Implementing Lightweight Routing for BSD TCP/IP

Antti Kantee <pooka@cs.hut.fi>
Helsinki University of Technology

Johannes Helander <jvh@microsoft.com>
Microsoft Research

ABSTRACT

We present a lightweight leaf node routing algorithm and implementation for the BSD networking stack and discuss why the new approach was difficult to implement. The key idea in the algorithm was to observe that end hosts will always send packets to the default gateway and therefore are not required to do actual routing. By removing routing functionality not required for embedded devices, the size of the compiled BSD routing code was reduced by over 430% resulting in a 22% overall reduction of the networking stack; this is more than the size of the TCP module. As a direct result, the savings enabled the inclusion of TCP functionality in embedded systems where is was not possible earlier. The features and performance of lightweight routing were shown to be similar to the historic BSD routing code in the embedded case.

1. Introduction

Despite ever-increasing hardware resources, there are still places where it pays to be as small as possible. One of these places is on an embedded device, where kilobytes of additional memory will cost some cents per unit manufactured. Once this cost is multiplied by the number of millions of units produced, the business motivations in investing to smaller code become very clear. From a software engineering point-of-view, however, smaller code can either have advantages or disadvantages. It depends on if the reduction was accomplished by architectural structurization or micro-optimization of the existing architecture and their respective proportions.

While it is relatively easy to write software to match a specification and a certain size target, the challenge is in that the real world is very real. It is far from uncommon to see quirks and workarounds in code which has to interoperate with counterparts from other vendors. This is due to specifications being written in spoken language and always containing too much room for interpretation. Hence, it reasons to say that writing software interoperable with real world implementations is difficult.

The BSD TCP/IP implementation has a proven track record in interoperating with the real world. In fact, it can be said to define TCP/IP interoperability. Therefore, when aiming for TCP/IP interoperability, using BSD code would be a good approach even if explicitly tiny implementations [1] are available. The BSD code does present some challenges for use and in our case this in fact is the large code size. Some of it can be shaved off using trivial modifications, but taking it beyond a certain point gets hard. Another option for making code more compact is to use code size reduction techniques [2] in compilers, but once again that will carry the reduction only a certain distance. To really reduce code size, the structure and feature-set must be attacked.

Routing is the process of establishing the course to get from point A to point B. In the IP networking stack this means deciding where to deliver packets when given a target IP address.

This decision is commonly made on all intermediate hops when the packet travels from one endpoint to another. Routing is typically implemented considering the most generic case: a router which must know where to forward packets for tens of thousands of different destination networks. For the typical end host, however, it is enough to be able to forward packets to the unique local network router.

In this paper we present a novel idea for implementing routing in leaf nodes. We show how it was implemented for the 4.4BSDLite2 BSD networking stack and discuss why it was difficult to do. This includes pointing out why the implementation is not ready for general-purpose BSD consumption. We also attempt to outline future improvements in the area for a more modular TCP/IP stack.

The rest of the paper is structured as follows: Chapter 2 gives a short overview of our platform, Microsoft Invisible Computing, and defines our requirements. Chapter 3 presents the current networking stack implementation and our trivial improvements and describes the situation with the routing code. Chapter 4 presents our solution for a major size improvement in the routing code and discusses its implementation. Chapter 5 presents the results and experimental figures for the implementation and finally Chapter 6 presents our conclusions and lines out the future work in the area.

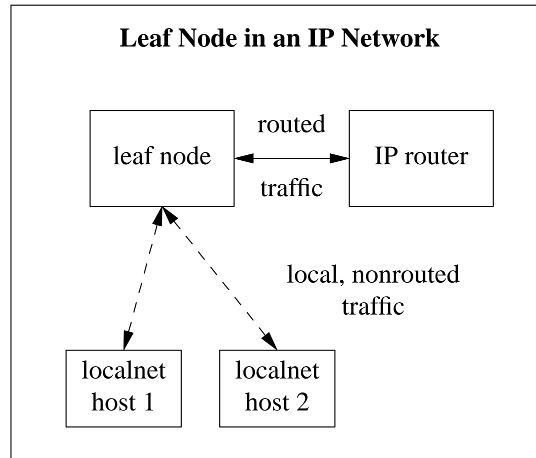
2. Microsoft Invisible Computing

Microsoft Invisible Computing [3] is a research prototype for making small devices part of the seamless computing world. It provides a compact middleware for constructing embedded web services applications and a small component based Real-Time Operating System with TCP/IP networking to make the middleware run straight on the metal on several embedded processors. The goal is to make it easy to build custom smart devices and consumer electronics, especially battery operated; and to support research in invisible computing, operating systems, networking, ubiquitous computing, sensor nets, distributed systems, object-oriented design, and wireless communication.

