3.2 Adaption layer

An "IP-over-X" adaptation layer needs to map IP datagrams to the services provided by the subnetwork, which is usually considered to be at layer 2 (L2) of a layered reference model.
A number of problems may need to be solved here:

- In a wireless network such as IEEE 802.15.4, a packet may be overheard by multiple receivers, not all of which may need to act on it; the L2 address provides an efficient way to make that decision. Once the IP layer has decided on the IP address of the next hop for a packet, one of the tasks of the adaptation layer is to find out to which link-layer address the packet needs to be addressed to so that it advances on its way to the intended IP-layer destination.
- The subnetwork may not immediately provide a path for packets to proceed to the next IP node. For instance, when mapping IP to connection-oriented networks such as ISDN or ATM (asynchronous transfer mode, a cell-switched link layer based on virtual circuits of 53-byte equal-size cells), the adaptation layer may need to set up connections (and may have to decide when to close them down again). While LoWPANs are not connection-oriented, in a Mesh-Under situation the adaptation layer may have to figure out the next L2 hop and may need to provide that hop with information about the further direction to forward the packet on.
- The IP packet needs to be packaged (*encapsulated*) in the subnetwork in such a way that the subnetwork can transport it and the L2 receiver can extract the IP packet again. This leads to a number of sub problems:
    - Links may be able to carry packets of other types than just IP datagrams. Also, there may be a need to distinguish different kinds of encapsulation. Most link layers provide some form of next-layer packet type information, such as the 16-bit ether type in Ethernet or the PID (protocol ID) in PPP.
    - IP packets may not fit into the data units that layer 2 can transport. An IP network interface is characterized by the maximum packet size that can be sent using that interface, the MTU (maximum transmission unit). Ethernet interfaces most often have an MTU of 1500 bytes. IPv6 defines a minimum value for the MTU of 1280 bytes, i.e. any maximum packet size imposed by the adaptation layer cannot be smaller than that, and at least 1280 byte or smaller packets have to go through. IEEE 802.15.4 can only transport L2 packets of up to 127 bytes (and a significant part of this can be consumed for L2 purposes). In order to be able to transport larger IPv6 packets, there needs to be a way to carve up the L3 packets and put their contents into multiple L2 packets. The next IP node then needs to put those parts of a packet together again and reconstruct the IP packet. This process is often called segmentation and reassembly; 6LoWPAN calls it *fragmentation.*
    - IP was designed so that each packet stands completely on its own. This leads to a header that may contain a lot of information that could be inferred from its context. In a LoWPAN, the typical IP/UDP header size of 48 bytes already consumes a significant part of the payload space available in a single IEEE 802.15.4 packet, leaving little for applications before fragmentation has to set in. The obvious fix may be to redesign (or avoid the use of) IP. A better approach is to eliminate large parts of the redundancy at the L3–L2 interface, and this has turned out to be a good architectural position to provide *header compression*. Existing IETF standards for header compression (such as ROHC mentioned above) are too heavyweight for LoWPAN Nodes

Link Layer

6LoWPAN attempts to be very modest in its requirements on the link layer. The basic service required of the link layer is for one node to be able to send packets of a limited size to another node within radio reach (i.e. a unicast packet). n a LoWPAN, a node A may be barely (or not at all) in radio range from another node C while both have reasonable error rates to and from a node B. Instead, 6LoWPAN's requirements on a link are relaxed to an assumption that, with respect to a node A and during a period of time, there is a set of nodes relatively

likely to be reachable from A. In this book, we call this set the *one-hop neighborhood* of A. In addition, there is an assumption that A can send a local *broadcast* packet that could be (but is not necessarily always) received by all nodes in A's one-hop neighborhood.
The IEEE 802.15.4 MAC layer defines four types of frames:
Data frames for the transport of actual data, such as IPv6 frames packaged according to the 6LoWPAN format specification;
Acknowledgment frames that are meant to be sent back by a receiver immediately after successful reception of a data frame, if requested by the acknowledgment request bit in the data frame MAC header
MAC layer command frames, used to enable various MAC layer services such as asso-ciation to and disassociation from a coordinator, and management of synchronized transmission;
Beacon frames, used by a coordinator to structure the communication with its associated nodes.

