OMNeT++ Community Summit, 2017

INET 4.0
New Features and Migration

University of Bremen – Germany – September 7-8, 2017

Levente Mészáros

Overview

Revisited Network Node Architecture

Introduction of Packet Tags

Redesigned Packet API

Original 2015 presentation

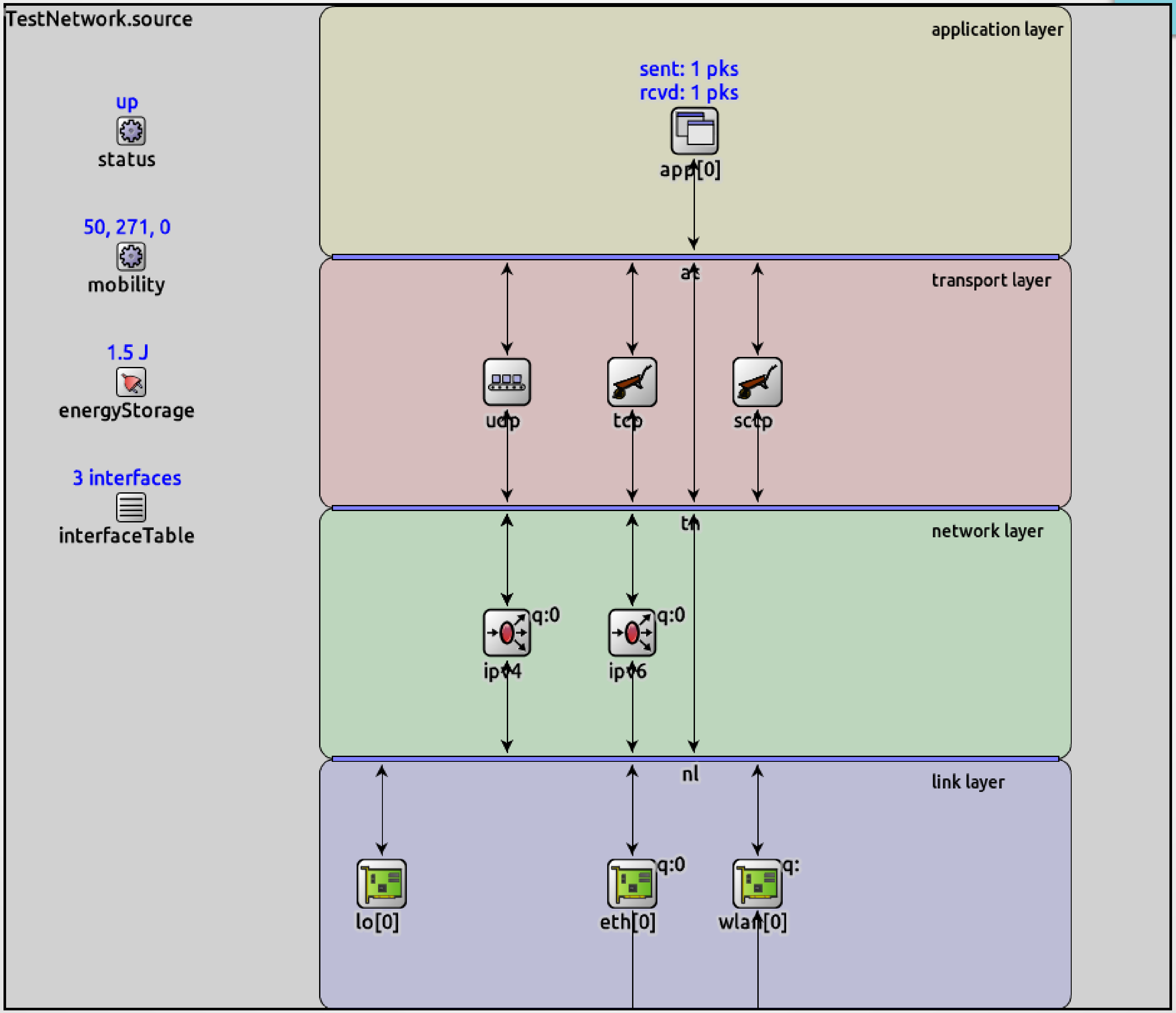
Motivation

- Applications must be able to use
  - different sockets and protocols simultaneously
  - raw sockets and lower layer protocols directly
- Protocols must be able to communicate with multiple applications and other protocols without implementing a dispatch mechanism
- Protocols of adjacent OSI layers must be able to communicate in a many-to-many relationship
- Network nodes must be more reusable to allow configuring different applications, protocols, and interfaces

Completed Changes

- Merged all application submodule vectors into one vector
- Removed dispatch mechanisms from existing protocols
- Added a new generic `MessageDispatcher` module
- Added dispatchers to base modules of network nodes
- Added dispatchers to network layer compound modules
- Added protocol registration to existing protocols
- Added interface registration to existing interfaces
- Added raw sockets to allow accessing lower layer protocols from applications through dispatchers

Revisited Standard Host



Migration Tasks

- Add your protocols to global C++ list of known protocols
- Register your protocols in dispatchers by calling `registerProtocol()` in `initialize()`
- Register your interfaces in dispatchers by calling `registerInterface()` in `initialize()`
- Dispatchers automatically learn where application sockets are based on intercepted open and close commands
- Add dispatchers to your network node modules if needed
  - dispatchers are completely optional, modules can still be organized in other simpler ways

