Chapter 8
Packets, Packet Headers, and Header Format

Generally, a packet consists of packet header and data payload. Packet header stores packet attributes (e.g., source and destination IP addresses) necessary for packet delivery, while data payload contains user information. Although this concept is typical in practice, NS2 models packets differently.

In most cases, NS2 extracts information from data payload and stores the information into packet header. This idea removes the need to process data payload at runtime. For example, instead of counting the number of bits in a packet, NS2 stores packet size in the variable `hdr_cmn::size_` (see Sect. 8.3.5), and accesses this variable at runtime.[1]

This chapter discusses how NS2 models packets. Section 8.1 gives an overview on NS2 packet modeling. Section 8.2 discusses the packet allocation and deallocation processes. Sections 8.3 and 8.4 show the details of packet header and data payload, respectively. We give a guideline of how to customize packets (i.e., to define a new payload type and activate/deactivate new and existing protocols) in Sect. 8.5. Finally, the chapter summary is given in Sect. 8.6.

8.1 An Overview of Packet Modeling Principle

8.1.1 Packet Architecture

Figure 8.1 shows the architecture of an NS2 packet model. From Fig. 8.1, a packet model consists of four main parts: actual packet, class `Packet`, protocol-specific headers, and packet header manager.

- **Actual Packet:** An actual packet refers to the portion of memory which stores packet header and data payload. NS2 does not directly access either the

[1]For example, class `LinkDelay` determines packet size from a variable `hdr_cmn::size_` when computing packet transmission time (see Line 11 of Program 7.3).

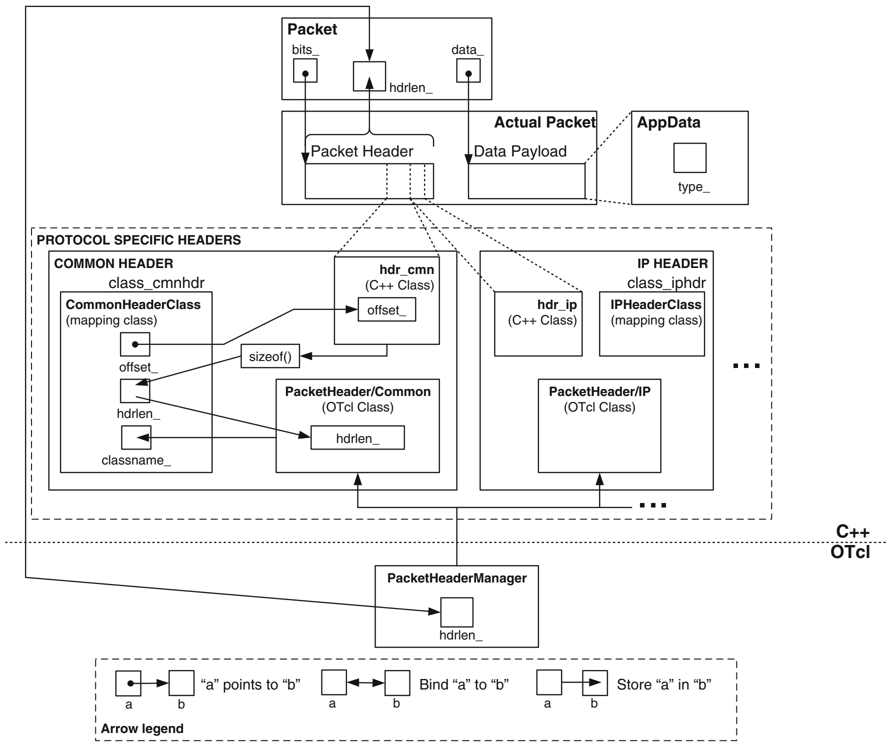

Fig. 8.1 Packet modeling in NS2

packet header or the data payload. Rather, it uses member variables "`bits_`" and "`data_`" of class `Packet` to access packet header and data payload, respectively. The details of packet header and data payload will be given in Sects. 8.3 and 8.4, respectively.

- **Class `Packet`:** Declared in Program 8.1, class `Packet` is the C++ main class which represents packets. It contains the following variables and functions:

  – **C++ Variables of Class `Packet`**

`bits_`	A string which contains packet header
`data_`	A pointer to an `AppData` object which contains data payload
`fflag_`	Set to `true` if the packet is currently referred to by other objects and `false` otherwise
`free_`	A pointer to the head of the packet free list
`ref_count_`	The number of objects which currently refer to the packet
`next_`	A pointer to the next packet in the link list of packets
`hdr_len_`	Length of packet header

Program 8.1 Declaration of class `Packet`

```
//~/ns/common/packet.h
class Packet : public Event {
private:
    unsigned char* bits_;
    AppData* data_;
    static void init(Packet*) {bzero(p->bits_, hdrlen_);}
    bool fflag_;
protected:
    static Packet* free_;
    int    ref_count_;
public:
    Packet* next_;
    static int hdrlen_;

    //Packet Allocation and Deallocation
    Packet() : bits_(0), data_(0), ref_count_(0), next_(0) { }
    inline unsigned char* const bits() { return (bits_); }
    inline Packet* copy() const;
    inline Packet* refcopy() { ++ref_count_; return this; }
    inline int& ref_count() { return (ref_count_); }
    static inline Packet* alloc();
    static inline Packet* alloc(int);
    inline void allocdata(int);
    static inline void free(Packet*);

    //Packet Access
    inline unsigned char* access(int off){return &bits_[off]);};
}
```

– **C++ Functions of Class `Packet`**

`init(p)`	Clear the packet header `*bits_` of the input packet `*p`.
`copy()`	Return a pointer to a duplicated packet.
`refcopy()`	Increase the number of objects, which refer to the packet, by one.
`alloc()`	Create a new packet and return the pointer to the created packet.
`alloc(n)`	Create a new packet with "`n`" bytes of data payload and return a pointer to the created packet.
`allocdata(n)`	Allocate "`n`" bytes of data payload to the variable `data_`.
`free(p)`	Deallocate the packet "`p`."
`access(off)`	Retrieve a reference to a certain point (specified by the offset "`off`") of the variable "`bits_`" (i.e., packet header).

- **Protocol Specific Header:** From Fig. 8.1, packet header consists of several protocol-specific headers. Each protocol-specific header uses a contiguous portion of packet header to store its packet attributes. In common with most TclObjects, there are three classes related to each protocol-specific header:
  - A C++ class (e.g., `hdr_cmn` or `hdr_ip`) provides a structure to store packet attributes.
  - An OTcl class (e.g., `PacketHeader/Common` or `PacketHeader/IP`) acts as an interface to the OTcl domain. NS2 uses this class to configure packet header from the OTcl domain.
  - A mapping class (e.g., `CommonHeaderClass` or `IPHeaderClass`) binds a C++ class to an OTcl class.

  We will discuss the details of protocol-specific header later in Sect. 8.3.5.
- **Packet Header Manager:** A packet header manager maintains a list of active protocols and configures all active protocol-specific headers to setup packet header. It has an instvar "`hdrlen_`" which indicates the length of packet header consisting of protocol-specific headers. The instvar "`hdrlen_`" is bound to a variable "`hdrlen_`" of class `Packet`. Any change in one of these two variables will result in an automatic change in another.
- **Data Payload:** From Line 4 in Program 8.1, the pointer "`data_`" points to data payload, which is of class `AppData`. We will discuss the details of data payload in Sect. 8.4.

8.1.2 A Packet as an Event: A Delayed Packet Reception Event

Derived from class `Event` (Line 1 in Program 8.1), class `Packet` can be placed on the simulation time line (see the details in Chap. 4). In Sect. 4.2, we mentioned two main classes derived from class `Event`: class `AtEvent` and class `Packet`. We also mentioned that an `AtEvent` object is an event created by a user from a Tcl simulation script. This section discusses details of another derived class of class `Event`: class `Packet`.

As discussed in Sect. 5.2.2, NS2 implements *delayed packet forwarding* by placing a packet reception event on the simulation timeline at a certain delayed time. Derived from class `Event`, class `Packet` can be placed on the simulation timeline to signify a delayed packet reception. For example, the following statement (see Line 8 in Program 7.4) schedules a packet reception event, where the NsObject `*target_` receives a packet `*p` at `txt+delay_` seconds in future:

```
s.schedule(target_, p, txt + delay_)
```

Note that a `Packet` pointer is cast to be an `Event` pointer before being fed as the second input argument of the function `schedule(...)`.

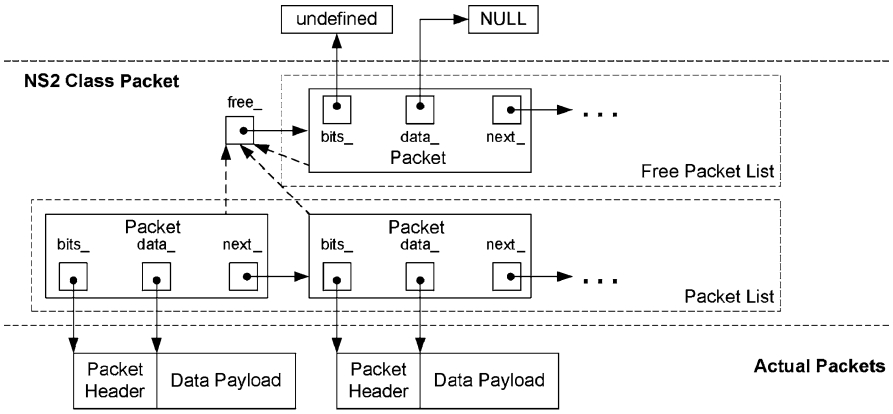

Fig. 8.2 A link list of packets and a free packet list

At the firing time, the Scheduler dispatches the scheduled event (i.e., `*p`) and invokes `target->handle(p)`, which executes "`target_->recv(p)`" to forward packet `*p` to the NsObject pointer `*target_`.

8.1.3 A Link List of Packets

Apart from the above four main packet components, a `Packet` object contains a pointer "`next_`" (Line 11 in Program 8.1), which helps formulating a link list of `Packet` objects (e.g., Packet List in Fig. 8.2). Program 8.2 shows the implementation of functions `enque(p)` and `deque()` of class `PacketQueue`. Function `enque(p)` (Lines 3–13) puts a `Packet` object `*p` to the end of the queue. If the `PacketQueue` is empty, NS2 sets "`hdrlen_`," "`tail_`," and "`p`" to point to the same place[2] (Line 5). Otherwise, Lines 7 and 8 set `*p` as the last packet in the `PacketQueue`, and shift variable "`tail_`" to the last packet pointer "`p`." Since the pointer "`tail_`" is the last pointer of `PacketQueue`, Line 10 sets the pointer `tail_->next_` to 0 (i.e, points to `NULL`).

Function `deque()` (Lines 14–21) retrieves a pointer to the packet at the head of the buffer. If there is no packet in the buffer, the function `deque()` will return a `NULL` pointer (Line 15). If the buffer is not empty, Line 17 will shift the pointer "`head_`" to the next packet, Line 19 will decrease the length of `PacketQueue` object by one, and Line 20 will return the `packet` pointer "`p`" which was set to the pointer "`head_`" in Line 16.

[2]Note that, `head_` and `tail_` are pointers to the first and the last `Packet` objects, respectively, in a `PacketQueue` object.

Program 8.2 Functions `enque` and `deque` of class `PacketQueue`

```
//~/ns/common/queue.h
class PacketQueue : public TclObject {
    ...
    virtual Packet* enque(Packet* p) {
        Packet* pt = tail_;
        if (!tail_) head_= tail_= p;
        else {
            tail_->next_= p;
            tail_= p;
        }
        tail_->next_= 0;
        ++len_;
        return pt;
    }

    virtual Packet* deque() {
        if (!head_) return 0;
        Packet* p = head_;
        head_= p->next_; // 0 if p == tail_
        if (p == tail_) head_= tail_= 0;
        --len_;
        return p;
    }
    ...
};
```

8.1.4 Free Packet List

Unlike most NS2 objects, a `Packet` object, once created, will not be destroyed until the simulation terminates. NS2 keeps `Packet` objects which are no longer in use in a *free packet list* (see Fig. 8.2). When NS2 needs a new packet, it first checks whether the free packet list is empty. If not, it will take a `Packet` object from the list. Otherwise, it will create another `Packet` object. We will discuss the details of how to *allocate* and *deallocate* a `Packet` object later in Sect. 8.2.

There are two variables which are closely related to the packet allocation/deallocation process: "`fflag_`" and "`free_`." Each `Packet` object uses a variable "`fflag_`" (Line 6 in Program 8.1) to indicate whether it is in use. The variable "`fflag_`" is set to `true`, when the `Packet` object is in use, and set to `false` otherwise. Shared by all the `Packet` objects, a static pointer "`free_`" (Line 8 in Program 8.1) is a pointer to the first packet on the free packet list. Each packet on the free packet list uses its variable "`next_`" to form a link list of free `Packet` objects. This link list of free packets is referred to as a *free packet list*. Although NS2 does not return memory allocated to a `Packet` object to the system, it does return the memory used by packet header (i.e., "`bits_`") and data payload (i.e., "`data_`") to the system (see Sect. 8.2.2), when the packet is deallocated. Since most memory required to store a `Packet` object is consumed by packet header and data payload, maintaining a free packet list does not result in a significant waste of memory.

8.2 Packet Allocation and Deallocation

Unlike most of the NS2 objects,[3] a `Packet` object is allocated and deallocated using static functions `alloc()` and `free(p)` of class `Packet`, respectively. If possible, function `alloc()` takes a `Packet` object from the free packet list. Only when the free packet list is empty, does the function `alloc()` creates a new `Packet` object using "`new`". Function `free(p)` deallocates a `Packet` object, by returning the memory allocated for packet header and data payload to the system and storing the not-in-use `Packet` pointer "`p`" in the free packet list for future reuse. The details of packet allocation and deallocation will be discussed below.

8.2.1 Packet Allocation

Program 8.3 shows details of the function `alloc()` of class `Packet`, the packet allocation function. The function `alloc()` returns a pointer to an allocated `Packet` object to the caller. This function consists of two parts: packet allocation in Lines 3–15 and packet initialization in Lines 16–22.

Consider the packet allocation in Lines 3–15. Line 3 declares "`p`" as a pointer to a `Packet` object and sets the pointer "`p`" to point to the first packet on the free packet list.[4] If the free packet list is empty (i.e., `p = 0`), NS2 will create a new `Packet` object (in Line 11) and allocate memory space with size "`hdrlen_`" bytes for the packet header in Line 12. The variable "`hdrlen_`" is not configured during the construction of a `Packet` object. Rather, it is set up in the Network Configuration Phase (see Sect. 8.3.8) and is used by the function `alloc()` to create packet header.

Function `alloc()` does not allocate memory space for data payload. When necessary, NS2 creates data payload using the function `allocdata(n)` (see Lines 8–14 in Program 8.4), which will be discussed in detail later in this section.

If the free packet list is nonempty, the function `alloc()` will execute Lines 5–9 in Program 8.3 (see also the diagram in Fig. 8.3). In this case, the function `alloc()` first makes sure that nobody is using the `Packet` object "`*p`," by asserting that

[3]Generally, NS2 creates and destroys most objects using procedures `new{...}` and `delete{...}`, respectively.

[4]Again, "`free_`" is the pointer to the first packet on the free packet list.

Program 8.3 Function `alloc` of class `Packet`

```
//~/ns/common/packet.h
inline Packet* Packet::alloc()
{
    //Packet Allocation
    Packet* p = free_;
    if (p != 0) {
        assert(p->fflag_ == FALSE);
        free_ = p->next_;
        assert(p->data_ == 0);
        p->uid_ = 0;
        p->time_ = 0;
    } else {
        p = new Packet;
        p->bits_ = new unsigned char[hdrlen_];
        if (p == 0 || p->bits_ == 0)
            abort();
    }

    //Packet Initialization
    init(p); // Initialize bits_[]
    (HDR_CMN(p))->next_hop_ = -2; // -1 reserved for
       IP_BROADCAST
    (HDR_CMN(p))->last_hop_ = -2; // -1 reserved for
       IP_BROADCAST
    p->fflag_ = TRUE;
    (HDR_CMN(p))->direction() = hdr_cmn::DOWN;
    p->next_ = 0;
    return (p);
}
```

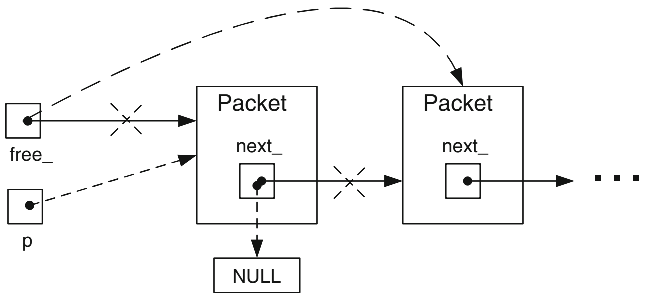

Fig. 8.3 Diagram of packet allocation when the free packet list is nonempty. The *dotted lines* show the actions caused by the function `alloc` of class `Packet`

Program 8.4 Functions `alloc`, `allocdata`, and `copy` of class `Packet`

```
    //~/ns/common/packet.h
 1  inline Packet* Packet::alloc(int n)
 2  {
 3      Packet* p = alloc();
 4      if (n > 0)
 5          p->allocdata(n);
 6      return (p);
 7  }

 8  inline void Packet::allocdata(int n)
 9  {
10      assert(data_ == 0);
11      data_ = new PacketData(n);
12      if (data_ == 0)
13          abort();
14  }

15  inline Packet* Packet::copy() const
16  {
17      Packet* p = alloc();
18      memcpy(p->bits(), bits_, hdrlen_);
19      if (data_)
20          p->data_ = data_->copy();
21      return (p);
22  }
```

"`fflag_`" is `false` (Line 5).[5] Then, Line 6 shifts the pointer "`free_`" by one position. Lines 8–9 initialize two variables ("`uid_`" and "`time_`") of class `Event` (i.e., the mother class of class `Packet`) to be zero. Line 21 removes the packet from the free list by setting `p->next_` to zero.

After the packet allocation process is complete, Lines 16–22 initialize the allocated `Packet` object. Line 16 invokes function `init(p)`, which initializes the header of packet `*p`. From Line 5 in Program 8.1, invocation of function `init(p)` executes "`bzero(p-> bits_,hdrlen_)`," which clears "`bits_`" to zero.[6] Line 19 sets `fflag_` to be `true`, indicating that the packet `*p` is now in use. Line 21 sets the pointer `p->next_` to be zero. Lines 17, 18, and 20 initialize the common header. We will discuss packet header in greater detail in Sect. 8.3.2.

[5]The C++ function `assert(cond)` can be used for an integrity check. It does nothing if the input argument "`cond`" is `true`. Otherwise, it will initiate an error handling process (e.g., showing an error on the screen).

[6]Function `bzero(...)` takes two arguments – the first is a pointer to the buffer and the second is the size of the buffer – and sets all values in a buffer to zero.

Apart from the function `alloc()`, other relevant functions include `alloc(n)`, `allocdata(n)`, and `copy()` (see Program 8.4). The function `allocdata(n)` data allocates a packet (Line 3), and invokes `allocdata(n)` (Line 5). The function `allocdata(n)` creates data payload with size "n" bytes (by invoking `new Packet Data(n)` in Line 11). We will discuss the details of data payload later in Sect. 8.4.

Function `copy()` returns a replica of the current `Packet` object. The only difference between the current and the replicated `Packet` objects is the unique ID (`uid_`) field. This function is quite useful, since we often need to create a packet which is the same as or slightly different from an original packet. This function first allocates a packet in Line 17. Then, it copies packet header and data payload to the created packet `*p` in Lines 18 and 20, respectively.

Despite its name, function `refcopy()` (Line 16 in Program 8.1) does not create a copy of a `Packet` object. Rather, it returns the pointer to the current `Packet` object and increment the variable `ref_count_` bg 1. The variable `ref_count_` keeps track of the number of objects which share the same `Packet` object. It is initialized to 0 in the constructor of class `Packet` (Line 13 in Program 8.1), and is incremented by one when the function `ref_copy()` (Line 16 in Program 8.1) is invoked, indicating that a new object starts using the current `Packet` object. Similarly, it is decremented by one when the function `free(p)` (see Sect. 8.2.2) is invoked, indicating that an object has stopped using the current `Packet` object.

8.2.2 *Packet Deallocation*

When a packet `*p` is no longer in use, NS2 deallocates the packet using a function `free(p)`. By deallocation, NS2 returns the memory used to store packet header and data payload to the system, sets the pointer "`data_`" to zero, and stores the `Packet` object in the free packet list. Note that although the value of "`bits_`" is not set to zero, the memory location stored in "`bits_`" is no longer accessible. It is very important not to use "`bits_`" after packet deallocation. Otherwise, NS2 will encounter a (memory share violation) runtime error.

Details of the function `free(Packet*)` are shown in Program 8.5. Before returning a `Packet` object to the free packet list, we need to make sure that

1. The packet is in use (i.e., `p->fflag_ = 1` in Line 3), since there is no point in deallocating a packet which has already been deallocated.
2. No object is using the packet. In other words, the variable `ref_count_` is zero (Line 4), where `ref_count_` stores the number of objects which are currently using this packet.
3. The packet is no longer on the simulation time line (i.e., `p->uid_<=0` in Line 5). Deallocating a packet while it is still on the simulation timeline will

Program 8.5 Function `free` of class `Packet`

```
    //~/ns/common/packet.h
 1  inline void Packet::free(Packet* p)
 2  {
 3      if (p->fflag_) {
 4          if (p->ref_count_ == 0) {
 5              assert(p->uid_ <= 0);
 6              if (p->data_ != 0) {
 7                  delete p->data_;
 8                  p->data_ = 0;
 9              }
10              init(p);
11              p->next_ = free_;
12              free_ = p;
13              p->fflag_ = FALSE;
14          } else {
15              --p->ref_count_;
16          }
17      }
18  }
```

cause event mis-sequencing and runtime error. Line 5 asserts that the event unique ID corresponding to the `Packet` object "p" (i.e., `p->uid_`) is non-positive, and therefore is no longer on the simulation timeline.[7]

NS2 allows more than one simulation object to share the same `Packet` object. To deallocate a packet, NS2 must ensure that the packet is no longer used by *any* simulation object. Again, NS2 keeps the number of objects sharing a packet in the variable `ref_count_`. If `ref_count_>0`, meaning an object is invoking the function `free(p)` while other objects are still using the packet `*p`, the function `free(p)` will simply reduce `ref_count_` by one, indicating that one object stops using the packet (Line 15).[8] On the other hand, if `ref_count_` is zero, meaning that no other object is using the packet, Lines 5–13 will then clear packet header and data payload and store the `Packet` object in the free packet list.

If all the above three conditions are satisfied, function `free(p)` will execute Lines 6–13 in Program 8.5. The schematic diagram for this part is shown in Fig. 8.4. Line 7 returns the memory used by data payload to the system. Line 8 sets the pointer "`data_`" to zero. Line 10 returns the memory used by header of the packet `*p` to the system by invoking the function `init(p)` (see Line 5 of Program 8.1). Lines 11 and 12 place the packet as the first packet on the free packet list. Finally, Line 13 sets "`fflag_`" to `false`, indicating that the packet is no longer in use.

[7]From Fig. 4.2, an event with positive unique ID (e.g, "`uid_`" is 2 or 6) was scheduled but has not been dispatched.

[8]If the `Packet` object is deallocated when `ref_count_ > 0`, simulation objects may later try to access the deallocated `Packet` object and cause a runtime error.

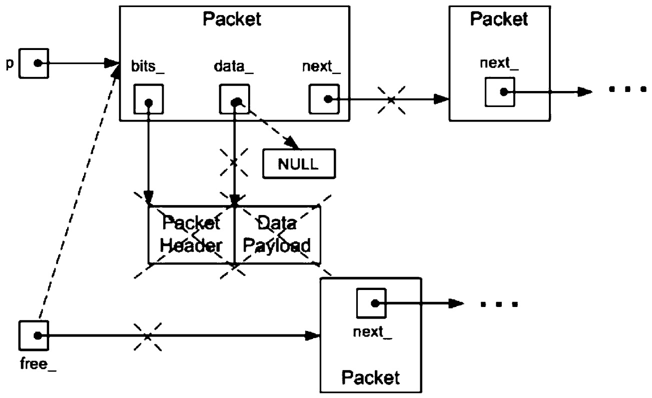

Fig. 8.4 The process of returning a packet to the packet free list. The *dotted lines* show the action caused by the function `free` of class `Packet`

8.3 Packet Header

As a part of a packet, packet header contains packet attributes such as packet unique ID and IP address. Again, packet header is stored in the variable "`bits_`" of class `Packet` (see Line 3 of Program 8.1). The variable "`bits_`" is declared as a string (i.e., a *Bag of Bits* (BOB)) and has no structure to store packet attributes. However, NS2 imposes a two-level structure on variable "`bits_`," as shown in Fig. 8.5.

The first level divides the entire packet header into protocol-specific headers. The location allocated to each protocol specific header on "`bits_`" is identified by its variable `offset_`. The second level imposes a packet attribute-storing structure on each protocol-specific header. On this level, packet attributes are stored as members of a C++ `struct` data type.

In practice, a packet contains only relevant protocol-specific headers. An NS2 packet, on the other hand, includes *all* protocol-specific headers into a packet header, regardless of packet type. Every packet uses the same amount of memory to store the packet header. The amount of memory is stored in the variable "`hdrlen_`" of class `Packet` in Line 12 of Program 8.1, and is declared as a static variable. The variable "`hdrlen_`" has no relationship to simulation packet size. For example, TCP and UPD packets may have different sizes. The values stored in the corresponding variable `hdr_cmn::size_` may be different; however, the values stored in the variable `Packet::hdrlen_` for both TCP and UDP packets are the same.

In the following, we first discuss the first level packet header composition in Sect. 8.3.1. Sections 8.3.2 and 8.3.3 show examples of protocol-specific headers: common packet header and IP packet header. Section 8.3.4 discusses one of the main packet attributes: payload type. Section 8.3.5 explains the details of protocol-specific header (i.e., the second level packet header composition). Section 8.3.6 demonstrates how packet attributes stored in packet header are

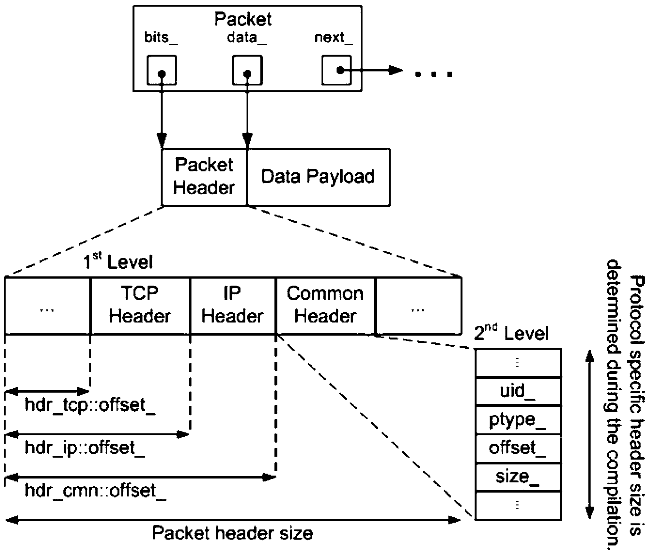

Fig. 8.5 Architecture of packet header: During the construction of the Simulator, the packet header size is determined and stored in the instvar `PacketHeaderManager::hdrlen_` which is bound to the variable `PacketHeaderManager::hdrlen_`. See the details in Step 2 in Sect. 8.3.8

accessed. Section 8.3.7 discusses one of the main packet header component, a packet header manager, which maintains the active protocol list and sets up the offset value for each protocol. Finally, Sect. 8.3.8 presents the packet header construction process.

8.3.1 An Overview of First Level Packet Composition: Offseting Protocol-Specific Header on the Packet Header

On the first level, NS2 puts together all relevant protocol-specific headers (e.g., common header, IP header, TCP header) and composes a packet header (see Fig. 8.5). Conceptually, NS2 allocates a contiguous part on the packet header for a protocol-specific header. Each protocol-specific header is offset from the beginning of packet header. The distance between the beginning of packet header and that of a protocol-specific header is stored in the member variable `offset_` of the protocol-specific header. For example, `hdr_cmn`, `hdr_ip`, and `hdr_tcp` – which represent common header, IP header, and TCP header – store their offset values in variables `hdr_cmn::offset_`, `hdr_ip::offset_`, and `hdr_tcp::offset_`, respectively.

Program 8.6 Declaration of C++ hdr_cmn struct data type

```
    //~/ns/common/packet.h
 1  struct hdr_cmn {
 2      enum dir_t { DOWN= -1, NONE= 0, UP= 1 };
 3      packet_t ptype_;     // payload type
 4      int size_;         // simulated packet size
 5      int uid_;          // unique id
 6      dir_t   direction_; // direction: 0=none, 1=up, -1=down
 7      static int offset_; // offset for this header

 8      inline static hdr_cmn* access(const Packet* p) {
 9          return (hdr_cmn*) p->access(offset_);
10      }
11      inline static int& offset() { return offset_; }
12      inline packet_t& ptype() { return (ptype_); }
13      inline int& size() { return (size_); }
14      inline int& uid() { return (uid_); }
15      inline dir_t& direction() { return (direction_); }
16  };
```

8.3.2 Common Packet Header

Common packet header contains packet attributes which are common to all packets. It uses C++ `struct` data type `hdr_cmn` to indicate how the packet attributes are stored. Program 8.6 shows a part of `hdr_cmn` declaration. The main member variables of `hdr_cmn` are as follows:

`ptype_`	The payload type (see Sect. 8.3.4).
`size_`	The packet size in bytes. Unlike actual packet transmission, the number of bits requires to hold a packet has no relationship to simulation packet size. During simulation, NS2 uses the variable `hdr_cmn::size_` as the packet size.
`uid_`	The ID which is unique to every packet.
`dir_t`	The direction to which the packet is moving. It is used mainly in wireless networks: A packet can move "`UP`" (i.e., `dir_t = 1`) toward higher layers or move "`DOWN`" toward lower layer components (i.e., `dir_t = -1`). It can also set to `0`, when not in use, by default, "`dir_t`" is set to "`DOWN`" (see Line 20 in Program 8.3).
`offset_`	The memory location relative to the beginning of packet header from which the common header is stored (see Sect. 8.3.1 and Fig. 8.5).

From Fig. 8.6, most functions of class `hdr_cmn` act as an interface to access its variables. Perhaps, the most important function of class `hdr_cmn` is the function `access(p)` in Lines 8–10. This function returns a pointer to the common protocol-specific header of the input `Packet` object `*p`. We will discuss the packet header access mechanism in greater detail in Sect. 8.3.6.

Program 8.7 Declaration of C++ hdr_ip struct data type

```
    //~/ns/common/ip.h
 1  struct hdr_ip {
 2      ns_addr_t   src_;
 3      ns_addr_t   dst_;
 4      int     ttl_;
 5      int     fid_;
 6      int     prio_;
 7      static int offset_;
 8      inline static int& offset() { return offset_; }
 9      inline static hdr_ip* access(const Packet* p) {
10          return (hdr_ip*) p->access(offset_);
11      }
12      ns_addr_t& src() { return (src_); }
13      nsaddr_t& saddr() { return (src_.addr_); }
14      int32_t& sport() { return src_.port_;}
15      ns_addr_t& dst() { return (dst_); }
16      nsaddr_t& daddr() { return (dst_.addr_); }
17      int32_t& dport() { return dst_.port_;}
18      int& ttl() { return (ttl_); }
19      int& flowid() { return (fid_); }
20      int& prio() { return (prio_); }
21  };
```

8.3.3 IP Packet Header

Represented by a C++ struct data type `hdr_ip`, IP packet header contains information about source and destination of a packet. Program 8.7 shows a part of `hdr_ip` declaration. IP packet header contains the following five main variables which contain IP-related packet information (see Lines 2–6 in Program 8.7):

`src_`	Source node's address indicated in the packet
`dst_`	Destination node's address indicated in the packet
`ttl_`	Time to live for the packet
`fid_`	Flow ID of the packet
`prio_`	Priority level of the packet

NS2 uses data type `ns_addr_t` defined in the file ~*ns*/config.h to store node address. From Program 8.8, `ns_addr_t` is a struct data type, which contains two members: `addr_` and `port_`. Both members are of type `int32_t`, which is simply an alias for `int` data type (see Line 5 and file ~*ns*/autoconf-win32.h). While `addr_` specifies the node address, `port_` identifies the attached port (if any).

