Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Linux IPv6 Networking

Past, Present, and Future

Hideaki Yoshifuji
The University of Tokyo
yoshfuji@linux-ipv6.org

Kazunori Miyazawa
Yokogawa Electric Corporation
miyazawa@linux-ipv6.org

Yuji Sekiya
The University of Tokyo
sekiya@linux-ipv6.org

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

Jun Murai
Keio University
jun@wide.ad.jp

Abstract

In order to deploy high-quality IPv6 protocol stack, we, USAGI Project[13], have analyzed and addressed issues on Linux IPv6 implementation. In this paper / in our talk in OLS2003, we describe the analysis of Linux IPv6 protocol stack, improvements and implementation of the IPv6 and IPsec protocol stack and patches which are integrated into the mainline kernel. We will explain the impacts of our API improvements on network applications.

We want to discuss on missing pieces and direction for future development.

As a demonstration we would like to provide IPv6 network connectivity to the OLS2003 meeting venue.

1 Introduction

Establishment of IPv6, as a next-generation Internet protocol to IPv4, started from the beginning of the 1990's. The aspect of IPv6 is on providing the solution to the protocol scalability, the greatest problem IPv4 facing as the Internet growing larger. In detail, IPv6 differ from IPv4 in following ways.

- 128bit address space.
- Forbidding of packet fragmentation in intermediate routers.
- Flexible feature extension using extension headers.
- Supporting security features by default.
- Supporting Plug & Play features by default.

Currently, IPv6 is at the final phase of standardization. Fundamental specifications are almost fixed and commercial products with IPv6 support are being deployed in the market. International leased lines for IPv6 are out as well. IPv6 has expanded the existing Internet by providing solutions to protocol scalability and beginning to grow as a standard for connecting everything, not just existing computers.

2 The Dawn of Linux IPv6 Stack

Linux IPv6 implementation was originally developed by Pedro Roque and integrated into mainline kernel at the end of 1996 in early 2.1 days and this was the one of the earliest implementation of IPv6 stack in the world.

In 1998, Linux IPv6 Users JP, which is a group of Linux IPv6 users in Japan, examined the status of IPv6 implementation in Linux and recognized the several grave issues[1].

- lack of scope in socket API; for example Linux does not have `sin6_scope_id` member in `sockaddr_in6{}`.
- So many bugs (Table 1), found by the TAHI [11] IPv6 Conformance Test Suite, especially in Neighbor Discovery and Stateless Address Auto-configuration.
- "default routes" are ignored on routers
- many missing features such as IPsec, Mobile IP.

These were because the stack had not been well-maintained / developed since 2.1 because there were not so widely used by Linux hackers. Thus, there had been few new features. Implementation had not followed the specification even the spec had been changed, and then, conformity to Specification became very low.

In 2.3 days, they, the Linux IPv6 Users JP, developed `sin6_scope_id` support. Their code was integrated into mainline kernel. There, however, were very few change in 2.3 days other than this.

Considering above circumstances, USAGI Project was lunched in October, 2000. USAGI Project is a project which aims to provide improved IPv6 stack on Linux; It seems to be required (almost) full-time task-force which commits Linux IPv6 development. There are similar organization called KAME [6], which provides IPv6 stack on BSD Operating systems such as FreeBSD, NetBSD, OpenBSD, and BSD/OS. However, KAME Project does not target their development on Linux. It is important to provide high-quality IPv6 stack on Linux, which is one of the most popular free open-source operating systems in the world, and widely used in embedded systems, for IPv6 to propagate.

Table 1: Summary of TAHI Conformance Test (linux-2.2.15, %)

Test Series	Pass	Warn	Fail
Spec.	94	6	0
ICMPv6	100	0	0
Neighbor Discovery	34	0	66
Autoconf	4	0	96
PMTU	50	0	50
IPv6/IPv4 Tunnel	100	0	0
Robustness	100	0	0

3 USAGI Challenges in Linux IPv6 Stack

Since USAGI Project started, we have continued analyzing issues we faced. In this section, we describes issues we found and our challenges to solve them.

3.1 ND and Addrconf

Neighbor Discovery (ND [8]) and Stateless Address Auto-configuration (Addrconf, [12]) are ones of the core features of IPv6. They take very important role to keep stable communication. However, the results of the Conformance Tests of Linux IPv6 stack were bad.

We've tried to fix the problems in the following way.

- Reinforcing checking illegal ND Messages
- Improving control times for ND state transition and address validation.

[1] We will discuss them later.

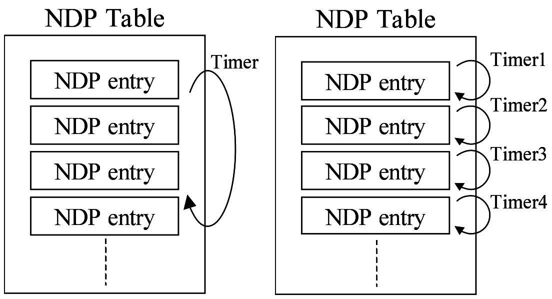

Figure 1: NDP Table: Linux vs USAGI

- Fixing ND state transition

3.1.1 Improving Timers for ND

The state of a neighbor is changed by events such as incoming Neighbor Advertisement message and timer expiration. It is required to manage timer accurately.

However, the existing Linux IPv6 protocol stack checks reachability of neighbor nodes with a single kernel timer(Figure 1 (Left)). Consequently, reachability were checked in constant intervals, regardless of the status for each node.

Therefore, USAGI Project improved this kernel timer to check each NDP entry independently as shown in Figure 1 (Right). Thus, resource management for a neighbor including mutual exception is simplified, and it is possible to enable and disable timer separately for each NDP entry, and prevent check made to unnecessary NDP entries. Moreover, it is possible to exchange messages correspondent to the status of each NDP entry as defined in the NDP specifications.

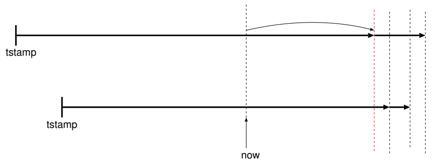

Figure 2: Dynamic Address Validation Timer

3.1.2 Improving Timers for Address Validation

As ND is, the state of an address is changed by events such as incoming Router Advertisement and time expiration. It is required to manage timer accurately, especially for Privacy Extensions [7].

However, the existing Linux IPv6 protocol stack performs validity checks with a long-term, constant, single kernel timer.

USAGI Project introduced new dynamic timer. When the timer expires, the timeout function visits each address for validation and determining the next timeout (Figure 2) with minimum and maximum interval between timeouts. Thus, accuracy of timer is improved. It is usual that several addresses request next timeout at (almost) the same time, introducing minimum interval between operations aggregates them and suppresses load of timer events.

Table 2 shows the results of these improvements and other minor fixes. Neighbor Discovery and Autoconf are significantly improved.

3.2 Routing Restructuring

3.2.1 Default Route Support on Routers

In routing table using radix tree[9], the top of the tree is the host which possesses the information regarding "default route." However, as shown in Figure 3, Linux IPv6

Table 2: Summary of TAHI Conformance Test (usagi24-s20020401, %)

Test Series	Pass	Warn	Fail
Spec.	100	0	0
ICMPv6	100	0	0
Neighbor Discovery	79	5	15
Autoconf	98	2	0
PMTU	50	0	50
IPv6/IPv4 Tunnel	100	0	0
Robustness	100	0	0

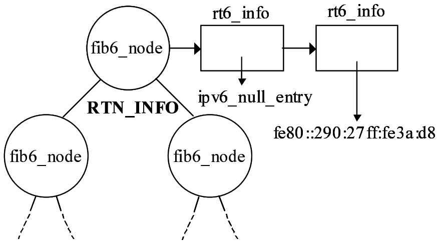

Figure 3: Linux IPv6 Routing Table Structure

protocol stack has a radix tree with fixed node information on top and it points to `ipv6_null_entry`. Therefore, when default route is added, the information is attached next to the `rt6_info{}` structure which contains `ipv6_null_entry`. This causes default route not to be referred.

In USAGI implementation, we replace the `ipv6_null_entry` with the new entry when adding a new routing entry on the top level root of the tree (Figure 4). When the last route is being deleted from the the top level root of the tree, we re-insert `ipv6_null_entry`. Thus, we can insert and remove the "default route" entries properly to/from the routing table.

3.3 Improvements on Router Selection

We pick one default router from the default router list and round-robin the default router list when it becomes unreachable. The default router is pointed by the `rt6_dflt_pointer`, which is guarded by `rt6_dflt_lock`, and default routers are stored on the top level root node of the routing tree (Figure 5). In this implementation, there were several issues.

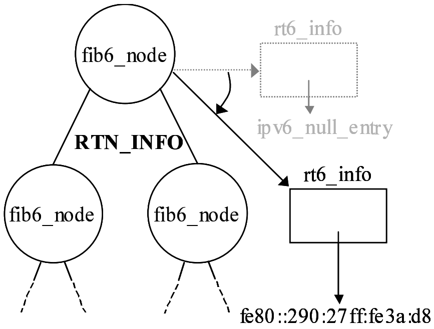

Figure 4: USAGI IPv6 Routing Table Structure

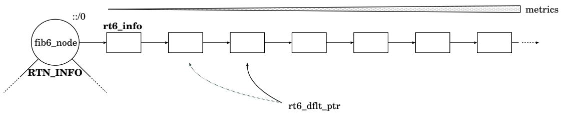

Figure 5: Default Routers in Linux

- `rt6_dflt_pointer` is reset when routing is modified; this happens very often and routers are not equally selected.
- We did not regard the metrics; we could not force using routes with smaller metrics (which is probably added manually.)

"Default Router Preferences, More-Specific Routes, and Load Sharing" [1] improves the ability of hosts to pick an appropriate router, especially when the host is multi-homed and the routers are on different links, and mandates load-share between routers with same "preferences."

To implement this specification, we stores preference (2 bits) of routes into the flags of the

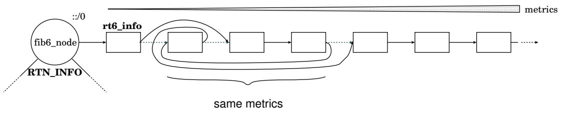

Figure 6: New Method for Route Round-robin

routing informations instead of reflecting it to the metrics; We would have to fix up routing table when receiving RA.

We also make a new generic round-robin code for the routes with same metrics (Figure 6). We use this for all routes and the `rt6_dflt_pointer` and `rt6_dflt_lock` are eliminated. Now we are free from above issues.

4 Linux IPv6 in 2.6

In this section, we describes key changes of IPv6 networking code between 2.4 and 2.6 [2] Then we try to describes how IPsec works.

4.1 Key Changes in 2.6.x

We have been developing IPv6 actively since end of 2.3.x era. However, only several selected changes were integrated into the mainline tree. One reason was that we were obscure and novice on kernel development.

After experiencing about two years of kernel development, we started integrating our efforts to the mainline kernel from the fall of 2002 more aggressively than before.

We have been being fixing several bugs such as:

- Verify ND options properly
- Refine IPv6 Address Validation Timer
- Fixing source address of MLD messages
- Avoiding garbage sin6_scope_id for MSG_ERRQUEUE message

In addition to these bug fixing, we've integrated following new features:

IPsec for IPv6
This is based on IPsec for IPv4, developed by David S. Miller et.al. See section 4.2.

Default route support on router
See section 3.2.1.

IPV6_V6ONLY support
See section 5.2.4.

ICMP6 rate limit support
Added rate-limit sysctl for ICMPv6 like for ICMP.

Privacy Extensions [7]
Assign randomized interface identifier to improve privacy.

AF-independent XFRM Infrastructure
Split up XFRM subsystem into af-independent portion and af-specific portion. Section 4.2.3.

Per-interface Statistics Infrastructure
Make a new infrastructure to provide per-interface statistics information.

