INET Framework Developer's Guide

Release 4.2

Nov 18, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is INET Framework

INET Framework is an open-source model library for the OMNeT++ simulation environment. It provides protocols, agents and other models for researchers and students working with communication networks. INET is especially useful when designing and validating new protocols, or exploring new or exotic scenarios.

INET supports a wide class of communication networks, including wired, wireless, mobile, ad hoc and sensor networks. It contains models for the Internet stack (TCP, UDP, IPv4, IPv6, OSPF, BGP, etc.), link layer protocols (Ethernet, PPP, IEEE 802.11, various sensor MAC protocols, etc), refined support for the wireless physical layer, MANET routing protocols, DiffServ, MPLS with LDP and RSVP-TE signalling, several application models, and many other protocols and components. It also provides support for node mobility, advanced visualization, network emulation and more.

Several other simulation frameworks take INET as a base, and extend it into specific directions, such as vehicular networks, overlay/peer-to-peer networks, or LTE.

1.2 Scope of this Manual

This manual is written for developers who intend to extend INET with new components, written in C++. This manual is accompanied by the INET Reference, which is generated from NED and MSG files using OMNeT++'s documentation generator, and the documentation of the underlying C++ classes, generated from the source files using Doxygen. A working knowledge of OMNeT++ and the C++ language is assumed.

CHAPTER

TWO

WORKING WITH PACKETS

2.1 Overview

The INET Packet API is designed to ease the implementation of communication protocols and applications by providing many useful C++ components. In the following sections, we introduce the Packet API in detail, and we shed light on many common API usages through examples.

Note: Code fragments in this chapter have been somewhat simplified for brevity. For example, some `const` modifiers and `const` casts have been omitted, setting fields have been omitted, and some algorithms have been simplified to ease understanding.

The representation of packets is essential for communication network simulation. Applications and communication protocols construct, deconstruct, encapsulate, fragment, aggregate, and manipulate packets in many ways. In order to ease the implementation of these behavioral patterns, INET provides a feature-rich general data structure, the `Packet` class.

The `Packet` data structure is capable of representing application packets, TCP segments, IP datagrams, Ethernet frames, IEEE 802.11 frames, and all kinds of digital data. It is designed to provide efficient storage, duplication, sharing, encapsulation, aggregation, fragmentation, serialization, and data representation selection. Additional functionality, such as support for enqueueing data for transmisson and buffering received data for reassembly and/or for reordering, is provided as separate C++ data structures on top of `Packet`.

2.2 Representing Data

The `Packet` data structure builds on top of another set of data structures called chunks. Chunks provide several alternatives to represent a piece of data.

INET provides the following built-in chunk C++ classes:

- `Chunk`, the base class for all chunk classes
- repeated byte or bit chunk (`ByteCountChunk`, `BitCountChunk`)
- raw bytes or bits chunk (`BytesChunk`, `BitsChunk`)
- ordered sequence of chunks (`SequenceChunk`)
- slice of another chunk designated by offset and length (`SliceChunk`)
- many protocol specific field based chunks (e.g. `Ipv4Header` subclass of `FieldsChunk`)

In addition, communication protocols and applications often define their own chunk types. User-defined chunks are normally defined in `msg` files as a subclass of `FieldsChunk`, which the OMNeT++ MSG compiler turns into C++ code. It is also possible to write a user defined chunk from scratch.

Chunks usually represent application data and protocol headers. The following examples demonstrate the construction of various chunks.

```
auto bitCountData = makeShared<BitCountChunk>(b(3), 0); // 3 zero bits
auto byteCountData = makeShared<ByteCountChunk>(B(10), '?'); // 10 '?' bytes
auto rawBitsData = makeShared<BitsChunk>();
rawBitsData->setBits({1, 0, 1}); // 3 raw bits
auto rawBytesData = makeShared<BytesChunk>(); // 10 raw bytes
rawBytesData->setBytes({243, 74, 19, 84, 81, 134, 216, 61, 4, 8});
auto fieldBasedHeader = makeShared<UdpHeader>(); // create new UDP header
fieldBasedHeader->setSrcPort(1000); // set some fields
```

In general, chunks must be constructed with a call to the `makeShared` function instead of the standard C++ `new` operator, because chunks are shared among packets using C++ shared pointers.

Packets most often contain several chunks, inserted by different protocols, as they are passed through the protocol layers. The most common way to represent packet contents is to form a compound chunk by concatenation.

```
auto sequence = makeShared<SequenceChunk>(); // create empty sequence
sequence->insertAtBack(makeShared<UdpHeader>()); // append UDP header
sequence->insertAtBack(makeShared<ByteCountChunk>(B(10), 0)); // 10 bytes
```

Protocols often need to slice data, for example to provide fragmentation, which is also directly supported by the chunk API.

```
auto udpHeader = makeShared<UdpHeader>(); // create 8 bytes UDP header
auto firstHalf = udpHeader->peek(B(0), B(4)); // first 4 bytes of header
auto secondHalf = udpHeader->peek(B(4), B(4)); // second 4 bytes of header
```

In order to avoid cluttered data representation due to slicing, the chunk API provides automatic merging for consecutive chunk slices.

```
auto sequence = makeShared<SequenceChunk>(); // create empty sequence
sequence->insertAtBack(firstHalf); // append first half
sequence->insertAtBack(secondHalf); // append second half
auto merged = sequence->peek(B(0), B(8)); // automatically merge slices
```

Alternative representations can be easily converted into one another using automatic serialization as a common ground.

```
auto raw = merged->peek<BytesChunk>(B(0), B(8)); // auto serialization
auto original = raw->peek<UdpHeader>(B(0), B(8)); // auto deserialization
```

The following MSG fragment is a more complete example which shows how a UDP header could be defined:

```
enum CrcMode
{
  CRC_DISABLED = 0; // CRC is not set, serializable
  CRC_DECLARED = 1; // CRC is correct without the value, not serializable
  CRC_COMPUTED = 2; // CRC is potentially incorrect, serializable
}

class UdpHeader extends FieldsChunk
{
  chunkLength = B(8); // UDP header length is always 8 bytes
  int sourcePort = -1; // source port field is undefined by default
  int destinationPort = -1; // destination port field is undefined by default
  B lengthField = B(-1); // length field is undefined by default
  uint16_t crc = 0; // checksum field is 0 by default
  CrcMode crcMode = CRC_DISABLED; // checksum mode is disabled by default
}
```

It's important to distinguish the two length related fields in the UdpHeader chunk. One is the length of the chunk itself (`chunkLength`), the other is the value in the length field of the header (`lengthField`).

2.3 Representing Packets

The `Packet` data structure uses a single chunk data structure to represent its contents. The contents may be as simple as raw bytes (`BytesChunk`), but most likely it will be the concatenation (`SequenceChunk`) of various protocol specific headers (e.g., `FieldsChunk` subclasses) and application data (e.g., `ByteCountChunk`).

Packets can be created by both applications and communication protocols. As packets are passed down through the protocol layers at the sender node, new protocol specific headers and trailers are inserted during processing.

```
auto emptyPacket = new Packet("ACK"); // create empty packet
auto data = makeShared<ByteCountChunk>(B(1000));
auto dataPacket = new Packet("DATA", data); // create new packet with data
auto moreData = makeShared<ByteCountChunk>(B(1000));
dataPacket->insertAtBack(moreData); // insert more data at the end
auto udpHeader = makeShared<UdpHeader>(); // create new UDP header
dataPacket->insertAtFront(udpHeader); // insert header into packet
```

In order to facilitate packet processing by communication protocols at the receiver node, `Packet` maintains two offsets into the packet data that divide the data into three regions: front popped part, data part, and back popped part. During packet processing, as the packet is passed through the protocol layers, headers and trailers are popped from the beginning and from the end of the packet, moving the corresponding offsets. This effectively reduces the remaining unprocessed part called the data part, but it doesn't affect the data stored in the packet.

```
packet->popAtFront<MacHeader>(); // pop specific header from packet
packet->popAtBack<MacTrailer>(); // pop specific trailer from packet
auto data = packet->peekData(); // peek remaining data in packet
```

2.4 Representing Signals

Protocols and applications use the `Packet` data structure to represent digital data during the processing within the network node. In contrast, the wireless transmission medium uses a different data structure called `Signal` to represent the physical phenomena used to transmit packets.

```
auto signal = new Signal(transmission);
signal->setDuration(duration);
signal->encapsulate(packet);
```

Signals always encapsulate a packet and also contain a description of the analog domain representation. The most important physical properties of a signal are the signal duration and the signal power.

