Transition toward IPv6

Main problems connected to the transition from IPv4 and IPv6
Techniques to easy their integration
Transition problems

- Transition
  - Required to allow interoperability with IPv4-only machines
  - Interoperability with IPv6-only machines is not a problem
  - IPv4 and IPv6 are different and incompatible protocols

- Migration of:
  - Network devices
  - Network
  - Host
  - Applications
Migration of network devices

- A problem only for devices L3(+)
  - Some problems may arise in L2 devices with some special functions (e.g. IGMP snooping, ...), mainly due to implementation problems

- Dual stack, “ships in the night” approach
  - Routing protocols and tables, access lists
Migration of the hosts

- **Dual Stack**
  - Very popular solution
  - IPv6 and IPv4 stack in each host
  - Complete support for both protocols
  - "dual layer" is currently the most used configuration

- **Limitations:**
  - It does not reduce the need for IPv4 addresses
  - Increased network complexity

- **Other dual-stack solutions:**
  - DSTM (Dual Stack Transition Mechanism)
  - ALG (Application Level Gateway)
Dual stack vs. Dual Layer

IPv4-mapped address (e.g. ::FFFF:a.b.c.d)

IPv4 address

IPv6 address

IPv4-IPv6 program

IPv4-IPv6 program

IPv4-mapped address (e.g. ::FFFF:a.b.c.d)
Migration of the network

Main solution: Tunnelling
- It allows to connect IPv6 subnets even when separated by IPv4 portions of the net

Tunneling mechanisms used:
- Configured Tunnelling
  - manual tunneling
- Automatic Tunnelling
  - “IPv4-compatible” addresses
  - 6over4 (RFC 2529)
  - 6to4
  - Tunnel Broker (RFC 3053)
  - ISATAP
  - Teredo
Tunnelling

IPv6 packet -> IPv4 tunnel -> IPv6 packet

- IPv6 in IPv4 (protocol type = 41)
- GRE
- ...

IPv6 packet -> IPv4 hdr -> IPv6 packet
Migration of the network

- Problem to be solved:
  
  To ship IPv6 packets through IPv4 "clouds"

- Not a problem
  
  To ship IPv4 packets through IPv6 "clouds"

- Assumption
  
  All the "active" nodes (those participating to the transition) are dual-stack ones
“IPv4-compatible” addresses

- Used to transport IPv6 packets over an IPv4 infrastructure
- Often designated as “automatic tunnelling” (error)
  - A pseudo-interface called “Automatic Tunneling Pseudo-Interface” is used
  - All the packets sent though it are “tunnelled” on the basis of the IPv6 destination address
    - The IPv6 address should be “IPv4-compatible”
    - Example: ::130.192.226.140
  - It is necessary to define a route to forward packets addressed to ::/96
  - In principle, it is possible to advertise more specific routes than ::/96, but this will lead to an explosion of routes in the network
“IPv4-compatible” addresses: captures

Traffic with direct route through the IPv4 net (scenario 1)

C:\> netsh interface ipv6 add route ::/96 "Automatic Tunneling Pseudo-Interface"
C:\> ipv6 rt
  ::/96  -> 2 pref 1if+0=1 life infinite, publish, no aging (manual)
  ::/0  -> 3/2002:c058:6301::c058:6301 pref 1if+2147483647=2147483648 life 2h/30m, publish, no aging (manual)
  ::/0  -> 3/2002:836b:213c:1:e0:8f08:f020:8 pref 1if+1180=1181 life 2h/30m, publish, no aging (manual)
  2002::/16  -> 3 pref 1if+1000=1001 life 2h/30m, publish, no aging (manual)
C:\> ping ::1 30.192.225.135

Traffic with route through an IPv4/IPv6 gateway (scenario 2)

C:\> netsh interface ipv6 add route ::/96 "Automatic Tunneling Pseudo-Interface" ::163.162.170.177
C:\> ipv6 rt
  ::/96  -> 2/::163.162.170.177 pref 1if+0=1 life infinite (manual)
  ::/0  -> 3/2002:c058:6301::c058:6301 pref 1if+2147483647=2147483648 life 2h/30m, publish, no aging (manual)
  ::/0  -> 3/2002:836b:213c:1:e0:8f08:f020:8 pref 1if+1180=1181 life 2h/30m, publish, no aging (manual)
  2002::/16  -> 3 pref 1if+1000=1001 life 2h/30m, publish, no aging (manual)
C:\> ping ::1 30.192.225.135
6over4

- Used to transport IPv6 packets over an IPv4 infrastructure
- "Virtual Ethernet"
  - Based on multicast
  - This requires the configuration of a "6over4" interface
- Addresses
  - NetID:InterfaceID/64
    - InterfaceID derives from the IPv4 address
- Currently not used
  - IPv4 multicast infrastructure often not available
6over4 and multicast IPv4

- Addresses 239.192.[next to last byte IPv6].[last byte IPv6]
  - Ex.: FF02::1 → 239.192.0.1
  - Ex.: FF02::1:FF28:9C5A (mcast address of the requested node) → 239.192.156.90
  - The IPv4 multicast address can be mapped on an Ethernet multicast
- Limited only to the corporate net (”site”), not the whole Internet
- The scope of the IPv4 multicast should be properly configured
  - IPv6 addresses with different prefixes may use the same multicast-v4
  - Link-local addresses (FE80::a.b.c.d) could have a global scope in IPv4
- IGMP used to signal the multicast group
- Neighbor discovery: extended to support "link-layers" IPv4
- Router advertisement/solicitation: used to configure global addresses
6to4

Your IPv4 address: 62.157.9.98

Decimal: 62 157 9 98
Hex: 3e 9d 09 62

Your IPv6 address: 2002:3e9d:0962:0001::1

6to4 prefix 80 bit address space
Structure of 6to4 prefix

<table>
<thead>
<tr>
<th>FP 001</th>
<th>TLA 0x002</th>
<th>V4ADDR</th>
<th>SLA ID</th>
<th>Interface ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>13</td>
<td>32</td>
<td>16</td>
<td>64</td>
</tr>
</tbody>
</table>

2002::/16

2002::/48

Internal nets

nodes
6to4: Basic scenario

Wide area IPv4 network

192.1.2.3

6to4 router
2002:c001:0203::/48
A

9.254.2.252

6to4 router
2002:09fe:02fc::/48
B
6to4: mixed scenario

Wide area IPv4 network

Native IPv6 wide area network

192.1.2.3
2002:c001:0203::/48
A

2001:0600::/48
IPv6 Site B
2002:09fe:fdfe::/48
2001:0600::/48

9.254.253.252
6to4

- Usage
  - Normally in host → 6to4 relay mode
  - IPv4 anycast addresses have been defined
    - 131.107.33.60
    - 192.88.99.1
    - Reverse path required
  - Not NAT compatible (unless a specific support is implemented)

C:\> ipv6 rt
  ::/0 -> 3/2002:c058:6301::c058:6301 pref 1if+2147483647=2147483648 life 2h/30m, publish, no aging (manual)
  ::/0 -> 3/2002:836b:213c:1:e0:8f08:f020:8 pref 1if+1171=1172 life 2h/30m, publish, no aging (manual)
  2002::/16 -> 3 pref 1if+1000=1001 life 2h/30m, publish, no aging (manual)
C:\> ping 2001:0610:0148:DEAD:0210:18FF:FE02:0E38 (www.6net.org)
ISATAP (intra-site automatic tunnel addressing protocol)

- In 6to4, native IPv6 intra-site connectivity is required
  - Router advertisement across tunnels is not allowed
- **ISATAP removes this limitation**
  - The router can be identified using a DNS query
  - Record A, name _isatap.domain.com
  - It is possible to interact with the router, using advertisement/solicitation messages, if its IPv4 address is known

- **Addresses**
  - ::0:5EFE:a.b.c.d
    - 00-00-5E: OUI IANA assigned
    - FE: type that indicates “included” networks
  - It allows a prefix /64 (as for other tunnel types)
Teredo

- IPv6 packets are encapsulated in IP/UDP to overcome NAT problems
- An external server to the private network is required
Tunnel broker
Tunnel broker: example
Why writing applications?

- Most of the work, so far, focused on network-related issues
  - Definition of the IPv6 protocol and related stuff
  - Network infrastructure
  - Operating System support
- A few people are currently working on the application side
- We must avoid the problem of ATM
  - Excellent technology, but no applications ready to work on it
- The socket interface
  - We’re network people, we don’t like .NET, Java, and friends
  - C/C++ only if you want flexibility and speed
The old programming code (BSD-style API)

#define PORT 2000             /* This definition is a number */

void server ()
{
    int Sock; /* Descriptor for the network socket */
    struct sockaddr_in SockAddr; /* Address of the server socket descr */

    if ( ( Sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        error("Server: cannot open socket.");
        return;
    }

    memset(& SockAddr, 0, sizeof(SockAddr));
    SockAddr.sin_family = AF_INET;
    SockAddr.sin_addr.s_addr= htonl(INADDR_ANY); /* all local addresses */
    SockAddr.sin_port = htons(PORT); /* Convert to network byte order */

    if (bind(Sock, (struct sockaddr *)&SockAddr, sizeof(SockAddr)) < 0) {
        error("Server: bind failure");
        return;
    }