4.2 IPsec

IP security provides security functionality for IP layer. An implementation of IPv4 IPsec by FreeS/WAN is available for years, however, the code was never merged into the mainline kernel. In 2000, IABG Project provided IPv6 support patch for FreeS/WAN. It, however, was "patched" and also unlikely to be merged into the mainline kernel.

[2] Some changes will be appeared in 2.4.21 (or later 2.4.x series).

We redesigned the architecture for multi-protocol, both IPv4 and IPv6, extensible IPsec. In our design, IPv4 and IPv6 share the Security Policy Database (SPD) and Security Association Database (SAD). CryptoAPI and its variants are used for cryptographic, digesting and compression/decompression algorithms.

4.2.1 Stackable Destination and XFRM

A new framework for processing IP packets has been introduced into linux-2.5.x. It is called "stackable destination" and XFRM.

Stackable destination is like a linked list of `dst{}`, which is made temporally and cached. We are able to insert another `dst{}` to original `dst{}` and make a stack of the `dst{}` structure. `dst{}` normally has a pointer to `xfrm_state{}`, whose output provides some functionality, i.e. transformation, for the packet.

XFRM stands for transformer. `xfrm_policy{}` and `xfrm_state{}` represent IPsec policy and IPsec SA respectively. `xfrm_state{}` is associated with `xfrm_policy{}` by `xfrm_tmpl{}`. SPD consists of `xfrm_policy{}`. SAD also consist of `xfrm_state{}`.

4.2.2 Packet Processing

The output process of IPsec fully uses this architecture. The order of primal functions are `xfrm_lookup()`, `xfrm_tmpl_resolve()`, `xfrm_bundle_create()` and `dst_output()`. `xfrm_lookup()` looks up `xfrm_policy{}` in SPD after routing resolution. At the moment the parameter `dst{}` in the stack points original `dst{}` structure. `xfrm_tmpl_resolve()` is called in `xfrm_lookup()` to resolve `xfrm_tmpl{}` in `xfrm_policy{}` which represents how the packet is processed and find `xfrm_state{}` matched up with `xfrm_tmpl{}`. This process is equivalent to looking up IPsec SA or IPsec SA bundle matched with IPsec policy. `xfrm_bundle_create()` creates the stackable destination and IPsec SA bundle if multiple SA are needed. These functions are called at routing resolution. `dst_output()` is called after building up the packet. Each output routine specified by the function pointer in the `dst{}` is called along with the chain of `dst{}`. This pointer points e.g `esp6_output()`. The output function is able to use `xfrm_state{}` from `dst{}` pointer in `sk_buff{}`.

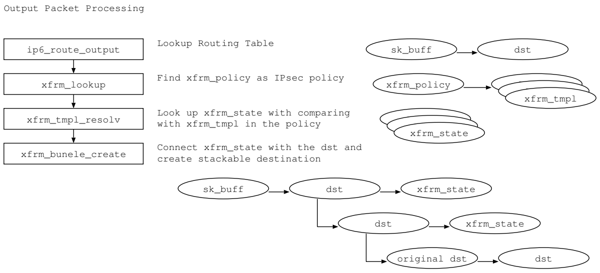

Figure 7: IPsec output process

The input process for IPsec is more simple than output. AH and ESP process routines are registered to `inet6_protos[]` at initiation. The kernel parse a packet and call the routines when protocol is AH or ESP. To unified extension header processing, all header types and handlers are registered in `inet6_protos[]` like upper layer protocol. IPsec packet process is looking up `xfrm_state{}` and process it. When it succeeds, used `xfrm_state{}` pointer keep in `sec_path{}` in `sk_buff{}` which contains the packet. After processing IPsec, the kernel call `xfrm_policy_check()` at entrance of upper layer process. In `xfrm_policy_check()` the kernel match up `xfrm_tmpl{}` in `xfrm_policy{}` and

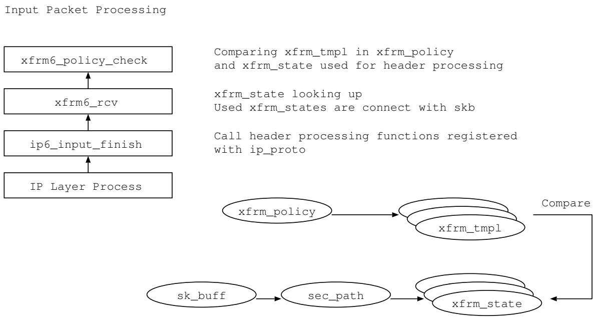

Figure 8: IPsec input process

`xfrm_state{}` kept in `sec_path{}`.

4.2.3 AF Indenendent XFRM Infrastructure

Since core functionality of the XFRM engine is common among address families, AF independent XFRM infrastructure has been introduced.

Address family specific XFRM functions are registered via address family information tables, e.g. `xfrm_policy_afinfo{}` and `xfrm_state_afinfo{}`. Common variables are also passed via the tables.

4.2.4 Key And Policy Management Interface

`PF_KEY` and `netlink(7)` interface are provided as IPsec interfaces. `PF_KEY` provides interface to maintain SAD and SPD. `PF_KEY` protocol is defined in RFC2367 but it is not so enough to maintain IPsec that implementation of `PF_KEY` is ordinary extended. The extension is different each implementation. Linux-2.5.x `PF_KEY` is compatible with KAME.

4.2.5 Test Results

On 24th April, 2003, Tom Lendacky reported to netdev mailing list that Test results of Linux-2.5 IPsec are very excellent(Table 3).

We have tried to fix the bugs in IPv6 IPsec fragmentation, and they should be fixed for now.

Test Series	Pass	Warn	Fail
ipsec	95	2	3
ipsec4	98	2	0
ipsec4-udp	96	4	0

Table 3: Summary of TAHI Conformance Test (linux-2.5.58, %)

5 Modern Programming Style for Network Applications

It is requested that applications should support both IPv4 and IPv6. In this section, we try to describe modern programming style for network applications.

5.1 Socket API and Protocol Independency

The Socket API is the framework of programming for communication including networking via the Internet. It was designed to be protocol independent. Communication is abstracted by the socket descriptor, and endpoint information, which is protocol dependent, is passed via opaque pointers to the generic socket address structure `sockaddr`.

IPv6 networking is also supported in this framework. New address family `AF_INET6` and IPv6 socket address structure `sockaddr_in6` are defined.

5.2 Protocol Independent Programming

The framework of the Socket API between the kernel and the user space is basically protocol independent. However, since the socket address structure and the naming space of the protocol depend on the protocol (or address), it was protocol dependent to lookup the name/address and to setup the protocol specific socket address structure. This prevented application from the protocol independency.

In RFC2133[3], new two name-lookup functions are defined: `getaddrinfo()` and `getnameinfo()`. It abstracts translation between name/service representation and socket address structure.

5.2.1 getaddrinfo(3)

`getaddrinfo(3)` [2] is the protocol independent function for forward lookup (name to address). This function looks up the "node" and "service" on condition that is specified by the "hints." It returns dynamically allocated linked list of `addrinfo{}`. Each `addrinfo{}` includes information for `socket(2)`, `connect(2)` (or `bind(2)`, if `AI_PASSIVE` flag is specified in hints).

Thus, application is not required to know the details of socket address structure, now. A client application walks though the list trying to create a socket and trying to connecting remote host until one of the attempt succeeds. Likewise, a server application walks through the list trying to create a socket and trying to binding local address.

5.2.2 getnameinfo(3)

`getnameinfo(3)` [2] is the protocol independent function for reverse lookup (address to name). This function takes socket address structure and looks up node name and service name on condition specified by the flags. By using this function, application is not required to know the details of each socket address structure for extracting addresses and / or port number.

For example, to extract numeric service number from the socket address structure, use `getnameinfo(3)` with `NI_NUMERICSERV`, and convert the resulting service number using `atoul(3)`.

Sample programs using `getaddrinfo(3)` and `getnameinfo(3)` are provided in Appendix.

