Network Programming

COSC 1176/1179

Lecture 9
IPv6 socket programming
Lecture Overview

During this lecture, we will learn

- Introduction to IPv6
- IPv6 socket programming
What is IPv6?
What is in IPv6?
Why IPv6: Address Shortage in IPv4

Why a new IP generation?

- IPv4 addresses become a scarce resource (32 bits, nominally about 4.3'' addresses)
  - The final block of IPv4 address has been allocated this year (2011).
  - The shortage of IPv4 addresses is a practical problem
  - An initial addressing crisis in IPv4 was alleviated by CIDR (→ more flexible allocation of address blocks) ...
  - ... giving IPv4 addressing a bit more lifetime

CIDR: Classless Interdomain Routing
IPv6 is not the only way for the evolution of the Internet

Competition by "fixes" to IPv4 (e.g. NAT)

IPv4
(incl. CIDR)

NAT

IPv6

Time

CIDR: Classless Interdomain Routing
NAT: Network Address Translator
IPv6 Design Goals

- simple header structure
- improved routing by address aggregation
- better multi-casting and any-casting
- possibility of using Quality of Service
- integrated security
- support and identification of traffic flows
Transition to IPv6 - 3 Basic Methods

- **Dual Stack**
  - Implemented in hosts to allow interworking
  - with IPv6 and IPv4 networks

- **Tunnels**
  - Used for IPv6 to IPv6 interworking via an IPv4 network

- **Translators**
  - Used for IPv6 to IPv4 interworking by translating the headers

"Bump-In-The-Stack" (BIS) - Allows v4 applic.s to run on IPv6

Diagram: IPv6 Network <-> IPv4 Tunnel <-> IPv6 Network

Diagram: IPv6 Network <-> Translator <-> IPv4 Network

Network Layer
- TCP/UDP
- IPv4
- IPv6
1. Dual Stack Terminals and Tunneling
   (Hosts with Dual IPStack = Support both IPv4 and IPv6)

2. Native IPv6 Devices (and Local Networks) plus Translation at v4/v6 Gateways
Types of IPv6 Addresses

- **Like IPv4...**
  - **Unicast**
    - An identifier for a single interface. A packet sent to a unicast address is delivered to the interface identified by that address.
  - **Multicast (a mandatory part of IPv6)**
    - An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.
  - **Anycast:**
    - An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to an anycast address is delivered to the "nearest" one, according to the routing protocols' measure of distance.

- Specified in the the v6 address architecture RFC.

Uni: Only One  MULTI: More than One  ANY: Only One but “closest” one
What is not in IPv6

- **Broadcast**
  - There is no broadcast in IPv6.
  - This functionality is taken over by multicast.

- One consequence of this is that the “all 0s” and “all 1s” addresses are legal in IPv6.

- There are other changes, we will see later.
IPv6 addresses
Representation of Addresses

- All IPv6 addresses are 128 bits
- Written as 8 sets of 4 hex digits (16 bits each) separated by colons
  - Leading zeros in group may be omitted
  - One contiguous set of all-zero groups may be replaced by ‘::’
  - Only one such group can be replaced

Example
- \(3ffe:3700:0200:00ff:0000:0000:0000:0001\)
- This can be written as
  - \(3ffe:3700:200:ff:0:0:0:1\) or
  - \(3ffe:3700:200:ff::1\)
- All three reduction methods are used here.
Examples of types of Unicast Addresses

- **Normal addresses**
  
  2001:0db8:85a3:0000:0000:8a2e:0370:7334

- **Unspecified address**
  - All zeros (::) - no such host
  - Used as source address during initialization
  - Also used in representing default

- **Compatible IPv4 addresses**
  - E.g. form ::ffff:a.b.c.d where a.b.c.d is the IPv4 address
  - Provides IPv6 stations with access to IPv4 stations

- **Loopback address**
  - Low-order one bit (::1)
  - Same as 127.0.0.1 in IPv4
Ipv6 Addresses

- **Multicast**
  - FF/8
  - FFfs \( f=\text{flags}, s=\text{scope} \)

- **Anycast**
  - subnet-prefix::0

- **Unicast**
  - is everything else, eg.
    - D202:23F8:17::12A1:0:A3

- **Site local**
  - like 10..., 172.16.., 192.168..
    - FE/8

- **Example address (used in documentation)**
  - 2001:DB8...
Embedding IPv6 addresses in URL's

To use a literal IPv6 address in a URL, the address should be enclosed in "[" and "]" characters. For example the following IPv6 addresses:

```
3ffe:2a00:100:7031::1
::fff:192.9.5.5
2010:836B:4179::836B:4179
```

would be represented as in the following example URLs: http://
```
http://[3ffe:2a00:100:7031::1]
http://[::fff:192.9.5.5]/ipng
http://[2010:836B:4179::836B:4179]
```

Source: RFC 2732
Coding for IPv6
A new address family name, AF_INET6, has been defined in \texttt{<sys/socket.h>}. 

The AF_INET6 is used in the first argument to the socket( ) function to indicate that an IPv6 socket is being created.

In AF_INET6 definition: new \texttt{sockaddr.in6} data structure. This is different from original \texttt{sockaddr.in} address data structure.

A new protocol family name, PF_INET6, is defined in \texttt{<sys/socket.h>}. The same value as the corresponding address family name:

\begin{verbatim}
#define PF_INET6       AF_INET6
\end{verbatim}
IPv6 Address Structure

- A new `in6_addr` structure holds a single IPv6 address and is defined as a result of including `<netinet/in.h>`:

  ```c
  struct in6_addr {
      uint8_t s6_addr[16]; /* IPv6 address */
  };
  ```

- This data structure contains an array of sixteen 8-bit elements, which make up one 128-bit IPv6 address.
- The IPv6 address is stored in network byte order.
Socket Address Struct (4.3BSD-Based Systems)

- The sockaddr_in structure is the protocol-specific address data structure for IPv4.
- The sockaddr_in6 structure holds IPv6 addresses
- Is defined in <netinet/in.h> header as:

```c
struct sockaddr_in6 {
    sa_family_t sin6_family; /* AF_INET6 */
    in_port_t sin6_port;    /* transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

sin6_family: The value of this field must be AF_INET6
sin6_port: contains the 16-bit UDP or TCP port number
sin6_flowinfo: intended to contain flow-related information
sin6_addr: holds one 128-bit IPv6 address
sin6_scope_id: identifies a set of interfaces
Socket Address Structure for 4.4BSD-Based Systems

It is defined as a result of including the `<netinet/in.h>` header.

```
struct sockaddr_in6 {
    uint8_t sin6_len; /* length of this struct */
    sa_family_t sin6_family; /* AF_INET6 */
    in_port_t sin6_port; /* transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

The only differences between this data structure and the 4.3BSD variant are the inclusion of the length field.
The Socket Functions

- Applications may create IPv6/TCP and IPv6/UDP sockets by simply using the constant AF_INET6 instead of AF_INET in the first argument of socket functions.

To create an IPv4/TCP socket, applications make the call:

```c
s = socket(AF_INET, SOCK_STREAM, 0);
```

To create an IPv6/TCP socket, applications make the call:

```c
s = socket(AF_INET6, SOCK_STREAM, 0);
```

To create an IPv4/UDP socket, applications make the call:

```c
s = socket(AF_INET, SOCK_DGRAM, 0);
```

To create an IPv6/UDP socket, applications make the call:

```c
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

- No changes to the syntax of the socket functions are needed to support IPv6.
Compatibility with IPv4 Applications

- Applications should be able to hold a combination of IPv4/TCP, IPv4/UDP, IPv6/TCP and IPv6/UDP sockets simultaneously within the same process.

- Applications using the original API should continue to operate as they did on systems supporting only IPv4. That is, they should continue to interoperate with IPv4 nodes.

- The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold a set prefix 0:0:0:0:0:FFFF. IPv4-mapped addresses are written as follows:

  ::FFFF:<IPv4-address>

- These addresses can be generated automatically by the getaddrinfo() function.

- Applications may use AF_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped IPv6 address.
IPv6 Wildcard Address

- With IPv4, one specifies the address as the symbolic constant INADDR_ANY (called the "wildcard" address) in the bind() call.

- Applications use `in6addr_any` similarly to the way they use INADDR_ANY in IPv4. For example, to bind a socket to port number 23, but let the system select the source address, an application could use the following code:

```c
struct sockaddr_in6 sin6;

sin6.sin6_family = AF_INET6;
sin6.sin6_flowinfo = 0;
sin6.sin6_port = htons(23);
sin6.sin6_addr = in6addr_any;

if (bind(s, (struct sockaddr*) &sin6, sizeof(sin6)) == -1)
    . . .
```
IPv6 server example: simpleserver6

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
const char NPMESSAGE[] = "Welcome to Network Programming.\n";

int main(int argc, char *argv[]) {
    int simpleSocket = 0;
    int simplePort = 0;
    int returnStatus = 0;
    struct sockaddr_in6 simpleServer;
    if (2 != argc) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(1);
    }
    simpleSocket = socket(PF_INET6, SOCK_STREAM, IPPROTO_TCP);
    if (simpleSocket == -1) {
        fprintf(stderr, "Could not create a socket!\n");
        exit(1);
    }
    else {
        fprintf(stderr, "Socket created!\n");
    }
```

© School of Computer Science and Information Technology
simplePort = atoi(argv[1]);

/* setup the address structure */
bzero(&simpleServer, sizeof(simpleServer));
simpleServer.sin6_family = PF_INET6;
inet_pton(PF_INET6, "::1", &(simpleServer.sin6_addr));
simpleServer.sin6_port = htons(simplePort);

/* bind to the address and port with our socket */
returnStatus = bind(simpleSocket, (struct sockaddr *)&simpleServer, sizeof(simpleServer));
if (returnStatus == 0) {
    fprintf(stderr, "Bind completed!\n");
} else {
    fprintf(stderr, "Could not bind to address!\n");
    close(simpleSocket);
    exit(1); }

/* lets listen on the socket for connections */
returnStatus = listen(simpleSocket, 5);
if (returnStatus == -1) {
    fprintf(stderr, "Cannot listen on socket!\n");
    close(simpleSocket);
    exit(1); }

char *inet_pton(int af, const char *src, void *dst);
The inet_pton() function converts an address from
presentation format to network format.

af   Specifies the address family. Currently, only
AF_INET and AF_INET6 are supported.

src  Points to the presentation-format address. The
format of the address is interpreted according to af.

dst  Points to the buffer into which the converted
address is stored.
while (1)
{
    struct sockaddr_in6 clientName = { 0 };
    int simpleChildSocket = 0;
    int clientNameLength = sizeof(clientName);
    /* wait here */

    simpleChildSocket = accept(simpleSocket,(struct sockaddr *)&clientName,
                                &clientNameLength);
    if (simpleChildSocket == -1) {
        fprintf(stderr, "Cannot accept connections!\n");
        close(simpleSocket);
        exit(1);
    }

    /* handle the new connection request */
    /* write out our message to the client */
    write(simpleChildSocket, NPMESSAGE, strlen(NPMESSAGE));
    close(simpleChildSocket);
}

return 0;
Example code

Code corner:

**Simpleserver6.c**


**Simpleclient6.c**


Platform: try these on machines having IPv6. Your linux/bsd machines.

Compilation (linux): gcc simpleserver6.cc -o simpleserver6
 gcc simpleclient6.cc -o simpleclient6
A number of new socket options are defined for IPv6. All of these new options are at the IPPROTO_IPV6 level. That is, the "level" parameter in the getsockopt() and setsockopt() calls is IPPROTO_IPV6 when using these options.

The constant name prefix IPV6_ is used in all of the new socket options.

The declaration for IPPROTO_IPV6, the new IPv6 socket options, and related constants defined in this section are obtained by including the header <netinet/in.h>.
A new setsockopt() option controls the hop limit used in outgoing unicast IPv6 packets. The name of this option is IPV6_UNICAST_HOPS, and it is used at the IPPROTO_IPV6 layer. Example:

```c
int hoplimit = 10;
if (setsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
               (char *) &hoplimit, sizeof(hoplimit)) == -1)
    perror("setsockopt IPV6_UNICAST_HOPS");
```

- When the IPV6_UNICAST_HOPS option is set with setsockopt(), the option value given is used as the hop limit for all subsequent unicast packets sent via that socket.
- If the option is not set, the system selects a default value.
Sending and Receiving Multicast Packets

- IPv6 applications may send multicast packets by simply specifying an IPv6 multicast address as the destination address, for example in the destination address argument of the sendto( ) function.

- Three socket options at the IPPROTO_IPV6 layer control some of the parameters for sending multicast packets.
  - **IPV6_MULTICAST_IF** - Set the interface to use for outgoing multicast packets.
  - **IPV6_MULTICAST_HOPS** - Set the hop limit to use for outgoing multicast packets.
  - **IPV6_MULTICAST_LOOP** - If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is looped back by the IP layer for local delivery.

- The reception of multicast packets:
  - **IPV6_JOIN_GROUP** - Join a multicast group on a specified local interface
  - **IPV6_LEAVE_GROUP** - Leave a multicast group on a specified interface.
IPv6 communications only - IPV6_V6ONLY

- AF_INET6 sockets may be used for both IPv4 and IPv6 communications. Some applications may want to restrict their use of an AF_INET6 socket to IPv6 communications only. For these applications the IPV6_V6ONLY socket option is defined.

```c
int on = 1;
if (setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY, (char *)&on, sizeof(on)) == -1)
    perror("setsockopt IPV6_V6ONLY");
else
    printf("IPV6_V6ONLY set\n");
```

- By default this option is turned off. When this option is turned on, the socket can be used to send and receive IPv6 packets only.
IPv6 Coding Gotchas
gethostbyname( )

`hostent` structure has single `h_addrtype` field, can return only IPv4 or IPv6 addresses!

Configured by system - typically returns IPv6 if possible, else IPv4.

gethostbyname() returns result in ONE static struct, not convenient for concurrent apps
int getaddrinfo(const char *hostname,
    const char *service,
    const struct addrinfo *hints,
    struct addrinfo **result)

struct addrinfo {
    int ai_flags;
    int ai_family; /* address family */
    int ai_socktype; /* stream, dgram… */
    int ai_protocol; /* IPv4 otr Ipv6 */
    size_t ai_addrlen;
    char *canonname; /* name of host */
    struct sockaddr *ai_addr; /* actual address struct */
    struct addtinfo *ai_next /* next one of these… */
}

Returns dynamically allocated (malloc) structs – re-entrant.
Coding Summary

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

const char NPMESSAGE[] = "Welcome to Network Programming.\n";

int main(int argc, char *argv[]) {
    int simpleSocket = 0;
    int simplePort = 0;
    int returnStatus = 0;
    struct sockaddr_in simpleServer;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(1);
    }
```
Coding Summary

```c
simpleSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (simpleSocket == -1) {
    fprintf(stderr, "Could not create a socket!\n");
    exit(1);
} else {
    fprintf(stderr, "Socket created!\n");
}
/* retrieve the port number for listening */
simplePort = atoi(argv[1]);

/* setup the address structure */
/* use INADDR_ANY to bind to all local addresses */
memset(&simpleServer,0, sizeof(simpleServer));
simpleServer.sin_family = AF_INET6;
simpleServer.sin_addr.s_addr = IN6ADDR_ANY;
simpleServer.sin_port = htons(simplePort);

/* bind to the address and port with our socket */
returnStatus = bind(simpleSocket,
    (struct sockaddr *)&simpleServer, sizeof(simpleServer));
```

- use PF_INET6
- use sin6_... fields
- may need getaddrinfo() or similar
if (returnStatus == 0) {
    fprintf(stderr, "Bind completed!\n");
} else {
    fprintf(stderr, "Could not bind to address!\n");
close(simpleSocket);
    exit(1);
}

/* lets listen on the socket for connections */
returnStatus = listen(simpleSocket, 5);

if (returnStatus == -1) {
    fprintf(stderr, "Cannot listen on socket!\n");
close(simpleSocket);
    exit(1);
}
while (1) {
    struct sockaddr_in clientName = {0};
    int simpleChildSocket = 0;
    int clientNameLength = sizeof(clientName);

    /* wait here */
    simpleChildSocket = accept(simpleSocket, (struct sockaddr *)&clientName, &clientNameLength);

    if (simpleChildSocket == -1) {
        fprintf(stderr, "Cannot accept connections!\n");
        close(simpleSocket);
        exit(1);
    }

    write(simpleChildSocket, NPMESSAGE, strlen(NPMESSAGE));
    close(simpleChildSocket);
}

close(simpleSocket);
return 0;
Summary

Big Question: Do we really need IPv6?
Summary

Big Question: Do we really need IPv6?

No? Then what else?
In this lecture, we have covered

- IPv6 basics
  - ipv4/ipv6 transition
  - Address format
- IPv6 sockets
- Socket functions, options etc.
- Porting IPv4 code to IPv6
References

First few slides on IPv6 are from:
Presentation / Background Information for TIPHON-17,
March 13-17, 2000, Sophia Antipolis
Source: Bernhard Petri, Siemens AG

RFC 3493 (socket related functions)