Overview

Revisited Network Node Architecture

Introduction of Packet Tags

Redesigned Packet API

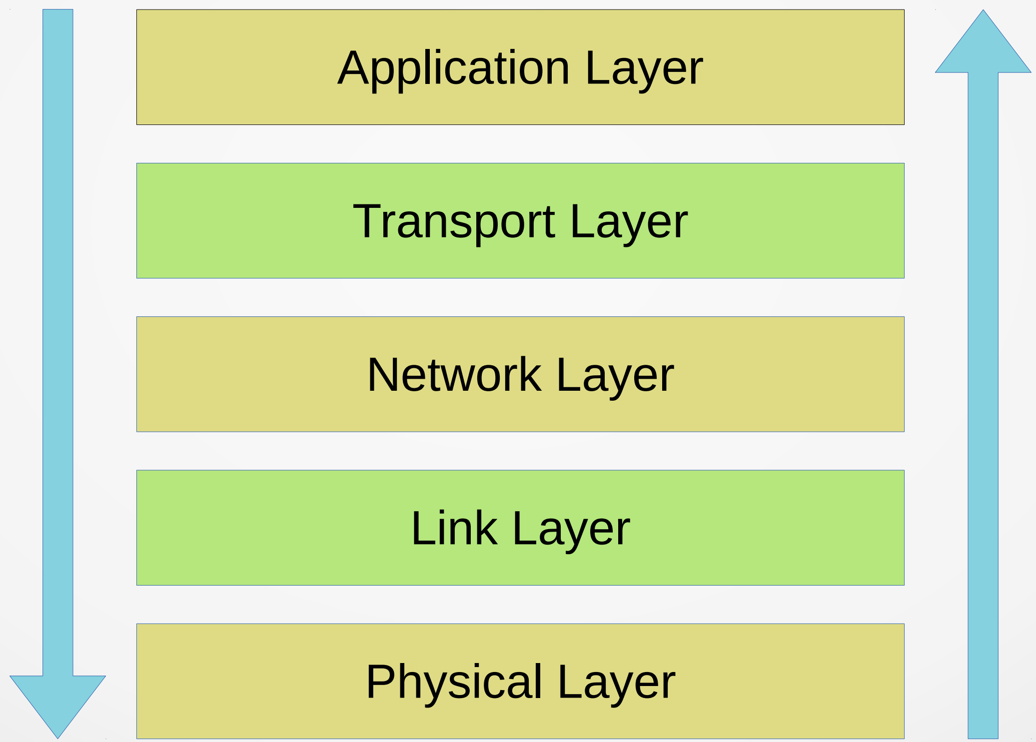
Original 2015 presentation

Motivation

- Cross-layer communication must be supported for many useful features
  - Applications must be able to control various service parameters (e.g. hop limit, QoS, outgoing interface)
  - Higher layer protocols must be able to control resource optimization parameters (e.g. transmission power)
  - Routing protocols must be able to access link quality indications (e.g. receive power)
- Protocol modules must be able to control the message dispatch mechanism
- Protocol modules must specify what protocol of a packet

Cross-Layer Communication

- As packets go through the layers



- Packets collect various request tags

- Packets collect various indication tags

Completed Changes

- Control infos are split into reusable tags in MSG files
  - tags focus on a single parameterization aspect
- Packets no longer carry control infos, they have several tags attached instead
  - Request tags are passed top-down (`Req` suffix)
  - Indication tags are passed bottom-up (`Ind` suffix)
  - Meta-info tags are passed around (`Tag` suffix)
- Tags pass through protocol layers
- Tags are removed where they are processed

Migration Tasks

- Split your existing packet control info classes in MSG files
  - Reuse existing tags if possible
  - Create new tags as needed
- Replace control infos with tags for both sending and processing packets in C++ code
- Remove tags individually where they are processed
- Remove all tags if a packet is reused or it leaves a node
- Add `DispatchProtocolReq` to instruct the dispatcher which protocol should process the packet next
- Add `PacketProtocolTag` to specify what kind of protocol is carried in the packet

Overview

Revisited Network Node Architecture
Introduction of Packet Tags
Redesigned Packet API

Motivation

- Protocols must be able to easily implement
  - **Fragmentation:** truncating packet length is a kludge
  - **Aggregation:** encapsulated packet field is insufficient
  - **Emulation:** processing raw packets separately is bad
- Protocols should not individually implement support for
  - byte count, raw bytes, object based, and **mixed packets and streams**
- Protocols should not directly use packet serialization
- Packet parts should not contain non-protocol related data
- Packet parts must be serializable on their own

API Goals

- Encapsulation
- Fragmentation
- Aggregation
- Serialization and deserialization
- Duplication and sharing
- Representation selection
- Emulation
- Queueing
- Reassembly and reordering

Representation Goals

- Length based and raw parts
- Optional and variant parts
- Successive and split parts
- Sharing individual parts
- Mixing differently represented parts
- Immutable parts
- Incorrectly received parts
- Incompletely received parts
- Improperly represented parts

Two-Layer API

- Chunks (lower layer API)
  - Provide different representations for packet parts
  - Can be combined to form larger chunks
  - Can be immutable to support efficient sharing
- Containers (upper layer API)
  - Provide packets, queues and buffers
  - Use one or more chunks for their contents
  - Use immutable chunks internally to support sharing
  - Merge and split chunks automatically
  - Share and reuse chunks automatically

Chunks Represent Packet Parts

- Operations
  - Insert and remove at the beginning and at the end
  - Peek arbitrary part and query length
  - Serialize and deserialize
- Chunks are designed for subclassing by the user
- Chunks can also be used to represent
  - Optional parts with separate optional chunks
  - Variant parts with subclassing chunks
  - Successive parts with a sequence of chunks

Count-Based Chunks

- They are used when the actual data is irrelevant
- `BitCountChunk` supports bit precision

61 bits

- `ByteCountChunk` supports byte precision

32 Bytes

Raw Data Chunks

- They are used for packet recording or hardware emulation
- `BitsChunk` provides raw data support for bits

10100110101111010101 0

- `BytesChunk` provides raw data support for bytes

CD 80 AB 02 75 23 A8 F7 FE B9 8C 04 00 23 FF

Field-Based Chunks

- They can still be generated using the MSG compiler
  - The `packet` keyword must be replaced with `class`
  - The class must subclass from `FieldsChunk`
  - The `byteLength` field is replaced with `chunkLength`
  - Field-Based chunks can form a class hierarchy

```
packet UDPPacket
{
    byteLength = 8;
    unsigned short srcPort;
    unsigned short destPort;
    int totalLengthField = -1;
}
```

→

```
class UdpHeader extends FieldsChunk
{
    chunkLength = byte(8);
    unsigned short srcPort;
    unsigned short destPort;
    int totalLengthField = -1;
}
```