Due to the deployment scenarios of, small size for the system is an absolute requirement. Otherwise several interesting cases would out of reach because of size limitations.

2.1. Requirements

Our requirements for the networking code are dictated by our operating environment. The current target environment is an IPv4-only network. Microsoft Invisible Computing is never being used in the capacity of an IP router, but rather as a data processor positioned as a leaf in the IP network. The system may have multiple interfaces, but at most one of them will be connected a smart router.



The main requirement for this work was the reduction of compiled object code size to better be in line with the target environment hardware size constraints.

Microsoft Invisible Computing Module Sizes

Module	ROM	RAM
BASE	21856	272
MACHDEP	5248	1680
NET	76904	2004
XML	12628	16
SOAP	52668	436
HTTP	22688	0
TCP	13744	132
DNS	11052	344
WSMAN	6740	0
SOAPMETA	17248	20
CRT	14036	32
DRIVERS	11360	308

2.2. Size Analysis

For the purpose of keeping the system small, its size is regularly monitored after a build to catch any "creeping featurism" early on. This information is also useful for doing size analysis.

Clearly, the largest individual module in the system is the network stack. Therefore it is the best candidate for reduction. The SOAP module is also large, but it was already written from ground up with minimalism in mind.

3. Networking Stack

As was identified in the previous chapter, the largest individual component of Microsoft Invisible Computing is the BSD networking stack and therefore the ideal candidate for size reduction.

Our target for the project was to make the networking stack smaller. Some trivial enhancements to the networking stack had already been made earlier (and were included the the table "Microsoft Invisible Computing Module Sizes"). These mostly included replacing macros with function calls and `#ifdef`'ing out all unnecessary bookkeeping code not required for the release build.

For example, the memory buffers (*mbufs*) used by the network code are written so that most operations handling them are macros. This reduces function call overhead when dealing with them. On past computer architectures reducing the number of function calls resulted in much faster code, but with present-day hardware inline expansion plays havok on memory footprint and cache behaviour. Additionally, with modern CPU architectures and their CPU/memory speed ratio, smaller code is usually faster. Furthermore, macro-style programming usually does very little to encourage the separation of an interface from the implementation. It also makes run-time loadable components an impossibility due to including the internal representation already in the "calling" code.

Also, another easy way to micro-optimize the routing code size is to replace toggles with inline-`#defined` values. On RISC architectures, this avoids a separate load instruction every time the toggle is referenced, in addition to the RAM required to store the variable. The obvious downside is losing the ability to toggle the values runtime.

Even though these methods provide valuable size reduction [4], it eventually becomes impossible to reduce the size any further with this approach.

The size of the compiled individual objects was analyzed to provide good candidates for reduction. While no individual routing module is at the top of the list, further examination indicates that the routing modules (radix, rtsock, route, and to some degree arp) combined are a sizeable chunk.

Object File Sizes (20 largest)

object file name	text size (bytes)
tcp_in.obj	7028
u_sock.obj	5308
endpoint.obj	4624
ip_out.obj	4468
cbuffer.obj	4292
if.obj	3832
dhcp_sub.obj	3832
radix.obj	3788
ip_in.obj	3612
u_sock2.obj	3516
in.obj	3492
rtsock.obj	3484
arp.obj	3392
u_mbuf.obj	3016
route.obj	2884
udp_req.obj	2588
ip_icmp.obj	2360
ethernet.obj	2316
tcp_out.obj	2200
in_pcb.obj	2148

3.1. 4.4BSD Routing Code

The current "off-the-shelf" BSD routing code has its roots in 4.4BSD [5]. That code was written with several targets in mind:

- nodes connected to the network through a single interface
- nodes connected to the network through multiple interfaces
- routers involved in packet forwarding
- support for multiple address families
- good performance of computer architectures of that era

While it works for all of the above, it contains extra payload for implementations not needing all the generic routing capabilities. Also, while the code is well described in literature [6], it is difficult to modify because it presents no clear interfaces but rather chooses to optimize itself for performance. Even though this was understandably a noble goal for the original implementation,

hardware developments have made most of the employed tricks unnecessary and sometimes even counterproductive.