The variables `src_` and `dst_` of IP header are of class `ns_addr_t`. Hence, "`src_.addr_`" and "`src_.port_`" store the node address and the port of the sending agent, respectively. Similarly, the packet will be sent to a receiving agent attached to port "`dst_.port_`" of a node with address "`dst_.addr_`."

Program 8.8 Declaration of C++ `ns_addr_t` struct data type, and `int32_t`

```
//~/ns/config.h
struct ns_addr_t {
    int32_t addr_;
    int32_t port_;
};

//~/ns/autoconf-win32.h
typedef int int32_t;
```

Lines 7–11 in Program 8.7 declare the variable `offset_`, function `offset (off)` and function `access(p)`, which are essential to access IP header of a packet. Lines 12–20 in Program 8.7 are functions that return the values of the variables.

8.3.4 *Payload Type*

Although stored in common header, payload type is attributed to the entire packet, not to a protocol-specific header. Each packet corresponds to only one payload type but may contain several protocol-specific headers. For example, a packet can be encapsulated by both TCP and IP protocols. However, its type can be either audio *or* TCP packet, *but not both*.

NS2 stores a payload type in a member variable `ptype_` of a common packet header. The type of the variable `ptype_` is enum `packet_t` defined in Program 8.9. Again, members of `enum` are integers which are mapped to strings. From Program 8.9, `PT_TCP` (Line 2) and `PT_UDP` (Line 3) are mapped to 0 and 1, respectively. Since `packet_t` declares `PT_NTYPE` (representing undefined payload type) as the last member, the value of `PT_NTYPE` is $N_p - 1$, where N_p is the number of `packet_t` members. NS2 provides 60 built-in payload types, meaning the default value of `PT_NTYPE` is 59.

From Lines 11–30 in Program 8.9, class `p_info` maps each member of `packet_t` to a description string. It has a static associative array variable, `name_` (Line 28). The index and value of `name_` are the payload type and the corresponding description string, respectively. Class `p_info` also has one important function `name(p)` (Lines 23–26), which translates a `packet_t` variable to a description string.

At the declaration, NS2 declares a global variable `packet_info` (using `extern`), which is of class `p_info` (Line 30). Accessible at the global scope, the variable `packet_info` provides an access to the function `name(p)` of class `p_info`. To obtain a description string of a `packet_t` object "p," one may invoke

```
packet_info.name(ptype)
```

Program 8.9 Declaration of `enum packet_t` type and class `p_info`

```
//~/ns/common/packet.h
enum packet_t {
    PT_TCP,
    PT_UDP,
    PT_CBR,
    PT_AUDIO,
    PT_VIDEO,
    PT_ACK,
    ...
    PT_NTYPE // This MUST be the LAST one
}

class p_info {
public:
    p_info() {
        name_[PT_TCP]= "tcp";
        name_[PT_UDP]= "udp";
        name_[PT_CBR]= "cbr";
        name_[PT_AUDIO]= "audio";
        name_[PT_VIDEO]= "video";
        name_[PT_ACK]= "ack";
        ...
        name_[PT_NTYPE]= "undefined";
    }
    const char* name(packet_t p) const {
        if ( p <= PT_NTYPE ) return name_[p];
        return 0;
    }
private:
    static char* name_[PT_NTYPE+1];
};
extern p_info packet_info; /* map PT_* to string name */
```

Example 8.1. Class `Agent` is responsible for creating and destroying network layer packets (see Chap. 9). It is the base class of TCP and UDP transport layer protocol modules. Class `Agent` provides a function `allocpkt()`, which is responsible for allocating (i.e., creating) a packet.

To print out the type of every allocated packet on the screen, we modify function `allocpkt()` of class `Agent` in file ~*ns*/common/agent.cc as follows:

```
//~/ns/common/agent.h
Packet* Agent::allocpkt() const
{
    Packet* p = Packet::alloc();
    initpkt(p);
    /*----- Begin Additional Codes -----*/
    hdr_cmn* ch = hdr_cmn::access(p);
    packet_t pt = ch->ptype();
8       printf("Example Test: Class Agent allocates a
                             packet with type %s\n", packet_
                               info.name(pt));
9       getchar();
10      /*----- End Additional Codes -----*/
11      return (p);
12 }
```

where Lines 5–10 are added to the original codes. Line 6 retrieves the reference "ch" to the common packet header (see Sect. 8.3.6). Line 7 obtains the payload type stored in the common header using the function ptype(), and assigns the payload type to variable "pt." Note that, the variable packet_info is a global variable of class p_info. When the variable "pt" is fed as an input argument, the statement packet_info.name(pt) returns the description string corresponding to the packet_t object "pt" (Line 8).

After recompiling the code, the simulation should show the type of every allocated packet on the screen. For example, when running the Tcl simulation script in Programs 2.1–2.2, the following result should appear on the screen:

```
>> ns myfirst_ns.tcl
Example Test: Class Agent allocates a packet with type cbr
Example Test: Class Agent allocates a packet with type cbr
Example Test: Class Agent allocates a packet with type cbr
...
```

8.3.5 Protocol-Specific Headers

A protocol-specific header stores packet attributes relevant to the underlying protocol only. For example, common packet header holds basic packet attributes such as packet unique ID, packet size, payload type, and so on. IP packet header contains IP packet attributes such as source and destination IP addresses and port numbers. There are 48 classifications of packet headers. The complete list of protocol-specific headers with their descriptions is given in [17].

Each protocol-specific header involves three classes: A C++ class, and OTcl class, and a mapping class.

8.3.5.1 Protocol-Specific Header C++ Classes

In C++, NS2 uses a struct data type to represent a protocol-specific header. It stores packet attributes and its offset value in members of the struct data type. It also provides a function access(p) which returns the reference to the protocol-specific header of a packet *p. Representing a protocol specific header, each `struct` data type is named using the format `hdr_<XXX>`, where `<XXX>` is an arbitrary string representing the type of a protocol-specific header. For example, the C++ class name for common packet header is `hdr_cmn`.

In the C++ domain, protocol specific headers are declared but not instantiated. Therefore, NS2 uses a `struct` data type (rather than a class) to represent protocol-specific headers. No constructor is required for a protocol-specific header. Hereafter, we will refer to `struct` and `class` interchangeably.

8.3.5.2 A Protocol-Specific Header OTcl Class

NS2 defines a shadow OTcl class for each C++ protocol specific header class. An OTcl class acts as an interface to the OTcl domain. It is named with the format `PacketHeader/<XXX>`, where `<XXX>` is an arbitrary string representing a protocol-specific header. For example, the OTcl class name for common packet header is `PacketHeader/Common`.

8.3.5.3 A Protocol-Specific Header Mapping Class

A mapping class is responsible for binding OTcl and C++ class names together. All the packet header mapping classes derive from class `PacketHeaderClass` which is a child class of class `TclClass`. A mapping class is named with format `<XXX>HeaderClass`, where `<XXX>` is an arbitrary string representing a protocol-specific header. For example, the mapping class name for common packet header is `CommonHeaderClass`.

Program 8.10 shows the declaration of class `PacketHeaderClass`, which has two key variables: `hrdlen_` in Line 8 and `offset_` in Line 9. The variable "`hdrlen_`" represents the length of the protocol-specific header.[9] It is the system memory needed to store a protocol-specific header C++ class. The variable `offset_` indicates the location on packet header where the protocol-specific header is used.

The constructor of class `PacketHeaderClass` in Lines 3 and 4 takes two input arguments. The first input argument `classname` is the name of the corresponding OTcl class name (e.g., `PacketHeader/Common`). The second one, `hdrlen`, is the length of the protocol-specific header C++ class. In Lines 3 and 4, the constructor feeds `classname` to the constructor of class `TclClass`, stores `hdrlen` in the member variable `hdrlen_`, and resets `offset_` to zero.

Function `method(argc,argv)` in Line 5 is an approach to take a C++ action from the OTcl domain. Functions `bind_offset(off)` in Line 6 and `offset(off)` in Line 7 are used to configure and retrieve, respectively, value

[9] While the variable `hdrlen_` in class `PacketHeaderClass` represents the length of a protocol specific header, the variable `hdrlen_` in class `Packet` represents total length of packet header.

Program 8.10 Declaration of class `PacketHeaderClass`

```
    //~/ns/common/packet.h
 1  class PacketHeaderClass : public TclClass {
 2  protected:
 3      PacketHeaderClass(const char* classname, int hdrlen) :
 4               TclClass(classname), hdrlen_(hdrlen),
                 offset_(0);{};
 5      virtual int method(int argc, const char*const* argv);
 6      inline void bind_offset(int* off) { offset_ = off; };
 7      inline void offset(int* off) {offset_= off;};
 8      int hdrlen_;         // # of bytes for this header
 9      int* offset_;        // offset for this header
10  public:
11      TclObject* create(int argc,const char*const* argv)
          {return 0;};
12      virtual void bind(){
13          TclClass::bind();
14          Tcl& tcl = Tcl::instance();
15          tcl.evalf("%s set hdrlen_ %d", classname_, hdrlen_);
16          add_method("offset");
17      };
18  };
```

of the variable "`offset_`". Function `create(argc,argv)` in Line 11 does nothing, since no protocol-specific header C++ object is created. It will be overridden by the derived classes of class `PacketHeaderClass`. Function `bind()` in Lines 12–17 glues the C++ class to the OTcl class. Line 13 first invokes the function `bind()` of class `TclClass`, which performs the basic binding actions. Line 15 exports variable "`hdrlen_`" to the OTcl domain. Line 16 registers the *OTcl method* `offset` using function `add_method`("`offset`").

Apart from the OTcl commands discussed in Sect. 3.4, an *OTcl method* is another way to invoke C++ functions from the OTcl domain. It is implemented in C++ via the following two steps. The first step is to define a function `method(ac,av)`. As can be seen from Program 8.11, the structure of function `method` is very similar to that of the function `command`. A *method* "`offset`" stores the input argument in the variable `*offset_` (Line 7 in Program 8.11). The second step in method implementation is to register the name of the *method* using a function "`add_method(str)`," which takes the method name as an input argument. For class `PacketHeaderClass`, the *method* `offset` is registered from within function `bind(...)` (Line 16 of Program 8.10).

A protocol-specific header is implemented using a `struct` data type, and hence does not derive function `command(...)` from class `TclObject`.[10] It resorts to OTcl *methods* defined in the mapping class to take C++ actions from the OTcl

[10]Since NS2 does not instantiate a protocol specific header object, it models a protocol specific header using `struct` data type.

Program 8.11 Function `method` of class `PacketHeaderClass`

```
//~/ns/common/packet.cc
int PacketHeaderClass::method(int ac, const char*const* av)
{
    Tcl& tcl = Tcl::instance();
    ...
    if (strcmp(argv[1], "offset") == 0) {
        if (offset_) {
            *offset_ = atoi(argv[2]);
            return TCL_OK;
        }
        tcl.resultf("Warning: cannot set
                          offset_ for %s",classname_);
        return TCL_OK;
    }
    ...
    return TclClass::method(ac, av);
}
```

Table 8.1 Classes and objects related to common packet header

Class/object	Name
C++ class	`hdr_cmn`
OTcl class	`PacketHeader/Common`
Mapping class	`CommonHeaderClass`
Mapping variable	`class_cmnhdr`

domain. We will show an example use of the method `offset` later in Sect. 8.3.8, when we discuss packet construction mechanism.

Consider, for example, a common packet header. Its C++, OTcl, and mapping classes are `hdr_cmn`, `PacketHeader/Common`, and `CommonPacketHeader Class`, respectively (see Table 8.1). Program 8.12 shows the declaration of class `CommonPacketHeaderClass`. As a child class of `TclClass`, a class mapping variable `class_cmnhdr` is instantiated at the declaration. Line 3 of the constructor invokes the constructor of its parent class `PacketHeaderClass`, which takes the OTcl class name (i.e., `PacketHeader/Common`) and the amount of memory needed to hold the C++ class (i.e., `hdr_cmn`) as input arguments. Here, "`sizeof (hdr_cmn)`" computes the required amount of memory, for `hdr_cmn`. The result of this statement is fed as the second input argument. In Line 6 of Program 8.10, the statement `bind_offset(&hdr_cmn::offset_)` sets the variable `offset_` to share the address with the input argument. Therefore, a change in `hdr_cmn::offset_` will result in an automatic change in the variable `*offset_` of class `CommonHeaderClass`, and vice versa.

Program 8.12 Declaration of class `CommonHeaderClass`

```
//~/ns/common/packet.cc
class CommonHeaderClass : public PacketHeaderClass {
public:
    CommonHeaderClass() : PacketHeaderClass("PacketHeader/
                              Common", sizeof(hdr_cmn)) {
        bind_offset(&hdr_cmn::offset_);
    }
} class_cmnhdr;
```

8.3.6 Packet Header Access Mechanism

This section demonstrates how packet attributes stored in packet header can be retrieved and modified. NS2 uses a two-level packet header structure to store packet attributes. On the first level, protocol-specific headers are stored within a packet header. On the second level, each protocol-specific header uses a C++ `struct` data type to store packet attributes.

Before proceeding further, let us have a look at how packet header can be modified.

Example 8.2. Given a pointer to a `Packet` object `*p`, the following statements set the packet size to be 1000 bytes.

```
hdr_cmn* ch = hdr_cmn::access(p);
ch->size_ = 1000;
```

The upper line retrieves the reference to the common header and stores the reference in the pointer "`ch`," while the lower line modifies the packet size through the field `size_` of the common packet header (through `*ch`). □

The header access mechanism consists of two major steps: (1) Retrieve a reference to a protocol-specific header, and (2) Follow the structure of the protocol-specific header to retrieve or modify packet attributes. In this section, we will explain the access mechanism through common packet header (see the corresponding class names in Table 8.1).

8.3.6.1 Retrieving a Reference to Protocol-Specific Header

NS2 obtains a reference to a protocol-specific header of a packet `*p` using a function `access(p)` of the C++ class `hdr_cmn`.

Example 8.3. Consider function `allocpkt()` of class `Agent` shown in Program 8.13, which shows the details of functions `allocpkt()` and

Program 8.13 Functions `allocpkt` and `initpkt` of class `Agent`

```
    //~/ns/common/agent.cc
 1  Packet* Agent::allocpkt() const
 2  {
 3      Packet* p = Packet::alloc();
 4      initpkt(p);
 5      return (p);
 6  }

 7  Packet* Agent::initpkt(Packet* p) const
 8  {
 9      hdr_cmn* ch = hdr_cmn::access(p);
10      ch->uid() = uidcnt_++;
11      ch->ptype() = type_;
12      ch->size() = size_;
13      ...
14      hdr_ip* iph = hdr_ip::access(p);
15      iph->saddr() = here_.addr_;
16      iph->sport() = here_.port_;
17      iph->daddr() = dst_.addr_;
18      iph->dport() = dst_.port_;
19      ...
20  }
```

`initpkt(p)`. Function `allocpkt()` in Lines 1–6 creates a `Packet` object and returns a pointer to the created object. It first invokes function `alloc()` of class `Packet` in Line 3 (see the details in Sect. 8.2.1). Then, Line 4 initializes the allocated packet by invoking the function `initpkt(p)`. Finally, Line 5 returns the pointer "`p`" which points to the initialized `Packet` object.

Function `initpkt(p)` follows the structure defined in the protocol-specific header C++ classes to set packet attributes to the default values. Lines 9 and 14 in Program 8.13 execute the first step in the access mechanism: retrieve references to common packet header "`ch`" and IP header "`iph`," respectively.

After obtaining the pointers "`ch`" and "`iph`," Lines 10–12 and Lines 15–18 carry out the second step in the access mechanism: access packet attributes through the structure defined in the protocol-specific headers. In this step, the relevant packet attributes such as unique packet ID, payload type, packet size, source IP address and port, and destination IP address and port, are configured through the pointers "`ch`" and "`iph`." Note that `uidcnt` (i.e., uid count) is a static member variable of class `Agent` which represents the total number of generated packets. We will discuss the details of class `Agent` later in Chap. 9. □

Figure 8.6 shows an internal mechanism of the function `hdr_cmn::access(p)` where "`p`" is a `Packet` pointer. When `hdr_cmn::access(p)` is executed Line 9 in Program 8.6 executes `p->access(offset_)`, where `offset_` is the member variable of class `hdr_cmn`, specifying the location on

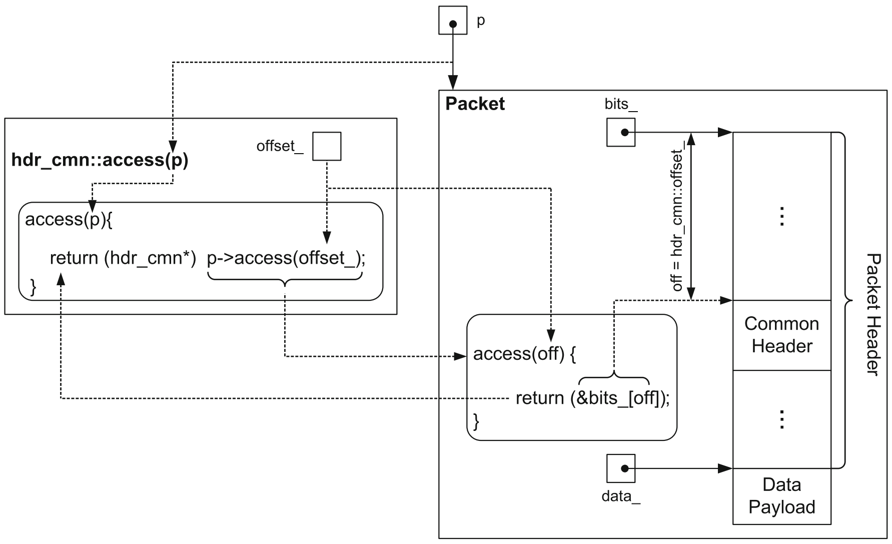

Fig. 8.6 The internal mechanism of the function `access(p)` of the `hdr_cmn` `struct` data type, where "`p`" is a pointer to a `Packet` object

the packet header allocated to the common header (see also Fig. 8.5). On the right-hand side of Fig. 8.6, the function `access(off)` simply returns `&bits_[off]`, where "`bits_`" is the member variable of class `Packet` storing the entire packet header. Since the input argument `offset_` belongs to `hdr_cmn`, the statement `access(offset_)` essentially returns `&bits_[hdr_cmn::offset_]`, which is the reference to the common header stored in the `Packet` object `*p`. This reference is returned as an `unsigned char*` variable. Then, class `hdr_cmn` casts the returned reference to `hdr_cmn*` data type and returns it to the caller.

Note that NS2 simplifies the retrival of protocal specific header reference, by defining pre-processing statements:

```
//~ns/common/packet.h
#define HDR_CMN(p) chdr_cmn::access(p))
#define HDR_ARP(p) chdr_arp::access(p))
...
```

8.3.6.2 Accessing Packet Attributes in a Protocol-Specific Header

After obtaining a reference to a protocol-specific header, the second step is to access the packet attributes according to the structure specified in the protocol-specific

Program 8.14 Declarations of C++ class `PacketHeaderManager` and mapping class `PacketHeaderManagerClass`

```
    //~/ns/common/packet.cc
 1  class PacketHeaderManager : public TclObject {
 2  public:
 3      PacketHeaderManager() {bind("hdrlen_",
                                 &Packet::hdrlen_);}
 4  };

 5  static class PacketHeaderManagerClass : public TclClass {
 6  public:
 7      PacketHeaderManagerClass() :
                                TclClass("PacketHeaderManager") {}
 8      TclObject* create(int, const char*const*) {
 9          return (new PacketHeaderManager);
10      }
11  } class_packethdr_mgr;
```

header C++ class. Since NS2 declares a protocol-specific header as a `struct` data type, it is fairly straightforward to access packet attributes once the reference to the protocol-specific header is obtained (see Example 8.3).

8.3.7 Packet Header Manager

A packet header manager is responsible for keeping the list of active protocols and setting the offset values of all the active protocols. It is implemented using a C++ class `PacketHeaderManager` which is bound to an OTcl class with the same name. Program 8.14 and Fig. 8.7 show the declaration of the C++ class `PacketHeaderManager` as well as the corresponding binding class, and the diagram of the OTcl class `PacketHeaderManager`, respectively.

The C++ class `PacketHeaderManager` has one constructor (Line 3) and has neither variables nor functions. The constructor binds the instvar “`hdrlen_`” of the OTcl class `PacketHeaderManager` to the variable “`hdrlen_`” of class `Packet` (see also Fig. 8.1). The OTcl class `PacketHeader Manager` has two main instvars: “`hdrlen_`” and “`tab_`.” The instvar “`hdrlen_`” stores the length of packet header. It is initialized to zero in Line 1 of Program 8.15, and is incremented as protocol-specific headers are added to the packet header. Representing the active protocol list, the instvar “`tab_`” (Line 2 in Program 8.16) is an associative array whose indexes are protocol-specific header OTcl class names and values are 1 if the protocol-specific header is active (see Line 12 in Program 8.5). If the protocol-specific header is inactive, the corresponding value of “`tab_`” will not be available (i.e., NS2 `unset`s all entries corresponding to inactive protocol-specific headers; see Line 7 in Program 8.20).