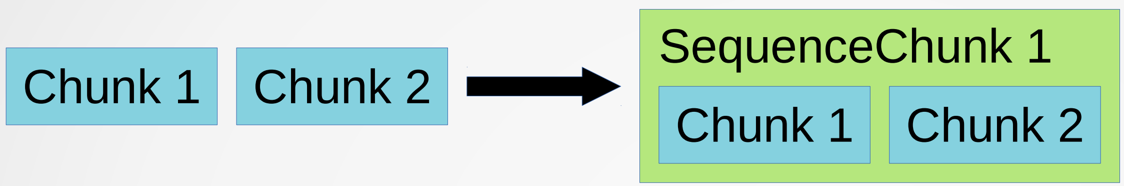
Field-Based Chunk Runtime Example

- Some fields are inherited from the `FieldsChunk` base class
- The raw data is automatically displayed if there is a serializer

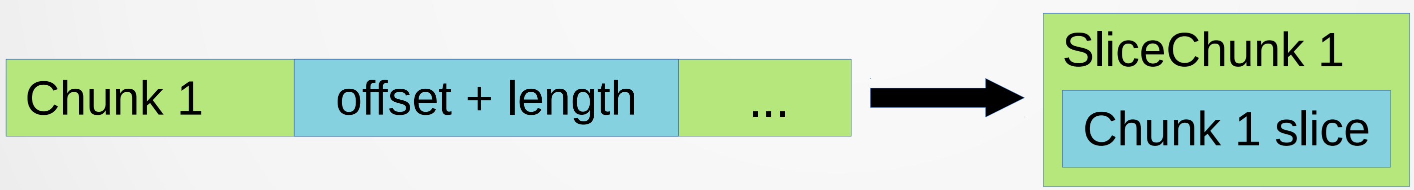
```
▾ (UdpHeader) inet::UdpHeader, length = 8 byte
    mutable = false (bool)
    complete = true (bool)
    correct = true (bool)
    properlyRepresented = true (bool)
    chunkLength = 64 (bit)
  ▾ bits[2] (string)
        [0] 0000 0100 0000 0100 0001 0011 1000 1000
        [1] 0000 0101 1100 1001 1110 1001 1111 0010
  ▾ bytes[1] (string)
        [0] 04 04 13 88 05 C9 E9 F2
    srcPort = 1028 [...]  (unsigned short)
    destPort = 5000 [...]  (unsigned short)
    totalLengthField = 1481 [...]  (int)
    crc = 59890 [...]  (uint16_t)
    crcMode = 3 (CRC_COMPUTED) [...]  (int)
```

Compound Chunks

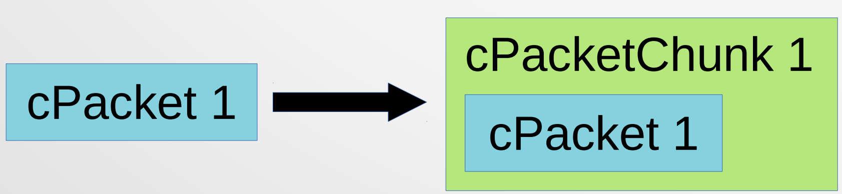
- `SequenceChunk` provides concatenation



- `SliceChunk` provides slicing using offset and length



- `cPacketChunk` provides support for `cPacket`



Automatic Merging and Splitting Rules

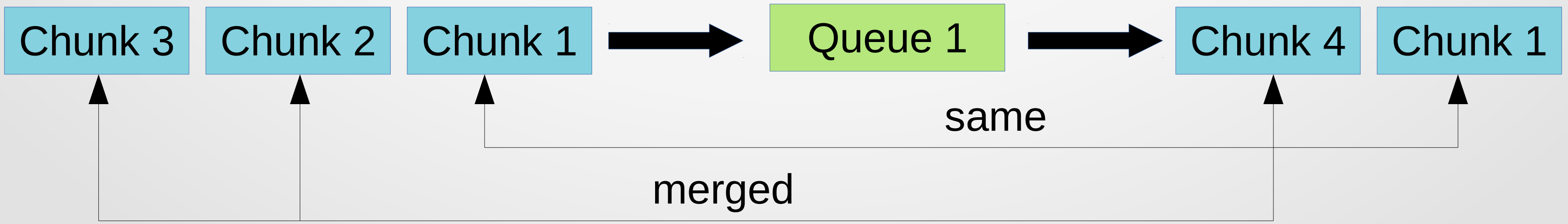
- Count-based chunks are merged and split on demand
- Raw data chunks are merged and split on demand
- Consecutive `SliceChunk`s are merged
- Subsequent `SequenceChunk`s are merged
- Nested `SequenceChunk`s are flattened
- `SequenceChunk` slice is flattened into a `SequenceChunk` potentially containing `SliceChunk`s at the ends
- etc.

Chunk API Usage Example

```
// create a new UDP header
auto header = std::make_shared<UdpHeader>();
// set some fields
header->setSrcPort(1000);
// get the first half of the 8 bytes header
auto slice = header->peek(byte(0), byte(4));
// create a new sequence
auto sequence = std::make_shared<SequenceChunk>();
// insert the first half into the sequence
sequence->insertAtEnd(slice);
// insert the second half into the sequence
sequence->insertAtEnd(header->peek(byte(4), byte(4)));
// get the complete header due to automatic merging
auto complete = sequence->peek(byte(0), byte(8));
// get the raw bytes from the complete header
auto raw = complete->peek<BytesChunk>(byte(0), byte(8));
// get the restored header from raw bytes
auto restored = raw->peek<UdpHeader>(byte(0), byte(8));
```