The kernel routing code itself can be thought of to be divided into three different modules: routing database, routing socket and in-kernel routing interface. Additionally, link layer addressing is joint at the hip to the routing code.

The Routing Database

The routing table information within the kernel is stored within the routing database. This is implemented as a radix tree in 4.4BSD and is typically found under `src/net/radix.c` in the source tree. The radix tree tries to optimize the amount of bit comparisons required to find the most specific match[1] for the given search key from the database.

The interface to this code is optimized more towards performance than to provide a modular database interface. For example, the `rn_search()` function returns the subtree in which the key resides - a clear bias toward a tree-shaped implementation. There is also no clear structure that is supposed to be passed to the radix tree code. Rather, it expects a `void *` with a `struct sockaddr` type memory layout beneath it. The details of the layout are configured at initialization time per protocol family using information from `struct domain`. A typed interface would be preferable to reduce the chance of programming error.

The routing database algorithm [7] is based on a modified version of the radix search trie [8]. However, several drawbacks involving modern hardware architectures have been identified [9], and clinging on to the historic code is not particularly relevant. Even so, it must be kept in mind that the existing implementation is reasonably efficient and, above all, it has proven to work in the real world.

Route Request Interface

The interface for requesting routing information is implemented in `net/route.c`. It essentially supports the following features:

- route query: `rtalloc()` and `rtalloc1()`
- freeing the allocated route: `rtfree()` (note: this does not remove the route from the database)
- rtdirection handling: `rtredirect()`
- interface for use by the routing socket: `rtrequest()`

The route asked for will be provided in `struct route` or `struct rtentry`. The difference between these two is that the former contains a pointer to the latter and a `struct sockaddr` describing the destination; we will see why this is necessary later. `struct rtentry` itself contains the essence of a route: for example the data storage elements used by the radix tree, the gateway, some statistics on the route, the interface used for the route, and so forth.

The Routing Socket

The 4.4BSD kernel does not implement a routing policy, it merely forwards packets according to a set of rules. Routing policies, i.e. decision on what the forwarding rules should look like, are made in userspace[2]. This means that the routing policy implementation in userspace must be able to communicate its decisions to the kernel and equally the kernel must be able to communicate any routing information it receives to userspace.

The method for communication is called a routing socket, i.e. a socket opened using the protocol family `PF_ROUTE`. Messages are then exchanged through it back and forth using `struct rt_msghdr` to describe each exchange. It is for example possible to set and change a route.

The routing messages used by the routing socket are spread also elsewhere into the kernel, although the exact message format is contained in the routing socket code. Other components involved must be able to receive information about routes going up or down and must be able to provide information if their state changes. For example, if an interface is detached, routing packets through it is no longer possible and the entire routing chain must be made to know about this.

Another purpose for routing sockets is to be a mechanism to communicate over the user-kernel barrier present in most modern operating systems. This is something which is not required for systems operating on machines without any form of memory protection, as is commonly the case with

[1] The one where the netmask stored in the database has least amount of 0-bits.

[2] Technically it would be possible to do it in the kernel also. Userspace programming is just easier most of the time.

microcontrollers and, in this special case, Microsoft Invisible Computing itself.

Link-layer Routing

In addition to doing network level routing, the routing code also handles link-layer routing. This means that for example the ARP cache for IPv4 Ethernet is tightly coupled to the routing code.

For all local networks in the system, the routing table contains a route with the approate address/netmask as the key and the respective interface as the gateway. These route entries have a cloning flag, `RTF_CLONING`, set. If the radix lookup produces a route with the cloning flag set, the flag signals that a route entry for the queried address should be created (‘‘cloned’’) and another lookup performed[3]. The lookup for the cloned route is a link layer resolution. As a special case, the ARP code has additional knowledge about this resolution process, since the ARP entries for the whole local network are not cached, but rather pulled in on a per-demand by doing an arpwhohas and interpreting the response (if any).

4. Lightweight Routing for BSD

The first attempt to make the routing code smaller was to replace the radix tree with a less complex structure, which would hopefully lead to reduced code size with equal performance in our target case. This was implemented as a Microsoft Invisible Computing Component Object Binary, COB, so the original radix code or the new lightweight code could be used interchangeably.

However, this facile approach produced very little in the form of results. The new code was only around 2kB smaller than the original. This was mostly due to the interfaces, which still required handling the complex data structures passed to and from the code. A more radical stratagem was therefore required.