Link-layer addressing

The link layer must have some concept of globally unique addressing. 6LoWPAN assumes that there is a very low likelihood of two devices coming up in the network with the same link-layer address. The fact that an address uniquely identifies a node does not mean that it is by itself useful for locating the node globally, i.e., the link-layer address is not *routable*, and it is not by itself useful for determining if a node is on the same or a different network. Data frames carry both a source and a destination address. The destination address is used by a receiver to decide whether the frame was actually intended for this receiver or for a different one. The source address is mainly used to look up the keying material for link-layer security, but may also play a role in mesh forwarding. IEEE 802.15.4 nodes are permanently identified by EUI-64 identifiers, which weigh in at 8 bytes. As a pair of 64-bit source and destination addresses already consumes one eighth of the usable space in a packet, IEEE 802.15.4 also defines a *short address* format These 16-bit addresses can be dynamically assigned during the bootstrapping of the network.

Link-layer management and operation

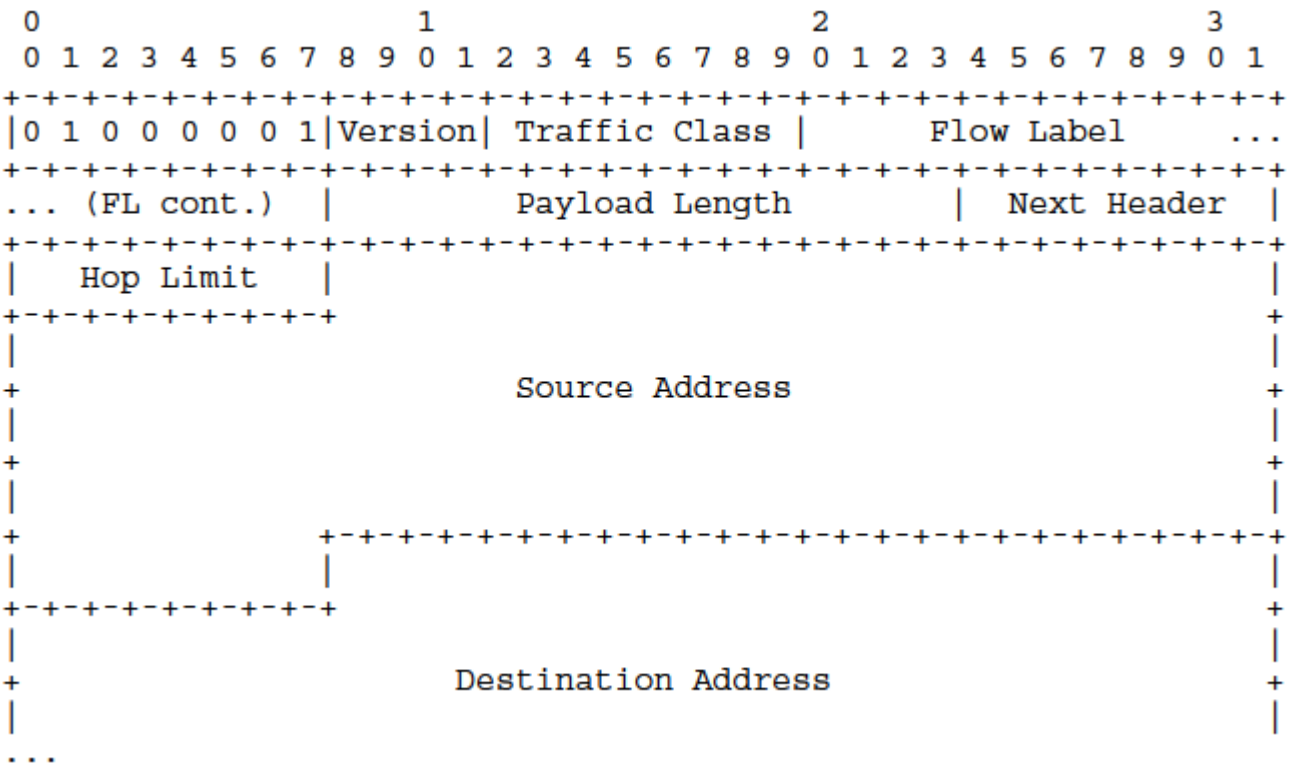

Figure 3.2.1 Uncompressed IPv6 packet with 6LoWPAN header.