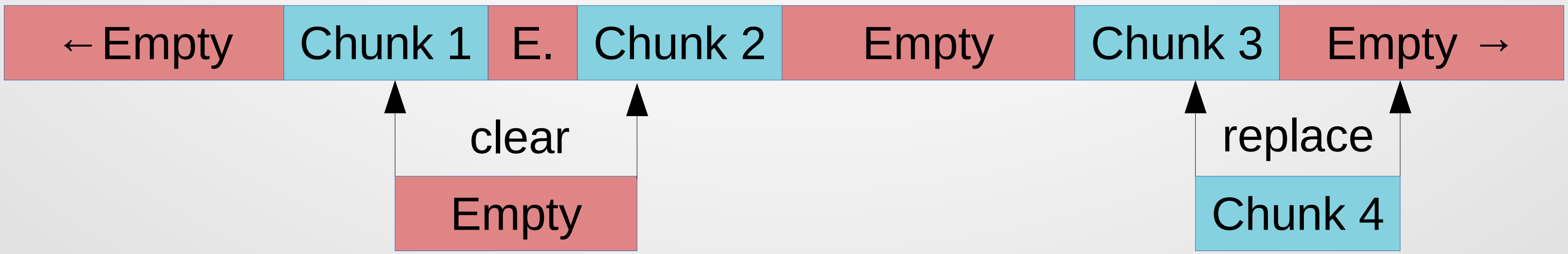
Queueing Chunks

- `ChunkQueue` provides FIFO queueing for in order chunks
- Operations
  - Peek various parts and query length
  - Push at the tail and pop at the head
  - Serialize and deserialize
- Representation
  - One immutable chunk to support sharing
  - Most likely a `SequenceChunk` or a `BytesChunk`



Buffering Chunks

- `ChunkBuffer` provides buffering for out of order chunks
- Operations
  - Peek various regions and query lengths
  - Replace a region
  - Clear a region
- Representation
  - One immutable chunk per region to support sharing
  - Most likely a `SequenceChunk` or a `BytesChunk`



Reassembling Chunks

- `ReassemblyBuffer` merges out of order parts into a whole
  - First part arrives

Chunk 1	Empty

  - Last part arrives

Chunk 1	Empty	Chunk 2

  - Middle part arrives

Chunk 1	E.	Chunk 3	Empty	Chunk 2

  - Arriving part fills the gap

Chunk 1	E.	Chunk 3	Chunk 4	Chunk 2

  - Arriving part overwrites existing parts

Chunk 1	Chunk 5	Chunk 4	Chunk 2

Reordering Chunks

- `ReorderBuffer` forms a stream from out of order parts
  - Expected part arrives

Chunk 1	Empty →

  - Out of order part arrives

Chunk 1	Empty	Chunk 2	Empty →

  - Another out of order part arrives

Chunk 1	Empty	Chunk 2	Empty	Chunk 3	Empty →

  - Arriving part fills in the gap

Chunk 1	Chunk 4	Chunk 2	Empty	Chunk 3	Empty →

  - Arriving part overwrites existing parts

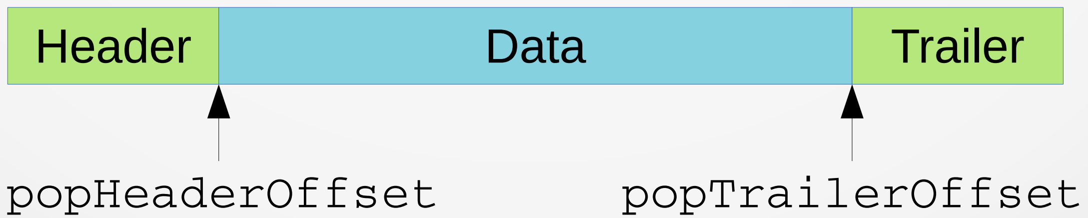
Chunk 1	Chunk 4	Chunk 2	Chunk 5	Empty →

INET Packet

- INET provides a new `inet::Packet` extending `cPacket`
- Operations
  - Peek various parts and query lengths
  - Insert and remove at the beginning and at the end
  - Serialize and deserialize
- Representation
  - Single immutable chunk to support sharing
  - Most likely a `SequenceChunk` or a `BytesChunk`

Packet Partitioning

- Packet provides header, data and trailer partitioning
- Partitioning is not shared among duplicates
- Partitioning is updated during processing
- Partitioning doesn't affect the actual packet data

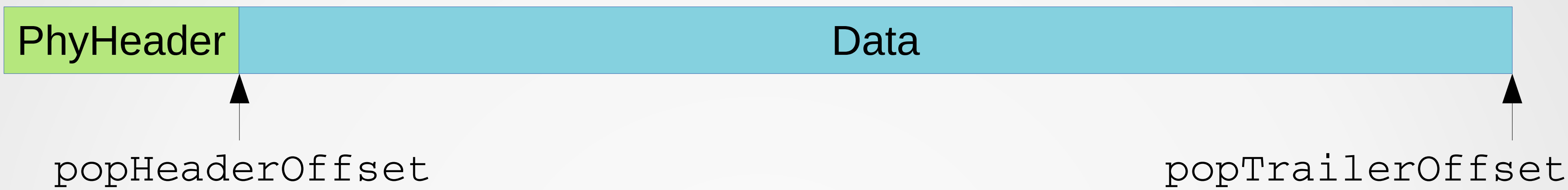


Packet Processing

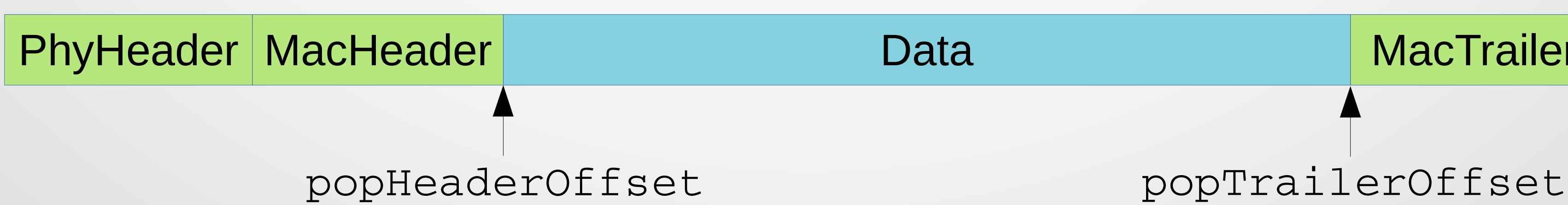
- Dispatch in protocol logic must be entirely based on data
  - Packet class is always `Packet`
    so `dynamic_cast<...>(packet)` cannot be used
  - Chunk class is always what is requested
    so `dynamic_cast<...>(chunk)` cannot be used
- Forwarding requires chunk duplication due to sharing
  - Received packet's chunks are immutable
  - Cannot call `setTimeToLive()` on immutable chunks

```
auto header = packet->removeHeader<IPv4Header>();
header->setTimeToLive(ipv4Header->getTimeToLive() - 1);
packet->insertHeader(header);
```