    /* ... */
The new programming style (RFC 3493 API)

```c
#define PORT "2000"                     /* This definition is a string */

void server ()
{
    int Sock; /* Descriptor for the network socket */
    struct addrinfo Hints, *AddrInfo; /* Helper structures */
    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family = AF_UNSPEC; /* or AF_INET / AF_INET6 */
    Hints.ai_socktype = SOCK_STREAM;
    Hints.ai_flags = AI_PASSIVE; /* ready to a bind() socket */

    if (getaddrinfo(NULL /* all local addr */, PORT, Hints, AddrInfo) != 0) {
        error("Server: cannot resolve Address / Port ");
        return;
    }

    // Open a socket with the correct address family for this address.
    if ((Sock=socket(AddrInfo->ai_family, AddrInfo->ai_socktype, AddrInfo->ai_protocol))<0){
        error("Server: cannot open socket.");
        return;
    }

    if (bind(Sock, AddrInfo->ai_addr, AddrInfo->ai_addrlen) < 0) {
        error("Server: bind failure");
        return;
    }

    /* ... */
```

Family-independent code

Fills some internal structures with family-independent data using literal / numeric host and port

Data returned by getaddrinfo() is used in a family-independent way
Modification to the system calls (1)

<table>
<thead>
<tr>
<th>Parameter changes</th>
<th>socket()</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bind(), connect(), accept()</td>
<td>socket(AF_INET, ...</td>
</tr>
<tr>
<td></td>
<td>sendto(), recvfrom()</td>
<td>➔ socket(AddrInfo-&gt;ai_family,...</td>
</tr>
<tr>
<td></td>
<td>setsockopt(), getsockopt()</td>
<td></td>
</tr>
<tr>
<td>Unchanged</td>
<td>recv(), send()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>listen(), select()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shutdown()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>htonl(), htons(), ntohl(), ntohs()</td>
<td></td>
</tr>
<tr>
<td>Replaced functions</td>
<td>gethostbyaddr(), gethostbyname()</td>
<td>➔ getaddrinfo(), freeaddrinfo()</td>
</tr>
<tr>
<td></td>
<td>gethostbyaddr(), getservbyport()</td>
<td>➔ getnameinfo()</td>
</tr>
<tr>
<td>Obsolete functions</td>
<td>inet_addr()</td>
<td>➔ getaddrinfo()</td>
</tr>
<tr>
<td></td>
<td>inet_ntoa()</td>
<td>➔ getnameinfo()</td>
</tr>
</tbody>
</table>

- Other helper functions are usually unchanged
  - gethostname(), getsockname(), getpeernamename()
  - getprotobynumber(), getprotobynumber(), getservbyname(), getservbyport()
Modification to the system calls (2)

- **Client side: fallback mechanism**
  - If a server does not respond to an IPv6 connection and it does have an IPv4 address, let’s try to connect in IPv4
    - Problems due to the timeout
  - This leads to a duplication of code and some additional control logic, which must be done by hand

- **Server side: dual-server capabilities**
  - If a server has both IPv4 and IPv6 addresses, it should accept both types of connections
  - Most of the OS requires this to be coded by hand
    - Notably exception: FreeBSD
  - Leads to duplicated code and some additional efforts in synchronization (we may have two waiting threads)
Capabilities detection at compile time

- Some application are distributed as a source code instead as a binaries
- Are IPv6 system calls and structures supported by the OS in which the compilation has to be done?
  - Autoconf – automake try to automate the building process
  - This is not always possible with automatic tools
  - The programmer may have to add several alternatives for the code in order to be able to compile on older platforms
    - The building process may activate portions of code by means of some #define
Adding IPv6 support to old IPv4 applications (1)

- We need to locate the code that needs to be changed
  - “string search” to locate the system calls related to the socket interface
    - This is simple
  - “visual inspection” for other parts of the code
    - This is not

- System calls related to the socket interface
  - Convert part of the code to become protocol independent
    - The most part of socket functions
  - Add special code for IPv6
    - Some functions (getsockopt(), setsockopt()) which behave differently in IPv4 and IPv6
Adding IPv6 support to old IPv4 applications (2)

- **Other code**
  - Custom control used as input for an IPv4 address
  - Parsing or URLs
    - Several allowed strings
      - http://130.192.16.81
      - http://truciolo.polito.it
    - The ":" symbol is a "port delimiter" in IPv4, while it is the "address separator" in IPv6
      - http://truciolo.polito.it:80
  - Application-layer protocol
    - Is this protocol defining a field that carries IPv4 addresses (e.g. peer-to-peer applications)?
  - Difficult to locate
Adding IPv6 support to old IPv4 applications: experimental results

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of code of the socket interface that need to be changed</th>
<th>Other lines of code that need to be changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeAMP</td>
<td>88 (59%)</td>
<td>40 URL parsing</td>
</tr>
<tr>
<td>Free MP3 player</td>
<td>The code is not well organized</td>
<td></td>
</tr>
<tr>
<td>GNUcleus</td>
<td>30 (83%)</td>
<td>Undefined</td>
</tr>
<tr>
<td>Free peer-to-peer application</td>
<td>The code related to the network is well defined into a C++ class</td>
<td>Protocol change</td>
</tr>
</tbody>
</table>

Far more than 50% of the code related to the socket interface must be changed.

And, for the rest, who knows?
Creating IPv6-only application

- The effort is slightly less than adding IPv6 support
  - We can adopt some system calls which are “deprecated” but allow converting the code easier
  - We are not forced to add some code to support both the IPv4 and IPv6 case

- It does not make sense
  - Who does have an IPv6-only network?
Writing new applications with IPv4 and IPv6 support

- For the most part, this is much easier than writing IPv4-only applications with the older BSD programming style
  - `getaddrinfo()` and `getnameinfo()` are very handy
  - Code is smaller and easier to understand than the one written according to the old socket interface

- Some code may be duplicated
  - `getsockopt()`, `setsockopt()`
  - URL parsing
Platform compatibility

- A network application may run on any network device
  - Important to write portable network code in order to be able to run everything on other platforms

- Most of the socket implementation is portable among platforms
  - Some minor issues still remain
**Platform differences (1)**

<table>
<thead>
<tr>
<th>Variables</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket()</td>
<td>returns an unsigned integer</td>
<td>socket() returns an integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error functions</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default choice</td>
<td>is WSAGetLastError() or GetLastError(); the gai_strerror() exists, but it is not thread safe and the previous functions should be used instead</td>
<td>gai_strerror() or the errno variable depending on the function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error messages</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error codes</td>
<td>are different from UNIX (although most of them have the same name)</td>
<td>Standard error codes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface management functions</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing; there are some proprietary functions: GetNumberOfInterfaces(), GetInterfaceInfo()</td>
<td>Standard if_nameindex() if_nametoindex() if_indextoname()</td>
<td></td>
</tr>
<tr>
<td>Additionally, there are the following: GetAdaptersInfo(), GetAdaptersAddresses(), GetAdapterIndex()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address management</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>getaddrinfo() with some specific flags</td>
<td>inet_ntop(), inet_pton()</td>
<td></td>
</tr>
</tbody>
</table>
## Platform differences (2)

<table>
<thead>
<tr>
<th>Socket and files</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets are not ‘standard’ files</td>
<td></td>
<td>The the same functions that are used to read and write files can be used with sockets</td>
</tr>
<tr>
<td>read() and write() do not work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadFile() and WriteFile() are working</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initialization / cleanup</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required: WSAStartup() and WSACleanup()</td>
<td></td>
<td>Not needed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Closing a socket</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>closesocket()</td>
<td></td>
<td>close()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Win32</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include “winsock.h” and “ws2_32.lib”</td>
<td></td>
<td>Several files in addition to “sys/socket.h”; socketlib</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server Socket</th>
<th>Win32, Linux</th>
<th>FreeBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>A server socket binds to a single address family; a server that has to accept IPv4/6 connections must open two server sockets</td>
<td></td>
<td>A single server socket can be created for both IPv4 and IPv6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>Linux</th>
<th>Win32, FreeBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generates a SIGPIPE signal whenever an error occurs when sending data on stream-oriented sockets</td>
<td></td>
<td>A “write” error on connected sockets generates only an error message</td>
</tr>
</tbody>
</table>
Platform differences (3)

- Several differences
  - Often “minor issues”, although “several minor issues” may become “a major problem”

- A C wrapper or C++ class that implements a socket interface may be an excellent solution
  - Not too much code to write
  - 1000 lines of code are enough
    - Just to hide some details

- Win32: things are getting slightly worse when migrating from old sockets to new ones
Applications that use a network abstraction

- Several applications do not use the socket interface directly
  - .NET or Java applications
  - MFC, QT or wxWindows-based applications

- The environment (either the run-time or the library) will manage the network connections
  - Usually, applications do not care about network protocols
  - These application should be able to run in IPv6 seamlessly
    - Obviously, the environment must support IPv6
  - Some problems (application-level protocols, GUI custom controls) may remain
What about if the application already exists and the source code is not available?