Some of the formats defined by 6LoWPAN are designed to carry further 6LoWPAN PDUs as their payload. When multiple headers need to be present, the question is which header should be transported as the payload

of which other header, i.e., in which order the headers should be nested. To make this work reliably, 6LoWPAN specifies a well-defined nesting order. If present, the various 6LoWPAN headers should be used in the following order:

- **Addressing:** the mesh header (10nnnnnn, see Section 2.5), carrying L2 original source and final destination addresses and a hop count, followed by a 6LoWPAN PDU;
- **Hop-by-hop processing:** headers that essentially are L2 hop-by-hop options such as the broadcast header (LOWPAN_BC0, 01010000, that carries a sequence number to be checked at each forwarding hop), followed by a 6LoWPAN PDU;
- **Destination processing:** the fragmentation header (11nnnnnn), carrying fragments that, after possibly having been carried through multiple L2 hops, need to be reassembled to a 6LoWPAN PDU on the destination node;
- **Payload:** headers carrying L3 packets such as IPv6 (01000001), LOW-PAN_HC1 (01000010, see Section 2.6.1), or LOWPAN_IPHC (011nnnnn).

Forwarding and Routing

Packets will often have to traverse multiple radio hops on their way through the LoWPAN.
This involves two related processes: *forwarding* and *routing*. Both can be performed at layer 2 or at layer 3.

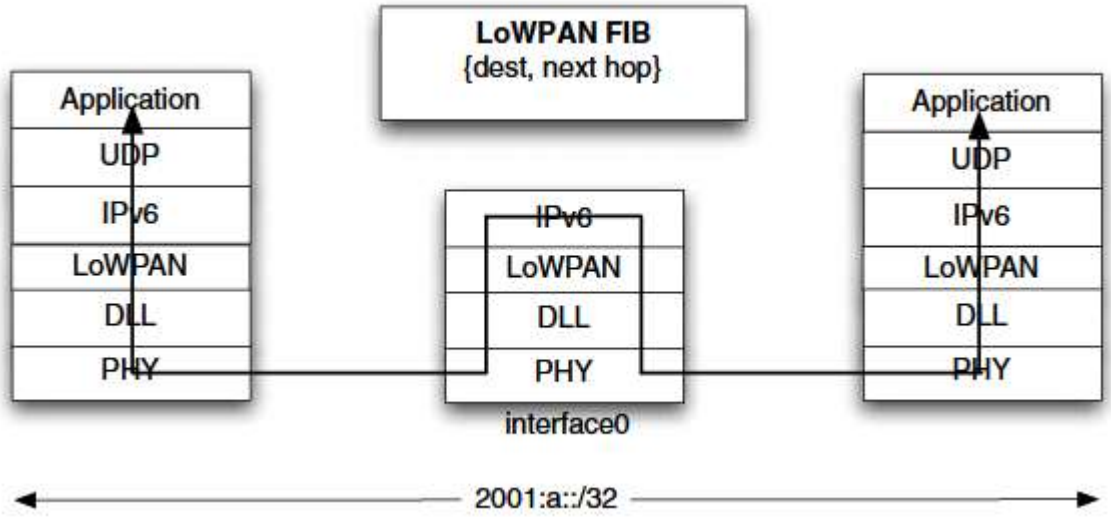

Figure 3.2.2 The LoWPAN routing model (L3 routing, "Route-Over").