Packet Processing Example

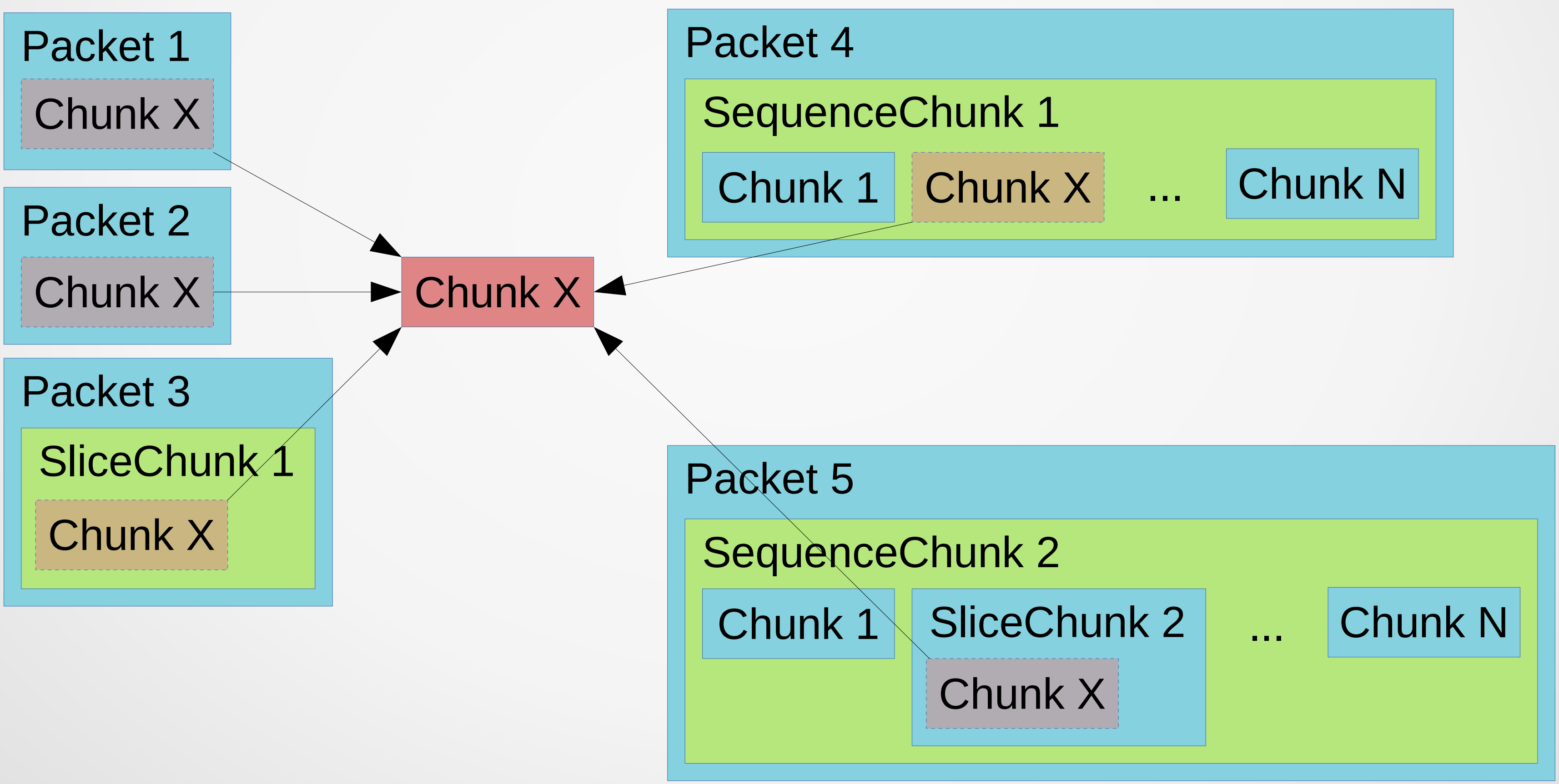



```
void Ieee80211Mac::decapsulate(Packet *packet)
{
    auto header = packet->popHeader<Ieee80211DataHeader>();
    header->getTransmitterAddress();
    packet->popTrailer<Ieee80211MacTrailer>(byte(4));
}
```



Sharing Chunks Among INET Packets

- Chunks are shared among containers with shared pointers

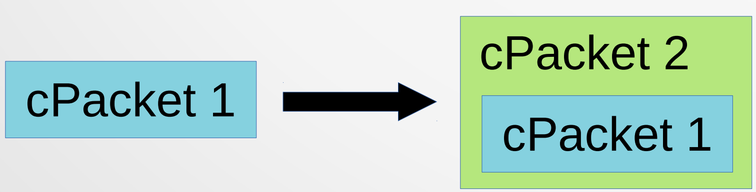


Encapsulation Using cPacket

- Maps to encapsulate ( )

```
Ieee80211DataFrame *Ieee80211MgmtAdhoc::encapsulate(cPacket *packet)
{
    Ieee80211DataFrame *frame = new Ieee80211DataFrame();
    frame->setTransmitterAddress(myAddress);
    frame->encapsulate(packet);
    return frame;
}
```