2.5 Representing Transmission Errors

An essential part of communication network simulation is the understanding of protocol behavior in the presence of errors. The Packet API provides several alternatives for representing errors. The alternatives range from simple, but computationally cheap, to accurate, but computationally expensive solutions.

- mark erroneous packets (simple)
- mark erroneous chunks (good compromise)
- change bits in raw chunks (accurate)

The first example shows how to represent transmission erros on the packet level. A packet is marked as erroneous based on its length and the associated bit error rate. This representation doesn't give too much chance for a protocol to do anything else than discard an erroneous packet.

```
Packet *ErrorModel::corruptPacket(Packet *packet, double ber)
{
  auto length = packet->getTotalLength();
  auto hasErrors = hasProbabilisticError(length, ber); // decide randomly
  auto corruptedPacket = packet->dup(); // cheap operation
  corruptedPacket->setBitError(hasErrors); // set bit error flag
  return corruptedPacket;
}
```

The second example shows how to represent transmission errors on the chunk level. Similarly to the previous example, a chunk is also marked as erroneous based on its length and the associated bit error rate. This representation allows a protocol to discard only certain parts of the packet. For example, an aggregated packet may be partially discarded and processed.

```
Packet *ErrorModel::corruptChunks(Packet *packet, double ber)
{
  b offset = b(0); // start from the beginning
  auto corruptedPacket = new Packet("Corrupt"); // create new packet
  while (auto chunk = packet->peekAt(offset)->dupShared()) { // for each chunk
    auto length = chunk->getChunkLength();
    auto hasErrors = hasProbabilisticError(length, ber); // decide randomly
    if (hasErrors) // if erroneous
      chunk->markIncorrect(); // set incorrect bit
    corruptedPacket->insertAtBack(chunk); // append chunk to corrupt packet
    offset += chunk->getChunkLength(); // increment offset with chunk length
  }
  return corruptedPacket;
}
```

The last example shows how to actually represent transmission errors on the byte level. In contrast with the previous examples, this time the actual data of the packet is modified. This allows a protocol to discard or correct any part based on checksums.

```
Packet *ErrorModel::corruptBytes(Packet *packet, double ber)
{
  vector<uint8_t> corruptedBytes; // bytes of corrupted packet
  auto data = packet->peekAllAsBytes(); // data of original packet
  for (auto byte : data->getBytes()) { // for each original byte do
    if (hasProbabilisticError(B(1), ber)) // if erroneous
      byte = ~byte; // invert byte (simplified corruption)
    corruptedBytes.push_back(byte); // store byte in corrupted data
  }
  auto corruptedData = makeShared<BytesChunk>(); // create new data
  corruptedData->setBytes(corruptedBytes); // store corrupted bits
  return new Packet("Corrupt", corruptedData); // create new packet
}
```

The physical layer models support the above mentioned different error representations via configurable parameters. Higher layer protocols detect errors by chechking the error bit on packets and chunks, and by standard CRC mechanisms.

2.6 Packet Tagging

Communication between protocols inside network nodes often require passing around meta information along with packets. To this end, packets are also capable of carrying various meta information called tags. Tags can either be attached to the whole packet or to a specific region. The former are called packet tags, the latter are called region tags.

The most important packet tag example is the one specifying the outermost protocol of the packet, which cannot be unambigously identified just by looking at the raw data. Other notable examples are: MAC address request, outgoing interface request, transmission power request, receive strength indication, incoming interface indication.

```
void Ipv4::sendDown(Packet *packet, Ipv4Address nextHopAddr, int interfaceId)
{
  auto macAddressReq = packet->addTag<MacAddressReq>(); // add new tag for MAC
  macAddressReq->setSrcAddress(selfAddress); // source is our MAC address
  auto nextHopMacAddress = resolveMacAddress(nextHopAddr); // simplified ARP
  macAddressReq->setDestAddress(nextHopMacAddress); // destination is next hop
  auto interfaceReq = packet->addTag<InterfaceReq>(); // add tag for dispatch
  interfaceReq->setInterfaceId(interfaceId); // set designated interface
  auto packetProtocolTag = packet->addTagIfAbsent<PacketProtocolTag>();
  packetProtocolTag->setProtocol(&Protocol::ipv4); // set protocol of packet
  send(packet, "out"); // send to MAC protocol module of designated interface
}
```

Tags are very simple C++ classes usually generated by the OMNeT++ MSG compiler. Tags come in three flavors:

- *requests* carry information from higher layer to lower layer (e.g. `MacAddressReq`).
- *indications* carry information from lower layer to higher layer (e.g. `InterfaceInd`).
- *plain tags* contain some meta information (e.g. `PacketProtocolTag`).
- *base classes* must not be attached to packets (e.g. `TagBase`).

```
class MacAddressReq extends TagBase
{
    MacAddress srcAddress;  // may be unspecified
    MacAddress destAddress; // always specified
}
```

2.7 Region Tagging

In order to collect some statistics, it is required to attach meta information to arbitrary regions of packets. For example, computing the end-to-end delay in a TCP stream requires to tag regions at the sender with the timestamp when they were created. Then the receiver computes the end-to-end delay for every region as the data arrives.

```
void ClientApp::send()
{
  auto data = makeShared<ByteCountChunk>(); // create new data chunk
  auto creationTimeTag = data->addTag<CreationTimeTag>(); // add new tag
  creationTimeTag->setCreationTime(simTime()); // store current time
  auto packet = new Packet("Data", data); // create new packet
  socket.send(packet); // send packet using TCP socket
}
```

In a TCP stream, the data can be arbitrarily split, reordered, and merged in the underlying network. The packet data representation takes care of maintaining the attached region tags as if they were individually attached to bits. In order to avoid cluttered data representation due to the above, the tag API provides automatic merging for similar consecutive tag regions.

```
void ServerApp::receive(Packet *packet)
{
  auto data = packet->peekData(); // get all data from the packet
  auto regions = data->getAllTags<CreationTimeTag>(); // get all tag regions
  for (auto& region : regions) { // for each region do
    auto creationTime = region.getTag()->getCreationTime(); // original time
    auto delay = simTime() - creationTime; // compute delay
    cout << region.getOffset() << region.getLength() << delay; // use data
  }
}
```

The above loop may run exactly once for the whole data, or it may run several times depending on how the data is provided at the sender and how the underlying network works.

2.8 Dissecting Packets

Understanding what's inside a packet is a very important and often used functionality. Simply using the representation may be insufficient, because the `Packet` may be represented with a `BytesChunk`, for exmple. The Packet API provides a `PacketDissector` class which analyzes a packet solely based on the assigned packet protocol and the actual data it contains.

The analysis is done according to the protocol logic as opposed to the actual representation of the data. The `PacketDissector` works similarly to a parser. Basically, it walks through each part (such as protocol headers) of a packet in order. For each part, it determines the corresponding protocol and the most specific representation for that protocol.

The `PacketDissector` class relies on small registered protocol-specific dissector classes (e.g. `Ipv4ProtocolDissector`) subclassing the required `ProtocolDissector` base class. Implementors are expected to use the `PacketDissector::ICallback` interface to notify the parser about the packet structure.

```
void startProtocolDataUnit(Protocol *protocol);
void endProtocolDataUnit(Protocol *protocol);
void markIncorrect();
void visitChunk(Ptr<Chunk>& chunk, Protocol *protocol);
void dissectPacket(Packet *packet, Protocol *protocol);
```

In order to use the `PacketDissector`, the user is expected to implement a `PacketDissector::ICallback` interface. The callback interface will be notified for each part of the packet as the `PacketDissector` goes through it.

```
auto& registry = ProtocolDissectorRegistry::globalRegistry;
PacketDissector dissector(registry, callback);
auto packetProtocolTag = packet->findTag<PacketProtocolTag>();
auto protocol = packetProtocolTag->getProtocol();
dissector.dissectPacket(packet, protocol);
```

2.9 Filtering Packets

Filtering packets based on the actual data they contain is another widely used and very important feature. With the help of the packet dissector, it is very simple to create arbitrary custom packet filters. Packet filters are generally used for recording packets and visualizing various packet related information.