Our target was to support only end nodes on the network. This changed the rules for the routing implementation. Routing on leaf nodes is actually an oxymoron. A leaf, per definition, cannot involve routing, since it is connected to the rest of the graph only from one point. The rest of the graph is either accessed through that point or not accessed at all.

The slight exception to the analysis above is that our node is only a leaf in the sense of the networking layer. As discussed in Chapter 3, the routing subsystem is also involved in making decisions about the link layer routing. Our "leaf" node will still be connected to an Ethernet, so support for link layer routing must be taken care of.

Based on the observation presented above, the conclusion was made: if routing is not done, code for it is not needed either. Size savings for non-existent code are substantial.

4.1. Lighweight Routing Algorithm

The old BSD routing code uses the routing table to make several decisions about the packet’s final destination. This information is encoded in the radix table and the decision of how to handle the packet is automatically done during the radix tree lookup. For example, if the lookup produces a link layer address, the code knows that the packet should be sent to a host on the local network, be it the default gateway or just some other host on the local net. Since we plan to have no radix tree, this information must otherwise be encoded into the system.

One important concept to keep in mind to avoid frequent confusion in the following discussion is the concept of the packet target in the routing code. While an IP packet header contains the final destination for the packet, the routing code is interested only in where the packet should be sent next. Therefore, when we are talking about the target, we mean the IP address of the next hop, not the final destination. Once the next hop is discovered, the packet is sent there and it is that host’s problem to look at the IP header to discover the final destination and again decide where it should be sent next.

Keeping the existing BSD semantics was a priority, so we modeled the new routing algorithm to do what the network stack used to do. The analysis lead to the following algorithm for routing a packet:

- If the target is a multicast address, send to the target address using a previously configured IP address as the multicast source.
- If we have an interface configured for the target address, send the packet to the loopback interface.

[3] Remember, the radix tree returns the most specific entry, so if a cloned route was already created for the address in question, the already-cloned entry will be returned and no further cloning done.

- If we have a point-to-point interface with destination address the same as the target address, send through the point-to-point interface to the target address.
- If we have an interface with the target address on the local network, send through that interface to the target address.
- Else, target the packet to the default gateway if one is configured.
- Otherwise, the target packet is discarded as being undelivarable.

Extracting from the above description, we need some configuration information to be able to operate:

- The configuration information for network interfaces
- The default gateway IP address
- Multicast interface information

In a normal BSD system, the interfaces and routes are configured using `ioctl()`'s and the routing socket with the tools *ifconfig* and *route*. While we still need a tool to configure networking interfaces, the routing table control tool can be greatly simplified because it is only required to perform two functions: set a gateway and delete a gateway.

4.2. Route Caching and Packet Forwarding

The old code does routing already on the transport layer and caches a `struct route` in `struct inpcb`. This is done so that lookup could be done for a certain connection once and then used thereafter. The other reason is source address selection: the packet destination must be known so that a source address for the packet can be selected.

This leads to some complications. First, for packet forwarding, the information available from the PCB is unavailable since the packet is not going through the transport layer at all. This means that the IP output routine must check if it has a valid route or not and do routing if it discovers it was passed a null route. Note, that source address selection will not be a problem for routers since the source address is already always present in the IP packet headers.

Second, the cached information is not always correct. It is possible to send packets to multiple different addresses from UDP sockets. Therefore, the cached information in the route structure must be verified each time cached route use is attempted and a relookup done if the cached target did not match the target at hand.

Since we do not concern ourselves with packet forwarding, the problem becomes slightly simpler: we can always pass valid route information to the `ip_output()` routine and do not have to do routing there any longer. However, we cannot easily move routing completely to the network layer, because we need the local address to select the right `inpcb`. It would clarify the structure greatly, though.

4.3. Multicast Addresses

Using multicast addresses [10] has some special treatment within the BSD networking stack. For receiving packets, it must be possible to join and part multicast groups and to check if we belong to a multicast group a packet was received for. This, though, is mostly in the domain of IGMP support and touches routing only slightly.

We already mentioned earlier that the routing algorithm explicitly checks for a multicast destination and instead of using the default gateway uses the multicast address as the nexthop destination. The ARP resolution routine then translates the multicast address to the respective ethermulti address and the packet is handled on a multicast capable router on the target network. This is what the old BSD code also did, including manually checking for multicast destination and possibly overriding the routing code decision to send the packet to the default gateway.

For selecting the source interface we use the same method as the BSD code. It is possible to set a system-wide default multicast output interface, although we do not currently provide a mechanism to set it on a per-socket basis.