- Result

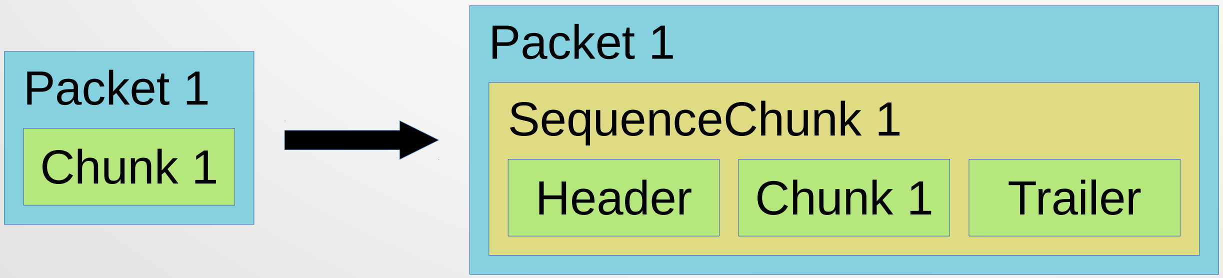


Encapsulation Using INET Packet

- Maps to concatenation (most of the time)

```
void Ieee80211Mac::encapsulate(Packet *packet) {
    auto header = std::make_shared<Ieee80211DataHeader>();
    header->setTransmitterAddress(mib->address);
    packet->insertHeader(header);
    auto trailer = std::make_shared<Ieee80211MacTrailer>();
    trailer->setFcsMode(FCS_DECLARED_CORRECT);
    packet->insertTrailer(trailer);
}
```

- Result



Encapsulated Packet Example

- Using cPacket

```
▼ ● Voice-97 (Ieee80211DataFrameWithSNAP)
    ▼ ● Voice-97 (IPv4Datagram)
        ▼ ● Voice-97 (UDPPacket)
              ● Voice-97 (ApplicationPacket)
```

- Using INET Packet

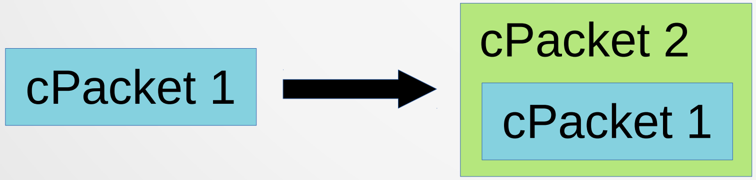
```
▼ ● Voice-97 (Packet)
    ▼ (SequenceChunk)
          (Ieee80211PhyHeader) inet::physicallayer::Ieee80211PhyHeader, length = 24 byte
          (Ieee80211DataHeader) inet::ieee80211::Ieee80211DataHeader, length = 26 byte
          (Ieee8022SnapHeader) inet::Ieee8022SnapHeader, length = 8 byte
          (IPv4Header) inet::IPv4Header, length = 20 byte
          (UdpHeader) inet::UdpHeader, length = 8 byte
          (ApplicationPacket) inet::ApplicationPacket, length = 100 byte
          (Ieee80211MacTrailer) inet::ieee80211::Ieee80211MacTrailer, length = 4 byte
```

Fragmentation Using cPacket

- Maps to `encapsulate()`, `setBitLength()` and offset

```
auto fragment = new Ieee80211DataFrame();
fragment->setFragmentNumber(index);
fragment->setFragmentOffset(offset);
fragment->encapsulate(frame);
fragment->setByteLength(length);
```

- Result



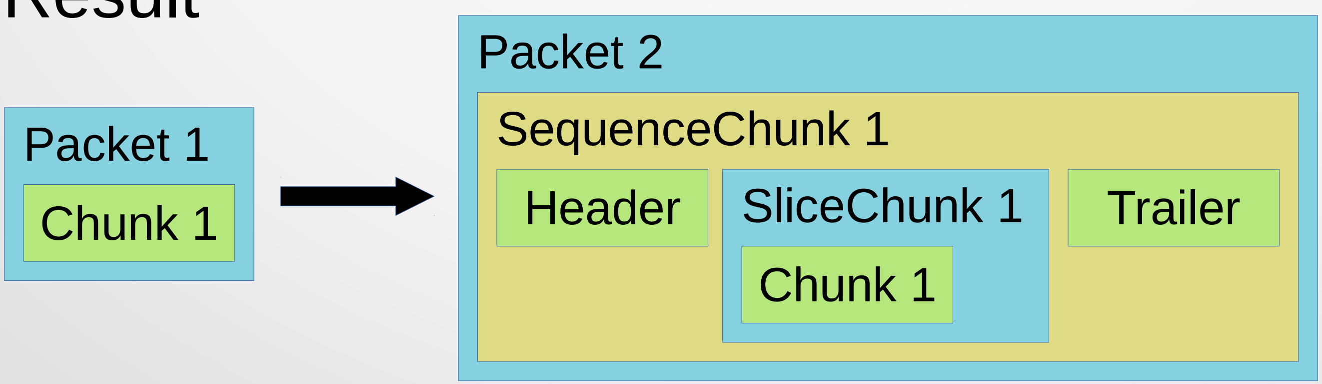
- Length of encapsulated packet > length of packet!

Fragmentation Using INET Packet

- Maps to slicing (most of the time)

```
auto fragment = new Packet();
auto header = std::make_shared<Ieee80211DataHeader>();
header->setFragmentNumber(index);
fragment->insertHeader(header);
auto data = frame->peekDataAt(offset, length);
fragment->append(data);
auto trailer = std::make_shared<Ieee80211MacTrailer>();
fragment->insertTrailer(trailer);
```

- Result



Fragmented Packet Example

- Using cPacket

  - Video-31 (Ieee80211DataFrameWithSNAP)
    - Video-31 (Ieee80211DataFrameWithSNAP)
      - Video-31 (IPv4Datagram)
        - Video-31 (UDPPacket)
          - Video-31 (ApplicationPacket)

- Using INET Packet

  - Video-31-frag0 (Packet)
    - (SequenceChunk)
      - (Ieee80211PhyHeader) inet::physicallayer::Ieee80211PhyHeader, length = 24 byte
      - (Ieee80211DataHeader) inet::ieee80211::Ieee80211DataHeader, length = 26 byte
      - (Ieee8022SnapHeader) inet::Ieee8022SnapHeader, length = 8 byte
      - (IPv4Header) inet::IPv4Header, length = 20 byte
      - (UdpHeader) inet::UdpHeader, length = 8 byte
      - (SliceChunk) SliceChunk, offset = 0 byte, length = 1434 byte, chunk = {inet::ApplicationPacket, length = 1998 byte}
        - (ApplicationPacket) inet::ApplicationPacket, length = 1998 byte
      - (Ieee80211MacTrailer) inet::ieee80211::Ieee80211MacTrailer, length = 4 byte

