Message Library Design Notes

David Mosberger

January 1996

Abstract

This document describes the current implementation of the x-kernel message library. The focus is on its data structures and the underlying principles. This document does not describe the message library's interface or how it is used. Please refer to the x-kernel Programmer's Manual [1] and the x-kernel Tutorial [2] for that purpose.

1 Introduction

Conceptually, a message is simply a linear sequence of bytes. A message therefore has contents (data) and a length (in bytes). The data in a message is never interpreted by the message library itself. The message library is optimized for the processing that is encountered in typical network protocols, such as TCP or IP. There are two key characteristics to network message processing: (a) on the outgoing side, headers are prepended to the message, and on the incoming side headers are removed from the beginning of the message, and (b) the amount of data in a message is typically large so that data copying should be avoided as much as possible. Given (a), it is reasonable to extend the simple message model to reserve space at the head of the message. Similarly, it is often necessary to discard a few bytes a the end of the message (e.g., to discard a CRC sum or a trailer). This leads to messages that end before the physical end of the buffer memory. With these two extensions, Figure 1 depicts this high-level view of a message.

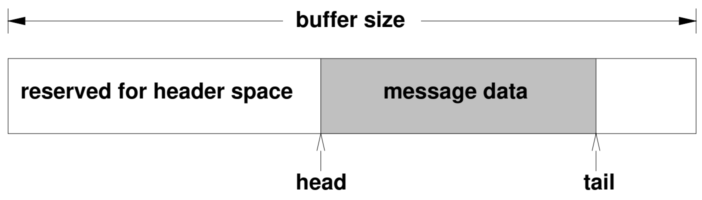

Figure 1: High-level view of a message.

1.1 Pushing Headers

With the above model, it is now possible to prepend (push) a header onto an existing message simply by decrementing head by the size of the header and then copying the header contents to the memory starting at offset head. This is efficient because network headers are small relative to the size of the message data. Now, what happens if the reserved space is all used up? One solution would be to allocate a new and bigger memory buffer, copy the existing data into that buffer and then prepend the new header. In practice this would be too slow because it requires copying all the data in the message. Instead, the message library extends the message model by implementing messages as a tree of memory buffers. Thus, when a new header would overflow the available header space, a new memory buffer is allocated and linked into the existing message by creating a new interior (pair) node. Thus, each header-space overflow adds two

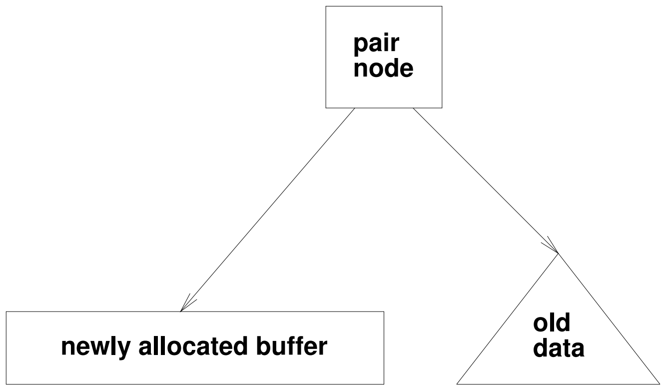

Figure 2: New buffer attached to the front of a message.

new nodes to the message tree (one interior node and a leaf node). For example, if the existing message tree is depicted as a triangle, a push overflow leads to the configuration shown in Figure 2.

Notice that each push overflow leads to a new interior node whose left subtree is a leaf. The resulting tree is a degenerate tree that is no more than a linear list. For searching, it would be better if it were balanced because that would minimize the average distance from the root. However, with the message library, searching is not common and in fact degenerate trees of the above kind (where the left subtree is always a leaf) are ideal because they allow tree traversal without recursion.

Notice that while the physical representation of the message now uses a tree, the logical view of messages as presented in the previous section is still valid. The only externally visible difference is that the message buffer now potentially consists of several smaller memory buffers that may be scattered in the computer's address space.

1.2 Popping Headers

So far, the message representation allows efficient implementation of pushing headers onto messages. Let's investigate the opposite operation: removing (popping) headers.

