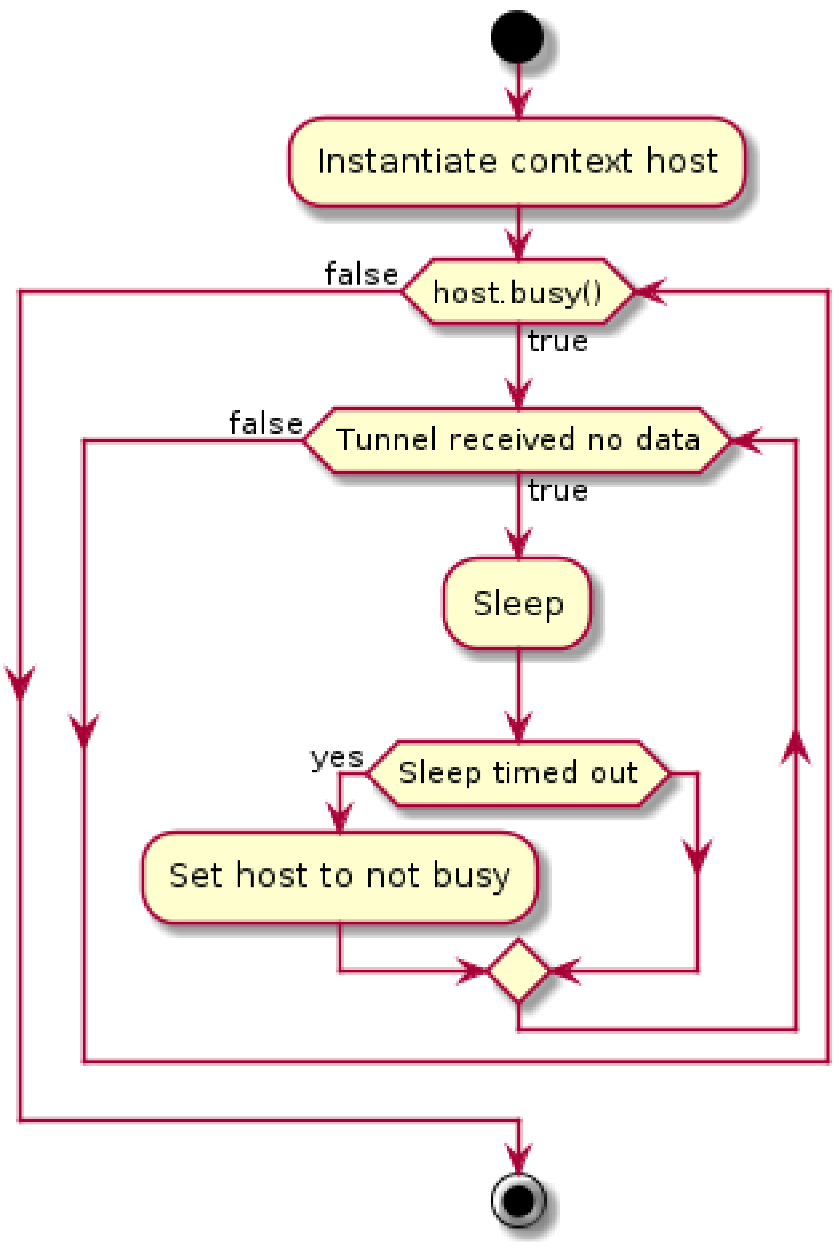
The following diagram shows the required steps to create a context host.


Instantiate current host's address
Create and bind dispatch table
Attach callbacks dispatch table
false
host.busy()
true
Process received payload
Send outgoing payloads

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.

5..3 Sequence diagrams

The following diagram shows the interaction between **forum users**, the **subscription broker** and the **content management** systen in order to manage subscriptions and generate notifications.

The following diagram shows the interaction between the backend and the database.

The following diagram shows the interaction between the backend and database stored procedures.

The following diagram shows the interaction between the authentication system and the database.

5..4 Deployment diagram

A **deployment diagram** in the Unified Modeling Language models the physical deployment of artifacts on nodes.

The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones.