RFC: 815

IP DATAGRAM REASSEMBLY ALGORITHMS

David D. Clark
MIT Laboratory for Computer Science
Computer Systems and Communications Group
July, 1982

1. Introduction

One of the mechanisms of IP is fragmentation and reassembly. Under certain circumstances, a datagram originally transmitted as a single unit will arrive at its final destination broken into several fragments. The IP layer at the receiving host must accumulate these fragments until enough have arrived to completely reconstitute the original datagram. The specification document for IP gives a complete description of the reassembly mechanism, and contains several examples. It also provides one possible algorithm for reassembly, based on keeping track of arriving fragments in a vector of bits. This document describes an alternate approach which should prove more suitable in some machines.

A superficial examination of the reassembly process may suggest that it is rather complicated. First, it is necessary to keep track of all the fragments, which suggests a small bookkeeping job. Second, when a new fragment arrives, it may combine with the existing fragments in a number of different ways. It may precisely fill the space between two fragments, or it may overlap with existing fragments, or completely duplicate existing fragments, or partially fill a space between two fragments without abutting either of them. Thus, it might seem that the reassembly process might involve designing a fairly complicated algorithm that tests for a number of different options.

In fact, the process of reassembly is extremely simple. This document describes a way of dealing with reassembly which reduces the bookkeeping problem to a minimum, which requires for storage only one buffer equal in size to the final datagram being reassembled, which can reassemble a datagram from any number of fragments arriving in any order with any possible pattern of overlap and duplication, and which is appropriate for almost any sort of operating system.

The reader should consult the IP specification document to be sure that he is completely familiar with the general concept of reassembly, and the particular header fields and vocabulary used to describe the process.

2. The Algorithm

In order to define this reassembly algorithm, it is necessary to define some terms. A partially reassembled datagram consists of certain sequences of octets that have already arrived, and certain areas still to come. We will refer to these missing areas as "holes". Each hole can be characterized by two numbers, hole.first, the number of the first octet in the hole, and hole.last, the number of the last octet in the hole. This pair of numbers we will call the "hole descriptor", and we will assume that all of the hole descriptors for a particular datagram are gathered together in the "hole descriptor list".

The general form of the algorithm is as follows. When a new fragment of the datagram arrives, it will possibly fill in one or more of the existing holes. We will examine each of the entries in the hole descriptor list to see whether the hole in question is eliminated by this incoming fragment. If so, we will delete that entry from the list. Eventually, a fragment will arrive which eliminates every entry from the list. At this point, the datagram has been completely reassembled and can be passed to higher protocol levels for further processing.

The algorithm will be described in two phases. In the first part, we will show the sequence of steps which are executed when a new fragment arrives, in order to determine whether or not any of the existing holes are filled by the new fragment. In the second part of this description, we will show a ridiculously simple algorithm for management of the hole descriptor list.

3. Fragment Processing Algorithm

An arriving fragment can fill any of the existing holes in a number of ways. Most simply, it can completely fill a hole. Alternatively, it may leave some remaining space at either the beginning or the end of an existing hole. Or finally, it can lie in the middle of an existing hole, breaking the hole in half and leaving a smaller hole at each end. Because of these possibilities, it might seem that a number of tests must be made when a new fragment arrives, leading to a rather complicated algorithm. In fact, if properly expressed, the algorithm can compare each hole to the arriving fragment in only four tests.

We start the algorithm when the earliest fragment of the datagram arrives. We begin by creating an empty data buffer area and putting one entry in its hole descriptor list, the entry which describes the datagram as being completely missing. In this case, hole.first equals zero, and hole.last equals infinity. (Infinity is presumably implemented by a very large integer, greater than 576, of the implementor’s choice.) The following eight steps are then used to insert each of the arriving fragments into the buffer area where the complete datagram is being built up. The arriving fragment is described by fragment.first, the first octet of the fragment, and fragment.last, the last octet of the fragment.

1. Select the next hole descriptor from the hole descriptor list. If there are no more entries, go to step eight.

2. If fragment.first is greater than hole.last, go to step one.

3. If fragment.last is less than hole.first, go to step one.

    - (If either step two or step three is true, then the newly arrived fragment does not overlap with the hole in any way, so we need pay no further attention to this hole. We return to the beginning of the algorithm where we select the next hole for examination.)

4. Delete the current entry from the hole descriptor list.

    - (Since neither step two nor step three was true, the newly arrived fragment does interact with this hole in some way. Therefore, the current descriptor will no longer be valid. We will destroy it, and in the next two steps we will determine whether or not it is necessary to create any new hole descriptors.)

5. If fragment.first is greater than hole.first, then create a new hole descriptor "new_hole" with new_hole.first equal to hole.first, and new_hole.last equal to fragment.first minus one.

- (If the test in step five is true, then the first part of the original hole is not filled by this fragment. We create a new descriptor for this smaller hole.)