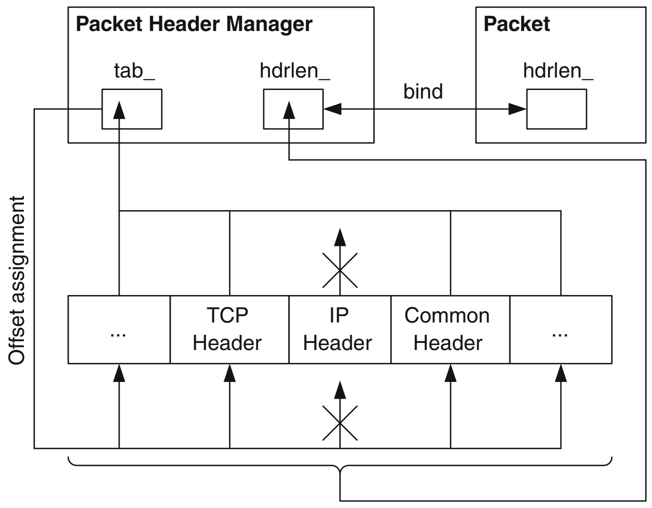

Fig. 8.7 Architecture of an OTcl `PacketHeaderManager` object

Program 8.15 Initialization of a `PacketHeaderManager` object

```
//~/ns/tcl/lib/ns-packet.tcl
PacketHeaderManager set hdrlen_ 0

foreach prot {
    Common
    Flags
    IP
   ...
}  {
    add-packet-header $prot
}

proc add-packet-header args {
    foreach cl $args {
        PacketHeaderManager set tab_(PacketHeader/$cl) 1
    }
}
```

8.3.8 *Protocol-Specific Header Composition and Packet Header Construction*

Packet header is constructed through the following three-step process:

Program 8.16 Function `create_packetformat` of class `Simulator` and function `allochdr` of class `PacketHeaderManager`

```
//~/ns/tcl/lib/ns-packet.tcl
Simulator instproc create_packetformat { } {
    PacketHeaderManager instvar tab_
    set pm [new PacketHeaderManager]
    foreach cl [PacketHeader info subclass] {
        if [info exists tab_($cl)] {
            set off [$pm allochdr $cl]
            $cl offset $off
        }
    }
    $self set packetManager_ $pm
}

PacketHeaderManager instproc allochdr cl {
    set size [$cl set hdrlen_]
    $self instvar hdrlen_
    set NS_ALIGN 8
    set incr [expr ($size + ($NS_ALIGN-1)) & ~($NS_ALIGN-1)]
    set base $hdrlen_
    incr hdrlen_ $incr
    return $base
}
```

Step 1: At the Compilation Time

During the compilation, NS2 translates all C++ codes into an executable file. It sets up all necessary variables (including the length of all protocol-specific headers) for all built-in protocol-specific headers, and includes all built-in protocol-specific headers into the active protocol list. There are three main tasks in this step.

Task 1: Construct All Mapping Variables, Configure the Variable `hdrlen_`, Register the OTcl Class Name, and Binds the Offset Value

Since all mapping variables are instantiated at the declaration, they are constructed during the compilation using their constructors. As an example, consider the common packet header[11] whose construction process shown in Program 8.10, Program 8.12, and Fig. 8.8 proceeds as follows:

1. Store the corresponding OTcl class name (e.g., `PacketHeader/Common`) in the variable `classname_` of class `TclClass`.
2. Determine the size (using function `sizeof (...)`) of the protocol-specific header, and store it in the variable "`hdrlen_`" of class `PacketHeaderClass`.
3. Bind the variable `offset_` of the `PacketHeader` to that of class `hdr_cmn`.

[11]NS2 repeats the following process for all protocol specific headers. For brevity, we show the construction process through common packet header only.

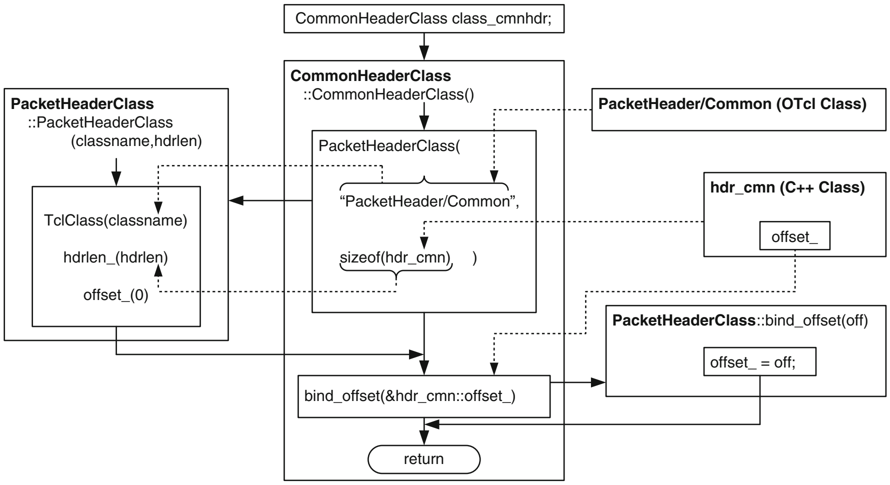
CommonHeaderClass class_cmnhdr;
CommonHeaderClass ::CommonHeaderClass()
PacketHeaderClass ::PacketHeaderClass (classname,hdrlen)
PacketHeader/Common (OTcl Class)
PacketHeaderClass(
"PacketHeader/Common",
sizeof(hdr_cmn))
TclClass(classname)
hdrlen_(hdrlen)
offset_(0)
hdr_cmn (C++ Class)
offset_
PacketHeaderClass::bind_offset(off)
offset_ = off;
bind_offset(&hdr_cmn::offset_)
return

Fig. 8.8 Construction of the static mapping variable `class_cmnhdr`

Task 2: Invocation of Function `bind()` of Class `TclClass` Which Exports the Variable `hdrlen_`

The main NS2 function (i.e., `main(argc,argv)`) invokes the function `init(...)` of class `Tcl`, which in turn invokes the function `bind()` of class `TclClass` of all mapping variables. The function `bind()` registers and binds an OTcl class name to the C++ domain (see file *~tclcl*/Tcl.cc). This function is overridden by class `PacketHeaderClass`.

As shown in Lines 12–17 of Program 8.10, class `PacketHeaderClass` overrides function `bind()` of class `TclClass`. Line 13 first invokes the function `bind()` of class `TclClass`. Line 15 exports the variable "`hdrlen_`" to the OTcl domain. Finally, Line 16 registers the OTcl method `offset`.

In case of class `CommonHeaderClass`, `classname_` is `PacketHeader/Common` and "`hdrlen_`" is 104 bytes. Therefore, Line 15 of Program 8.10 executes the following OTcl statement:

```
PacketHeader/Common set hdrlen_ 104
```

which sets instvar "`hdrlen_`" of class `PacketHeader/Common` to be 104. Note that this instvar "`hdrlen_`" is not bound to the C++ domain.

After Task 1 and Task 2 are completed, the related protocol-specific classes, namely `hdr_cmn`, `PacketHeader/Common`, and `CommonHeaderClass`, would be as shown in Fig. 8.9. The mapping object `class_cmnhdr` is of class `CommonHeaderClass`, which derives from classes `PacketHeaderClass` and `TclClass`, respectively. It inherits variables `classname_`, `hdrlen_`, and `offset_` from its parent class. After object construction is complete, variable `classname_` will store the name of the OTcl common packet header

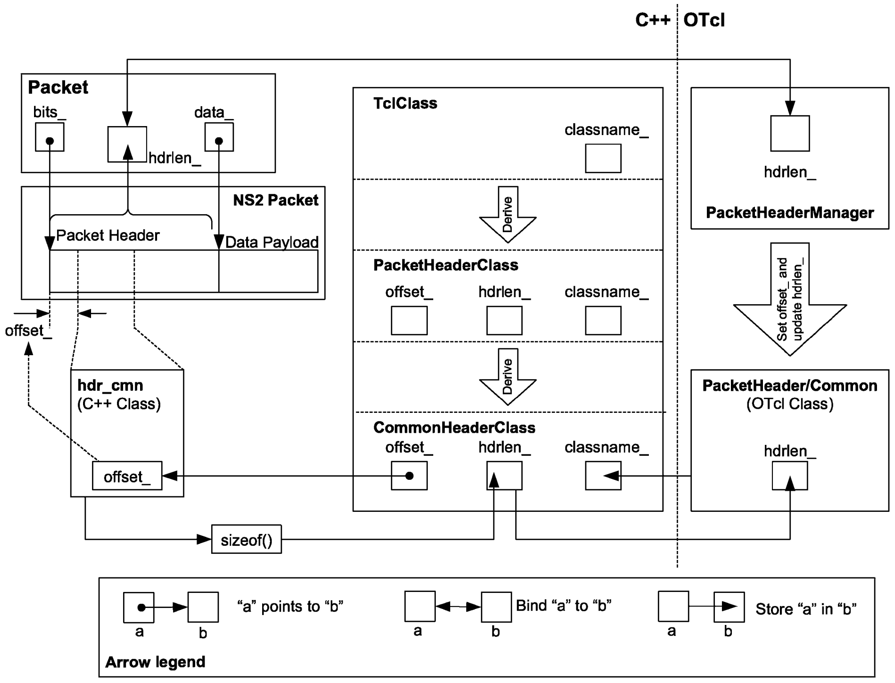

Fig. 8.9 A schematic diagram of a static mapping object `class_cmnhdr`, class `hdr_cmn`, class `PacketHeader/Common`, and class `Packet`

class (i.e., `PacketHeader/Common`), `hdr_len_` will store the amount of memory in bytes needed to store common header, and `offset_` will point to `hdr_cmn:: offset_`. However, at this moment, the offset value is set to zero. The dashed arrow in Fig. 8.9 indicates that the value of variable `hdr_cmn:: offset_` will be later set to store an offset from the beginning of a packet header to the point where the common packet header is stored. Also, after the function `Tcl::init()` invokes the function `bind()` of class `PacketHeaderClass`, the instvar "`hdrlen_`" of class `PacketHeader/Common` will store the value of the variable "`hdrlen_`" of class `CommonHeaderClass`. Note that tasks 1 and 2 only set up C++ OTcl class, and mapping class. However, the packet header manager is not configured at this phase.

Task 3: Sourcing the File ~ns*/tcl/lib/ns-packet.tcl to Setup an Active Protocol List*

As discussed in Sect. 3.7, NS2 sources all scripting Tcl files during the compilation process. In regards to packet header, Program 8.15 shows a part of the file ~*ns*/tcl/lib/ns-packet. Here, Line 8 invokes procedure `add-packet-`

header{prot} for all built-in protocol-specific headers indicated in Lines 3–6. Line 12 sets the value of the associative array "tab_" whose index is the input protocol-specific header name to be 1.

