Programming Project: Load-balancing Forwarding Lookup

Version 1.0

Due: 2 May 2006

1 Overview

For this assignment you will implement a modified longest-prefix-matching system, such as might be used in a router. However, the addresses to be looked up are not IPv4 or IPv6 addresses, but IPv685 addresses. The semantics of IPv685 addressing is similar to that of IPv4, but there are two important differences:

- IPv685 addresses are 40 bits long (i.e. \( W = 40 \)).
- Bits of the address are transmitted least significant bit first. That is, the first bit transmitted is the least significant bit of the address, the second is the second most significant, etc. Similarly, the first byte transmitted/received is the least significant, while the last byte transmitted/received is the most significant. (IPv4 addresses, in contrast, are transmitted in big-endian, or most-significant-first, order.) This means that, when manipulating bits in bytes, the most significant bit is 0x01, the next most significant is 0x02, the third most significant is 0x04, etc. (As with IPv4 addresses, which byte in a word is most significant depends on the processor architecture.)

An additional difference from traditional longest-match systems is that your lookup function must implement load distribution across two or more routes to destinations. Some prefixes in the database may have more than one associated routes. When the lookup function is called with packets that match such a prefix, the returned route should be different for different packets. This must be done in such a way that the packets are distributed across the routes. However, it is necessary to ensure that packets belonging to the same TCP connection all take the same path through the network. Therefore, the lookup function must always return the same route for packets with the same source and destination address.

Your code may implement this capability in any way you choose. For example, you might implement it by means of a hashing function that maps each (source, destination) address pair to a unique index, or you might choose to store information in the table about active flows and the routes to which they are assigned. If you use a hash function, it must approximate a “random” function—that is, it must distribute pairs across the possible routes evenly. If you store state, you must be prepared to reclaim that state periodically; the system will provide information about when it is safe to reclaim state associated with a flow.

In addition to the lookup function, you must also implement the procedures that build and maintain the data structures that hold the prefix database.

2 Packet Layout

The packet format relevant to this assignment is simple: the first field in the packet is the destination address, in little-endian byte order (i.e. byte 0 is the least significant byte. The second five bytes are the source address, also in little-endian order. There may be other information following these fields, but that information should never be referenced by the lookup code.
### 3 Data Types

The relevant C type declarations are:

```c
#define ASIZE 5

typedef struct {
  unsigned char abytes[ASIZE];
} ADDR;

typedef union {
  unsigned short rt_w;
} RTPTR;
```

Because the number of possible next-hops is small, the RTPTR (route) type is only a two-byte object. **In fact, no more than 12 bits of any RTPTR will be significant**, i.e. there are no more than 4096 different routes in the system at any time. In what follows we refer to the contents of a RTPTR structure as a “route pointer”, even though it is not big enough to be a pointer on a 32-bit machine. The “null” route pointer is identified with the value `(unsigned short) 0`. It is guaranteed to differ from every valid route pointer.

Note also that the compiler aligns instances of data structures on word boundaries, so for example an array of ADDRs will end up occupying more than five bytes per element in the array. You may want to play with different data structure layouts in order to achieve better space usage.

### 4 Function Specifications

You will implement the following functions:

1. **RTPTR FTLookup(void *packetptr)**. The single input is a pointer to the first byte of a packet. The return value is a RTPTR, which is null if there is no matching entry in the prefix database. If FTLookup() is called with the same destination and source addresses several consecutive times, it **must** return the same route pointer.

2. **void FTAddPrefix(ADDR a, int prefLen, RTPTR *routes, int numRoutes)**. Adds a prefix along with the associated route pointer(s). The `prefLen` parameter indicates the number of significant bits in the prefix.

   Note that this counts from what would normally be the “right” of the word. So if `a` contains `0x0018ac19df` and `prefLen` is 30, the prefix is actually `1111101110011000001101000110`, i.e. the bits are “read” right-to-left. This should present no problem when comparing destination addresses to prefixes, since both are presented in the same order; you just need to make sure you do the shifting and masking correctly.

   The `routes` parameter points to an array of RTPTR structures, whose length is indicated by the `numRoutes` parameter. **The value of `numRoutes` is guaranteed not to exceed 10.** In practice, the number will typically be smaller, like 3 or 4.
3. `void FTNewRoute(ADDR a, int prefLen, RTPTR r)`. Associates a route pointer with a prefix that is already in the database (and already has multiple routes—see next section). After this call, the given route pointer can be returned for addresses for which the given prefix is the best match. Note that this function is never called to put a new prefix in the database.

4. `void FTRemovePrefix(ADDR a, int prefLen)`. Removes a prefix and all of its associated route pointers from the database. The prefix must have been inserted previously using `FTAddPrefix()`. (In other words, it is OK for your program to crash if `FTRemovePrefix()` is called on a prefix that had not been previously added.) After this call returns, calls to `FTLookup()` will never return the formerly-associated route pointer.

5. `void FTRemoveRT(ADDR a, int prefLen, RTPTR r)`. Removes one of multiple route pointers associated with the given prefix. Precondition: the given prefix is in the database, and the given route pointer is one of its (multiple) associated route pointers.

6. `void FlowFinished(ADDR src, ADDR dest)`.

## 5 Further Details

The status of a prefix as “single-route” or “multiple-route/load-balancing” is fixed the first time it is added to the database. That is, if a prefix is first added with a single route pointer, then neither `FTNewRoute()` nor `FTRemoveRoute()` will ever be called on it. If a prefix is initially added with multiple route pointers, then either `FTRemoveRoute()` or `FTNewRoute()` may be called multiple times on it. However, the number of associated route pointers will never decrease below 0; though it may be decreased to 1 (and subsequently increased again).

Since database updates are typically done off-line, you may assume that lookups will never be invoked when updates are in progress. Thus your code need not deal with mutual exclusion.

Unfortunately, there is no bound on the number of different flows that may match a given prefix at any time. (Well, there is, but it is too large to be useful.) However, having more than 250 simultaneous flows matching the same destination prefix is expected to be extremely rare.

The distribution of destinations among forwarded packets is expected to be consistent with those of a backbone router: a high degree of multiplexing, with little locality. The number of concurrently-active flows could be as high as 150,000 (not all of these will be using load-balanced routes, of course).

## 6 Evaluation

Your code will be evaluated on its correctness, speed, and memory usage. Lookup speed will be measured by building a database (of approximately 100K prefixes) and calling the lookup function on a hundred thousand or so simulated packets. Update speed will be evaluated on a second timing run in which database updates (additions and deletions) are interspersed among lookups. When you turn in the code, you will also turn in a description and analysis of its memory usage, in particular how much space is required for each prefix in the average and worst cases.