5.2.3 getifaddrs(3)

Other issue to support IPv6 in application how we know addresses on the node on which it is running. SIOCGIFADDR is used for IPv4, however, `ifreq{}` is not enough to store IPv6 socket address structure. There might be possibility to introduce new `ioctl(2)` to manage lager addresses, however, it is nasty to get information via buffer of fixed length. Thus, `getifaddrs(3)` was invented.[3] This function grubs network address information including netmask etc. on the node. MAC, IPv4 and IPv6 are supported for now. Application walks through the linked list returned from this function, looking for appropriate information using the family, flags etc.

Sample programs using `getifaddrs(3)` is provided in Appendix.

[3] BSDI's invention; this is not standardized yet.

5.2.4 IPV6_V6ONLY Socket Option

IPv6 sockets may be used for both IPv4 and IPv6 communications. IPv4-mapped IPv6 address is defined [5] to enable IPv6 application IPv4 address of an IPv4 node is represented as an IPv4-mapped IPv6 address in such applications.

Linux supports this feature and port space of TCP (or UDP) has been completely shared between IPv4 and IPv6 [4].

However, some applications may want to restrict their use of an IPv6 socket to IPv6 communication only. For these applications, `IPV6_V6ONLY` is defined in RFC3493 [2].

In 2.6, `IPV6_V6ONLY` socket option is supported. In this implementation, the 'IPv4-mapped" feature is enabled by default as before, and as spec says. If the `IPV6_V6ONLY` socket option is set to the IPv6 socket, the socket will not care about IPv4 address space at all.

As mentioned before, spec says this "IPv4-mapped" feature is enable by default. However, there are OSes, such as NetBSD, which do not enable that feature by default, or even which do not support that feature at all. These OSes are not RFC compliant, but, unfortunately, it is the real. So, application would need to take care of this situation. For example:

- Try to setup both IPv6 and IPv4 sockets.
- Set `IPV6_V6ONLY` socket option to the IPv6 socket, prior to perform `bind(2)`.
- It is not fatal to failed to set `IPV6_V6ONLY` socket option.
- Don't take it fatal unless all socket creation resulted in error.

The sample server provided in the Appendix is written in this manner.

6 Future Plans

Finally, we list our future plans. Here's our future plan, especially for this year.

- stabilizing IPv6 IPsec
- introducing IP Mobility to mainline
- completing porting ND fix to (pre-)linux-2.6 and submitting patches to mainline
- introducing generic IPv4,6 tunnel interface
- examining and stabilizing IPv6 Netfilter
- completing Prefix Delegation
- improving API support [2, 10]
- implementing IPv6 Multicast Routing

As of writing this paper, we're working hard stabilizing IPv6 and IPv6 IPsec stack in (pre-)linux-2.6.

We're also discussing how the Mobile IP should be implemented with the maintainers and HUT [4] people.

XFRM is flexible and promising framework in netwroking. We are able to and going to implement Mobile IP, generic tunnel etc.

We have been waiting for new documents for the APIs. It has taken long time to publish the new documents for APIs, however, new version are (about to) available. We'll follow them.

[4] In some OSes, such as FreeBSD 4.x, IPv4 socket can override IPv6 socket of the same port. We believe that this fashion is vulnerable to "binding closer" type attacks.

Multicast routing is probably the biggest missing piece; we will try to implement this, too.

References

[1] R. Drave and R. Hinden. Default router preferences, more-specific routes, and load sharing. Work in Progress, June 2002.

[2] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC3493, March 2003.

[3] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC2133, April 1997.

[4] GO Project. MIPL Mobile IPv6 for Linux. http://www.mipl.mediapoli.com/.

[5] R. Hinden and S. Deering. Ip version 6 addressing architecture. RFC2373, July 1998.

[6] KAME Project. KAME Project Web Page. http://www.kame.net.

[7] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in ipv6. RFC3041, January 2001.

[8] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC2461, December 1998.

[9] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, pages 93–104, 1991.

[10] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei. Advanced sockets api for ipv6. Work in Progress, March 2003.

