Socket Programming (2/2)
Outline

1. Introduction to Network Programming
2. Network Architecture – Client/Server Model
3. TCP Socket Programming
4. UDP Socket Programming
5. IPv4/IPv6 Programming Migration
UDP Socket Programming
Review of Transport Layer Services

- Connection-Oriented Transport Services
  - Transmission Control Protocol (TCP)
  - Stream Control Transmission Protocol (SCTP)

- Connectionless Transport Services
  - User Datagram Protocol (UDP)
Connectionless Transport Services

- No (Logical) Connection Establishment – Packet Switching
- Each Datagram is individually transmitted – Out-of-order
- Unreliable Data Transfer – Datagram Loss
- No Connection State – No Flow control and Congestion Control
## Comparisons of Transport Services

<table>
<thead>
<tr>
<th>Service Types</th>
<th>Connection-Oriented</th>
<th>Connectionless</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Protocols</td>
<td>TCP</td>
<td>UDP</td>
</tr>
<tr>
<td>Connection Establishment</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Reliable Data Transfer</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>In-order Data Delivery</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Flow Control/ Congestion Control</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Data Transmission Controlled by</td>
<td>TCP</td>
<td>Application</td>
</tr>
</tbody>
</table>
User Datagram Protocol (UDP)

- UDP provides an unreliable, connectionless service to the invoking application
- No handshaking between sending and receiving transport layer entities before sending data segments
- UDP is defined in RFC 768 (http://www.ietf.org/rfc/rfc0768.txt)
- UDP-based Internet Services/Applications: DNS, SNMP, RIP, Internet Telephony
Comparison of UDP and TCP Segments

Flow Control and Congestion Control

No Support

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Destination Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Application Data (Message)

UDP Segment Structure

TCP Support

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Destination Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td>Acknowledgement Number</td>
</tr>
<tr>
<td>Header Length</td>
<td>Unused</td>
</tr>
<tr>
<td>Receive Window</td>
<td></td>
</tr>
<tr>
<td>Internet Checksum</td>
<td>Urgent Data Pointer</td>
</tr>
<tr>
<td>Options</td>
<td>Data</td>
</tr>
</tbody>
</table>

TCP Segment Structure
Preview of UDP Sockets’ Functions

- UDP Sockets – Interfaces between application layer and UDP transport layer
- Using socket() to a UDP socket without connection establishment
- Using sendto() and recvfrom() to send and/or receive user datagrams from peer hosts

```c
sockfd = socket(PF_INET, SOCK_DGRAM, 0);
recvfrom(sockfd, echobuffer, BUFFER_SIZE, 0, (struct sockaddr *)&peeraddr, &length);
```

source ip, source port
# Categories of UDP Sockets’ Functions

<table>
<thead>
<tr>
<th>Host Type</th>
<th>UDP Client</th>
<th>UDP Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket Creation</td>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td>Port Binding</td>
<td>bind()</td>
<td></td>
</tr>
<tr>
<td>Sending &amp; Receiving</td>
<td>sendto()</td>
<td>recvfrom()</td>
</tr>
<tr>
<td>Datagrams</td>
<td>recvfrom()</td>
<td>sendto()</td>
</tr>
<tr>
<td>Socket Closing</td>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>
Operations of UDP Client/Server

UDP Client

socket()
sendto()
recvfrom()
close()

UDP Server

socket()
bind()
recvfrom()

Wait another UDP Client

sendto()

CCU COMM ANT LAB
Elementary UDP Socket Functions
Review of UDP Unreliable Data Transfer

- No (Logical) Connection Establishment – Packet Switching
- Each Datagram is individually transmitted – Out-of-order Sequence
- Unreliable Data Transfer – Datagram Loss
- No Connection State – No Flow control and Congestion Control
IP and Port Addressing for UDP

- UDP does not create a (virtual) logical channel before data transfer
  - No peer service port binding information
- Each datagram should be explicitly specified its addressing information
  - Source IP, Source Port
  - Destination IP, Destination Port
UDP Socket Creation – socket()

- socket -- create an endpoint for communication
- int socket(int domain, int type, int protocol)
  - domain: Specify a communications domain - PF_INET
  - type: Specify the semantics of communication - SOCK_DGRAM
  - protocol: Specify a particular protocol to be used with the socket – 0
sockfd = socket(PF_INET, SOCK_DGM, 0) ;
sockfd = socket(PF_INET, SOCK_DGM, 0);  

sockfd – socket descriptor (handler)  
integer  
-1 means that socket is creating failure
sockfd = socket(PF_INET, SOCK_DGM, 0);

Parameter : domain
Domain value: PF_INET – Internet Protocol version 4
PF_INET6 – Internet Protocol version 6
PF_LINK – Link Layer Interface
sockfd = socket(PF_INET, SOCK_DGM, 0);

Parameter: type
Type value:
- SOCK_STREAM - TCP
- SOCK_DGRAM - UDP
- SOCK_RAW - Raw Socket
UDP Socket Service Binding – bind()