4.4. ARP

ARP [11] is the link-layer routing mechanism used in and IPv4 + Ethernet environment. On the fundamental level it is a translation service. An address of some format goes in and an address of some other format comes out. In our case these are the IPv4 address of the packet nexthop destination and the Ethernet address of the destination. Some caching is also necessary to avoid doing costly lookups every time a packet is sent.

We chose to implement ARP just as it is described in the previous paragraph - a very simple lookup database. Due to the removal of the

radix tree, we could not use the old solution where the ARP cache was kept in the radix tree. Instead, we store all the addresses in a linked list. This structure was mostly an accident, as any other data structure would have worked much better. Even so, the typical real world case for us produces no difference in performance between a linked list and a more sophisticated data structure.

With the removal the radix tree and kernel routing interface the ARP module no longer has any knowledge about our network-level "routing" module. Its operation is entirely driven by the ethernet output routine, which tries to resolve the link layer addresses of packets as they are being sent.

The main interface for making ARP queries, `arpresolve()`, still exists in our new implementation. It either returns the translated address queried for or fires off a query for the address and creates a new embryonic ARP table entry. If a reply arrives, the embryonic ARP table entry is filled out. Since the old ARP code used the generic route expiration functionality, we had to implement similar timers in the new code. The `arp_rtrequest()` function, which had a clear routing socket bias, was replaced with `arp_addentry()` and `arp_delentry()` implementing functionality evident from the designations.

5. Results & Analysis

To recall, our target was to have equal functionality for leaf nodes with similar performance and significantly reduced code size.

5.1. Implementation

The current implementation is extremely intrusive and it is not possible to support the old and new routing code based on a compile-time option[4]. This is mostly due to incompatibility in the approaches between the original implementation and our implementation. While it is not a tempting idea to be restrained only to leaf nodes, it is on level with the current requirements for the networking part of the system.

[4] Of course, it would be possible to write a script that would wrap all the "-" lines from the diff behind `#ifdef OLDROUTING` and the "+" ones behind `#ifndef OLDROUTING`, but that would be an utter maintenance nightmare and therefore practically impossible.

5.2. Features

Our routing algorithm described in Chapter 4 is able to route packets from a leaf node to the local network(s) and the default gateway. It supports sending and receiving multicast packets and can join and depart multicast groups. This retains all of the features necessary for us from the old BSD routing code.

5.3. Code size

We analyze the code size savings by comparing the before and after size for the networking stack on our target platform.

Old/New Compiled Size

	old (bytes)	new (bytes)
entire stack†	77100	63276
radix.obj	3788	0
rtsock.obj	3484	0
arp.obj	3392	2656
route.obj	2884	480

The compiled ROM code size is almost 14kB smaller. Represented as a percentage, the new code is almost 22% smaller than the old code. The new routing code itself is over 430% smaller when compared to the old one!

As a comparison, the compiled size for the new TCP code for our target platform is 13276 bytes. Previously vendors were inclined to leave TCP out of systems and implement required functionality on top of UDP when then functionality of TCP would in reality have been required. Our result enables the inclusion of TCP without any loss in functionality or need to increase hardware resources to accomodate a larger code foorprint. The use of real TCP will typically also make the application smaller, since some of the desired TCP functionality, such as reliable transport, no longer needs to be implemented on the application layer.

An interesting point can be made by comparing the old and new ARP code sizes. The new ARP resolution code is almost 750 bytes (28%) smaller than the old ARP code. This is due to not having to implement the complex interfaces to the routing socket and radix tree.

† Not including the TCP module.

5.4. Performance

To measure the performance differences between the old and new code, a program which sends a UDP packet to several consecutive IP addresses was devised. Every UDP packet sent will cause a routing lookup because the target address does not not match the cached address.

We tested two cases: sending packets to the local network and sending packets to outside of the local network, i.e. sent to the gateway. All tests were executed while running Microsoft Invisible Computing as a regular process on top of Windows XP.

Routing Overhead Measurements

test	old (seconds)	new (seconds)
neighbour	8.98	13.84
remote	238.36	245.73

The test "neighbour" measured sending a packet to 128 consecutive addresses on the local network and looping this 16000 times. The test "remote" sent a packet to 67108864 (0x4000000) consecutive addresses outside the local network. The stack was modified to only do routing, not actually send the packets, for the duration of these tests. ARP lookup was done, but a dummy result was always returned when a match was found instead of invoking an arpwhohas query.

