# INET Framework Migration Guide

***Release 4.7***

**INET community**

**Nov 02, 2026**

# CONTENTS


**1 Network Node Architecture** **3**
- 1.1 Extending the Known Protocols . . . 3
- 1.2 Registering Protocols for Dispatching . . . 3
- 1.3 Attaching Tags for Dispatching . . . 4
- 1.4 Configuring Applications . . . 4
- 1.5 Configuring Protocols . . . 4

**2 Tags API** **5**
- 2.1 Splitting Control Infos . . . 5
- 2.2 Communicating Through Protocol Layers . . . 5
- 2.3 Determining the Protocol of Packets . . . 6

**3 Packet API** **7**
- 3.1 Protocol Header Classes . . . 7
- 3.2 Immutability of Chunks . . . 8
- 3.3 Serializing Packets . . . 8
- 3.4 Handling Checksums . . . 8

Release: 4.7

CHAPTER

ONE

# NETWORK NODE ARCHITECTURE

The internal structure of network nodes has been considerably changed. With the new architecture, applications can directly talk to any protocol down to the link layer, and protocols don't have to deal with dispatching to other protocols.

The old `NodeBase` module has been split up into the following base modules:

- `NodeBase` contains mobility, status and energy related submodules
- `LinkLayerNodeBase` adds network interfaces to `NodeBase`
- `NetworkLayerNodeBase` adds network protocols to `LinkLayerNodeBase`
- `TransportLayerNodeBase` adds transport protocols to `NetworkLayerNodeBase`
- `ApplicationLayerNodeBase` adds applications to `TransportLayerNodeBase`

Protocol modules inside the network nodes are separated from each other by a `MessageDispatcher` module. This module is responsible for dispatching packets and commands to the intended receiver module based on various tags:

- `SocketReq` specifies the sender socket
- `SocketInd` specifies the receiver socket
- `InterfaceReq` specifies the receiver interface
- `DispatchProtocolReq` specifies the receiver protocol

The `MessageDispatcher` is also used inside network layer compound modules such as the `Ipv4NetworkLayer`. This usage is not accidental, it solves dispatching ARP, ICMP, and IPv4 packets to the appropriate protocol modules.

## 1.1 Extending the Known Protocols

Internally, protocols first must be added to the list of known protocols in the Protocol class before they can be used. Some protocols (such as IP) have a mapping between protocol-specific integer identifiers and actual protocols. These mappings should be created as a `ProtocolGroup`. Here are some examples of how to do this:

```
const Protocol Protocol::ipv4("ipv4" , "IPv4);

const ProtocolGroup ProtocolGroup::ethertype("ethertype", {
    { 0x0800, &Protocol::ipv4 },
    { 0x0806, &Protocol::arp },
    ...
});
```

## 1.2 Registering Protocols for Dispatching

Modules must register supported protocols with the `MessageDispatcher` to operate properly. This is done by calling `inet::registerProtocol(...)` for each supported protocol on each gate in `initialize()`. Interfaces (usually MAC protocols modules) must also register by calling `inet::registerInterface(...)` for the

corresponding `NetworkInterface` and gate in `initialize()`. On the other hand, sockets are learned by the `MessageDispatcher` automatically on the fly.

## 1.3 Attaching Tags for Dispatching

When a protocol sends a packet or command to another protocol or interface, it must attach the appropriate tag for the `MessageDispatcher`. The dispatcher uses the attached tags to look up the intended receiver in its registration list and forwards the message on the appropriate gate. Here are some examples of how to do this:

```
packet->addTag<DispatchProtocolReq>()->setProtocol(receiverProtocol); // send to␣
↪protocol
packet->addTag<InterfaceReq>()->setInterfaceId(interfaceId); // send to interface
packet->addTag<SocketInd>()->setSocketId(socketId); // send to socket
```

## 1.4 Configuring Applications

The old application submodule vectors (`pingApp`, `udpApp`, `tcpApp`) have been merged into a single application vector (`app`). The merged vector can contain all kinds of applications, which are free to use any protocol they see fit.

This change requires updating the configuration of applications in INI files. In the simplest case, this can be done by simply replacing the application vector names. If the example uses more than one kind of application in a single network node, then the submodule vector indexes must also be updated.