- Assign a local protocol address to a socket
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)`
  - `sockfd`: socket descriptor created by `socket()`
  - `my_addr`: Local address information including IP address and port value
  - `addrlen`: byte length of local address information – `my_addr`
- On success, zero is returned. On error, -1 is returned
- It is usually used in server port binding
RetValue = bind(sockfd, (struct sockaddr *)&srvaddr, length);
RetValue = bind(sockfd, (struct sockaddr *)&srvaddr, length);

If RetValue is equal to 0, binding local address, local port to the socket is Successfully
If RetValue is less than 0, binding process is failed.
RetVal = bind(sockfd,(struct sockaddr *)&srvaddr,length);
RetValue = bind(sockfd,(struct sockaddr *)&srvaddr,length);
RetValue = bind(sockfd, (struct sockaddr *)&srvaddr, length);
sockfd = socket(PF_INET, SOCK_DRGM, 0)

RetVal = bind(sockfd, (struct sockaddr *)&srvaddr, length);
RetVal = bind(sockfd, (struct sockaddr *)&srvaddr, length);
	srvaddr.sin_family → Protocol Family (PF_INET)
	srvaddr.sin_addr.s_addr → Local IP (0,0.0.0)
	srvaddr.sin_port → Local Port (1260)
RetValue = bind(sockfd,(struct sockaddr *)&srvaddr,length);

Length of srvaddr structure
Comparison of TCP and UDP Socket’s IP and Port Binding

- TCP and UDP sockets use same function to create socket interface and IP and port information binding at server side.
- TCP uses connect() to create a communication channel.
  - Communication channel is identified by the binding of local IP, local Port, remote IP, and remote Port.
- UDP has no connection process to bind their IP and Port between local and remote hosts.

![Diagram showing TCP and UDP communication process with IP and Port binding.](image-url)
UDP Socket Data Sending – sendto()

- Send a message on a socket
- `ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);

  - `s:` File descriptor of the sending socket
  - `buf:` Content of sending message
  - `len:` Byte length of sending message
  - `flags:` Control information of flags
  - `to:` Addressing information of destination host
  - `tolen:` Byte length of to structure
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);

On success, these calls return the number of characters sent. On error, -1 is returned.
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);

Destination Addressing Information.

PF_INET
“140.123.101.30”
1690
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
sockfd : created by
socket(PF_INET, SOCK_DGM, 0) ;

RetVal = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);

buffer: block data
Example: strcpy(buffer, “Hello”);
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
control information of `sendto()` function
Flag value normally is 0

\[
\text{RetValue} = \text{sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);
}\]
RetValue = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &server, length);

server.sin_family -> Protocol Family (PF_INET)
server.sin_addr.s_addr -> Destination IP (140.123.101.30)
sserver.sin_port -> Destination Port (1690)
RetValue = sendto(sockfd, buffer, strlen(buffer), 0,
(struct sockaddr*) &server, length);
UDP Socket Data Receiving – recvfrom()

- Receive a message from a socket
- `ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);`
  - `s:` File descriptor of the sending socket
  - `buf:` Content of receiving message
  - `len:` length of allocated buffer to receive message
  - `flags:` Control information of flags
  - `from:` Addressing information of sending host
  - `fromlen:` Byte length of from structure
RetValue = recvfrom(sockfd,buffer, strlen(buffer),0,
  (struct sockaddr*) &peer,length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);

On success, these calls return the number of characters received. On error, -1 is returned.
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
Addressing Information of Sending Host

PF_INET
"140.123.101.24"
1340

RetVal = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
Addressing Information of Source Host

PF_INET
"140.123.101.24"
1340

RetValue = recvfrom(sockfd,buffer, strlen(buffer),0,
(struct sockaddr*) &peer,length);
sockfd : created by
socket(PF_INET, SOCK_DGM, 0) ;

RetVal = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);

buffer: allocated space to receive incoming message
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);

(strlen(buffer)) : size of allocated space
control information of recvfrom() function
Flag value normally is 0

RetVal = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);

peer.sin_family → Protocol Family (PF_INET)
peer.sin_addr.s_addr → Source IP (140.123.101.24)
peer.sin_port → Source Port (1340)
RetValue = recvfrom(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*) &peer, length);

length = size of peer structure
Comparison of TCP and UDP Socket Data Sending and Receiving
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
<tr>
<td>Receiving Message</td>
<td>recv()</td>
<td>recvfrom()</td>
</tr>
<tr>
<td>Protocol</td>
<td>TCP</td>
<td>UDP</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
send(int s, const void *buf, size_t len, int flags)
```

```c
sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)
```

**UDP**
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td><code>send()</code></td>
<td><code>sendto()</code></td>
</tr>
</tbody>
</table>

### TCP

```c
send(int s, const void *buf, size_t len, int flags)
```

### UDP

```c
sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)
```

Same socket and sending buffer parameters
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
send(int s, const void *buf, size_t len, int flags)
```

```c
sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)
```

**UDP**
### Protocol

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td><strong>Sending Message</strong></td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

send(int s, const void *buf, size_t len, int flags)

const struct sockaddr *to, socklen_t tolen

**UDP**

UDP should specify addressing information to identify the destination at every sendto() process.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```
send(int s, const void *buf, size_t len, int flags)
```