In the normal case, popping a header simply involves incrementing the head index and returning a pointer to the old location. Similar to the push case, this works, provided there is enough data in the first leaf. If this is not the case, there are two possible scenarios: (a) the header data comes from a single other leaf, or (b) the header data comes from several other leaves. In case (a), the message library simply finds the correct leaf and returns a pointer to the header data (this is the one case where the message tree has to be searched). For efficiency, the message library also caches the location of the leaf in which it found the header data. If another header is popped (which is likely), then it is normally the case that the data for the new header comes from the same leaf as the current header, so this results in bypassing the step of having to find the leaf in the message tree. In case (b), there is no choice: the message library has to copy the header data into a new, contiguous buffer. Conceptually, this is simple. The leaves in the old message are copied until enough data has been accumulated for the header. In the example below, the data in leaves l1, l2, and l3 are copied into the new leaf called header, as illustrated in Figure 3.

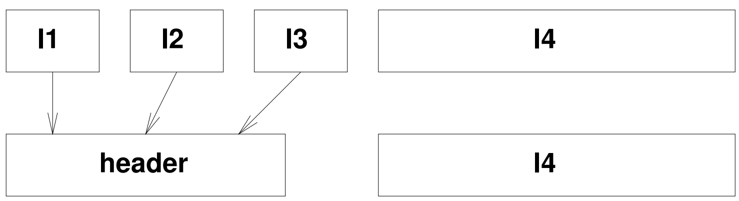

Figure 3: Accumulating leaves into a new buffer.

After the data is copied, the message tree is updated such that the new contiguous header leaf replaces the first three leaves in the original tree. Conceptually, the final message appears as shown in Figure 4.

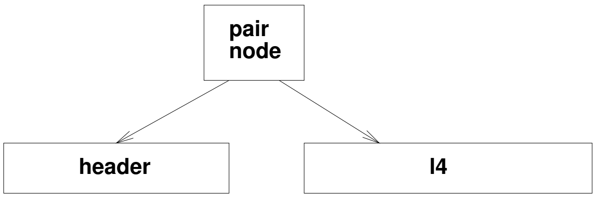

Figure 4: Message after leaves are accumulated.

Notice that removing the original small leaves requires a recursive node deletion algorithm. Since this is slow, the current message library implementation modifies the tree such that the small leaves remain in the tree, but are ignored as far as message data is concerned. In essence, the message library first discards all the bytes in the small leaves and then creates the pair node that connects the new header leaf into the tree. A more accurate picture of the final tree is given in Figure 5. The right pair node is setup such that the subtree containing l1, l2, and l3 will be ignored as far as message data is concerned.

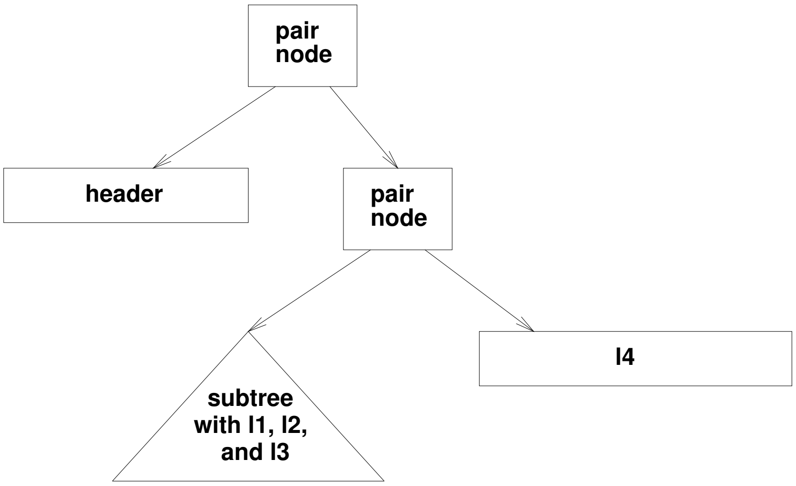

Figure 5: Detailed view of message after leaves are accumulated.

