Tcl and IPv6

Reinhard Max

- Work for SUSE since 1997
- Responsible for various RPMs:
  - Tcl/Tk
  - SQLite
  - PostgreSQL
  - ClamAV
- Added IPv6 support to [socket] for Tcl 8.6 over the last three years

Oustervote

Who hasn't heared of IPv6 before?

Oustervote

Who has used IPv6 networking?

Oustervote

Who has coded for IPv4 in C?

Oustervote

Who has coded for IPv6 in C?

Oustervote

Who has coded for IPv4 in Tcl?

Oustervote

Who has coded for IPv6 in Tcl?

Oustervote

Who has a Tcl code base that needs to get ported to IPv6 sooner or later?

Oustervote

Who has a Tcl code base that needs to get ported to IPv6 sooner or later?

Chances are good that you get IPv6 for free just by switching to Tcl 8.6.

Oustervote

Who has a C code base that needs to get ported to IPv6 sooner or later?

Oustervote

Who has a C code base that needs to get ported to IPv6 sooner or later?

Sorry, slightly more work for you.

IPv4 vs. IPv6

IPv4 vs. IPv6

- 2^32 IP addresses
- Large routing tables
- Needs NAT to overcome address space exhaustion
- Painful renumbering
- ...

- 2^128 IP addresses
- Aggregate routing
- Plenty of addresses, no NAT needed
- Easy renumbering
- ...

What do IP addresses look like?

- IPv4: dotted decimal notation
  - 123.4.56.7
- IPv6: 16 bit hex groups separated by colons
  - 123:4567:89ab:cdef:dead:beef:42:1
  - One sequence of zeroes can be shortcut as double colons:
    1234:0:0:0:0:0:0:5678 → 1234::5678

Special addresses

- Loopback: 127.0.0.1 vs. ::1
- Wildcard: 0.0.0.0 vs. ::

[socket] up to Tcl 8.5

```
% socket -server dummy 4242
sock3
% fconfigure sock3 -sockname
0.0.0.0 0.0.0.0 4242

% socket localhost 4242
sock4
% fconfigure sock4 -sockname
127.0.0.1 localhost 56100
% fconfigure sock4 -peername
127.0.0.1 localhost 4242
```

[socket] in Tcl 8.6

```
% socket -server dummy 4242
sock6a2fa0
% fconfigure sock6a2fa0 -sockname
0.0.0.0 0.0.0.0 4242 :: :: 4242

% socket localhost 4242
sock6ab2c0
% fconfigure sock6ab2c0 -sockname
::1 localhost 56100
% fconfigure sock6ab2c0 -peername
::1 localhost 4242
```

But watch out!

But watch out!

http://example.com/

http://1.2.3.4/

http://example.com:8080/

http://1.2.3.4:8080/

But watch out!

http://example.com/

http://1.2.3.4/

http://example.com:8080/

http://1.2.3.4:8080/

http://2001:DB8::8080/ (fail!)

But watch out!

http://example.com/

http://1.2.3.4/

http://example.com:8080/

http://1.2.3.4:8080/

http://2001:DB8::8080/ (fail!)

http://[2001:DB8::8080]/

http://[2001::DB8::]:8080/

http://[2001:DB8::8080]:8080/

The C side of things

IPv4 only Client

```
int sock;
struct sockaddr_in addr;
struct hostent *host;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
host = gethostbyname(name);
memcpy(&addr.sin_addr, host->h_addr,
       host->h_length);
sock = socket(AF_INET, SOCK_STREAM, 0);
connect(sock,(struct sockaddr*)&addr,
        sizeof(addr));
```

Cross Protocol Client

```
int sock;
struct addrinfo *a, *p, hints;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = 0;
getaddrinfo(host, service, &hints, &a);
for (p=a, p, p=p->ai_next) {
    sock = socket(p->ai_family,
                  p->ai_socktype,
                  p->ai_protocol);
    connect(sock, p->ai_addr, ai_addrlen);
}
```

Cross Protocol Client (2)

```
struct addrinfo { /* shortened */
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    struct sockaddr *ai_addr;
    struct addrinfo *ai_next;
};
```

IPv4 only Server

```
int sock;
struct sockaddr_in addr;
addr.inaddr.sin_addr.s_addr =
     htonl(INADDR_ANY);
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
sock = socket(AF_INET,SOCK_STREAM,0);
bind(sock, (struct sockaddr*)&addr,
     sizeof(addr));
listen(sock, 20);
```

Cross Protocol Server

```
int sock;
struct addrinfo *a, *p, hints;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(hostname, portname, &hints, &a);
for (p=a, p, p=p->ai_next) {
    sock = socket(p->ai_family,
                  p->ai_socktype,
                  p->ai_protocol);
    bind(sock, p->ai_addr, p->ai_addrlen);
    listen(sock, 20);
}
```

Client Server differences

- Client
  - Loop the remote addresses until one succeeds.
- Server
  - Set the AI_PASSIVE flag
  - Loop over the local addresses and keep all that succeeded
  - Use select() or poll() to wait for incoming connections on multiple listening sockets at once.

Questions?