```
sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)
```

**UDP**
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td><strong>Sending Message</strong></td>
<td>send()</td>
<td>sendto()</td>
</tr>
<tr>
<td><strong>Receiving Message</strong></td>
<td>recv()</td>
<td>recvfrom()</td>
</tr>
</tbody>
</table>

**TCP**

```
recv(int s, void *buf, size_t len, int flags)
recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

**UDP**
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
recv(int s, void *buf, size_t len, int flags)
```

```c
int s, void *buf, size_t len, int flags
```

```c
recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

**UDP**

Same socket and receiving buffer parameters
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
recv(int s, void *buf, size_t len, int flags)
```

```c
recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

**UDP**
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
recv(int s, void *buf, size_t len, int flags)
```

```c
struct sockaddr *from, socklen_t *fromlen
```

```c
recvfrom(int s, void *buf, size_t len, int flags,
         struct sockaddr *from, socklen_t *fromlen);
```

**UDP**

UDP can get specific source information of receiving message at every recvfrom() process.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td>Sending Message</td>
<td>send()</td>
<td>sendto()</td>
</tr>
</tbody>
</table>

**TCP**

```c
recv(int s, void *buf, size_t len, int flags)
```

```c
recvfrom(int s, void *buf, size_t len, int flags,
         struct sockaddr *from, socklen_t *fromlen);
```

**UDP**
Using the following operations, bind(), listen(), connect(), and accept(), to create a logical transport channel with specific addressing information between two hosts.
Using the following operations, bind(), listen(), connect(), and accept(), to create a logical transport channel with specific addressing information between two hosts.
Using the following operations, bind(), listen(), accept() and connect(), to create a logical transport channel with specific addressing information between two hosts.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

**UDP**
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

Client

```
Application

socket

Transport
```

Server

```
Application

Local IP: 140.123.101.100
Local Port: 1460

Transport
```

**UDP**
**Protocol**

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

---

**UDP**
### Protocol Table

<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

### TCP

Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

**Client**

Application

`socket`

**Transport**

**Server**

Local IP: 140.123.101.100
Local Port: 1460

**Application**

`socket`

**Transport**

### UDP

**Application**

`socket`

**Transport**
TCP

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

UDP
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td><strong>Connection-Oriented</strong></td>
<td><strong>Connectionless</strong></td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

**UDP**

![TCP diagram](image)

Client
- Local IP: 140.123.101.243
- Local Port: 3865

Server
- Local IP: 140.123.101.100
- Local Port: 1460
**TCP**

Using the following operations, bind(), listen(), accept(), and **connect()**, to create a logical transport channel with specific addressing information between two hosts.

**UDP**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

**Service**

**TCP**

Client
- Application
- Transport
- Local IP: 140.123.101.243
- Local Port: 3865

Server
- Application
- Transport
- Local IP: 140.123.101.100
- Local Port: 1460
<table>
<thead>
<tr>
<th>Protocol</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

TCP
Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

UDP
Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.
### Protocol Comparison

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
<tr>
<td><strong>Sending Message</strong></td>
<td><code>send()</code></td>
<td><code>sendto()</code></td>
</tr>
<tr>
<td><strong>Receiving Message</strong></td>
<td><code>recv()</code></td>
<td><code>recvfrom()</code></td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, `bind()`, `listen()`, `accept()`, and `connect()`, to create a logical transport channel with specific addressing information between two hosts.

**UDP**

---

**Channel Addressing Information**

- **Client**: IP: 140.123.101.243, Port: 3865
- **Server**: IP: 140.123.101.100, Port: 1460
### Protocol

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Connection-Oriented</td>
<td>Connectionless</td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

---

**UDP**

Only use bind() to address service port, no other procedures to establish a logical transport channel with specific addressing information. The destination addressing information must be injected into the sendto() function.
### Protocol

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service</strong></td>
<td><strong>Connection-Oriented</strong></td>
<td><strong>Connectionless</strong></td>
</tr>
</tbody>
</table>

**TCP**

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

**UDP**

Only use bind() to address service port, no other procedures to establish a logical transport channel with specific addressing information. The destination addressing information must be injected into the sendto() function.
**TCP**

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

Only use bind() to address service port, no other procedures to establish a logical transport channel with specific addressing information. The destination addressing information must be injected into the sendto() function.
Protocol | TCP | UDP
---|---|---
Service | Connection-Oriented | Connectionless

**TCP**

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

---

**UDP**

Only use **bind()** to address service port, no other procedures to establish a logical transport channel with specific addressing information. The destination addressing information must be injected into the sendto() function.
TCP

Using the following operations, bind(), listen(), accept(), and connect(), to create a logical transport channel with specific addressing information between two hosts.

No Specific Channel Addressing Information

Only use bind() to address service port, no other procedures to establish a logical transport channel with specific addressing information. The destination addressing information must be injected into the sendto() function.
UDP Socket Release – close()

- close - Close a file descriptor
- int close(int fd);
  - fd: socket descriptor
RetValue = close(sockfd) ;
RetValue = close(sockfd);
RetValue = close(sockfd);  
returns zero on success. On error, -1 is returned
RetValue = close(sockfd) ;
RetValue = \texttt{close}(sockfd) ;
RetValue = \texttt{close}(sockfd) ;
sockfd : created by socket(PF_INET, SOCK_DGM, 0) ;

RetValue = close(sockfd) ;
Review of UDP Client/Server Operation
UDP Client

UDP Server

socket()

bind()

recvfrom()
UDP Client

UDP Server

socket()

socket()
bind()
recvfrom()

Blocking!!
wait incoming
userdatagram
UDP Client

socket()

sendto()

UDP Server

socket()
bind()
recvfrom()
Blocking!!
wait incoming
userdatagram
UDP Client

- socket()
- sendto()
- recvfrom()

UDP Server

- socket()
- bind()
- recvfrom()

Blocking!!
wait incoming userdatagram
UDP Client

- socket()
- sendto()
- recvfrom()

UDP Server

- socket()
- bind()
- recvfrom()
UDP Client

socket()

sendto()

recvfrom()

UDP Server

socket()

bind()

recvfrom()
UDP Client

socket()

sendto()

recvfrom()

UDP Server

socket()

bind()

recvfrom()

sendto()
**UDP Client**

- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`

**UDP Server**

- `socket()`
- `bind()`
- `recvfrom()`

Wait another UDP Client
UDP Client

 socket()

UDP Server

 socket()
 bind()
 recvfrom()
IPv6 Socket Programming
IPv6 Socket Programming
IPv4 Exhaustion Counter (English)

INTEC Inc., provides a blogpost version of "IPv4 Exhaustion Counter" that visualize the status of IPv4 address exhaustion. This blogpost is licensed under Creative Commons License [Attribution-NonCommercial-NoDerivs 2.1 Japan].

[Overview]
"IPv4 Exhaustion Counter" is a blogpost that visualize the status of IPv4 address exhaustion which mashed up with the "IANA IPv4 Address Space Registry" provided by IANA and "IPv4 Address Report" researched by Mr. Geoff Huston of APNIC.

[Deployment]
Please copy a following deployment code (HTML tag) into design template or entry of your blog page, and HTML file of your Web page.

[Deployment Code]
```html
<iframe src="http://netcore.com.jp/IPv4ExhaustCounter.php" width="200" height="400" frameborder="0"
```

[Explanation]
1. It shows a name of RIR (Regional Internet Registry) with the nearest exhaustion day.
2. It shows unallocated and total IPv4 address blocks in each RIR.
3. It shows the X-day, the days until X-day and the amount of IPv4 address in the nearest exhaustion RIR (estimation).
4. It shows whether you access via IPv4 or IPv6.

Ref: IANA What is RIR?

[IANA Version]
Please use a following deployment code (HTML tag) for IPv4 Exhaustion Counter (IANA)

[Notice]
The contents or design of this tool may change/terminate without notice. This tool can't display on other sites that infringe the rights of any other entity, or offend public order and morals.

Get free downloads for IPv4 Exhaustion Counter for iPhone and iPod touch.
2011年6月8日 世界IPv6日

為增進所有網際網路社群對於IPv4位址枯竭的認知，並敦促所有的相關網際網路應用服務業者盡速在IPv6網路上進行各項服務及應用，主要負責網際網路相關的標準、教育、及政策的制定及推廣的國際網路學會(Internet Society, ISOC) 宣布將2011年6月8日訂為全球IPv6日(IPv6 World Day)，並邀請全球網際網路相關業者一起在IPv6網路上進行所提供的服務測試，目前包括：Facebook、Google、Yahoo、Akamai、Limelight Networks等知名網站將於今年的6月8日共同進行IPv6服務的測試，預計之後將有來自全球的網際網路社群陸續加入。本次的測試也將是網際網路史上第一次全球性規模的IPv6測試。

在6月8日當天所有的參加者將提供24小時的IPv6網路服務，根據IANA的預測，過去大家習慣使用的IPv4位址將在2011年初發放完畢，因此，所有的網際網路相關產業應該全面採用IPv6位址並在IPv6網路上提供服務，否則將面臨成本的增加及部分使用者無法使用服務的危機。包括ISP業者、網際網路硬體製造商、OS系統業者、及其他的網際網路相關的產業都應當及早進行IPv6的移轉，以因應IPv6網路時代的來臨。
IPv4 Addressing Problems

- Does IPv4 Addressing Space be exhausted?
  - Every Equipment (Host Needs) a Unique IPv4 Address
  - Network Usage is an Exponential Growth
IPv4 Addressing - Temporary Solution

- Sub-Netting
- Classless Inter Domain Routing – CIDR
- Dynamic Address Allocation – Dynamic Host Configuration Protocol (DHCP)
- Network Address Translation (NAT) – Private Networks
IPv4 Addressing – Permanent Solution

- Extend IPv4 Addressing Space – New Addressing Model
- Revise IPv4 Originating Design Defects
- Compromise the Developing Trend of Network Technology
IPv4 Originating Defects

- Variable Header Length
  - Option Fields
- Cyclic Redundancy Check (CRC)
  - Overloading
- Maximum Transmission Unit (MTU)
  - IP Packet Fragmentation/Resassembly