Step 2: During the Network Configuration Phase

In regards to packet header construction, the main task in the Network Configuration Phase is to setup variables offset_ of all active protocol-specific headers and formulate a packet header format. Subsequent packet creation will follow the packet format created in this step.

The offset configuration process takes place during the simulator construction. From Line 2 of Program 4.11, the constructor of the Simulator invokes the instproc create_packetformat{} of class Simulator.

As shown in Program 8.16, the instproc create_packetformat{} creates a PacketHeaderManager object "pm" (Line 3). Here, the constructor of C++ class PacketHeaderManager is invoked. From Program 8.14, the constructor binds its OTcl instvar "hdrlen_" to the variable "hdrlen_" of the C++ class Packet.

After creating a PacketHeaderManager object "pm," the instproc create_packetformat{} computes the offset value of all active protocol-specific headers using the instproc allochdr{cl} (Line 6), and configures the offset values of all protocol specific headers (Line 7). The foreach loop in Line 4 runs for all built-in protocol-specific headers which are child classes of class PacketHeader. Line 5 filters out those which are not in the active protocol list (see Sect. 8.3.7). Lines 6 and 7 are executed for all active protocol-specific headers specified in the variable "tab_" (which was configured in Step 1 – Task 3) of the PacketHeaderManager object "pm." Line 7 configures offset values using the OTcl *method* offset (see Program 8.11) of protocol specific header mapping classes. The OTcl *method* offset stores the input argument in the variable *offset_ of the protocol-specific header mapping class (e.g., CommonHeaderClass).

Lines 12–19 in Program 8.16 and Fig. 8.10 show the OTcl source codes and the diagram, respectively, of the instproc allochdr{cl} of an OTcl class PacketHeaderManager. The instproc allochdr{cl} takes one input argument "cl" (in Line 12) which is the name of a protocol-specific header, computes the memory requirement, and returns the offset value corresponding to the input argument "cl." Line 13 stores header length of a protocol-specific header "cl" (e.g., the variable "hdrlen_" of class PacketHeader/Common) in a local variable "size."[12] Based on "size," Lines 15 and 16 compute the amount of

[12]The variable hdrlen_ of a protocol specific header OTcl class was configured in Step 1 – Task 2.

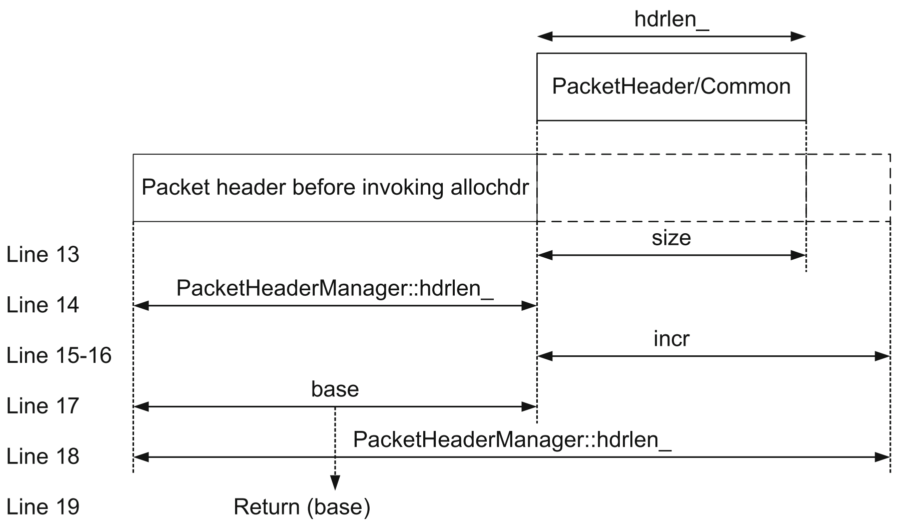

Fig. 8.10 A diagram representing the instproc `allochdr` of class `PacketHeaderManager`. Line numbers shown on the *left* correspond to the lines in Program 8.16. The action corresponding to each line is shown on the *right*

memory (`incr`) needed to store the header.[13] Line 17 stores the current packet header length (excluding the input protocol-specific header) in a local variable "`base`." Since "`base`" is an offset distance from the beginning of packet header to the input protocol-specific header, it is returned to the caller as the offset value in Line 19. After Line 18 increases the header length (i.e., the instvar "`hdrlen_`" of class `PacketHeaderManager`) by "`incr`."

During the `Simulator` construction, the packet header manager also updates its variable "`hdrlen_`" (Line 19 in Program 8.16). Note that the instvar "`hdrlen_`" of class `PacketHeaderManager` was set to zero at the compilation (Line 1 in Program 8.15). As Lines 6 and 7 in Program 8.16 repeat for every protocol-specific header, the offset value is added to the instvar "`hdrlen_`" of an OTcl class `PacketHeaderManager`. At the end, the instvar "`hdrlen_`" will represent the total header length, which embraces all protocol-specific headers.

Step 3: During the Simulation Phase

During the Simulation Phase, NS2 creates packets based on the format defined in the former two steps. For example, an `Agent` object creates and initializes a packet using its function `allocpkt()`. Here, a packet is created using the function

[13]The variable "`incr`" could be greater than "`size`," depending on the underlying hardware.

alloc() of class Packet and initialized using the function initpkt(p) of class Agent. Again, the function alloc() takes a packet from the free packet list, if it is nonempty. Otherwise, it will create a new packet using "new". After retrieving a packet, it clears the values stored in the packet header and data payload. The function initpkt(p) assigns default values to packet attributes such as packet unique ID, payload type, and packet size (see Program 8.13). The initialization is performed by retrieving a reference (e.g., "ch") to the relevant protocol-specific header and accessing packet attributes using the predefined structure.

8.4 Data Payload

Implementation of data payload in NS2 differs from actual data payload. In practice, user information is transformed into bits and are stored in data payload. Such the transformation is not necessary in simulation, since NS2 stores the user information in the packet header. NS2 rarely needs to maintain data payload. In Line 11 of Program 7.3, packet transmission time, i.e., the time required to send out a packet, is computed as $\frac{\text{packet size}}{\text{bandwidth}}$. Class LinkDelay determines the size of a packet by hdr_cmn::size_ (not by counting the number of bits stored in packet header and data payload) to compute packet transmission time. In most cases, users do not need to explicity deal with data payload.

NS2 also provides a support to hold data payload. In Line 4 of Program 8.1, class Packet provides a pointer "data_" to an AppData object.[14] Program 8.17 shows the declaration of an abstract class AppData. Class AppData has only one member variable type_ in Line 11. Among its functions, and one is a pure virtual function copy() shown in Line 18. Indicating the type of application, the variable type_ is of type enum AppDataType defined in Lines 1–8. The function copy() duplicates an AppData object to a new AppData object. It is a pure virtual function, and must be overridden by child instantiable classes of class AppData. Function size() in Line 17 returns the amount of memory required to store an AppData object.

Class AppData provides two constructors. One is in Line 13, where the caller feeds an AppData type as an input argument. Another is in Line 14, where a reference to a AppData object is fed as an input argument. In both the cases, the constructor simply sets the variable type_ to a value as specified in the input argument.

Program 8.18 shows the declaration of class PacketData, a child class of class AppData. This class has two new member variables: "data_" (a string variable which stores data payload) in Line 3 and datalen_ (the length of "data_") in Line 4. Line 25 defines a function data() which simply returns "data_." Lines 26 and 27 override the virtual functions size() and copy(), respectively, of

[14]However, no memory is allocated to the AppData object unless it is needed.

Program 8.17 Declaration of `enum AppDataType` and class `AppData`

```
    //~/ns/common/ns-process.h
 1  enum AppDataType {
 2       ...
 3       PACKET_DATA,
 4       HTTP_DATA,
 5       ...
 6       ADU_LAST
 7
 8  };

 9  class AppData {
10  private:
11       AppDataType type_;       // ADU type
12  public:
13       AppData(AppDataType type) { type_ = type; }
14       AppData(AppData& d) { type_ = d.type_; }
15       virtual ~AppData() {}
16       AppDataType type() const { return type_; }
17       virtual int size() const { return sizeof(AppData); }
18       virtual AppData* copy() = 0;
19  };
```

class `AppData`. Function `size()` simply returns `datalen_`. Function `copy()` creates a new `PacketData` object which has the same content as the current `PacketData` object, and returns the pointer to the created object to the caller.

Class `PacketData` has two constructors. One is to construct a new object with size "`sz`," using the constructor in Lines 6–12. This constructor simply sets the default application data type to be `PACKET_DATA` (Line 6), stores "`sz`" in "`datalen_`" (Line 7), and allocates memory of size "`datalen_`" to "`data_`" (Line 9). Another construction method[15] is to create a copy of an input `PacketData` object (Lines 13–20). In this case, the constructor feeds an input `PacketData` object "`d`" to the parent class (Line 13), copies the variable `datalen_` (Line 14), and duplicates its data payload (Line 17).[16]

NS2 creates a `PacketData` object through two functions of class `Packet`: `alloc(n)` and `allocdata(n)`. In Program 8.4, the function `alloc(n)` allocates a packet in Line 3 and creates data payload using the function `allocdata(n)` in Line 5. The function `allocdata(n)` creates a `PacketData` object of size "`n`," by executing "`new PacketData(n)`" in Line 11.

Program 8.19 shows four functions which can be used to manipulate data payload. Functions `accessdata()` (Lines 4–9) and `userdata()` (Line 10) are both data payload access functions. The difference is that the function

[15]Function `copy()` in Line 27 uses this constructor to create a copy of a `PacketData` object.

[16]Function `memcpy(dst,src,num)` copies "`num`" data bytes from the location pointed by "`src`" to the memory block pointed by "`dst`."

Program 8.18 Declaration of class `PacketData`

```
    //~/ns/common/packet.h
 1  class PacketData : public AppData {
 2  private:
 3      unsigned char* data_;
 4      int datalen_;
 5  public:
 6      PacketData(int sz) : AppData(PACKET_DATA) {
 7          datalen_ = sz;
 8          if (datalen_ > 0)
 9              data_ = new unsigned char[datalen_];
10          else
11              data_ = NULL;
12      }
13      PacketData(PacketData& d) : AppData(d) {
14          datalen_ = d.datalen_;
15          if (datalen_ > 0) {
16              data_ = new unsigned char[datalen_];
17              memcpy(data_, d.data_, datalen_);
18          } else
19              data_ = NULL;
20      }
21      virtual ~PacketData() {
22          if (data_ != NULL)
23              delete []data_;
24      }
25      unsigned char* data() { return data_; }
26      virtual int size() const { return datalen_; }
27      virtual AppData* copy() { return new PacketData(*this);}
28  };
```

`accessdata()` returns a direct pointer to *a string* "`data_`" which contains data payload while the function `userdata()` returns a pointer to *an* `AppData` *object* which contains data payload. Function `setdata(d)` (Lines 11–15) sets the pointer "`data_`" to point to the input argument "d." Finally, function `datalen()` in Line 16 returns the size of data payload.