[11] TAHI Project. Test and Verification for IPv6. http://www.tahi.org.

[12] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC2462, December 1998.

[13] USAGI Project. USAGI Project Web Page. http://www.linux-ipv6.org.

7 Appendix: Sample Application Written in Modern Manner

7.1 Client

```
/*
 * Sample Modern Client
 *
 * Usage:
 * % ./modern-client host.example.com daytime
 *
 * $Id: modern-client.c,v 1.1 2003/05/13 20:06:58 yoshfuji Exp $
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

int main(int argc, char **argv) {
    char *host, *port;
    struct addrinfo hints, *ai0, *ai;
    int s;
    int gai;

    /* check arguments */
    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Usage: %s [host] portnum\n", argv[0]);
        exit(1);
    }

    if (argc == 3) {
        host = argv[1];
        port = argv[2];
    } else {
        host = NULL;    /* loopback address */
        port = argv[1];
    }

    /* look-up name */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    gai = getaddrinfo(host, port, &hints, &ai0);
    if (gai) {
        fprintf(stderr,
                "getaddrinfo(): %s port %s: %s\n",
                host, port, gai_strerror(gai));
        exit(1);
    }

    /* loop connecting remote entity */
    s = -1;
    for (ai = ai0; ai; ai = ai->ai_next) {
        /* create a socket */
        s = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (s == -1)
            continue;

        /* connect */
        if (connect(s, ai->ai_addr, ai->ai_addrlen) == 0)
            break;

        close(s);
        s = -1;
    }

    /* free address information */
    freeaddrinfo(ai0);

    /* check if we have failed */
    if (s == -1) {
        fprintf(stderr, "Cannot connect to %s port %s\n",
                host != NULL ? host : "(null)",
                port);
        exit(1);
    }

    /* process loop */
    while (1) {
        ssize_t cc;
        char buf[1024];

        /* read from remote host */
        cc = read(s, buf, sizeof(buf));
        if (cc == -1) {
            perror("read");
            close(s);
            exit(1);
        } else if (cc == 0) {
            break;
        }

        /* output response */
        if (write(STDOUT_FILENO, buf, cc) == -1) {
            perror("write");
            close(s);
            exit(1);
        }
    }

    close(s);
    exit(0);
}
```