After the copying and adjusting the tree, the new header leaf contains enough data for the header that is being popped. The message library can now increment the head index and return the pointer to the contiguous header memory just as in the previous cases.

1.3 Trimming Messages

Other operations that are common include determining the length of the data in a message and truncating/discarding data. Truncation involves dropping the last n bytes, whereas discarding involves dropping the first m bytes in a message. To support these operations efficiently, the message *fragment* abstraction is introduced. A fragment refers to a subrange of the data in a message. To that effect, the fragment datastructure contains a pointer to the message tree that contains the message data and two indices: head, the index of the first byte in the subrange, and tail, the index of the first byte outside of the subrange. With these definitions, the length of a fragment is given by the expression tail - head. Truncation is implemented by decrementing the tail index, and the disard operation is implemented by incrementing the head index. Notice that the head and tail fields in the fragment data structure correspond exactly to the the head and tail variables described in the beginning of this section. Thus, as a first approximation, the message data structure is simply a message fragment.

1.4 Visiting Message Data

The final operation, that is, unfortunately, often on the critical path of network packet processing, is visiting all the data in a message tree. In general, visiting a tree requires a recursive algorithm. For efficiency, the current implementation avoids recursive function invocations by explicitly managing a recursion stack. In the common case where a message consists of a single leaf, or in the degenerate tree case where all left subtrees are leaves, this yields highly efficient code. The state that needs to be maintained during a message traversal is a stack of fragments. The stack is initialized with the fragment in the message to be visited. The fragment at the top of the stack is the one to be visited next. After popping a fragment from the top of the stack, there are two possibilities: (a) the fragment points to a pair node in which case it is simply necessary to push the fragments for the right and the left subtrees onto the stack (in this order). In case (b), the fragment points to a leaf node so the data in the leaf can be processed directly.

2 Data Structures

We are now at the point where we can present the message library's data structures. We start with the most fundamental structure: the message fragment.

```
struct MsgFrag {
    int      head;
    int      tail;
    MsgNode  tree;
};
```

As explained in the Introduction, a fragment refers to a subrange of the data in a message tree. The message tree is pointed to by field tree. The fields head and tail are the indices of the first byte in the subrange and the first byte outside of the subrange, respectively. The data length is thus given by the expression tail - head.

2.1 The Message Structure

The message structure is given as follows:

```
struct Msg {
    struct MsgFrag  f;
    MsgNodeLeaf     first;          /* left-most leaf in tree */
    int             firstOffset;    /* offset to first leaf */
    bool            firstIsMine;    /* is first leaf writable? */
    struct Attrs    attrs;
};
```

The central field is f, the message fragment containing the data for this message. Because the head of a message is manipulated frequently, it makes sense to cache the state related to the first leaf that is not empty. For this reason, the message structure also contains the fields first and firstOffset. Similarly, field firstIsMine is used to optimize the pushing of headers. These fields are described in more detail below. The final field, attrs, is used to hold message attributes. These are arbitrary name/value pairs that can be used to associate other information with messages.

Field first points to the first (left-most) leaf that is either not empty or that is the next one to be written on a header push. To be precise, first points to the leaf that contains the data byte that index f.head refers to. Field firstOffset is the offset (relative to the first byte in the message tree) of the first byte in leaf first. It is normally zero, but whenever popping a header results in an underflow, this field gets incremented. Field firstIsMine is true if the first leaf is owned by this message. The invariant here is that for any given leaf, there is at most one message for which firstIsMine is TRUE. There may be leaves for which there is no owner, but there are never two or more owners for the same leaf. Ownership gives the right to push header data onto the first leaf. Ownership does not give the right to change existing message data, however. This is because message trees (and therefore leaves) are shared by messages as much as possible. Due to this sharing, it is necessary that any message byte is written at most once. Figure 6 illustrates the key-fields in the message data structure.