In order to simplify filtering, the Packet API provides a generic expression based packet filter which is implemented in the `PacketFilter` class. The expression syntax is the same as other OMNeT++ expressions, and the data filter is matched against individual chunks of the packet as found by the packet dissector.

For example, the packet filter expression "ping*" matches all packets having the name prefix 'ping', and the packet chunk filter expression "inet::Ipv4Header and srcAddress(10.0.0.*)" matches all packets that contain an IPv4 header with a '10.0.0' source address prefix.

```
PacketFilter filter; // patterns for the whole packet and for the data
filter.setPattern("ping*", "Ipv4Header and sourceAddress(10.0.0.*)");
filter.matches(packet); // returns boolean value
```

2.10 Printing Packets

During model development, packets often need to be displayed in a human readable form. The Packet API provides a `PacketPrinter` class which is capable of forming a human readable string representation of `Packet`'s. The `PacketPrinter` class relies on small registered protocol-specific printer classes (e.g. `Ipv4ProtocolPrinter` subclassing the required `ProtocolPrinter` base class.

The packet printer is automatically used by the OMNeT++ runtime user interface to display packets in the packet log window. The packet printer contributes several log window columns into the user interface: 'Source', 'Destination', 'Protocol', 'Length', and 'Info'. These columns display packet data similarly to the well-known Wireshark protocol analyzer.

```
PacketPrinter printer; // turns packets into human readable strings
printer.printPacket(std::cout, packet); // print to standard output
```

The `PacketPrinter` provides a few other functions which have additional options to control the details of the resulting human readable form.

2.11 Recording PCAP

Exporting the packets from a simulation into a PCAP file allows further processing with 3rd party tools. The Packet API provides a `PcapDump` class for creating PCAP files. Packet filtering can be used to reduce the file size and increase performance.

```
PcapDump dump;
dump.openPcap("out.pcap", 65535, 0); // maximum length and PCAP type
dump.writePacket(simTime(), packet); // record with current time
```

2.12 Encapsulating Packets

Many communication protocols work with simple packet encapsulation. They encapsulate packets with their own protocol specific headers and trailers at the sender node, and they decapsulate packets at the reciver node. The headers and trailers carry the information that is required to provide the protocol specific service.

For example, when sending a packet, the Ethernet protocol encapsulates an IP datagram by prepending the packet with an Ethernet header, and also by appending the packet with an optional padding and an Ethernet FCS. The following example shows how a MAC protocol could encapsulate a packet:

```
void Mac::encapsulate(Packet *packet)
{
  auto header = makeShared<MacHeader>(); // create new header
  header->setChunkLength(B(8)); // set chunk length to 8 bytes
  header->setLengthField(packet->getDataLength()); // set length field
  header->setTransmitterAddress(selfAddress); // set other header fields
  packet->insertAtFront(header); // insert header into packet
  auto trailer = makeShared<MacTrailer>(); // create new trailer
  trailer->setChunkLength(B(4)); // set chunk length to 4 bytes
  trailer->setFcsMode(FCS_MODE_DECLARED); // set trailer fields
  packet->insertAtBack(trailer); // insert trailer into packet
}
```

When receiving a packet, the Ethernet protocol removes an Ethernet header and an Ethernet FCS from the received Ethernet frame, and passes the resulting IP datagram along. The following example shows how a MAC protocol could decapsulate a packet:

```
void Mac::decapsulate(Packet *packet)
{
  auto header = packet->popAtFront<MacHeader>(); // pop header from packet
  auto lengthField = header->getLengthField();
  cout << header->getChunkLength() << endl; // print chunk length
  cout << lengthField << endl; // print header length field
  cout << header->getReceiverAddress() << endl; // print other header fields
  auto trailer = packet->popAtBack<MacTrailer>(); // pop trailer from packet
  cout << trailer->getFcsMode() << endl; // print trailer fields
  assert(packet->getDataLength() == lengthField); // if the packet is correct
}
```

Although the `popAtFront` and `popAtBack` functions change the remaining unprocessed part of the packet, they don't have effect on the actual packet data. That is when the packet reaches high level protocol, it still contains all the received data.

2.13 Fragmenting Packets

Communication protocols often provide fragmentation to overcome various physical limits (e.g. length limit, error rate). They split packets into smaller pieces at the sender node, which send them one-by-one. They form the original packet at the receiver node by combining the received fragments.

For example, the IEEE 802.11 protocol fragments packets to overcome the increasing probability of packet loss of large packets. The following example shows how a MAC protocol could fragment a packet:

```
vector<Packet *> *Mac::fragment(Packet *packet, vector<b>& sizes)
{
  auto offset = b(0); // start from the packet's beginning
  auto fragments = new vector<Packet *>(); // result collection
  for (auto size : sizes) { // for each received size do
    auto fragment = new Packet("Fragment"); // header + data part + trailer
    auto header = makeShared<MacHeader>(); // create new header
    header->setFragmentOffset(offset); // set fragment offset for reassembly
    fragment->insertAtFront(header); // insert header into fragment
    auto data = packet->peekAt(offset, size); // get data part from packet
    fragment->insertAtBack(data); // insert data part into fragment
    auto trailer = makeShared<MacTrailer>(); // create new trailer
    fragment->insertAtBack(trailer); // insert trailer into fragment
    fragments->push_back(fragment); // collect fragment into result
    offset += size; // increment offset with size of data part
  }
  return fragments;
}
```

When receiving fragments, protocols need to collect the coherent fragments of the same packet until all fragments becomes available. The following example shows how a MAC protocol could form the original packet from a set of coherent fragments:

```
Packet *Mac::defragment(vector<Packet *>& fragments)
{
  auto packet = new Packet("Original"); // create new concatenated packet
  for (auto fragment : fragments) {
    fragment->popAtFront<MacHeader>(); // pop header from fragment
    fragment->popAtBack<MacTrailer>(); // pop trailer from fragment
    packet->insertAtBack(fragment->peekData()); // concatenate fragment data
  }
  return packet;
}
```

2.14 Aggregating Packets

Communication protocols often provide aggregation to better utilize the communication channel by reducing protocol overhead. They wait for several packets to arrive at the sender node, then they form a large aggregated packet which is in turn sent at once. At the receiver node the aggregated packet is split into the original packets, and they are passed along.

For example, the IEEE 802.11 protocol aggregates packets for better channel utilization at both MSDU and MPDU levels. The following example shows a version of how a MAC protocol could create an aggregate packet:

```
Packet *Mac::aggregate(vector<Packet *>& packets)
{
  auto aggregate = new Packet("Aggregate"); // create concatenated packet
  for (auto packet : packets) { // for each received packet do
    auto header = makeShared<SubHeader>(); // create new subheader
    header->setLengthField(packet->getDataLength()); // set subframe length
    aggregate->insertAtBack(header); // insert subheader into aggregate
    auto data = packet->peekData(); // get packet data
    aggregate->insertAtBack(data); // insert data into aggregate
  }
  auto header = makeShared<MacHeader>(); // create new header
  header->setAggregate(true); // set aggregate flag
  aggregate->insertAtFront(header); // insert header into aggregate
  auto trailer = makeShared<MacTrailer>(); // create new trailer
  aggregate->insertAtBack(trailer); // insert trailer into aggregate
  return aggregate;
}
```

The following example shows a version of how a MAC protocol could disaggregate a packet:

```
vector<Packet *> *Mac::disaggregate(Packet *aggregate)
{
  aggregate->popAtFront<MacHeader>(); // pop header from packet
  aggregate->popAtBack<MacTrailer>(); // pop trailer from packet
  vector<Packet *> *packets = new vector<Packet *>(); // result collection
  b offset = aggregate->getFrontOffset(); // start after header
  while (offset != aggregate->getBackOffset()) { // up to trailer
    auto header = aggregate->peekAt<SubHeader>(offset); // peek sub header
    offset += header->getChunkLength(); // increment with header length
    auto size = header->getLengthField(); // get length field from header
    auto data = aggregate->peekAt(offset, size); // peek following data part
    auto packet = new Packet("Original"); // create new packet
    packet->insertAtBack(data); // insert data into packet
    packets->push_back(packet); // collect packet into result
    offset += size; // increment offset with data size
  }
  return packets;
}
```

2.15 Serializing Packets

In real communication systems packets are usually stored as a sequence of bytes directly in network byte order. In contrast, INET usually stores packets in small field based C++ classes (generated by the OMNeT++ MSG compiler) to ease debugging. In order to calculate checksums or to communicate with real hardware, all protocol specific parts must be serializable to a sequence of bytes.

The protocol header serializers are separate classes from the actual protocol headers. They must be registered in the `ChunkSerializerRegistry` in order to be used. The following example shows how a MAC protocol header could be serialized to a sequence of bytes:

```
void MacHeaderSerializer::serialize
  (MemoryOutputStream& stream, Ptr<Chunk>& chunk)
{
  auto header = staticPtrCast<MacHeader>(chunk);
  stream.writeUint16Be(header->getType()); // unsigned 16 bits, big endian
  stream.writeMacAddress(header->getTransmitterAddress());
  stream.writeMacAddress(header->getReceiverAddress());
}
```

Deserialization is somewhat more complicated than serialization, because it must be prepared to handle incomplete or even incorrect data due to errors introduced by the network. The following example shows how a MAC protocol header could be deserialized from a sequence of bytes:

```
Ptr<Chunk> MacHeaderSerializer::deserialize(MemoryInputStream& stream)
{
  auto header = makeShared<MacHeader>(); // create new header
  header->setType(stream.readUint16Be()); // unsigned 16 bits, big endian
  header->setTransmitterAddress(stream.readMacAddress());
  header->setReceiverAddress(stream.readMacAddress());
  return header;
}
```

2.16 Emulation Support

In order to be able to communicate with real hardware, packets must be converted to and from a sequence of bytes. The reason is that the programming interface of operating systems and external libraries work with sending and receiving raw data.

All protocol headers and data chunks which are present in a packet must have a registered serializer to be able to create the raw sequence of bytes. Protocol modules must also be configured to either disable or compute checksums, because serializers cannot carry out the checksum calculation.

The following example shows how a packet could be converted to a sequence of bytes to send through an external interface:

```
vector<uint8_t>& ExternalInterface::prepareToSend(Packet *packet)
{
  auto data = packet->peekAllAsBytes(); // convert to a sequence of bytes
  return data->getBytes(); // actual bytes to send
}
```

The following example shows how a packet could be converted from a sequence of bytes when receiving from an external interface:

```
Packet *ExternalInterface::prepareToReceive(vector<uint8_t>& bytes)
{
  auto data = makeShared<BytesChunk>(bytes); // create chunk with bytes
  return new Packet("Emulation", data); // create packet with data
}
```

In INET, all protocols automatically support hardware emulation due to the dual representation of packets. The above example creates a packet which contains a single chunk with a sequence of bytes. As the packet is passed through the protocols, they can interpret the data (e.g. by calling `peekAtFront`) as they see fit. The Packet API always provides the requested representation, either because it's already available in the packet, or because it gets automatically deserialized.

2.17 Queueing Packets

Some protocols store packet data temporarily at the sender node before actual processing can occur. For example, the TCP protocol must store the outgoing data received from the application in order to be able to provide transmission flow control.

The following example shows how a transport protocol could store the received data temporarily until the data is actually used:

```
class TransportSendQueue
{
  ChunkQueue queue; // stores application data
  B sequenceNumber; // position in stream

  void enqueueApplicationData(Packet *packet);
  Packet *createSegment(b length);
};

void TransportSendQueue::enqueueApplicationData(Packet *packet)
{
  queue.push(packet->peekData()); // store received data
}

Packet *TransportSendQueue::createSegment(b maxLength)
{
  auto packet = new Packet("Segment"); // create new segment
  auto header = makeShared<TransportHeader>(); // create new header
  header->setSequenceNumber(sequenceNumber); // store sequence number for␣
↪reordering
  packet->insertAtFront(header); // insert header into segment
  if (queue.getLength() < maxLength)
    maxLength = queue.getLength(); // reduce length if necessary
  auto data = queue.pop(maxLength); // pop requested amount of data
  packet->insertAtBack(data); // insert data into segment
  sequenceNumber += data->getChunkLength(); // increase sequence number
  return packet;
}
```

The `ChunkQueue` class acts similarly to a binary FIFO queue except it works with chunks. Similarly to the `Packet` it also automatically merge consecutive data and selects the most appropriate representation.

2.18 Buffering Packets

Protocols at the receiver node often need to buffer incoming packet data until the actual processing can occur. For example, packets may arrive out of order, and the data they contain must be reassembled or reordered before it can be passed along.

INET provides a few special purpose C++ classes to support data buffering:

- `ChunkBuffer` provides automatic merging for large data chunks from out of order smaller data chunks.
- `ReassemblyBuffer` provides reassembling for out of order data according to an expected length.
- `ReorderBuffer` provides reordering for out of order data into a continuous data stream from an expected offset.

All buffers deal with only the data, represented by chunks, instead of packets. They automatically merge consecutive data and select the most appropriate representation. Protocols using these buffers automatically support all data representation provided by INET, and any combination thereof. For example, `ByteCountChunk`, `BytesChunk`, `FieldsChunk`, and `SliceChunk` can be freely mixed in the same buffer.

2.19 Reassembling Packets

Some protocols may use an unreliable service to transfer a large piece of data over the network. The unreliable service requires the receiver node to be prepared for receiving parts out of order and potentially duplicated.

For example, the IP protocol must store incoming fragments at the receiver node, because it must wait until the datagram becomes complete, before it can be passed along. The IP protocol must also be prepared for receiving the individual fragments out of order and potentially duplicated.

The following example shows how a network protocol could store and reassemble the data of the incoming packets into a whole packet:

```
class NetworkProtocolDefragmentation
{
  ReassemblyBuffer buffer; // stores received data

  void processDatagram(Packet *packet); // processes incoming packes
  Packet *getReassembledDatagram(); // reassembles the original packet
};

void NetworkProtocolDefragmentation::processDatagram(Packet *packet)
{
  auto header = packet->popAtFront<NetworkProtocolHeader>(); // remove header
  auto fragmentOffset = header->getFragmentOffset(); // determine offset
  auto data = packet->peekData(); // get data from packet
  buffer.replace(fragmentOffset, data); // overwrite data in buffer
}

Packet *NetworkProtocolDefragmentation::getReassembledDatagram()
{
  if (!buffer.isComplete()) // if reassembly isn't complete
    return nullptr; // there's nothing to return
  auto data = buffer.getReassembledData(); // complete reassembly
  return new Packet("Datagram", data); // create new packet
}
```

The `ReassemblyBuffer` supports replacing the stored data at a given offset, and it also provides the complete reassembled data with the expected length if available.

2.20 Reordering Packets

Some protocols may use an unreliable service to transfer a long data stream over the network. The unreliable service requires the sender node to resend unacknowledged parts, and it also requires the receiver node to be prepared for receiving parts out of order and potentially duplicated.

For example, the TCP protocol must buffer the incoming data at the receiver node, because the TCP segments may arrive out of order and potentially duplicated or overlapping, and TCP is required to provide the data to the application in the correct order and only once.

The following example shows how a transport protocol could store and reorder the data of incoming packets, which may arrive out of order, and also how such a protocol could pass along only the available data in the correct order:

```
class TransportReceiveQueue
{
  ReorderBuffer buffer; // stores receive data
  B sequenceNumber;

  void processSegment(Packet *packet);
  Packet *getAvailableData();
```

(continues on next page)

(continued from previous page)

```
};

void TransportReceiveQueue::processSegment(Packet *packet)
{
  auto header = packet->popAtFront<TransportHeader>(); // pop transport header
  auto sequenceNumber = header->getSequenceNumber();
  auto data = packet->peekData(); // get all packet data
  buffer.replace(sequenceNumber, data); // overwrite data in buffer
}

Packet *TransportReceiveQueue::getAvailableData()
{
  if (buffer.getAvailableDataLength() == b(0)) // if no data available
    return nullptr;
  auto data = buffer.popAvailableData(); // remove all available data
  return new Packet("Data", data);
}
```

The `ReorderBuffer` supports replacing the stored data at a given offset, and it provides the available data from the expected offset if any.

2.21 Dispatching Packets

Protocols also communicate with each other inside the network node by sending packets, requests, and confirmations. INET is very flexible in terms of how protocols can be connected to each other. Protocols can be connected directly, or they can be connected through one or more MessageDispatcher modules.

This flexiblity allows creating very simple network nodes where the protocol stack is a chain. But it also allows creating more complicated network nodes where protocols are grouped into protocol layers to provide many-to-one and many-to-many relationships. It's also possible to use dispatcher modules hierarchically inside compound modules, or to connect all protocols to a single central dispatcher module.

The `DispatchProtocolTag` must be attached to a packet, request or confirmation to allow the MessageDispatcher to direct the message to the inteded recipient. The following example shows how a MAC protocol could send up a packet to the designated protocol without actually knowing where that protocol is in the network node:

```
void Mac::sendUp(Packet *packet)
{
  auto req = packet->addTagIfAbsent<DispatchProtocolReq>();
  req->setProtocol(&Protocol::ipv4); // set destination protocol
  req->setServicePrimitive(SP_INDICATION); // determine receiving gate
  send(packet, "upperLayerOut");
}
```

The MessageDispatcher finds the designated protocol module and its gate based on the `registerProtocol` calls it has received during the initialization of all connected protocol modules.

CHAPTER

THREE

USING SOCKETS

3.1 Overview

The INET Socket API provides special C++ abstractions on top of the standard OMNeT++ message passing interface for several communication protocols.

Sockets are most often used by applications and routing protocols to acccess the corresponding protocol services. Sockets are capable of communicating with the underlying protocol in a bidirectional way. They can assemble and send service requests and packets, and they can also receive service indications and packets.

Applications can simply call the socket class member functions (e.g. `bind()`, `connect()`, `send()`, `close()`) to create and configure sockets, and to send and receive packets. They may also use several different sockets simulatenously.

The following sections first introduce the shared functionality of sockets, and then list all INET sockets in detail, mostly by shedding light on many common usages through examples.

Note: Code fragments in this chapter have been somewhat simplified for brevity. For example, some `virtual` modifiers and `override` qualifiers have been omitted, and some algorithms have been simplified to ease understanding.

3.1.1 Socket Interfaces

Although sockets are always implemented as protocol specific C++ classes, INET also provides C++ socket interfaces. These interfaces allow writing general C++ code which can handle many different kinds of sockets all at once.

For example, the `ISocket` interface is implemented by all sockets, and the `INetworkSocket` interface is implemented by all network protocol sockets.

3.1.2 Identifying Sockets

All sockets have a socket identifier which is unique within the network node. It is automatically assigned to the sockets when they are created. The identifier can accessed with `getSocketId()` throughout the lifetime of the socket.

The socket identifier is also passed along in `SocketReq` and `SocketInd` packet tags. These tags allow applications and protocols to identify the socket to which `Packet`'s, service `Request`'s, and service `Indication`'s belong.

3.1.3 Configuring Sockets

Since all sockets work with message passing under the hoods, they must be configured prior to use. In order to send packets and service requests on the correct gate towards the underlying communication protocol, the output gate must be configured:

```
socket.setOutputGate(gate("socketOut")); // configure socket output gate
socket.setCallback(this); // set callback interface for message processing
```

In contrast, incoming messages such as service indications from the underlying communication protocol can be received on any application gate.

To ease application development, all sockets support storing a user specified data object pointer. The pointer is accessible with the `setUserData()`, `getUserData()` member functions.

Another mandatory configuration for all sockets is setting the socket callback interface. The callback interface is covered in more detail in the following section.

Other socket specific configuration options are also available, these are discussed in the section of the corresponding socket.

3.1.4 Callback Interfaces

To ease centralized message processing, all sockets provide a callback interface which must be implemented by applications. The callback interface is usually called `ICallback`, and it's defined as an inner class of the socket it belongs to. These interfaces often contain some generic notification methods along with several socket specific methods.

For example, the most common callback method is the one which processes incoming packets:

```
class ICallback // usually the inner class of the socket
{
    void socketDataArrived(ISocket *socket, Packet *packet);
};
```

3.1.5 Processing Messages

In general, sockets can process all incoming messages which were sent by the underlying protocol. The received messages must be processed by the socket where they belong to.

For example, an application can simply go through each knonwn socket in any order, and decide which one should process the received message as follows:

```
if (socket.belongsToSocket(message)) // match message and socket
    socket.processMessage(message); // invoke callback interface
```

Sockets usually deconstruct the received messages and update their state accordingly if necessary. They also automatically dispatch received packets and service indications for further processing to the appropriate functions in the corresponding `ICallback` interface.

3.1.6 Sending Data

All sockets provide one or more `send()` functions which send packets using the current configuration of the socket. The actual means of packet delivery depends on the underlying communication protocol, but in general the state of the socket is expected to affect it.

For example, after the socket is properly configured, the application can start sending packets without attaching any tags, because the socket takes care of the necessary technical details:

```
socket.send(packet); // by means of the underlying communication protocol
```

3.1.7 Receiving Data

For example, the application may directly implement the `ICallback` interface of the socket and print the received data as follows:

```
class App : public cSimpleModule, public ICallback
{
    void socketDataArrived(ISocket *socket, Packet *packet);
};

void App::socketDataArrived(ISocket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

3.1.8 Closing Sockets

Sockets must be closed before deleting them. Closing a socket allows the underlying communication protocol to release allocated resources. These resources are often allocated on the local network node, the remote nework node, or potentially somewhere else in the network.

For example, a socket for a connection oriented protocol must be closed to release the allocated resources at the peer:

```
socket.close(); // release allocated local and remote network resources
```

3.1.9 Using Multiple Sockets

If the application needs to manage a large number of sockets, for example in a server application which handles multiple incoming connections, the generic `SocketMap` class may be useful. This class can manage all kinds of sockets which implement the `ISocket` interface simultaneously.

For example, processing an incoming packet or service indication can be done as follows:

```
auto socket = socketMap.findSocketFor(message); // lookup socket to process
socket->processMessage(message); // dispatch message to callback interface
```

In order for the `SocketMap` to operate properly, sockets must be added to and removed from it using the `addSocket()` and `removeSocket()` methods respectively.

3.2 UDP Socket

The `UdpSocket` class provides an easy to use C++ interface to send and receive UDP datagrams. The underlying UDP protocol is implemented in the Udp module.

3.2.1 Callback Interface

Processing packets and indications which are received from the Udp module is pretty simple. The incoming message must be processed by the socket where it belongs as shown in the general section.

The `UdpSocket` deconstructs the message and uses the `UdpSocket::ICallback` interface to notify the application about received data and error indications:

```
class ICallback // inner class of UdpSocket
{
    void socketDataArrived(UdpSocket *socket, Packet *packet);
    void socketErrorArrived(UdpSocket *socket, Indication *indication);
};
```

3.2.2 Configuring Sockets

For receiving UDP datagrams on a socket, it must be bound to an address and a port. Both the address and port is optional. If the address is unspecified, than all UDP datagrams with any destination address are received. If the port is -1, then an unused port is selected automatically by the Udp module. The address and port pair must be unique within the same network node.

Here is how to bind to a specific local address and port to receive UDP datagrams:

```
socket.bind(Ipv4Address("10.0.0.42"), 42); // local address/port
```

For only receiving UDP datagrams from a specific remote address/port, the socket can be connected to the desired remote address/port:

```
socket.connect(Ipv4Address("10.0.0.42"), 42); // remote address/port
```

There are several other socket options (e.g. receiving broadcasts, managing multicast groups, setting type of service) which can also be configured using the `UdpSocket` class:

```
socket.setTimeToLive(16); // change default TTL
socket.setBroadcast(true); // receive all broadcasts
socket.joinMulticastGroup(Ipv4Address("224.0.0.9")); // receive multicasts
```

3.2.3 Sending Data

After the socket has been configured, applications can send datagrams to a remote address and port via a simple function call:

```
socket.sendTo(packet42, Ipv4Address("10.0.0.42"), 42); // remote address/port
```

If the application wants to send several datagrams, it can optionally connect to the destination.

The UDP protocol is in fact connectionless, so when the Udp module receives the connect request, it simply remembers the remote address and port, and use it as default destination for later sends.

```
socket.connect(Ipv4Address("10.0.0.42"), 42); // remote address/port
socket.send(packet1); // send packets via connected socket
// ...
socket.send(packet42);
```

The application can call connect several times on the same socket.

3.2.4 Receiving Data

For example, the application may directly implement the `UdpSocket::ICallback` interface and print the received data as follows:

```
class UdpApp : public cSimpleModule, public UdpSocket::ICallback
{
    void socketDataArrived(UdpSocket *socket, Packet *packet);
};
```

(continues on next page)

(continued from previous page)

```
void UdpApp::socketDataArrived(UdpSocket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

3.3 TCP Socket

The `TcpSocket` class provides an easy to use C++ interface to manage TCP connections, and to send and receive data. The underlying TCP protocol is implemented in the Tcp, TcpLwip, and TcpNsc modules.

3.3.1 Callback Interface

Messages received from the various Tcp modules can be processed by the `TcpSocket` where they belong to. The `TcpSocket` deconstructs the message and uses the `TcpSocket::ICallback` interface to notify the application about the received data or service indication:

```
class ICallback // inner class of TcpSocket
{
    void socketDataArrived(TcpSocket* socket, Packet *packet, bool urgent);
    void socketAvailable(TcpSocket *socket, TcpAvailableInfo *info);
    void socketEstablished(TcpSocket *socket);
    // ...
    void socketClosed(TcpSocket *socket);
    void socketFailure(TcpSocket *socket, int code);
};
```

3.3.2 Configuring Connections

The Tcp module supports several TCP different congestion algorithms, which can also be configured using the `TcpSocket`:

```
socket.setTCPAlgorithmClass("TcpReno");
```

Upon setting the individual parameters, the socket immediately sends sevice requests to the underlying Tcp protocol module.

3.3.3 Setting up Connections

Since TCP is a connection oriented protocol, a connection must be established before applications can exchange data. On the one side, the application listens at a local address and port for incoming TCP connections:

```
socket.bind(Ipv4Address("10.0.0.42"), 42); // local address/port
socket.listen(); // start listening for incoming connections
```

On the other side, the application connects to a remote address and port to establish a new connection:

```
socket.connect(Ipv4Address("10.0.0.42"), 42); // remote address/port
```

3.3.4 Accepting Connections

The Tcp module automatically notifies the `TcpSocket` about incoming connections. The socket in turn notifies the application using the `ICallback::socketAvailable` method of the callback interface. Finally, incoming TCP connections must be accepted by the application before they can be used:

```
class TcpServerApp : public cSimpleModule, public TcpSocket::ICallback
{
    TcpSocket serverSocket; // server socket listening for connections
    SocketMap socketMap; // container for all accepted connections

    void socketAvailable(TcpSocket *socket, TcpAvailableInfo *info);
};

void TcpServerApp::socketAvailable(TcpSocket *socket, TcpAvailableInfo *info)
{
    auto newSocket = new TcpSocket(info); // create socket using received info
    // ...
    socketMap.addSocket(newSocket); // store accepted connection
    serverSocket.accept(info->getNewSocketId()); // notify Tcp module
}
```

After the connection is accepted, the Tcp module notifies the application about the socket being established and ready to be used.

3.3.5 Sending Data

After the connection has been established, applications can send data to the remote application via a simple function call:

```
socket.send(packet1);
// ...
socket.send(packet42);
```

Packet data is enqueued by the local Tcp module and transmitted over time according to the protocol logic.

3.3.6 Receiving Data

Receiving data is as simple as implementing the corresponding method of the `TcpSocket::ICallback` interface. One caveat is that packet data may arrive in different chunk sizes (but the same order) than they were sent due to the nature of TCP protocol.

For example, the application may directly implement the `TcpSocket::ICallback` interface and print the received data as follows:

```
class TcpApp : public cSimpleModule, public TcpSocket::ICallback
{
    void socketDataArrived(TcpSocket *socket, Packet *packet, bool urgent);
};

void TcpApp::socketDataArrived(TcpSocket *socket, Packet *packet, bool urgent)
{
    EV << packet->peekData() << endl;
}
```

3.4 SCTP Socket

The `SctpSocket` class provides an easy to use C++ interface to manage SCTP connections, and to send and receive data. The underlying SCTP protocol is implemented in the Sctp module.

3.4.1 Callback Interface

Messages received from the Sctp module can be processed by the `SctpSocket` where they belong to. The `SctpSocket` deconstructs the message and uses the `SctpSocket::ICallback` interface to notify the application about the received data or service indication:

```
class ICallback // inner class of SctpSocket
{
    void socketDataArrived(SctpSocket* socket, Packet *packet, bool urgent);
    void socketEstablished(SctpSocket *socket);
    // ...
    void socketClosed(SctpSocket *socket);
    void socketFailure(SctpSocket *socket, int code);
};
```

3.4.2 Configuring Connections

The `SctpSocket` class supports setting several SCTP specific connection parameters directly:

```
socket.setOutboundStreams(2);
socket.setStreamPriority(1);
socket.setEnableHeartbeats(true);
// ...
```

Upon setting the individual parameters, the socket immediately sends sevice requests to the underlying Sctp protocol module.

3.4.3 Setting up Connections

Since SCTP is a connection oriented protocol, a connection must be established before applications can exchange data. On the one side, the application listens at a local address and port for incoming SCTP connections:

```
socket.bind(Ipv4Address("10.0.0.42"), 42); // local address/port
socket.listen(true); // start listening for incoming connections
```

On the other side, the application connects to a remote address and port to establish a new connection:

```
socket.connect(Ipv4Address("10.0.0.42"), 42);
```

3.4.4 Accepting Connections

The Sctp module automatically notifies the `SctpSocket` about incoming connections. The socket in turn notifies the application using the `ICallback::socketAvailable` method of the callback interface. Finally, incoming SCTP connections must be accepted by the application before they can be used:

```
class SctpServerApp : public cSimpleModule, public SctpSocket::ICallback
{
    SocketMap socketMap;
    SctpSocket serverSocket;

```

(continues on next page)

(continued from previous page)

```
    void socketAvailable(SctpSocket *socket, SctpAvailableInfo *info);
};

void SctpServerApp::socketAvailable(SctpSocket *socket, SctpAvailableInfo *info)
{
    auto newSocket = new SctpSocket(info);
    // ...
    socketMap.addSocket(newSocket);
    serverSocket.accept(info->getNewSocketId());
}
```

3.4.5 Sending Data

After the connection has been established, applications can send data to the remote applica- tion via a simple function call:

```
socket.send(packet1);
// ...
socket.send(packet42);
```

Packet data is enqueued by the local Sctp module and transmitted over time according to the protocol logic.

3.4.6 Receiving Data

Receiving data is as simple as implementing the corresponding method of the `SctpSocket::ICallback` interface. One caveat is that packet data may arrive in different chunk sizes (but the same order) than they were sent due to the nature of SCTP protocol.

For example, the application may directly implement the `SctpSocket::ICallback` interface and print the received data as follows:

```
class SctpApp : public cSimpleModule, public SctpSocket::ICallback
{
    void socketDataArrived(SctpSocket *socket, Packet *packet, bool urgent);
};

void SctpApp::socketDataArrived(SctpSocket *socket, Packet *packet, bool urgent)
{
    EV << packet->peekData() << endl;
}
```

3.5 IPv4 Socket

The `Ipv4Socket` class provides an easy to use C++ interface to send and receive IPv4 datagrams. The underlying IPv4 protocol is implemented in the Ipv4 module.