L2 forwarding ("Mesh-Under")

When routing and forwarding happen at layer 2, they are performed based on layer-2 addresses, i.e., 64-bit EUI-64 or 16-bit short addresses. The IETF is not usually working on layer-2 routing ("mesh routing") protocols. To forward the packet to its eventual layer-2 destination, the node needs to know its address, the *final destination address*.
Also, to perform a number of services including reassembly, nodes need to know the address of the original layer-2 source, the *originator address*. Since each forwarding step overwrites the link-layer destination address by the address of the next hop and the link-layer source address by the address of the node doing the forwarding, this information needs to be stored somewhere else. 6LoWPAN defines the *mesh header* for this. In addition to the addresses, the mesh header stores a layer-2 equivalent of an IPv6 Hop Limit. Since the diameters of useful wireless multihop networks are usually small, the format is optimized for *hops left* values below 15 by only allocating four bits to that value. If a value of 15 or larger is needed, the 6LoWPAN packet encoder needs to insert an extension byte. This value must be decremented by a forwarding node before sending the packet on its next hop; if the value reaches zero, the packet is discarded silently. (Note that the lack of any error message means that the *traceroute* functionality that was one of the success factors of IP networking cannot be implemented for 6LoWPAN Mesh-Under.) In an implementation, there is no need to remove the extension byte when the hops left value drops to 14 or less by the decrementing process; the packet can be sent on as is or it can be optimized by removing the byte and moving the value into the dispatch byte.

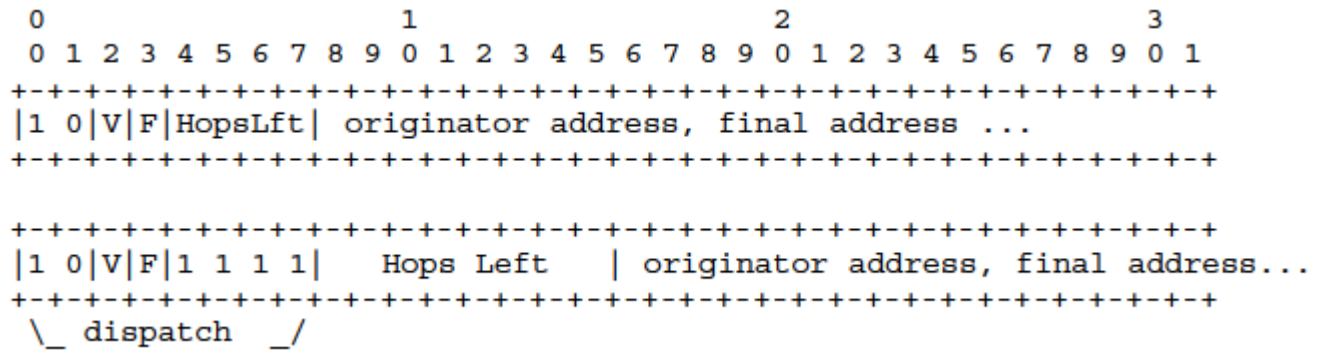

Figure 3.2.3 Mesh addressing type and header.

L3 routing ("Route-Over")

Layer-3 Route-Over forwarding is illustrated in Figure 3.2.2. In contrast to layer-2 mesh forwarding, layer-3 Route-Over forwarding does not require any special support from the adaptation layer format. Before the layer-3 forwarding engine sees the packet, the adaptation layer has done its work and decapsulated the packet – at least conceptually (implementations may be able to perform some optimizations by keeping the encapsulated form if they know how to rewrite it into the proper encapsulated form for the next layer-3 hop). Note that this in particular means that fragmentation and reassembly are performed at each hop in Route-Over forwarding – it is hard to imagine otherwise, as the layer-3 addresses are part of the initial bytes of the IPv6 header, which is present only in the first fragment of a larger packet. Again, implementations may be able to optimize this process by keeping virtual reassembly buffers that remember just the IPv6 header including the relevant addresses (and the contents of any fragments that arrived out of order before the addresses).

Fragmentation and Reassembly

IPv4 requires each node originating host or router) to be able to *fragment* packets into several smaller ones, and each final destination must be able to *reassemble* these fragments. To distinguish the whole (unfragmented) packet from its fragments, the former is often called the *datagram*. Note that the very small minimum MTU of IPv4 is often confused with the larger minimum *reassembly* buffer size: every IPv4 destination *must be able to receive a datagram of 576 [bytes] either in one piece or in fragments to be reassembled*

The fragmentation format

An 8-bit *datagram_offset* indicates the position of the fragment in the reassembled IPv6 packet; as in the IP layer, this counts in 8-byte units, so eight bits can cover the entire 2047 bytes. As with all 6LoWPAN frames, the first byte (dispatch byte) of a fragment indicates the type of frame; eight of the possible values for that byte (11100nnn) have been allocated for fragments with *datagram_size* spilling over into the dispatch byte so that no padding is required to accommodate the other bits. The resulting format, shown in Figure 3.2.4, is used for all but the initial fragment of a 6LoWPAN fragment sequence.