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>Header Length</td>
</tr>
<tr>
<td>Type of Service</td>
<td>Datagram Length</td>
</tr>
<tr>
<td>16-bit Identifier</td>
<td>Flags</td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
</tr>
<tr>
<td></td>
<td>Header Checksum</td>
</tr>
<tr>
<td></td>
<td>32-bit Source IP Address</td>
</tr>
<tr>
<td></td>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td></td>
<td>Options (if any)</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
</tbody>
</table>
Next Generation Internet Protocol – IPv6

- Internet Protocol Version 6 – RFC 2460
- Expanded Addressing Capability – 128 bits
- Autoconfiguration Mechanisms
- Simplification of Header Format – Fixed 40 bytes
- Improved Support for Extensions and Options
- Extension for Authentication and Privacy
- Flow Label Capability
IPv4 and IPv6 Headers
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ver</td>
<td>Version</td>
</tr>
<tr>
<td>Header Length</td>
<td>16-bit Identifier</td>
</tr>
<tr>
<td>Type of Service</td>
<td>Flags 13-bit Fragmentation Offset</td>
</tr>
<tr>
<td>Datagram Length</td>
<td>Time-to-Live</td>
</tr>
<tr>
<td></td>
<td>Upper-Layer Protocol</td>
</tr>
<tr>
<td></td>
<td>Header Checksum</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
</tr>
<tr>
<td>Options (if any) + Padding</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>Traffic Class</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>Next Header</th>
<th>Hop Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

128-bit Source IP Address

128-bit Destination IP Address
<table>
<thead>
<tr>
<th>Ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 16-bit Identifier
- 13-bit Fragmentation Offset
- Time-to-Live
- Upper-Layer Protocol
- Header Checksum

<table>
<thead>
<tr>
<th>Ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Payload Length
- Next Header
- Hop Limit

- 128-bit Source IP Address
- 128-bit Destination IP Address

- Options (if any) + Padding

- 32-bit Source IP Address
- 32-bit Destination IP Address
## Protocol Version

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identifier</td>
<td></td>
<td></td>
<td>13-bit Fragmentation Offset</td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
<td></td>
<td>Header Checksum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Destination IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Options (if any) + Padding</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Payload Length</td>
<td>Next Header</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128-bit Destination IP Address</td>
<td></td>
</tr>
</tbody>
</table>
### Quality of Service

**Header Length**

- **ver**: 4 bits
- **Header Length**: 16 bits

**TOS**

- **Flags**
- **13-bit Fragmentation Offset**

**Datagram Length**

- **16-bit Identifier**
- **Upper-Layer Protocol**
- **Header Checksum**

**Time-to-Live**

- **32-bit Source IP Address**

**Upper-Layer Protocol**

- **32-bit Destination IP Address**

**Options (if any) + Padding**

**Payload Length**

- **Next Header**
- **Hop Limit**

**Flow Label**

- **128-bit Source IP Address**
- **128-bit Destination IP Address**
Payload Length

<table>
<thead>
<tr>
<th>ver</th>
<th>Header</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16-bit Identifier</th>
<th>Flags</th>
<th>13-bit Fragmentation Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Time-to-Live</th>
<th>Upper-Layer Protocol</th>
<th>Header Checksum</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>32-bit Source IP Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>32-bit Destination IP Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (if any) + Padding</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>Next Header</th>
<th>Hop Limit</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>128-bit Source IP Address</th>
</tr>
</thead>
</table>

| 128-bit Destination IP Address |
## Lifetime of Packet

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identifier</td>
<td>FLAGS</td>
<td>13-bit Fragmentation Offset</td>
<td></td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
<td>Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any) + Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload Length</td>
<td>Next Header</td>
<td>Hop Limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Protocol Indicator

### Header Length

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Time-to-Live
- **16-bit Identifier**
- **13-bit Fragmentation Offset**
- **Upper-Layer Protocol**
- **Header Checksum**
- **32-bit Source IP Address**
- **32-bit Destination IP Address**
- **Options (if any) + Padding**

### Flow Label

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>Next Header</th>
<th>Hop Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 128-bit Source IP Address | 128-bit Destination IP Address |
# Source Address

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ver</td>
<td>Version</td>
</tr>
<tr>
<td>Header Length</td>
<td>Length of IP header</td>
</tr>
<tr>
<td>TOS</td>
<td>Type of Service</td>
</tr>
<tr>
<td>Datagram Length</td>
<td>Length of data portion</td>
</tr>
<tr>
<td>16-bit Identifier</td>
<td>16-bit Identifier</td>
</tr>
<tr>
<td>13-bit Fragmentation</td>
<td>Fragmentation offset</td>
</tr>
<tr>
<td>Offset</td>
<td>Fragment offset</td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Time-to-Live</td>
</tr>
<tr>
<td>Upper-Layer Protocol</td>
<td>Upper-layer protocol</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>Header checksum</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>Source IP address</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>Destination IP address</td>
</tr>
<tr>
<td>Options (if any)</td>
<td>Options (if any)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Payload Length</td>
<td>Next Header</td>
</tr>
<tr>
<td></td>
<td>Hop Limit</td>
<td></td>
</tr>
</tbody>
</table>

128-bit Source IP Address

128-bit Destination IP Address
### Destination Address

<table>
<thead>
<tr>
<th></th>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16-bit Identifier</td>
<td>F L A G S</td>
<td>13-bit Fragmentation Offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
<td>Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options (if any) + Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Payload Length</td>
<td>Next Header</td>
<td>Hop Limit</td>
</tr>
<tr>
<td></td>
<td>128-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Header Checksum**

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16-bit Identifier</th>
<th>13-bit Fragmentation Offset</th>
<th>Header Checksum</th>
<th>Time-to-Live</th>
<th>Upper-Layer Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit Source IP Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>32-bit Destination IP Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (if any) + Padding</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>Next Header</th>
<th>Hop Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>128-bit Source IP Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>128-bit Destination IP Address</th>
</tr>
</thead>
</table>

**Advanced Network Technology**
## Datagram Fragmentation/Reassembly

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16-bit Identifier</td>
<td></td>
<td>13-bit Fragmentation Offset</td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
<td>Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options (if any) + Padding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Payload Length</td>
<td>Next Header</td>
</tr>
<tr>
<td></td>
<td>Hop Limit</td>
<td></td>
</tr>
</tbody>
</table>

128-bit Source IP Address

128-bit Destination IP Address
Variable Header Length with Option Fields

<table>
<thead>
<tr>
<th>ver</th>
<th>Header Length</th>
<th>TOS</th>
<th>Datagram Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identifier</td>
<td>F L A G S</td>
<td>13-bit Fragmentation Offset</td>
<td></td>
</tr>
<tr>
<td>Time-to-Live</td>
<td>Upper-Layer Protocol</td>
<td>Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any) + Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ver</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload Length</td>
<td>Next Header</td>
<td>Hop Limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IPv4 and IPv6 use the same socket functions

With Different Protocol Family and Socket Address Structure
IPv4 and IPv6 Socket Address Structures

- Different Protocol Family
- Different Addressing Length