- Most of the applications do not make the source code available
- If we want to use IPv6 networks, we must have IPv6 applications
  - Several projects around the world are still missing the point, creating IPv6 networks with no traffic on them
- Some applications are ready
  - Some examples on Win32:
    - Internet Explorer, Mozilla Firebird and Thunderbird
    - Internet Information Server (Windows Server 2003), Apache
    - ...
The most common deployment scenario

IPv4/IPv6 network
IPv4/IPv6
Server application
IPv4/IPv6
DNS
IPv4
Client application
IPv4
Client application

We must upgrade servers first
Avoids the timeout due to the IPv4 fallback

Domain mydomain.com

myserver AAAA 2001:760:400::1
myserver A 130.192.19.81

IPv4/IPv6 address
IPv6 address
IPv4 address
The TCP/UDP port forwarder (bouncer) (1)

**IPv4 application**
Wait on:
- network protocol IPv4
- transport protocol TCP (or UDP)
- port X

**Port forwarder Client**
Connect to:
- host 127.0.0.1
- transport protocol TCP (or UDP)
- port X

**Port forwarder Server**
Wait on:
- network protocol IPv6
- transport protocol TCP (or UDP)
- port X

TCP/UDP Port forwarder

**Server Host**

IPv6 address

Incoming IPv6 connection on port X

IPv4 address

Incoming IPv4 connection on port X
The TCP/UDP port forwarder (bouncer) (2)

- Very simple
- Does not requires any modification to the legacy IPv4 application
- Compatible with almost all TCP/UDP applications
  - It does not understand the application data (it is not a proxy)
- Hides the connecting network address
  - Needed for statistics, logging, filtering
    - E.g. SMTP server which accepts only connections from local hosts
- Does not work with applications that do not follow the pure client-server model
  - E.g. FTP in “active mode” because the server opens the data connection toward the client
- Fragmented packets
  - IPv6 headers are bigger than IPv4 ones
  - Packets generated by the IPv4 server may need to be fragmented by the bouncer
Bouncer and clients

- In general, the deployment of a ‘bouncer’ is not limited to a server
- However, it requires an explicit (and fixed) configuration
  - of the address of the bouncer
  - of the server we want to connect to
- Using a bouncer to translate client connections
  - A bouncer can be used to translate IPv4 requests coming from a legacy client into IPv6 streams
  - This approach works only if the client tries to connect always to the same server
    - E.g. DNS, POP, SMTP, IMAP, web browsing through a proxy
  - It does not work if the client can contact several servers
    - E.g. Standard web browsing, FTP, ...
- In any case, the bouncer is one of the best methods due to its simplicity
Other methods (1)

- Most important approaches
  - Bump In the Stack (BIS)
    - Packet-level translator embedded in the stack
    - IPv6 packets are translated into IPv4 ones and delivered to the application
    - It allows also initiating the connection
      - In case the contacted machine is IPv6, the DNS resolver returns a fake IPv4 address to the application; IPv4 packets directed to this host are intercepted and transformed in IPv6
  - Bump In the API (BIA)
    - More efficient approach: the socket library is modified in order to recognize if the host we want to contact is IPv4 or IPv6
    - The library generates the packets according to the right protocol without any need of translation
    - In case of a server, the library checks if the server is waiting on IPv4 or IPv6 socket and delivers the data accordingly
Other methods (2)

- Other approaches
  - TCP/UDP Relay
    - Similar to a bouncer, but the address of the relay is returned by the DNS on an application-dependent way
  - Network Address Translator – Protocol Translator
  - SOCKS
  - Application-level gateways

- Problems
  - Intrusiveness
    - content inspection
    - explicit client capabilities (e.g. proxy)
  - Not targeted to seamlessly migrating server applications
Conclusions

- Effort required to add IPv6 support to an old IPv4-only application is not negligible
  - Far more than 50% of the lines of code need to be changed
  - Hidden costs (input forms, application-dependent protocols, etc.)

- Creation of new IPv4 and IPv6 applications from scratch
  - The socket interface is simpler than before
  - Some common issues:
    - Fallback: for clients
    - Dual-socket bind: for servers

- If we want to add IPv6 support to a closed source application
  - The bouncer mechanism may be the best choice