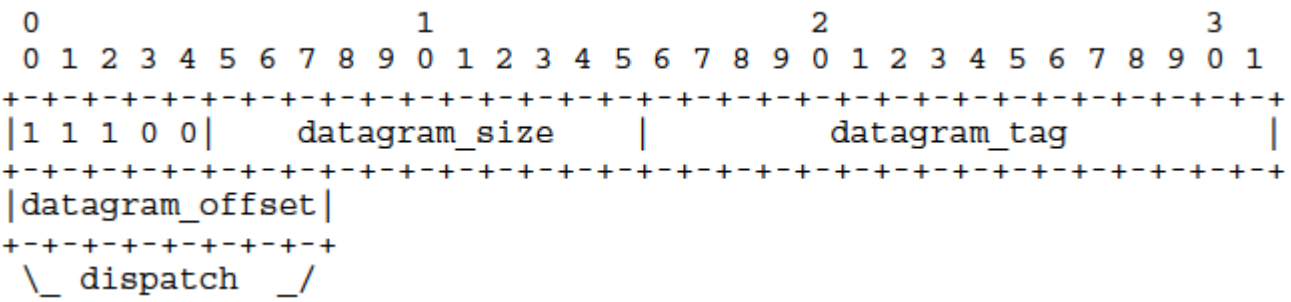

Figure 3.2.4 Non-initial 6LoWPAN fragment.

Assuming that most packets sent in a LoWPAN will be relatively small even if fragmented, a significant part of the fragments will be initial fragments with a fragment offset of all zeros. An optimization allows eliding that number; another eight possible dispatch values (11000nnn) were consumed for an alternative fragment format that implies a *datagram_offset* of zero (Figure 3.2.2).

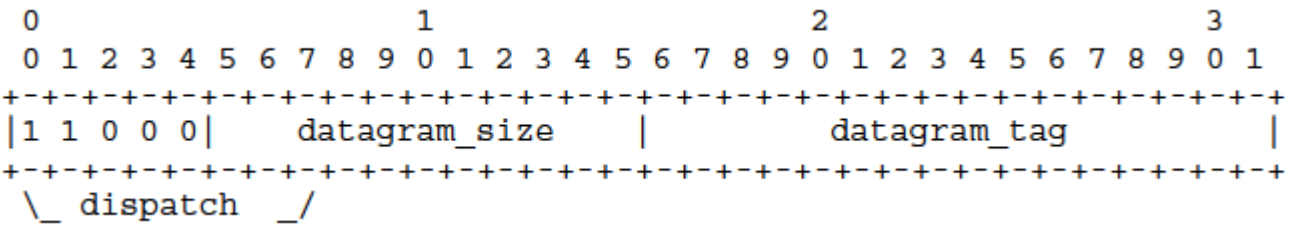

Figure 3.2.5 Initial 6LoWPAN fragment.

A node that needs to send a 6LoWPAN PDU that is too big to fit into a link-layer frame might use the following procedure:

Set variable *packet_size* to the size of the IPv6 packet (header and payload), and *header_size* to the size of the 6LoWPAN headers that need to be in the first fragment only, such as the dispatch byte and uncompressed or compressed IPv6 headers (including non-compressed fields in the latter case). If part of the IPv6 packet header or payload (such as the UDP header) is compressed away into the compressed header, adjust *header_size* down by that amount (which will usually make *header_size* negative!). In summary, *header_size* + *packet_size* is the size of the 6LoWPAN PDU that would result if it could be sent unfragmented.