For the local network case, performance difference and be attributed to the difference between the computational complexity of radix tree and the O(n) performance of the linked lists of the new ARP code. If the latter was replaced with an O(log n) structure, performance would be equivalent. In most real world situations this will have very little effect, since communication is typically at most with a handful of local network hosts - not hundreds. Because a linked list implementation was already present in the system, the size gain of code reuse outweighs the handicap of performance degredation evident only in test cases crafted to be maximally adverse.

For the remote case, performance appears to be roughly the same and the difference in measurements can be attributed to noise because of running Microsoft Invisible Computing as a process. This is not surpising, since for both the new and old versions routing on a leaf node is all but a NOP. We are confident that equal or superior performance for a given embedded target architecture could be easily accomplished with some performance tuning, if in fact the performance were to be discovered to be worse at all.

6. Conclusions and Future Work

We presented a method for implementing routing on IP leaf nodes without implementing an IP routing table at all. The key was to explicitly teach the code what kind of routing we want instead of using routing table entries to express the same information. The benefit was huge, over 430%, code savings due to not having to express rules on an abstract level when they could be expressed on a concrete level. This enables, for example, the inclusion of TCP support in embedded devices where it previously was not present due to size constraints; it is not always necessary to write everything from scratch even though the target is to be compact.

The routing code presented works for leaf nodes, but left a more general approach to be desired. Support for general-purpose routers should be investigated as an alternate implementation. It may not be worthwhile to carry the old routing code along at all, but rather to start from a clean slate or the work presented here.

The interface between the routing code and the networking stack proper was developed into a direction which, with additional work, will enable the development of pluggable routing modules optimized for speed, size and/or features depending on the target environment.

We identified two places within the networking stack for a use of a database: IP routing and ARP translation. These were previously implemented using the same database, the radix tree. However, there are many more places within a kernel that could benefit from a more general database type of component instead of relying on ad-hoc data retrieval structures, which more often than not happen to be linked lists. Even a simple, lightweight data storage algorithm, such as A-trees [12], would provide better options for generic data storage and retrieval when readily available in all BSD operating systems.

Finally, routing could also be considered a more general-purpose component. There is no need to artificially separate OS and application routing in an embedded environment. A simple example is the Internet Relay Chat service, but the topic of routing is becoming significantly popular

with the emergence of various peer-to-peer technologies [13]. Advantages, in addition to the obvious size savings, would be better testing coverage and reduced maintenance costs.

Acknowledgements

We wish to thank Alessandro Forin for inspirational and insightful conversations and Sasha Nosov for motivation.

Further Information

Microsoft Invisible Computing, including the routing implementation described here, is available as source code under the Microsoft Shared Source License from the website: `http://research.microsoft.com/invisible/`

References

1. Adam Dunkels, *Full TCP/IP for 8-Bit Architectures,* First International Conference On Mobile Applications, Systems and Services (MOBISYS 2003) (May 2003).
2. Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto, "Survey of code-size reduction methods," *ACM Computing Surveys,* Vol 35, Issue 3, pp. 223 -- 267 (September 2003).
3. Johannes Helander and Alessandro Forin, *MMLite: A Highly Componentized System Architecture,* pp. 96 -- 103, Eight ACM SIGOPS European Workshop (1998).
4. Ruby Li, *How to Reduce Code Size (and Memory Cost) Without Sacrificing Performance,* Embedded.com (November 2005).
5. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).
6. Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated, Vol2.,* Addison-Wesley (1995).
7. Keith Sklower, *A Tree-Based Packet Routing Table for Berkeley UNIX,* pp. 93 -- 99, USENIX Association Conference Proceedings (1991).
8. Robert Sedgewick, *Algorithms in C,* Addison-Wesley (1990).
9. André Oppermann, *Optimizing the FreeBSD IP and TCP Stack,* Fourth European BSD Conference (not in proceedings) (2005).
10. S. Deering, *Host Extensions for IP Multicasting,* RFC 1112 (1989).
11. David C. Plummer, *An Ethernet Address Resolution Protocol,* RFC 826 (1982).
12. Alistair Crooks, *The A-Tree - a Simpler, More Efficient B-Tree,* pp. 185 -- 201, Proceedings of the 3rd European BSD Conference (2004).
13. Stephanos Androutsellis-Theotokis and Diomidis Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Computing Surveys,* Vol 36, Issue 4, pp. 335 -- 371 (December 2004).