```
IPv4
140.123.101.100
AF_INET

struct sockaddr_in {
    u_int8_t   sin_len;
    u_int8_t   sin_family;
    struct sockaddr_in sin;
    ...
    sin.sin_family = AF_INET;
    sin.sin_port = htons(32);
    sin.sin6_addr = INADDR_ANY;
}

IPv6
fe80::0000:0000:0000:020c:f1ff:fe28:2f15
    (fe80::20c:f1ff:fe28:2f15):
AF_INET6

struct sockaddr_in6 {
    u_int8_t   sin6_len;
    u_int8_t   sin6_family;
    struct sockaddr_in6 sin6;
    ...
    sin6.sin6_family = AF_INET6;
    sin6.sin6_port = htons(32);
    sin6.sin6_addr = any6_addr;
}```
socket()
socket()

connect()
setsockopt()

bind()
listen()
IPv4 Address : 140.123.101.100
IPv6 Address : fe80::20c:f1ff:fe28:2f15

sockaddr_in
sockaddr_in6
IPv4 Address : 140.123.101.100
IPv6 Address : fe80::20c:f1ff:fe28:2f15

socket(), bind(), listen(), accept(), send(), recv(), setsockopt(), sendto(), recvfrom(), connect(), close()

struct sockaddr_in {
  u_int8_t   sin_len ;
  u_int8_t   sin_family ;
  u_int16_t sin_port ;
  struct in_addr sin_addr ;
  int8_t       size_zero[8] ;
} ;

struct sockaddr_in6 {
  u_int8_t   sin6_len ;
  u_int8_t   sin6_family ;
  u_int16_t sin6_port ;
  u_int32_t sin6_flowinfo ;
  struct in6_addr sin6_addr ;
  u_int32_t sin6_scope_id ;
} ;
IPv4 Address: 140.123.101.100
IPv6 Address: fe80::20c:f1ff:fe28:2f15

**struct sockaddr_in:**

- u_int8_t sin_len
- u_int8_t sin_family
- u_int16_t sin_port
- struct in_addr sin_addr
- int8_t size_zero[8]

**struct sockaddr_in6:**

- u_int8_t sin6_len
- u_int8_t sin6_family
- u_int16_t sin6_port
- u_int32_t sin6_flowinfo
- struct in6_addr sin6_addr
- u_int32_t sin6_scope_id

- Length: 16 bytes
- 32-bit IPv4 Address
- Unused
- Fixed-length (16 bytes)

- Length: 28 bytes
- 32-bit IPv6 Address
- Flow Label
- 128-bit IPv6 Address
- Scope ID
- Fixed-length (28 bytes)
IPv4 Address: 140.123.101.100
IPv6 Address: fe80::20c:f1ff:fe28:2f15