For example, the existing configuration:

```
*.host1.numPingApps = 1
*.host1.pingApp[0].destAddr = "host7"
*.host1.numUdpApps = 1
*.host1.udpApp[0].typename = "UdpSink"
```

The updated configuration:

```
*.host1.numApps = 2
*.host1.app[0].typename = "PingApp"
*.host1.app[0].destAddr = "host7"
*.host1.app[1].typename = "UdpSink"
```

## 1.5 Configuring Protocols

Transport layer and network layer protocols can be enabled/disabled with separate boolean flags (`hasTcp`, `hasUdp`, `hasSctp`, `hasIpv4`, `hasIpv6`). The number of network interfaces can be set with separate parameters (`numLoInterfaces`, `numPppInterfaces`, `numEthInterfaces`, `numWlanInterfaces`, `numTunInterfaces`) or they are set indirectly with the number of connected (pppg, ethg) gates.

CHAPTER

TWO

# TAGS API

Packets no longer carry control info data structures. They have a set of tags attached instead. A tag is usually a very small data structure that focuses on a single parameterization aspect of one or more protocols.

Some notable tag examples:

- `SocketReq`, `SocketInd` specifies the socket
- `MacAddressReq`, `MacAddressInd` specifies source and destination MAC addresses
- `L3AddressReq`, `L3AddressInd` specifies source and destination network addresses
- `SignalPowerReq`, `SignalPowerInd` specifies send and receive signal power
- `DispatchProtocolReq`, `DispatchProtocolInd` specifies intended receiver protocol
- `PacketProtcolTag` specifies the protocol of the packet

Tags come in three flavors:

- requests (called `SomethingReq`) carry information from a higher layer to lower layer protocols
- indications (called `SomethingInd`) carry information from a lower layer to higher layer protocols
- plain tags (called `SomethingTag`) contain some meta information
- base classes (called `SomethingTagBase`) must not be attached to packets

## 2.1 Splitting Control Infos

When migrating a protocol, the old control info data structures, which were attached to packets, must be replaced with a set of tags. Implementors should use already existing tags if possible; otherwise, they are free to create new ones as they see fit.

Any code that sets, reads, or removes control info objects of packets must be replaced with code that adds, reads, or removes the appropriate tags.

Setting control info on commands need not be changed but may be adapted for consistency.

## 2.2 Communicating Through Protocol Layers

Tags can pass through protocol layers and reach far away from the originator protocol in both the downward and upward direction. In general, tags are removed where they are processed, usually turning into some data in a packet. Of course, protocols are free to ignore any tag they wish based on their configuration and state.

When a packet is reused for any purpose (e.g. forwarding, loopback interface, echo application), most likely, all tags on the packet should be removed. The reason is that the implementor can never be sure what kind of tags are attached to a packet and what unintended effects those tags will have at a later stage in some protocol.

Finally, it's important to note that tags are not transmitted from one network node to another. All physical layer protocols are required to delete all tags (except the `PacketProtocolTag`) from a packet before sending it to the peer or the medium. In other words, tags are only meant to be processed in the same network node.

## 2.3 Determining the Protocol of Packets

With the new packet API, packets can no longer be differentiated using the C++ *dynamic_cast* operator with the desired type. The reason is that all packets are instances of the `Packet` class. In fact, this is quite understandable if one views packets as a sequence of bytes. Any sequence of bytes, no matter how it is represented by a `Packet`, can be interpreted by any protocol, even if the packet was not intended to be processed by that protocol. Therefore, before a protocol is sending out a packet using any of its gates, it must attach a `PacketProtocolTag` to it. Here is an example of how to do this:

```
packet->addTagIfAbsent<PacketProtocolTag>()->setProtocol(&Protocol::ipv4); // updates␣
↪tag
```

CHAPTER

THREE

# PACKET API

INET provides a new packet API that supports efficient construction, sharing, duplication, encapsulation, aggregation, fragmentation, and serialization. The data structure also supports dual representation by default. That is, data can be accessed as raw bytes and also as field-based classes. Internally, packets store their data in different kinds of chunks.

