Aplicativos web e IPv6

Programação IPv6 para web

Linguagens de programação tais como C, Python, Perl e o kit de ferramentas Qt podem trabalhar com IPv6. Mas você precisa se assegurar de que está usando as funções, classes e métodos corretos. Vamos mostrar como levar o seu aplicativo legado para o mundo do IPv6.

por Peter Hrenka

A migração para a Internet IPv6 só vai funcionar se as aplicações que estão rodando na web também migrarem. Este artigo explica como os desenvolvedores podem readequar programas tanto para IPv4 quanto para IPv6. Você vai ver como trabalhar com IPv6 em C, Python, C++, Perl e com o kit de ferramentas Qt.

A maioria dos programas usados como exemplo não necessitam de uma conexão nativa em IPv6 à Internet, o que significa que você pode testar seu código localmente, em qualquer distribuição Linux recente.

C 101

Os programadores ainda precisarão emitir uma chamada de sistema via `socket()` no IPv6 para definir um destino no formato de descritor de arquivos. Se esse destino tiver de ser compatível com o IPv6, o domínio deve ser `PF_INET6` ou `AF_INET6`. Os tipos aqui podem ser os mesmos usados no IPv4, `SOCK_STREAM` para UDP e `SOCK_DRAM` para TCP. As coisas começam a ficar empolgantes de verdade quando o socket realmente quer iniciar uma conexão. No lado do cliente é `connect()` e no lado do servidor é `bind()`. Ambos com a expectativa de obter um `const struct sockaddr*`. Mas onde encontrá-lo?

Entre outras coisas, o RFC 3493 [1] descreve a função `getaddrinfo()`, que retorna estas estruturas `sockaddr`. O `getaddrinfo()` substitui o `gethostbyname()`, que é obsoleto e incapaz de realizar multithreading (ou, para ser mais preciso: incapaz de reentrância). De modo geral, a função `getaddrinfo()` ajuda a migrar gradualmente, sem dor, do IPv4 para o IPv6. A função trabalha com resolução de nomes, parsing (análise sintática), criação de estruturas de endereços e parâmetros para chamadas subsequentes de rede.

A **listagem 1** introduz a função `getaddrinfo()` e seus parâmetros. A `node` aponta para uma string que descreve o endereço com maior detalhamento. A string é normalmente um hostname na forma de `www.linuxmagazine.com.br`. Também são permitidas notações de endereços IPv4 ou IPv6, tais como `127.0.0.1` ou `FF80::0201:02FF:FE03:0405%eth0`. O parâmetro `service` descreve o número da porta.o Os designadores de serviço, como aqueles listados em `/etc/services`, também são permitidos – por exemplo, `http` ou `ssh` – bem como números de portas em notação decimal.

O parâmetro opcional `hint` pode apontar para uma `addrinfo structure` tal como a que é definida na **listagem 2**. `res` também usa essa estrutura, que dá suporte a vários resultados ligados por `ai_next` para chamar `freeaddrinfo()` quando os endereços não são mais necessários.

Os desenvolvedores especificam o `AF_INET` como sendo o `ai_family` para receber endereços IPv4 e o `AF_INET6` para receber endereços IPv6. Caso precise dos dois tipos de endereço, você deve especificar o `AF_UNSPEC`, que vai lhe dar primeiro o(s) endereço(s) IPv6 e depois o(s) IPv4. A estrutura `res` nunca possui `AF_UNSPEC`. Em vez

disso, mantém a família de endereços tangíveis do endereço de retorno.

Para o `ai_socktype`, você deve inserir o `SOCK_STREAM` para UDP, ou o `SOCK_DGRAM` para TCP, na estrutura `hint`. Ao inserir um `0`, você verá várias entradas com os possíveis tipos para cada endereço, se houver. O `ai_protocol` permite ao programador requisitar um protocolo, tal como `FP_INET` ou `FP_INET6`. Geralmente é normal usar o valor `0` para receber resultados que batam com os outros parâmetros. Um aspecto prático é que os campos `ai_family`, `ai_socktype` e `ai_protocol` correspondem precisamente aos parâmetros `domain`, `type` e `protocol` na chamada de sistema `socket()`, o que significa que você pode simplesmente deixá-los para lá.