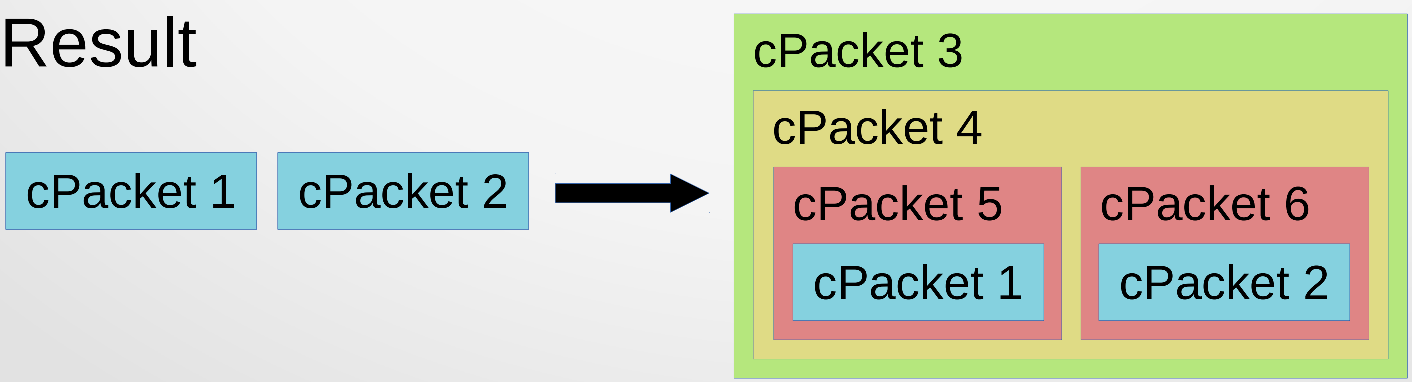
  - Video-31-frag1 (Packet)
    - (SequenceChunk)
      - (Ieee80211PhyHeader) inet::physicallayer::Ieee80211PhyHeader, length = 24 byte
      - (Ieee80211DataHeader) inet::ieee80211::Ieee80211DataHeader, length = 26 byte
      - (SliceChunk) SliceChunk, offset = 1434 byte, length = 564 byte, chunk = {inet::ApplicationPacket, length = 1998 byte}
        - (ApplicationPacket) inet::ApplicationPacket, length = 1998 byte
      - (Ieee80211MacTrailer) inet::ieee80211::Ieee80211MacTrailer, length = 4 byte

Aggregation Using cPacket

- Maps to explicitly added fields

```
auto amsdu = new Ieee80211AMsdu();
amsdu->setSubframesArraySize(frames->size());
for (auto frame : frames)
{
    Ieee80211MsduSubframe msduSubframe;
    msduSubframe.setLength(frame->getBitLength());
    msduSubframe.encapsulate(frame);
    amsdu->setSubframes(i, msduSubframe);
}
amsdu->setByteLength(aMsduLength);
auto aggregatedFrame = new Ieee80211DataFrame("A-MSDU");
aggregatedFrame->setAMsduPresent(true);
aggregatedFrame->encapsulate(amsdu);
```

- Result

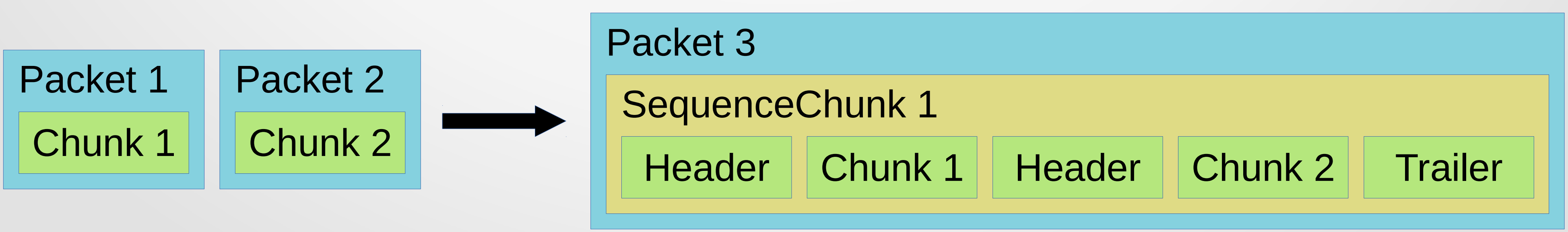


Aggregation Using INET Packet

- Maps to concatenation (most of the time)

```
auto amsdu = new Packet();
for (auto frame : frames) {
    auto data = frame->peekData();
    auto header = std::make_shared<Ieee80211MsduSubframeHeader>();
    header->setLength(data->getChunkLength());
    amsdu->append(header);
    amsdu->append(data);
}
auto header = std::make_shared<Ieee80211DataHeader>();
header->setAMsduPresent(true);
amsdu->insertHeader(header);
amsdu->insertTrailer(std::make_shared<Ieee80211MacTrailer>());
```

- Result



Aggregated Packet Example

- Using cPacket

- A-MSDU (Ieee80211DataFrame)
  - (Ieee80211AMsdu)
    - Voice-26 (Ieee80211MsduSubframe)
      - Voice-26 (IPv4Datagram)
        - Voice-26 (UDPPacket)
          - Voice-26 (ApplicationPacket)
    - Voice-27 (Ieee80211MsduSubframe)
      - Voice-27 (IPv4Datagram)
        - Voice-27 (UDPPacket)
          - Voice-27 (ApplicationPacket)