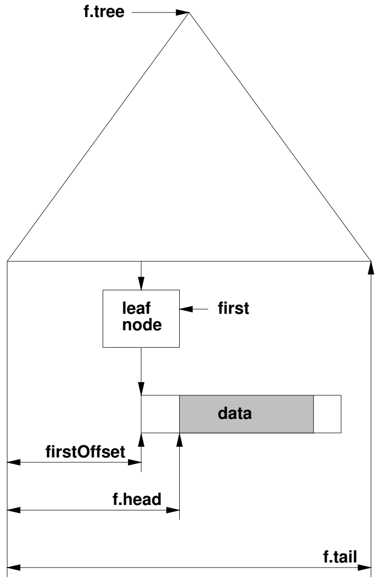

Figure 6: Key fields in message structure.

There are several points worth emphasizing.

- The number of bytes that are available for header data is given by the expression f.head - firstOffset.
- The first field does not necessarily point to the left-most leaf in the tree (e.g., consider header popping that causes an underflow).
- f.tail is not necessarily the index of the last data byte in the message tree (e.g., consider message truncation).

2.2 The Message Tree

There are two types of nodes in the tree: interior nodes (PAIR nodes) and leaves. For storage management reasons, leaves are divided into three subtypes: PAGE, BUF, and EMPTY.

There is one and only one EMPTY leaf that is used for all buffers of length zero. PAGE leaves are used to identify data areas that were allocated by the message library. The message library allocates memory from the heap in units of pages. (This page size is not necessarily related to the size of virtual memory pages; it is often much smaller than a virtual memory page). Allocating memory at the granularity of pages reduces internal fragmentation in the heap manager. Also, when creating a PAGE buffer of n bytes, the amount of memory requested is rounded up to ensure that at least m bytes are available for headers, where m is maximum size expected to be taken up by protocol headers. In the current implemention, m is given by constant MAX_HDR_STACK_SIZE. In contrast to PAGE leaves, BUF leaves are used for data whose memory was allocated outside the message library. Such nodes have a deallocator function associated with them that is called as soon as the message library determines that the memory is no longer referenced by any of the messages in existence.

The enumeration type that indicates a node's type is given below:

```
enum NodeType {
    MSG_NODE_JUNK = 0,  /* to catch programming errors */
    MSG_NODE_PAIR,      /* joins two subtrees */
    MSG_NODE_PAGE,      /* leaf with one or more pages of data */
    MSG_NODE_BUF,       /* leaf using a special deallocator */
    MSG_NODE_EMPTY      /* used by all zero-length leaves */
};
```

The JUNK node type is there because messages that are not properly initialized are most likely to contain zeroes. Thus, choosing the value 0 for this dummy-type increases the likelihood of catching programming errors.

The exact declaration of a node depends on its type. However, all message node structures start with this common part.

```
struct MsgNodePart {
    enum NodeType  type;
    u_int          refCnt;
};
```

The type field allows the full declaration of this node to be determined. The refCnt field is a reference count that gives the number of permanent references to this node. Reference counting is needed for message tree nodes because they can be shared among multiple messages. When the reference count reaches zero, the node is deleted. If the node is an interior node, the subtrees are visited recusively and all nodes that are no longer needed are deleted as well. The current implementation avoids recursion as much as possible and manages an explicit stack when recursion is unavoidable. To maximize performance, memory for the stack is allocated via alloca(), not the regular heap allocator.

2.3 Interior Nodes

An interior (PAIR) node is simply a pair of fragments. Field l is the left fragment that is visited first in a message traversal and r is the right fragment:

```
struct MsgNodePairPart {
    struct MsgFrag  l;
    struct MsgFrag  r;
};
```

One important point is that the head field in the left fragment is always zero. This is because the message structures that refer to this tree also contain a fragment. The offset of the first byte in a message is therefore subsumed into the f.head field in the message structures. This opens up the possibility of allowing the owner of a leaf to push header data onto a message without requiring changes to any of the possibly existing PAIR nodes. The fact that the head field in the left fragment is always zero also implies that the length of such a fragment is given directly by the value of the tail field. (Recall that in general, the length is given by tail - head). The message library makes heavy use of this invariant and is optimized accordingly.

Now, suppose p is a PAIR node. Then the n-th byte in the data represented by p is found as follows: if n is less than l.tail, the byte is located at offset n in the left subtree (l.tree). Otherwise, the byte is located at offset n - l.tail + r.head in the right subtree (r.tree).