3.5.1 Callback Interface

Messages received from the Ipv4 module must be processed by the socket where they belong as shown in the general section. The `Ipv4Socket` deconstructs the message and uses the `Ipv4Socket::ICallback` interface to notify the application about the received data:

```
class ICallback // inner class of Ipv4Socket
{
    void socketDataArrived(Ipv4Socket *socket, Packet *packet);
};
```

3.5.2 Configuring Sockets

In order to only receive IPv4 datagrams which are sent to a specific local address or contain a specific protocol, the socket can be bound to the desired local address or protocol.

For example, the following code fragment shows how the INET PingApp binds to the ICMPv4 protocol to receive all incoming ICMPv4 Echo Reply messages:

```
socket.bind(&Protocol::icmpv4, Ipv4Address()); // filter for ICMPv4 messages
```

For only receiving IPv4 datagrams from a specific remote address, the socket can be connected to the desired remote address:

```
socket.connect(Ipv4Address("10.0.0.42")); // filter for remote address
```

3.5.3 Sending Data

After the socket has been configured, applications can immediately start sending IPv4 datagrams to a remote address via a simple function call:

```
socket.sendTo(packet, Ipv4Address("10.0.0.42")); // remote address
```

If the application wants to send several IPv4 datagrams to the same destination address, it can optionally connect to the destination:

```
socket.connect(Ipv4Address("10.0.0.42")); // remote address
socket.send(packet1);
// ...
socket.send(packet42);
```

The IPv4 protocol is in fact connectionless, so when the Ipv4 module receives the connect request, it simply remembers the remote address, and uses it as the default destination address for later sends.

The application can call `connect()` several times on the same socket.

3.5.4 Receiving Data

For example, the application may directly implement the `Ipv4Socket::ICallback` interface and print the received data as follows:

```
class Ipv4App : public cSimpleModule, public Ipv4Socket::ICallback
{
    void socketDataArrived(Ipv4Socket *socket, Packet *packet);
};

void Ipv4App::socketDataArrived(Ipv4Socket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

3.6 IPv6 Socket

The `Ipv6Socket` class provides an easy to use C++ interface to send and receive IPv6 datagrams. The underlying IPv6 protocol is implemented in the Ipv6 module.

3.6.1 Callback Interface

Messages received from the Ipv6 module must be processed by the socket where they belong as shown in the general section. The `Ipv6Socket` deconstructs the message and uses the `Ipv6Socket::ICallback` interface to notify the application about the received data:

```
class ICallback // inner class of Ipv6Socket
{
    void socketDataArrived(Ipv6Socket *socket, Packet *packet);
};
```

3.6.2 Configuring Sockets

In order to only receive IPv6 datagrams which are sent to a specific local address or contain a specific protocol, the socket can be bound to the desired local address or protocol.

For example, the following code fragment shows how the INET PingApp binds to the ICMPv6 protocol to receive all incoming ICMPv6 Echo Reply messages:

```
socket.bind(&Protocol::icmpv6, Ipv6Address()); // filter for ICMPv6 messages
```

For only receiving IPv6 datagrams from a specific remote address, the socket can be connected to the desired remote address:

```
socket.connect(Ipv6Address("10:0:0:0:0:0:0:42")); // filter for remote address
```

3.6.3 Sending Data

After the socket has been configured, applications can immediately start sending IPv6 datagrams to a remote address via a simple function call:

```
socket.sendTo(packet, Ipv6Address("10:0:0:0:0:0:0:42")); // remote address
```

If the application wants to send several IPv6 datagrams to the same destination address, it can optionally connect to the destination:

```
socket.connect(Ipv6Address("10:0:0:0:0:0:0:42")); // remote address
socket.send(packet1);
// ...
socket.send(packet42);
```

The IPv6 protocol is in fact connectionless, so when the Ipv6 module receives the connect request, it simply remembers the remote address, and uses it as the default destination address for later sends.

The application can call `connect()` several times on the same socket.

3.6.4 Receiving Data

For example, the application may directly implement the `Ipv6Socket::ICallback` interface and print the received data as follows:

```
class Ipv6App : public cSimpleModule, public Ipv6Socket::ICallback
{
    void socketDataArrived(Ipv6Socket *socket, Packet *packet);
};

void Ipv6App::socketDataArrived(Ipv6Socket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

3.7 L3 Socket

The L3Socket class provides an easy to use C++ interface to send and receive datagrams using the conceptual network protocols. The underlying network protocols are implemented in the NextHopForwarding, Flooding, ProbabilisticBroadcast, and AdaptiveProbabilisticBroadcast modules.

3.7.1 Callback Interface

Messages received from the network protocol module must be processed by the associated socket where as shown in the general section. The L3Socket deconstructs the message and uses the L3Socket::ICallback interface to notify the application about the received data:

```
class ICallback // inner class of L3Socket
{
    void socketDataArrived(L3Socket *socket, Packet *packet);
};
```

3.7.2 Configuring Sockets

Since the L3Socket class is network protocol agnostic, it must be configured to connect to a desired network protocol:

```
L3Socket socket(&Protocol::flooding);
```

In order to only receive datagrams which are sent to a specific local address or contain a specific protocol, the socket can be bound to the desired local address or protocol. The conceptual network protocols can work with the ModuleIdAddress class which contains a moduleId of the desired network interface.

For example, the following code fragment shows how the INET PingApp binds to the Echo protocol to receive all incoming Echo Reply messages:

```
socket.bind(&Protocol::echo, ModuleIdAddress(42));
```

For only receiving datagrams from a specific remote address, the socket can be connected to the desired remote address:

```
socket.connect(ModuleIdAddress(42)); // filter for remote interface
```

3.7.3 Sending Data

After the socket has been configured, applications can immediately start sending datagrams to a remote address via a simple function call:

```
socket.sendTo(packet, ModuleIdAddress(42)); // remote interface
```

If the application wants to send several datagrams to the same destination address, it can optionally connect to the destination:

```
socket.connect(ModuleIdAddress(42)); // remote interface
socket.send(packet1);
//..
socket.send(packet42);
```

The network protocols are in fact connectionless, so when the protocol module receives the connect request, it simply remembers the remote address, and uses it as the default destination address for later sends.

The application can call `connect()` several times on the same socket.

3.7.4 Receiving Data

For example, the application may directly implement the `L3Socket::ICallback` interface and print the received data as follows:

```
class L3App : public cSimpleModule, public L3Socket::ICallback
{
    void socketDataArrived(L3Socket *socket, Packet *packet);
};

void L3App::socketDataArrived(L3Socket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

3.8 TUN Socket

The `TunSocket` class provides an easy to use C++ interface to send and receive datagrams using a TUN interface. The underlying TUN interface is implemented in the Tun module.

A TUN interface is basically a virtual network interface which is usually connected to an application (from the outside) instead of other network devices. It can be used for many networking tasks such as tunneling, or virtual private networking.

3.8.1 Callback Interface

Messages received from the Tun module must be processed by the socket where they belong as shown in the general section. The `TunSocket` deconstructs the message and uses the `TunSocket::ICallback` interface to notify the application about the received data:

```
class ICallback // inner class of TunSocket
{
    void socketDataArrived(TunSocket *socket, Packet *packet);
};
```

3.8.2 Configuring Sockets

A `TunSocket` must be associated with a TUN interface before it can be used:

```
socket.open(interface->getId());
```

3.8.3 Sending Packets

As soon as the TunSocket is associated with a TUN interface, applications can immediately start sending datagrams via a simple function call:

```
socket.send(packet);
```

When the application sends a datagram to a TunSocket, the packet appears for the protocol stack within the network node as if the packet were received from the network.

3.8.4 Receiving Packets

Messages received from the TUN interface must be processed by the corresponding TunSocket. The TunSocket deconstructs the message and uses the TunSocket::ICallback interface to notify the application about the received data:

```
class TunApp : public cSimpleModule, public TunSocket::ICallback
{
    void socketDataArrived(TunSocket *socket, Packet *packet);
};

void TunApp::socketDataArrived(TunSocket *socket, Packet *packet)
{
    EV << packet->peekData() << endl;
}
```

When the protocol stack within the network node sends a datagram to a TUN interface, the packet appears for the application which uses a TunSocket as if the packet were sent to the network.

CHAPTER

FOUR

TESTING

INET contains a comprehensive test suite. The test suite can be found under the `tests` folder. Testing is usually done from the command line after setting up the environment with:

```
$ cd ~/workspace/omnetpp
$ . setenv
$ cd ~/workspace/inet
$ . setenv
```

4.1 Regression Testing

The most often used tests are the so called fingerprint tests, which are generally useful for regression testing. Fingerprint tests are a low-cost but effective tool for regression testing of simulation models during development. For more details on fingerprint testing, please refer to the OMNeT++ manual.

The INET fingerprint tests can be found under the `tests/fingerprint` folder. Each test is basically a simulation scenario described by one line in a CSV file. INET contains several such CSV files for different groups of tests. One CSV line describes a simulation by specifying the working directory, command line arguments, simulation time limit, expected fingerprint and test result (e.g. pass or fail), and several user defined tags to help filtering.

Fingerprint tests can be run using the **`fingerprinttest`** script found in the above folder. To run all fingerprint tests, simply run the script without any arguments:

```
$ ./fingerprinttest
```

The script also has various command line arguments, which can be understood by asking for help with:

```
$ ./fingerprinttest -h
```

Running all INET fingerprint tests takes several minutes even on a modern computer. By default, the script utilizes all available CPUs, the tests are run in parallel (non-deterministic order).

The simplest way to make the tests run faster, is to run only a subset of all fingerprint tests. The script provides a filter parameter (-m) which takes a regular expression that is matched against all information of the test.

Another less commonly used technique is running the fingerprint tests in release mode. One disadvantage of release mode is that certain assertions, which are conditionally compiled, are not checked.

For example, the following command runs all wireless tests in release mode:

```
$ ./fingerprinttest --release -m wireless
```

While the tests are running, the script prints some information about each test followed by the test result. The test result is either PASS, FAIL or ERROR (plus some optional details):

```
/examples/bgpv4/BgpAndOspfSimple/ -f omnetpp.ini -c config1 -r 0 ... : PASS
/examples/bgpv4/BgpCompleteTest/ -f omnetpp.ini -c config1 -r 0 ... : FAIL
```

When testing is finished, the script also prints a short summary of the results with the total number of tests, failures and errors:

```
Failures:
  /examples/bgpv4/BgpCompleteTest/ -f omnetpp.ini -c config1 -r 0
----------------------------------------------------------------------
Ran 2 tests in 3.615s

FAILED (failures=1)
```

Apart from the information printed on the console, fingerprint testing also produces three CSV files. One for the updated fingerprints, one for the failed tests, and one for the test which couldn't finish without an error. The updated file can be used to overwrite the original CSV file to accept the new fingerprints.

Usually, when a simulation model is being developed, the fingerprint tests should be run every now and then, to make sure there are no regressions (i.e. all tests run with the expected result). How often tests should be run depends on the kind of development, tests may be run from several times a day to a few times a week.

For example, if a completely new simulation model is developed with new examples, then the new fingerprint tests can be run as late as when the first version is pushed into the repository. In contrast, during an extensive model refactoring, it's adivsable to run the affected tests more often (e.g. after each small step). Running the tests more often helps avoiding getting into a situation where it's difficult to tell if the new fingerprints is acceptable or not.

In any case, certain correct changes in the simulation model break fingerprint tests. When this happens, there are two ways to validate the changes:

- One is when the changes are further divided until the smallest set of changes are found, which still break fingerprints. In this case, this set is carefully reviewed (potentially by multiple people), and if it is found to be correct, then the fingerprint changes are simply accepted.
- The other is to actually change the fingerprint calculation in a way that ignores the effects of the set of changes. To this end, you can change the fingerprint ingredients, use filtering, or change the fingerprint calculation algorithm in C++. These options are covered in the OMNeT++ manual.

  With this method, the fingerprint tests must be re-run before the changes are applied using the modified fingerprint calculation. The updated fingerprint CSV files must be used to run the fingerprint tests after applying the changes. This method is often time consuming, but may be worth the efforts for complex changes, because it allows having a far greater confidence in correctness.

For a simple example, a timing change which neither changes the order of events nor where the events happen, can be easily validated by changing the fingerprint and removing time from the ingredients (default is tplx) as follows:

```
./fingerprinttest -a --fingerprint=0 --fingerprint-ingredients=plx
```

Another common example is when a submodule is renamed. Since the default fingerprint ingredients contain the full module path of all events, this change will break fingerprint tests. If the module is not optional or not accessed by other modules using its name, then removing the module path from the fingerprint ingredients (i.e. using tlx) can be used to validate this change.

For a more complicated example, when the IEEE 802.11 MAC (very complex) model is refactored to a large extent, then it's really difficult to validate the changes. If despite the changes, the model is expected to keep its external behavior, then running the fingerprint tests can be used to prove this to some extent. The easy way to do this is to actually ignore all events executed by the old and the new IEEE 802.11 MAC models:

```
# please note the quoting uses both ' and " in the right order
./fingerprinttest -a --fingerprint-modules='"not(fullPath(**.wlan[*].mac.**))"'
```

INET uses continuous integration on Github. Fingerprint and other tests are run automatically for changes on the master and integration branches. Moreover, all submitted pull requests are automatically tested the same way. The result of the test suite is clearly marked on the pull request with check mark or a red cross.

In general, contributors are expected to take care of not breaking the fingerprint tests. In case of a necessary fingerprint change, the CSV files should be updated in separate patches.

CHAPTER

FIVE

APPENDIX: AUTHOR'S GUIDE

5.1 Overview

This chapter is intended for authors and contributors of this *INET Developer's Guide*, and covers the guidelines for deciding what type of content is appropriate for this *Guide* and what is not.

The main guiding principle is to avoid redundancy and duplication of information with other pieces of documentation, namely:

- Standards documents (RFCs, IEEE specifications, etc.) that describe protocols that INET modules implement;
- *INET User's Guide*, which is intended for users who are interested in assembling simulations using the components provided by INET;
- *INET Framework Reference*, directly generated from NED and MSG comments by OMNeT++ documentation generator;
- Showcases, tutorials and simulation examples (`showcases/`, `tutorials/` and `examples/` folders in the INET project)

Why is duplication to be avoided? Multiple reasons:

- It is a waste of our reader's time they have to skip information they have already seen elsewhere
- The text can easily get out of date as the INET Framework evolves
- It is extra effort for maintainers to keep all copies up to date

5.2 Guidelines

5.2.1 Do Not Repeat the Standard

When describing a module that implements protocol X, do not go into lengths explaining what protocol X does and how it works, because that is appropriately (and usually, much better) explained in the specification or books on protocol X. It is OK to summarize the protocol's goal and principles in a short paragraph though.

In particular, do not describe the *format of the protocol messages*. It surely looks nice and takes up a lot of space, but the same information can probably be found in a myriad places all over the internet.

5.2.2 Do Not Repeat NED Documentation

Things like module parameters, gate names, emitted signals and collected statistics are appropriately and formally part of the NED definitions, and there is no need to duplicate that information in this *Guide*.

Detailed information on the module, such as *usage details* and the list of *implemented standards* should be covered in the module's NED documentation, not in this *Guide*.

5.2.3 Do Not Repeat C++ Documentation

Describing every minute detail of C++ classes, methods, arguments are expected to be appropriately present in their *doxygen* documentation.

5.2.4 What then?

Concentrate on giving a "big picture" of the implementation: what it is generally capable of, how the parts fit together, how to use the provided APIs, what were the main design decisions, etc. Give simple yet meaningful examples and just enough information about the API that after a quick read, users can "bootstrap" into implementing their own protocols and applications. If they have questions afterwards, they will/should refer to the C++ documentation.