The new API uses the following classes at the chunk level:

- `Chunk`
- `ByteCountChunk`, `BytesChunk`, `BitCountChunk`, `BitsChunk`
- `FieldsChunk`
- `SliceChunk`
- `SequenceChunk`
- `cPacketChunk` (for backward compatibility)
- message compiler generated classes (subclassing `FieldsChunk`)

The new API uses the following classes at the packet level:

- `Packet`
- `ReorderBuffer`
- `ReassemblyBuffer`
- `ChunkBuffer`
- `ChunkQueue`

The new API uses the following classes for serialization:

- `ChunkSerializer`
- one subclass of `ChunkSerializer` for each `Chunk` subclass listed above
- `ChunkSerializerRegistry`

## 3.1 Protocol Header Classes

The most substantial change regarding protocols is that protocol-specific headers (or messages) are no longer subclasses of `cPacket`. Protocol headers subclass the `Chunk` class instead, and they are simply added to Packets during processing. Variable references to `Chunk` objects must use shared pointers (`Ptr<SomeChunk>`) types. Here is an example of how to do this:

```
auto ipv4Header = makeShared<IPv4Header>(); // creates mutable chunk
ipv4Header->setSourceAddress(sourceAddress);
packet->insertAtFront(ipv4Header);
```

(continues on next page)

(continued from previous page)

```
const auto& ipv4Header = packet->peekAtFront<IPv4Header>(); // return immutable chunk
auto sourceAddress = ipv4Header->getSourceAddress();
```

Sometimes processing in a protocol module requires multiple utility functions and classes. Some functions may need the packet and the protocol header at the same time. Only passing the protocol header is not sufficient because due to the shared nature of chunks, they don't have an owner packet. Only passing the packet requires the called function to peek at the protocol header, which might unnecessarily slow down execution. In such cases, it is a good idea to pass the packet and the protocol header in separate parameters. Whether this is desirable or not highly depends on the complexity of the protocol and the organization of its implementation.

## 3.2 Immutability of Chunks

Another important difficulty to note is that chunks can only be added to packets if they are immutable. This requirement comes from the fact that packets support peeking into their data regardless of how the data is represented. The result of peek operations is required to stay consistent with the original content of the packet. Moreover, the content of packets can be arbitrarily shared with other packets that may be potentially present in different network nodes. Unfortunately, these properties forbid arbitrary changes once the chunk has been added to the packet. Of course, internally, packets do their best to reuse any chunk data structure if possible.

When the need arises to change the contents of the packet, such as forwarding a packet in a network protocol, the best thing to do is the following. Remove the part that is to be updated, create a mutable copy, update it according to the protocol, and add the updated part back to the packet. In fact, this is like saying that forwarding a packet is the same as sending out another packet that shares some structure with the received one. Here is an example of how to do this:

```
auto ipv4Header = packet->removeAtFront<IPv4Header>(); // duplicate is necessary
ipv4Header->setTimeToLive(ipv4Header->getTimeToLive() - 1); // mutable chunk
packet->insertAtFront(ipv4Header);
```

## 3.3 Serializing Packets

The old packet serializer classes have been replaced with new classes subclassing from the `ChunkSerializer` class. The old serializers used to not only serialize the packet they were responsible for but they also recursed into the encapsulated packet. This is no longer the case as serializers are only responsible for the corresponding chunk that they handle.

Actually transforming a packet to a sequence of bytes doesn't involve directly calling the serialization API. In fact, calling the serialization API in most cases is not needed. For example, retrieving the whole contents of a packet as a sequence of bytes is as simple as follows:

```
packet->peekAt<BytesChunk>(byte(0), packet->getPacketLength()); // generic peek
packet->peekAllBytes(); // shorthand
```

This property of the API greatly simplifies code that serializes packets into trace files, such as PCAP. Finally, the new API allows testing the protocol implementations for proper emulation support. Configuring all network interfaces to send out packets (in place of the original packets) which contain a single `BytesChunk` only is easy to do. At the receiver modules, there's no need to change anything in the protocol implementations. The reason being that the packet API transparently handles the dual representation, and it converts the sequence of bytes to the requested chunk types as needed.

## 3.4 Handling Checksums

The old serializer classes used to compute and verify checksums on the fly. This caused some confusion, especially with the proper support of pseudo headers. With the new API, this is no longer the case. The new serializers are only responsible for transforming from one representation (sequence of bytes) to another (fields), and vice versa.

Computing and verifying checksums is up to the protocol implementations, and it is independent of the actual representation of the header. In general, protocols should have parameters to declare the checksum correct/incorrect or to actually compute and verify it. Of course, for emulation, one should enable computing and verifying checksums.