8.5 Customizing Packets

8.5.1 Creating Your Own Packet

When designing a new protocol, programmers may need to change the packet format. This section gives a guideline of how packet header, data payload, or both can be modified. Note that, it is recommended *not to* use data payload in simulation. If possible, include information related to the new protocol in a protocol-specific header.

Program 8.19 Functions `accessdata`, `userdata`, `setdata` and `datalen` of class `Packet`

```
//~/ns/common/packet.h
class Packet : public Event {
    ...
public:
    inline unsigned char* accessdata() const {
        if (data_ == 0)
            return 0;
        assert(data_->type() == PACKET_DATA);
        return (((PacketData*)data_)->data());
    }
    inline AppData* userdata() const {return data_;}
    inline void setdata(AppData* d) {
        if (data_ != NULL)
            delete data_;
        data_ = d;
    }
    inline int datalen() const { return data_ ? data_
                          ->size() : 0; }
    ...
};
```

8.5.1.1 Defining a New Packet Header

Suppose we would like to include a new protocol-specific header, namely "My Header," into the packet header. We need to define a C++ class (e.g., `hdr_myhdr`), an OTcl class (e.g., `PacketHeader/MyHeader`), and a mapping class (e.g., `MyHeaderClass`), and include the OTcl class into the active protocol list. In particular, we need to perform the following four steps:

- **Step 1:** Define a C++ protocol-specific header structure (e.g., see Program 8.6).
  - Pick a name for the C++ `struct` data type, say `struct hdr_myhdr`.
  - Declare a variable `offset_` to identify where the protocol-specific header reside in the entire header.
  - Define a function `access(p)` which returns the reference to the protocol-specific header (see Lines 8–10 in Program 8.6).
  - Include all member variables required to hold new packet attributes.
  - [Optional] Include a new payload type into `enum packet_t` and class `p_info` (e.g., see Program 8.9). Again, a new payload type does not need to be added for every new protocol-specific header.
- **Step 2:** Pick an OTcl name for the protocol specific header, e.g., `PacketHeader/MyHeader`.

- **Step 3:** Bind the OTcl name with the C++ protocol-specific header structure. Derive a mapping class `MyHeaderClass` from class `PacketHeaderClass` (e.g., see class `CommonHeaderClass` in Program 8.12).
  - At the construction, feed the OTcl name (i.e., `PacketHeader/MyHeader`) and the size needed to hold the protocol-specific header (i.e., `sizeof(hdr_myhdr)`) to the constructor of class `PacketHeaderClass` (e.g., see Line 3 in Program 8.12).
  - From within the constructor of the mapping class, invoke function `bind_offset(...)` feeding the address of the variable `offset_` of the C++ `struct` data type as an input argument (i.e., invoke `bind_offset(&hdr_myhdr::offset_)`).
  - Instantiate a mapping variable `class_myhdr` at the declaration.
- **Step 4:** Activate the protocol-specific header from the OTcl domain. Add the OTcl name to the list defined within the packet header manage. In particular, modify Lines 2–9 of Program 8.15 as follows:

```
foreach prot {
    Common
    Flags
    ...
    MyHeader
} {
    add-packet-header $prot
}
```

  where only the suffix of the new protocol-specific header (i.e., `MyHeader`) is added to the `foreach` loop.

8.5.1.2 Defining a New Data Payload

Data payload can be created in four levels:

1. None: NS2 rarely uses data payload in simulation. To avoid any complication it is suggested not to use data payload in simulation.
2. Use class `PacketData`: The simplest form of storing data payload is to use class `PacketData` (see Program 8.18). Class `Packet` has a variable "`data_`" whose class is `PacketData`, and provides functions (in Program 8.19) to manipulate the variable "`data_`."
3. Derive a class (e.g., class `MyPacketData`) from class `PacketData`: This option is suitable when new functionalities (i.e., functions and variables) in addition to those provided by class `PacketData` are needed. After deriving a new `PacketData` class, programmers may also derive a new class (e.g., class `MyPacket`) from class `Packet`, and override the variable "`data_`" of class `Packet` to be a pointer to a `MyPacketData` object.

4. Define a new data payload class: A user can also define a new payload type if needed. This option should be used when the new payload has nothing in common with class `PacketData`. The following are the main tasks needed to define and use a new payload type `MY_DATA`.

   - Include the new payload type (e.g., `MY_DATA`) into `enum AppDataType` data type (see Program 8.17).
   - Derive a new payload class `MyData` from class `AppData`.
     - Feed the payload type `MY_DATA` to the constructor of class `AppData`.
     - Include any other necessary functions and variables.
     - Override functions `size()` and `copy()`.
   - Derive a new class `MyPacket` from class `Packet`
     - Declare a variable of class `MyData` to store data payload.
     - Include functions to manipulate the above `MyData` variable.

8.5.2 *Activate/Deactivate a Protocol-Specific Header*

By default, NS2 includes *all* built-in protocol-specific headers into packet header (see Program 8.15). This inclusion can lead to unnecessary wastage of memory especially in a *packet-intensive* simulation, where numerous packets are created. For example, common, IP, and TCP headers together use only 0.1 kB, while the default packet header consumes as much as 1.9 kB [17]. Again, NS2 does not return the memory allocated to a `Packet` object until the simulation terminates. Selectively including protocol-specific header can lead to huge memory saving.

The packet format is defined when the Simulator is created. Therefore, a protocol-specific headers must be activated/deactivated before the creation of the Simulator. NS2 provides the following OTcl procedures to activate/ deactivate protocol-specific headers:

- To add a protocol-specific header `PacketHeader/MH1`, execute

  ```
  add-packet-header MH1
  ```

  In Lines 10–14 of Program 8.15, the above statement includes `PacketHeader/MH1` to the variable "`tab_`" of class `PacketHeaderManager`.
- To remove a protocol-specific header `PacketHeader/MH1` from the active list, execute

  ```
  remove-packet-header MH1
  ```

  The details of procedure `remove-packet-header{args}` are shown in Lines 1–9 of Program 8.20. Line 7 removes the entries with the index `Packet Header/MH1` from the variable "`tab_`" of class `PacketHeaderManager`.

Program 8.20 Procedures `remove-packet-header`, and `remove-all-packet-header`

```
    //~ns/tcl/ns-packet.tcl
 1  proc remove-packet-header args {
 2      foreach cl $args {
 3          if { $cl == "Common" } {
 4              warn "Cannot exclude common packet header."
 5              continue
 6          }
 7          PacketHeaderManager unset tab_(PacketHeader/$cl)
 8      }
 9  }

10  proc remove-all-packet-headers {} {
11      PacketHeaderManager instvar tab_
12      foreach cl [PacketHeader info subclass] {
13          if { $cl != "PacketHeader/Common" } {
14              if [info exists tab_($cl)] {
15                  PacketHeaderManager unset tab_($cl)
16              }
17          }
18      }
19  }
```

- To remove all protocol-specific headers, execute

  ```
  remove-all-packet-header
  ```

  In Lines 10–19 of Program 8.20, the procedure `remove-all-packet-header{}` uses `foreach` to remove all protocol-specific headers (except for common header) from the active protocol list.

8.6 Chapter Summary

Consisting of packet header and data payload, a packet is represented by a C++ class `Packet`. Class `Packet` consists of pointers "`bits_`" to its packet header and "`data_`" to its data payload. It uses a pointer "`next_`" to form a link list of packets. It also has a pointer "`free_`" which points to the first `Packet` object on the free packet list. When a `Packet` object is no longer in use, NS2 stores the `Packet` object in the free packet list for future reuse. Again, `Packet` objects are not destroyed until the simulation terminates. When allocating a packet, NS2 first tries to take a `Packet` object from the free packet list. Only when the free packet list is empty, will NS2 create a new `Packet` object.

During simulation, NS2 usually stores relevant user information (e.g., packet size) in packet header, and rarely uses data payload. It is recommended not to use data payload if possible, since storing all information in packet header greatly simplifies the simulation yet yields the same simulation results.

Packet header consists of several protocol-specific headers. Each protocol-specific header occupies a contiguous part in packet header and identifies the occupied location using its variable `offset_`. NS2 uses a packet header manager (represented by an OTcl class `PacketHeaderManager`) to maintain a list of active protocols, and define packet header format using the list when the Simulator is created. The packet header construction process proceeds in the three following steps:

1. *At the Compilation*: NS2 defines the following three classes for each of protocol-specific headers:
   - *A C++ class*: NS2 uses C++ `struct` data type to define how packet attributes are stored in a protocol specific header. One of the important member variables is `offset_`, which indicates the location of the protocol-specific header on the packet header.
   - *An OTcl class*: During the Network Configuration Phase, the packet header manager configures packet header from the OTcl domain.
   - *A mapping class*: A mapping class binds the OTcl and C++ class together. It declares a *method* `offset`, which is invoked by a packet header manager from the OTcl domain to configure the value of the variable `offset_` of the C++ class `PacketHeaderClass`.
2. *At the Network Configuration Phase*: A user may add/remove protocol specific headers to/from the active protocol list. When the Simulator is created, the packet header manager computes and assigns appropriate offset values to all protocol-specific headers specified in the active list.
3. *At the Simulation Phase*: NS2 follows the above packet header definitions when allocating a packet.

8.7 Exercises

1. What are actual packets, class `Packet`, data payload, packet header, protocol-specific header? Explain their similarities/differences/relationships. Draw a diagram to support your answer.
2. Consider standard IP packet header as specified in [21]. How does NS2 store this packet header information?
3. What is the free packet list? How does it relate to the variable "`free_`" of class `Packet`? Draw a diagram to support your answer.
4. What is the purpose of the variable "`fflag_`" of class `Packet`? When is it used? How is it used?
5. Explain the packet destruction process. Draw a diagram to support your answer.

6. What is the purpose of the variable "`offset_`" defined for each protocol-specific header? Explain your answer via few examples of protocol specific header.
7. Where and how does NS2 define active protocol list? Write few statements to activate/deactivate protocol-specific headers.
8. Design a new packet header which can record a collection of time values. Pass the packet with new header through a network. When the packet enters a `LinkDelay` object, add the current simulation time into the time value collection and print out all the values in the collection.
9. Design a new data payload type. For simplicity, set every bit in the payload to "1". Run a program to test your answer.
10. Write C++ statements to perform the following tasks:

    a. Show the following information for a packet `*p` on the screen: Size, source and destination IP addresses and ports, payload type, and flow ID.
    b. Create a new packet.
    c. Destroy the packet `*p`.