### SOCKADDR_INET

<table>
<thead>
<tr>
<th>length</th>
<th>AF_INET</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit Port #</td>
<td></td>
</tr>
<tr>
<td>32-bit IPv4 Address</td>
<td></td>
</tr>
<tr>
<td>(unused)</td>
<td></td>
</tr>
</tbody>
</table>

#### Fixed-length (16 bytes)

### SOCKADDR_INET6

<table>
<thead>
<tr>
<th>length</th>
<th>AF_INET6</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit port #</td>
<td></td>
</tr>
<tr>
<td>32-bit Flow Label</td>
<td></td>
</tr>
<tr>
<td>128-bit IPv6 Address</td>
<td></td>
</tr>
<tr>
<td>32-bit Scope ID</td>
<td></td>
</tr>
</tbody>
</table>

#### Fixed-length (28 bytes)
IPv4 and IPv6 Coexisting
Only IPv4 Protocol Stack

IPv4 Client

TCP

IPv4

Datalink

IPv4 Server

TCP

IPv4

Datalink
Only IPv6 Protocol Stack

IPv6 Client
  └── TCP
    └── IPv6
      └── Datalink

IPv6 Server
  └── TCP
    └── IPv6
      └── Datalink
IPv6 Client

TCP

IPv6

Datalink

IPv6 Server

TCP

IPv6

Datalink
IPv4/IPv6 Client

TCP

IPv4

IPv6

Datalink

IPv4/IPv6 Server

TCP

IPv4 mapped IPv6 Address

IPv4

IPv6

Datalink
Comparison of IPv4 and IPv6
Socket Functions
IPv4 Address Format

- IPv4 Address Length: 32 bits (4 bytes)
- “140.123.101.100” – Dotted-Decimal String
- Network Byte Ordered IPv4 Internet Address Structure
  ```c
  struct in_addr {
    in_addr_t s_addr;
  }
  ```
  - in_addr_t: uint32_t (Unsigned 32-bit integer)
- Conversion of Network Byte Address and Address String
- Byte Manipulation Functions
  - int inet_aton()
  - in_addr_t inet_addr()
  - char *inet_ntoa()
int inet_aton(const char *cp, struct in_addr *inp);

Converts the Internet host address \texttt{cp} from the standard numbers-and-dots notation into binary data and stores it in the structure that \texttt{inp} points to.

\texttt{inet_aton()} returns non-zero if the address is valid, zero if not.

\texttt{struct in_addr dest;}
\texttt{inet_aton("140.123.101.114", &dest.sin_addr);}
in_addr_t inet_addr(const char *cp);

Convert the Internet host address `cp` from numbers-and-dots notation into binary data in network byte order.

If the input is invalid, INADDR_NONE (usually -1) is returned.

```c
struct in_addr dest;
dest.sin_addr = inet_addr("140.123.101.114");
```
char *inet_ntoa()

char *inet_ntoa(struct in_addr in);

Convert the Internet host address in given in network byte order to a string in standard numbers-and-dots notation.

The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

struct in_addr in;
char ipstr[17];
memset(ipst, 17, 0);
in.s_addr = inet_addr("140.123.101.114");
strcpy(ipstr, inet_ntoa(in));
IPv6 Address Format

- IPv6 Address Length: 128 bits (16 bytes)
- “fe80:0000:0000:0000:0202:b3ff:fe1e:8329” – Colon-Hexadecimal String
- “fe80:0:0:0:202:b3ff:fe1e:8329”
- “fe80::202:b3ff:fe1e:8239” – Abbreviation
- Network Byte Ordered IPv6 Internet Address Structure
  - struct in6_addr {
    uint8_t s6_addr[16];
  }
  - uint8_t (Unsigned 8-bit integer)
**IPv6 Address Format**

- Conversion of Network Byte Address and Address String
- Byte Manipulation Functions
  - `int inet_pton()`
  - `const char *inet_ntop()`

<table>
<thead>
<tr>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP Address String:</strong> “140.123.101.114”</td>
<td>“fe80:0000:0000:0000:0202:b3ff:fe1e:8329”</td>
</tr>
<tr>
<td><code>inet_pton(AF_INET, src, des)</code></td>
<td><code>inet_pton(AF_INET6, src, des)</code></td>
</tr>
<tr>
<td>Network Byte Ordered Address</td>
<td>8C 7B 65 72</td>
</tr>
</tbody>
</table>
int inet_pton()

int inet_pton(int af, const char *src, void *dst);
Convert the character string src into a network address
structure in the af address family, then copies the
network address structure to dst

<table>
<thead>
<tr>
<th>parameters</th>
<th>af</th>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4</td>
<td>AF_INET</td>
<td>char</td>
<td>struct in_addr{} ;</td>
</tr>
<tr>
<td>IPv6</td>
<td>AF_INET6</td>
<td>char</td>
<td>struct in6_addr{} ;</td>
</tr>
</tbody>
</table>

struct in_addr in4 ;
struct in6_addr in6 ;
inet_pton(AF_INET, "140.123.101.114", &in4);
inet_pton(AF_INET6, "fe80:0000:0000:0000:0202:b3ff:fe1e:8329", &in6)
const char *inet_ntop() 

const char *inet_ntop(int af, const void *src, char *dst, socklen_t cnt);

Convert the network address structure src in the af address family into a character string, which is copied to a character buffer dst, which is cnt bytes long.