2.4 Leaves

The common state that the three leaf node types share is factored into the following structure.

```
struct MsgNodeLeafPart {
    int   size;     /* size of buffer */
    char  *buf;     /* the buffer */
};
```

The buf field points to the beginning of the memory buffer represented by this node. The size field specifies the size of the buffer in bytes.

A PAGE node contains the memory for its data in the node structure itself. As C doesn't support variable length arrays the structure given below declares the buffer to be only one byte long. In reality, the buffer is size bytes long. (This is the size field in the MsgNodeLeafPart structure). This works correctly so long as this structure appears at the very end of the enclosing data structure. The declaration is given below.

```
struct MsgNodePagePart {
    char  buffer[1];      /* longer in actuality */
};
```

BUF type nodes contain a reference to the function that is used to free (deallocate) the data buffer once it is no longer needed by the message library. The address of the deallocation function is stored in the free field in the following structure.

```
struct MsgNodeBufPart {
    MsgDeallocator  free;
};
```

The full declarations of the node structures are given below. As expected, there is one structure per node type. In addition, there are two structures (MsgNode and MsgNodeLeaf) that factor state that is common to more than one node type.

```
struct MsgNode {
    struct MsgNodePart      c;
};

struct MsgNodePair {
    struct MsgNodePart      c;
    struct MsgNodePairPart  pair;
};

struct MsgNodeLeaf {
    struct MsgNodePart      c;
    struct MsgNodeLeafPart  leaf;
};

struct MsgNodePage {
    struct MsgNodePart      c;
    struct MsgNodeLeafPart  leaf;
    struct MsgNodePagePart  page;
};

struct MsgNodeBuf {
    struct MsgNodePart      c;
    struct MsgNodeLeafPart  leaf;
    struct MsgNodeBufPart   buf;
};
```

Thus, the relationships illustrated in Figure 7 hold, where each edge denotes the "*is a*" relation; e.g., MsgNodeLeaf *is a* MsgNode.

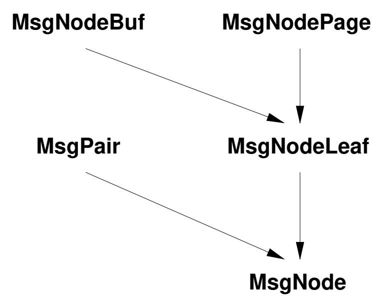

Figure 7: Relationship among various structures.

2.5 Message Walk Context

The contents of a message can be visited using the msgWalk functions. As described in the previous section, visiting a message tree requires maintaining a stack of fragments. The structure that implements this conceptual stack is given below.

```
struct MsgWalk {
    struct MsgFrag      f;       /* frag to be visited next */
    struct MsgFragStack stack;
};
```

To understand the reason for maintaining field f besides the actual fragment stack stored in field stack, consider the case where a message consists only of PAIR nodes whose left subtrees are leaves. Such a message can be traversed using a single fragment that keeps track of the position to be visited next. In essence, this case corresponds to traversing a linear list. Memory for the actual stack needs to be allocated on the heap, which is relatively expensive. Thus, by maintaining field f, memory (de-)allocation can be avoided for common case messages.

3 Final Comments

There are several points that are worth emphasizing. These points capture the invariants that provide the underpinnings of the message library. As such, they should be helpful in reading and understanding the code.

- Message data can have arbitrary alignment. It is not safe to assume, for example, that initializing a message to a length of 8 will yield a pointer to memory that is aligned on an 8-byte boundary.
- Due to sharing, message data must not be written more than once.
- Pair nodes, and therefore the fragments contained therein, are immutable. Once created and initialized, they don't change. Only the fragment in the message structures can change over time.
- The head field in the left fragment of a PAIR node must always be zero. As a result, the tail field in such a fragment directly gives the length of the fragment.

References

[1] Network Systems Research Group, Department of Computer Science, University of Arizona. *x-kernel Programmer's Manual (Version 3.3)*, Jan. 1996.

[2] L. L. Peterson, B. S. Davie, and A. C. Bavier. *x-kernel Tutorial*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.