- Using INET Packet

- A-MSDU (Packet)
  - (SequenceChunk)
    - (Ieee80211PhyHeader) inet::physicallayer::Ieee80211PhyHeader, length = 24 byte
    - (Ieee80211DataHeader) inet::ieee80211::Ieee80211DataHeader, length = 26 byte
    - (Ieee80211MsduSubframeHeader) inet::ieee80211::Ieee80211MsduSubframeHeader, length = 14 byte
    - (Ieee8022SnapHeader) inet::Ieee8022SnapHeader, length = 8 byte
    - (IPv4Header) inet::IPv4Header, length = 20 byte
    - (UdpHeader) inet::UdpHeader, length = 8 byte
    - (ApplicationPacket) inet::ApplicationPacket, length = 100 byte
    - (ByteCountChunk) ByteCountChunk, length = 2 byte
    - (Ieee80211MsduSubframeHeader) inet::ieee80211::Ieee80211MsduSubframeHeader, length = 14 byte
    - (Ieee8022SnapHeader) inet::Ieee8022SnapHeader, length = 8 byte
    - (IPv4Header) inet::IPv4Header, length = 20 byte
    - (UdpHeader) inet::UdpHeader, length = 8 byte
    - (ApplicationPacket) inet::ApplicationPacket, length = 100 byte
    - (Ieee80211MacTrailer) inet::ieee80211::Ieee80211MacTrailer, length = 4 byte

Serialization

- Serialization is implemented in separate serializer classes
- Mapping is stored in global `ChunkSerializerRegistry`
  - `UdpHeader → UdpHeaderSerializer`
- Serializers simply convert to and from a raw stream
  - May handle multiple chunks
  - May handle variant parts
  - Must not be recursive
  - Must not contain any protocol logic
  - Must not compute or verify CRC

Serialization Example

- UDP header serializer

```
const auto& udpHeader = std::static_pointer_cast<const UdpHeader>(chunk);
stream.writeUint16Be(udpHeader->getSourcePort());
stream.writeUint16Be(udpHeader->getDestinationPort());
stream.writeUint16Be(udpHeader->getTotalLengthField());
auto crcMode = udpHeader->getCrcMode();
if (crcMode != CRC_DISABLED && crcMode != CRC_COMPUTED)
    throw cRuntimeError("Cannot serialize UDP header without turned off o
stream.writeUint16Be(udpHeader->getCrc());
```

- Examples of getting the raw bytes from a packet

```
packet->peekAt<BytesChunk>(byte(0), packet->getPacketLength());
packet->peekAllBytes();
```

Serialized Packet Example

```
▼ ● arpREPLY.arpREPLY (Packet)
    totalLength = 720 (bit)
    headerPoppedLength = 0 (bit)
    dataLength = 720 (bit)
    trailerPoppedLength = 0 (bit)
  ▼ contents (SequenceChunk)
      mutable = false (bool)
      complete = true (bool)
      correct = true (bool)
      properlyRepresented = true (bool)
      chunkLength = 720 (bit)
    ▶ bits[23] (string)
    ▼ bytes[6] (string)
        [0] 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F
        [1] 3F 3F 3F 3F 3F 3F 3F 3F 88 01 00 2C 10 00 00 00
        [2] 00 00 0A AA 00 00 00 01 0A AA 00 00 00 02 00 00
        [3] 00 00 AA AA 03 00 00 00 08 06 00 01 08 00 06 04
        [4] 00 02 0A AA 00 00 00 01 0A 00 00 01 0A AA 00 00
        [5] 00 02 0A 00 00 02 00 00 00 00
    ▼ chunks[5] (Chunk)
      ▶ [0] (Ieee80211PhyHeader) : inet::physicallayer::Ieee80211PhyHeader, length = 24 byte
      ▶ [1] (Ieee80211DataHeader) : inet::ieee80211::Ieee80211DataHeader, length = 26 byte
      ▶ [2] (Ieee8022SnapHeader) : inet::Ieee8022SnapHeader, length = 8 byte
      ▶ [3] (ARPPacket) : inet::ARPPacket, length = 28 byte
      ▶ [4] (Ieee80211MacTrailer) : inet::ieee80211::Ieee80211MacTrailer, length = 4 byte
```

Emulation Support

- Senders create packets containing one `BytesChunk`
- Receivers does not handle raw packets in any special way
  - No need to `dynamic_cast<RawPacket>(packet)`
  - No need to deserialize packets, happens transparently
  - Incorrect interpretation of raw packets is possible!
- Testing emulation support using fingerprints
  - Replace packets leaving network nodes with a copy containing one `BytesChunk`

Checksum Handling

- Checksums can be
  - Disabled
  - Declared correct
  - Declared incorrect
  - Computed
- Checksums are computed and verified in protocol modules
  - Parameters are added to the protocol module to control the checksum handling behavior
- Proper serialization requires disabled or computed checksums!

Error Representation

- There are several ways to represent packet reception errors in physical layers
  - Marking the whole packet erroneous by calling `cPacket::setBitError()`
  - Marking an already represented part of the packet erroneous by calling `Chunk::markIncorrect()`
  - Converting only the erroneous part to a `BytesChunk` and altering some of the bytes
  - Converting the whole packet to a `BytesChunk` and altering some of the bytes

Completed Protocol Changes

- Converted all packets to chunks in MSG files
- Refactored all protocols to use INET packets except for `PacketDrill` and `SCTP`
  - Refactored encapsulation, fragmentation and aggregation implementations
  - Replaced queues and buffers with the ones that use chunks where appropriate
  - Refactored data streams (e.g. TCP) to support any combination of mixed data representation
  - Eliminated `RawPacket` handling that was used to support emulation

Completed Other Changes

- Refactored all applications to use INET packets
- Refactored all header serializers to use chunks
  - Moved CRC computation and verification from serializers to protocol modules
- Refactored PCAP recording and packet printers
- Updated all examples and tests
- Validated changes using fingerprint tests

Protocol Migration Tasks

- Convert protocol defined packets to chunks in MSG files
- Remove payload fields from chunks in MSG files
- Refactor `encapsulate()` to insert chunks
- Refactor `decapsulate()` to pop chunks
- Replace `new ...()` packet allocations with `std::make_shared<...>()` chunk allocations
- Passing chunks around may be insufficient due to sharing
  - Pass both packet and chunk as separate arguments
- Take care of the immutability of received packets' chunks

Serializer Migration Tasks

- Convert packet serializers to chunks serializers
  - Remove recursion to encapsulated packet
- Move checksum handling from serializers to protocols
  - Add extra CRC mode field to headers
  - Add CRC mode parameters to protocol module
  - Move generating pseudo headers from serializers to protocols

Questions and Answers

INET 4.0 is coming

Thank you for your attention!