<table>
<thead>
<tr>
<th>parameters</th>
<th>af</th>
<th>src</th>
<th>dst</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4</td>
<td>AF_INET</td>
<td>struct in_addr{} ;</td>
<td>char</td>
<td>INETADDRSTRLEN(14)</td>
</tr>
<tr>
<td>IPv6</td>
<td>AF_INET6</td>
<td>struct in6_addr{} ;</td>
<td>char</td>
<td>INET6ADDRSTRLEN(48)</td>
</tr>
</tbody>
</table>

struct in_addr in4 ;
struct in6_addr in6 ;
char ipstr[256] ;

inet_ntop(AF_INET, in4, &ipstr, INETADDRSTRLEN);
inet_pton(AF_INET6, in6, &ipstr, INET6ADDRSTRLEN)
Comparison of Socket Address Structures

IPv4

<table>
<thead>
<tr>
<th>length</th>
<th>AF_INET</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit port#</td>
<td></td>
</tr>
<tr>
<td>32-bit IPv4 address</td>
<td></td>
</tr>
<tr>
<td>(unused)</td>
<td></td>
</tr>
</tbody>
</table>

fixed-length (16 bytes)

IPv6

<table>
<thead>
<tr>
<th>length</th>
<th>AF_INET6</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit port#</td>
<td></td>
</tr>
<tr>
<td>32-bit flow label</td>
<td></td>
</tr>
<tr>
<td>128-bit IPv6 address</td>
<td></td>
</tr>
<tr>
<td>32-bit scope ID</td>
<td></td>
</tr>
</tbody>
</table>

fixed-length (28 bytes)

Storage

<table>
<thead>
<tr>
<th>length</th>
<th>AF_XXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>(opaque)</td>
<td></td>
</tr>
</tbody>
</table>

longest on system
IPv4/IPv6 Socket Functions

- **Same Socket Functions**
  - Socket Creation
  - Socket Connection
  - Message Sending/Receiving

- **Different Protocol Domains and Socket Address Structure**
  - IPv4 Protocol Domain - AF_INET
  - IPv4 Socket Address Structure - sockaddr_in{}
  - IPv6 Protocol Domain - AF_INET6
  - IPv6 Socket Address Structure - sockaddr_in6{}

<table>
<thead>
<tr>
<th>Types</th>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Family</td>
<td>AF_INET</td>
<td>AF_INET6</td>
</tr>
<tr>
<td>Address Information Structure</td>
<td>in_addr</td>
<td>In6_addr</td>
</tr>
<tr>
<td>Socket Address Structure</td>
<td>sockaddr_in</td>
<td>sockaddr_in6</td>
</tr>
<tr>
<td></td>
<td>sockaddr_storage</td>
<td></td>
</tr>
</tbody>
</table>
IPv4/IPv6 Program Migration

- Direct change Domain Protocol
  - AF_INET (IPv4) \(\rightarrow\) AF_INET6 (IPv6)
  - IP_PROTO_IP (IPv4) \(\rightarrow\) IP_PROTO_IP6 (IPv6)
- Direct change Socket Address Structure and Internet Address Structure
  - sockaddr_in{} (IPv4) \(\rightarrow\) sockaddr_in6{}/sockaddr_storage (IPv6)
- Conversion of Network Byte Ordered Address and IP String
  - inet_aton(), inet_addr(), inet_ntoa() – IPv4
  - inet_pton(), inet_ntop() – IPv6
- Host Name and IP Address Translations Function
  - gethostbyname(), gethostbyaddr () – IPv4
  - getaddrinfo(), getnameinfo() – IPv6
- Conversion of data message containing IP address information
  - FTP, SIP, H323, ...
IPv4/IPv6 Program Migration

- Utility Tool – Microsoft’s checkv4.exe
  - Included in Microsoft Platform SDK
- List the suggestion of code modification
  - Protocol Domain / Protocol Type
  - Socket Address Structures
  - Network Byte Ordered Functions
  - Host Name and IP Address Translation Functions
IPv4/IPv6 TCP Operations
IPv4/IPv6 UDP Operations
UDP Client

- `socket`
- `sendto`
- `recvfrom`

UDP Server

- `socket`
- `bind`
- `recvfrom`
- `sendto`

IPv4
- `AF_INET`
- `in_addr{}`
- `sockaddr_in{}`

IPv6
- `AF_INET6`
- `in6_addr{}`
- `sockaddr_in6{}`
在IPv4/IPv6的程式設計，最主要的差異在於
(A). Socket Address Structure 與 Internet Address Format()
(B). Socket 建立階段：socket(), bind(), listen(), accept()
(C). Socket 資料送收階段：send()/recv(), sendto()/recvfrom()
(D). 以上皆是