7.2 Server

```
/*
 * Sample Modern Server
 *
 * Usage:
 * % ./modern-server :: 12345
 *
 * $Id: modern-server.tex,v 1.1 2003/05/15 03:54:13 yoshfuji Exp $
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <sys/select.h>

#ifndef MAX_SOCKNUM
# define MAX_SOCKNUM FD_SETSIZE
#endif

static const char *message = "Hello, world!\n";

int main(int argc, char **argv) {
    char *host, *port;
    struct addrinfo hints, *ai0, *ai;
    int gai;
    int socknum = 0, *socklist = NULL;
    int maxfd = -1;
    fd_set fds_init, fds;
    int i;

    /* check arguments */
    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Usage: %s [host] portnum\n", argv[0]);
        exit(1);
    }

    if (argc == 3) {
        host = argv[1];
        port = argv[2];
    } else {
        host = NULL;    /* unspecified address */
        port = argv[1];
    }
    /* resolve address */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    gai = getaddrinfo(host, port, &hints, &ai0);
    if (gai) {
        fprintf(stderr,
                "getaddrinfo(): %s port %s: %s\n",
                host != NULL ? host : "(null)",
                port,
                gai_strerror(gai));
        exit(1);
    }

    /* initialize fd_set for select(2) */
    FD_ZERO(&fds_init);

    /* loop waiting for connection */
    for (ai = ai0; ai; ai = ai->ai_next) {
        int s;
        int *newlist;
#ifdef IPV6_V6ONLY
        int on = 1;
#endif

        /* create a socket */
        s = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (s == -1)
            continue;

#ifdef IPV6_V6ONLY
        if (ai->ai_family == AF_INET6 &&
            setsockopt(s,
                       IPPROTO_IPV6, IPV6_V6ONLY,
                       &on, sizeof(on)) == -1) {
            perror("setsockopt(IPV6_V6ONLY)");
            /*
             * Some systems do not support his option;
             * This error should no be fatal.
             */
        }
#endif

        /* listen */
        if (bind(s, ai->ai_addr, ai->ai_addrlen) == -1) {
            close(s);
            continue;
        }

        if (listen(s, 5) == -1) {
            close(s);
            continue;
        }

        if (s >= FD_SETSIZE || socknum >= MAX_SOCKNUM) {
            close(s);
            fprintf(stderr, "too many file/socket descriptors\n");
            break;
        }

        /* re-allocate list of socket */
        newlist = realloc(socklist, sizeof(int)*(socknum+1));
        if (newlist == NULL) {
            perror("realloc");
            close(s);
            break;      /* XXX: terminate immidiately? */
        }

        socklist = newlist;
        socklist[socknum++] = s;

        /* set fd_set */
        FD_SET(s, &fds_init);

        if (maxfd < s)
            maxfd = s;
    }

    /* free address information */
    freeaddrinfo(ai0);

    /* check if we have failed */
    if (socknum == 0) {
        fprintf(stderr,
                "Cannot allocate any listen sockets on %s port %s\n",
                host != NULL ? host : "(null)",
                port);
        exit(1);
    }

    while (1) {
      int i;

      fds = fds_init;

      if (select(maxfd + 1, &fds, NULL, NULL, NULL) == -1) {
          perror("select");
          continue;
      }

      for (i = 0; i < socknum; i++) {
          int sock = socklist[i];

          /* look up listener.
           * XXX: this is not fair between listers
           */
          if (FD_ISSET(sock, &fds)) {
              int newfd;
              struct sockaddr_storage ss;
              socklen_t sslen;
              ssize_t cc;
              char hostbuf[NI_MAXHOST];
              int gni;

              sslen = sizeof(ss);
              newfd = accept(sock, (struct sockaddr *)&ss, &sslen);
              if (newfd == -1) {
                  perror("accept");
                  continue;
              }

              gni = getnameinfo((struct sockaddr *)&ss, sslen,
                                hostbuf, sizeof(hostbuf),
                                NULL, 0,
                                NI_NUMERICHOST);
              if (gni)
                  strcpy(hostbuf, "???");       /*FIXME!*/

              printf("accept from %s\n", hostbuf);

              cc = write(newfd, message, strlen(message));
              if (cc == -1) {
                  perror("write");
              } else if (cc != strlen(message)) {
                  fprintf(stderr,
                          "write returned %d "
                          "while %d is expected.\n",
                          cc, strlen(message));
              }

              close(newfd);
          }
      }
    }

    /* we should not reache here */
    for (i = 0; i < socknum; i++)
        close(socklist[i]);
    free(socklist);

    exit(0);
}
```

7.3 getifaddrs(3)

```
#include <stdlib.h>
#include <ifaddrs.h>
```

```
int main() {
    struct ifaddrs *ifa0, *ifa;
    int ret;

    ret = getifaddrs(&ifa0);
    if (ret) {
        perror("getifaddrs()");
        exit(1);
    }

    for (ifa = ifa0; ifa; ifa = ifa->ifa_next) {
        if (!ifa->ifa_addr)
            continue;
        switch(ifa->ifa_addr->sa_family) {
          case AF_INET:
            /* ifa->ifa_addr points sockaddr_in{} */
            /* ... */
            break;
          case AF_INET6:
            /* ifa->ifa_addr points sockaddr_in6{} */
            /* ... */
            break;
#if defined(AF_PACKET)
          case AF_PACKET:
            /* ifa->ifa_addr points sockaddr_ll{} */
            /* ... */
            break;
#endif
#if defined(AF_LINK)
          case AF_LINK:
            /* ifa->ifa_addr points sockaddr_dl{} */
            /* ... */
            break;
#endif
          default:
            /* not supported */
            ;
        }
    }
    freeifaddrs(ifa0);
    exit(0);
}
```