6. If fragment.last is less than hole.last and fragment.more fragments is true, then create a new hole descriptor "new_hole", with new_hole.first equal to fragment.last plus one and new_hole.last equal to hole.last.

   - (This test is the mirror of step five with one additional feature. Initially, we did not know how long the reassembled datagram would be, and therefore we created a hole reaching from zero to infinity. Eventually, we will receive the last fragment of the datagram. At this point, that hole descriptor which reaches from the last octet of the buffer to infinity can be discarded. The fragment which contains the last fragment indicates this fact by a flag in the internet header called "more fragments". The test of this bit in this statement prevents us from creating a descriptor for the unneeded hole which describes the space from the end of the datagram to infinity.)

7. Go to step one.

8. If the hole descriptor list is now empty, the datagram is now complete. Pass it on to the higher level protocol processor for further handling. Otherwise, return.

4. Part Two: Managing the Hole Descriptor List

The main complexity in the eight step algorithm above is not performing the arithmetical tests, but in adding and deleting entries from the hole descriptor list. One could imagine an implementation in which the storage management package was many times more complicated than the rest of the algorithm, since there is no specified upper limit on the number of hole descriptors which will exist for a datagram during reassembly. There is a very simple way to deal with the hole descriptors, however. Just put each hole descriptor in the first octets of the hole itself. Note that by the definition of the reassembly algorithm, the minimum size of a hole is eight octets. To store hole.first and hole.last will presumably require two octets each. An additional two octets will be required to thread together the entries on the hole descriptor list. This leaves at least two more octets to deal with implementation idiosyncrasies.

There is only one obvious pitfall to this storage strategy. One must execute the eight step algorithm above before copying the data from the fragment into the reassembly buffer. If one were to copy the data first, it might smash one or more hole descriptors. Once the algorithm above has been run, any hole descriptors which are about to be smashed have already been rendered obsolete.

5. Loose Ends

Scattering the hole descriptors throughout the reassembly buffer itself requires that they be threaded onto some sort of list so that they can be found. This in turn implies that there must be a pointer to the head of the list. In many cases, this pointer can be stored in some sort of descriptor block which the implementation associates with each reassembly buffer. If no such storage is available, a dirty but effective trick is to store the head of the list in a part of the internet header in the reassembly buffer which is no longer needed. An obvious location is the checksum field.

When the final fragment of the datagram arrives, the packet length field in the internet header should be filled in.

6. Options

The preceding description made one unacceptable simplification. It assumed that there were no internet options associated with the datagram being reassembled. The difficulty with options is that until one receives the first fragment of the datagram, one cannot tell how big the internet header will be. This is because, while certain options are copied identically into every fragment of a datagram, other options, such as "record route", are put in the first fragment only. (The "first fragment" is the fragment containing octet zero of the original datagram.)

Until one knows how big the internet header is, one does not know where to copy the data from each fragment into the reassembly buffer. If the earliest fragment to arrive happens to be the first fragment, then this is no problem. Otherwise, there are two solutions. First, one can leave space in the reassembly buffer for the maximum possible internet header. In fact, the maximum size is not very large, 64 octets. Alternatively, one can simply gamble that the first fragment will contain no options. If, when the first fragment finally arrives, there are options, one can then shift the data in the buffer a sufficient distance for allow for them. The only peril in copying the data is that one will trash the pointers that thread the hole descriptors together. It is easy to see how to untrash the pointers.

The source and record route options have the interesting feature that, since different fragments can follow different paths, they may arrive with different return routes recorded in different fragments.

Normally, this is more information than the receiving Internet module needs. The specified procedure is to take the return route recorded in the first fragment and ignore the other versions.

7. The Complete Algorithm

In addition to the algorithm described above there are two parts to the reassembly process. First, when a fragment arrives, it is necessary to find the reassembly buffer associated with that fragment. This requires some mechanism for searching all the existing reassembly buffers. The correct reassembly buffer is identified by an equality of the following fields: the foreign and local internet address, the protocol ID, and the identification field.

The final part of the algorithm is some sort of timer based mechanism which decrements the time to live field of each partially reassembled datagram, so that incomplete datagrams which have outlived their usefulness can be detected and deleted. One can either create a demon which comes alive once a second and decrements the field in each datagram by one, or one can read the clock when each first fragment arrives, and queue some sort of timer call, using whatever system mechanism is appropriate, to reap the datagram when its time has come.

An implementation of the complete algorithm comprising all these parts was constructed in BCPL as a test. The complete algorithm took less than one and one-half pages of listing, and generated approximately 400 nova machine instructions. That portion of the algorithm actually involved with management of hole descriptors is about 20 lines of code.

The version of the algorithm described here is actually a simplification of the author’s original version, thanks to an insightful observation by Elizabeth Martin at MIT.