Set variable *max_frame* to the space left in the link-layer frame after accounting for PHY, MAC, address and security headers and trailers, as well as any 6LoWPAN headers that may need to be prepended to each fragment or full packet (such as mesh headers for Mesh-Under). Note that *max_frame* may depend on the actual next-hop destination and the security and address size settings applicable for that. (Unless *header_size* + *packet_size* > *max_frame*, no fragmentation is needed and the PDU can simply be packaged into a link-layer frame.)

Increment a global *datagram_tag* variable to a new value that will be used in all fragments of this PDU.

Now the tricky part is to send just so much data that the first fragment is nicely filled but ends on a multiple of 8 bytes *within the IPv6 packet*. Set variable *max_frag_initial* to ^ *(max_frame* − 4 − *header_size)/*8^ ∗ 8 (leaving four bytes of space for the initial fragment header).

Send the first *max_frag_initial* + *header_size* bytes of the 6LoWPAN PDU in an initial fragment, prepending the four-byte initial fragment header to those bytes.

Set variable *position* to *max_frag_initial*.

Set variable *max_frag* to ^*(max_frame* − 5*)/*8^ ∗ 8 (leaving five bytes of space for each non-initial fragment header).

As long as *packet_size* − *position* > *max_frame* − 5:

- Send the next *max_frag* bytes in a non-initial fragment, prepending the five-byte non-initial fragment header to those bytes, filling in the value of *datagram_offset* from *position/*8.
- Increment *position* by *max_frag*.

Send the remaining bytes in a non-initial fragment, filling in the *datagram_offset* from *position*/8.

Fragment reception and reassembly might operate by this procedure:

- Build a *four-tuple* consisting of:
  – the source address,
  – the destination address
  – the *datagram_size*, and
  – the *datagram_tag*.
- If no reassembly buffer has been created for this four-tuple, create one, using the *datagram_size* as the buffer size, and initialize as empty a corresponding list of fragments received.
- For the initial fragment:
  - ✓ set variable *datagram_offset* to zero;
  - ✓ discard the four-byte fragment header;
  - ✓ perform any decoding and decompression on the contained dispatch byte and any compressed headers, as if this were a full packet, but using the full *datagram_size* for the reconstruction of length fields such as the IPv6 payload length and the UDP length;
  - ✓ set temporary variables *data* to the contents and *frag_size* to the size of the resulting decompressed packet.
- For non-initial fragments:
  - ✓ set variable *datagram_offset* to the value of the field from the packet;
  - ✓ discard the five-byte fragment header;
  - ✓ set temporary variables *data* to the contents and *frag_size* to the size of the data portion of the fragment received, i.e., minus the size of the five-byte header.
- Set variable *byte_offset* to *datagram_offset* ∗ 8.
- Check that *frag_size* either:
  - ✓ is a multiple of 8 (allowing additional fragments to line up with the end), or
  - ✓ *byte_offset* + *frag_size* = *datagram_size* (i.e., this is the final fragment);
  - ✓ if neither is true, fail (there would be no way to fill in the remaining bytes).
- If any of the entries in the list of fragments received before overlaps the interval [*byte_offset*, *byte_offset* + *frag_size)*:
  - ✓ If the overlapping entry is identical, discard the current fragment as a duplicate.
  - ✓ If not, fail.
- Otherwise, add the interval to the list.
- Copy the contents of *data* to the buffer positions starting from *byte_offset*.
- If the list of intervals now covers the whole span, the reassembly is complete, and the buffer contains a reassembled IPv6 packet of size *datagram_size*. Perform any final processing that requires the whole packet such as reconstructing a compressed-away UDP checksum.

Avoiding the fragmentation performance penalty

Fragmentation is undesirable for a number of reasons, ,the problem most often discussed is the decoupling between unit of loss (the fragment) and unit of retransmission (the entire packet), with the related inefficiencies. Possibly even more important in a resource-constrained embedded environment, the uncertainty of when the remaining fragments for a reassembly buffer will be received makes management of the resources assigned to reassembly buffers very difficult. This is probably less of an issue for more resource-heavy edge routers, but can make reception of fragmented packets by battery-operated systems with limited RAM quite unreliable. As a rule of thumb, fragmentation is marginally acceptable for packets originating from such devices (where the main problem is the reduced probability of the whole reassembled packet arriving intact), but should be avoided for packets being sent to them.