O campo `ai_addrlen` armazena o valor retornado do comprimento da estrutura do `sockaddr_in`. O `ai_addr` contém um ponteiro para um `sockaddr_in` para IPv4, ou um `sockaddr_in6` para IPv6 dentro da estrutura `res`. Este apontador pode ser usado junto ao `ai_addrlen` como parâmetro para as chamadas de sistema `connect()`, `bind()`, `sendto()` ou `recvfrom()`. O campo `ai_canonname` da estrutura `res` contém um apontador para o nome canônico do host, caso você o requisite pelos `ai_flags`. Dê uma olhada no **quadro 1** para ter uma ideia sobre as flags.

Efeitos colaterais

Se você quiser escrever programas compatíveis com IPv6, deve evitar certas funções. Por exemplo, evite aquelas que esperam ou retornam uma `struct in_addr` como parâmetro (**listagem 3**).

As funções alternativas `inet_pton()` e `inet_ntop()` existem por causa de `inet_aton()` e `inet_ntoa()`, que convertem notações numéricas ASCII e `struct in_addr` para lá e para cá (**listagem 4**).

Python

A versão 2.2 do Python adicionou uma função `getaddrinfo()` ao módulo padrão do `socket` [2]. Felizmente, as características perversas do C (o que inclui parâmetros obsoletos dentro das estruturas `hints` ou `res`) não se aplicam ao Python. Alguns parâmetros opcionais desta linguagem manipulam a função da estrutura `hints` da interface C:

```
socket.getaddrinfo(host,
➥ port,family=0,socktype=0,
➥ U proto=0, flags=0)
```

Listagem 1: getaddrinfo()

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);
```

Listagem 2: struct addrinfo

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

Listagem 3: Funções incompatíveis com IPv6

```
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in)
```

Listagem 4: inet_pton e inet_ntop

```
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

Listagem 5: getaddrinfo.py

```
#!/usr/bin/env python
import sys
from socket import *

host = sys.argv[1]
port = 80 if len(sys.argv)<3
else sys.argv[2]

for addrinfo in
getaddrinfo(host, port, AF_
UNSPEC, SOCK_STREAM):
    family, socktype, proto,
canonname, sockaddr = addrinfo
    socketObject =
socket(family, socktype, proto)
    #socketObject =
socket(*addrinfo[:3])
    if socketObject is None:
continue
    haveConnection = False
    try:
        socketObject.
connect(sockaddr)
        socketObject.close()
        haveConnection = True
    except:
        pass
    familyString = "IPv6" if
family==AF_INET6 else "IPv4"
    args = familyString,
sockaddr[0], haveConnection
    print("{0} address {1},
connect = {2}".format(*args))
```

As constantes requeridas `AF_` e `AI_` também estão disponíveis no módulo `socket`.

O valor de retorno do `socket.getaddrinfo()` é uma lista de cinco sets, na seguinte forma:

```
(family, socktype, proto,
➥ canonname, sockaddr)
```

Os três primeiros parâmetros podem ser passados para o `socket.socket()` para criar um objeto `socket`. O `canonname` retorna o valor do nome canônico se for requisitado via `flags`. O último parâmetro, `sockaddr`, corresponde ao `ai_addr`, o qual é retornado pelo Python como um set com diferentes tamanhos, dependendo da família de endereços. A primeira entrada neste conjunto é sempre uma string que contém um endereço

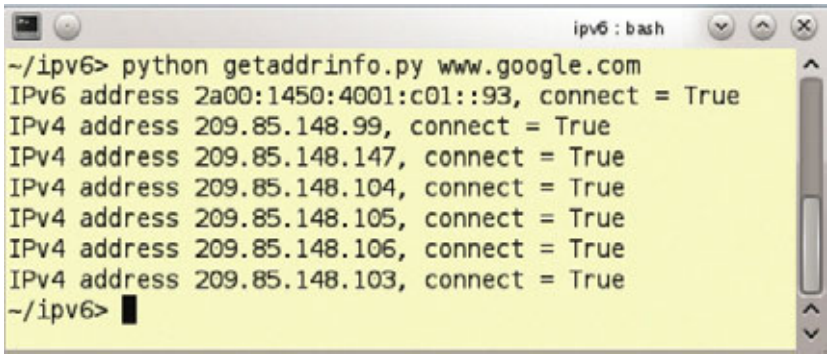

Figura 1 O Google pode ser encontrado tanto pelo IPv4 como pelo IPv6, tal como mostra este programa exemplo em Python.

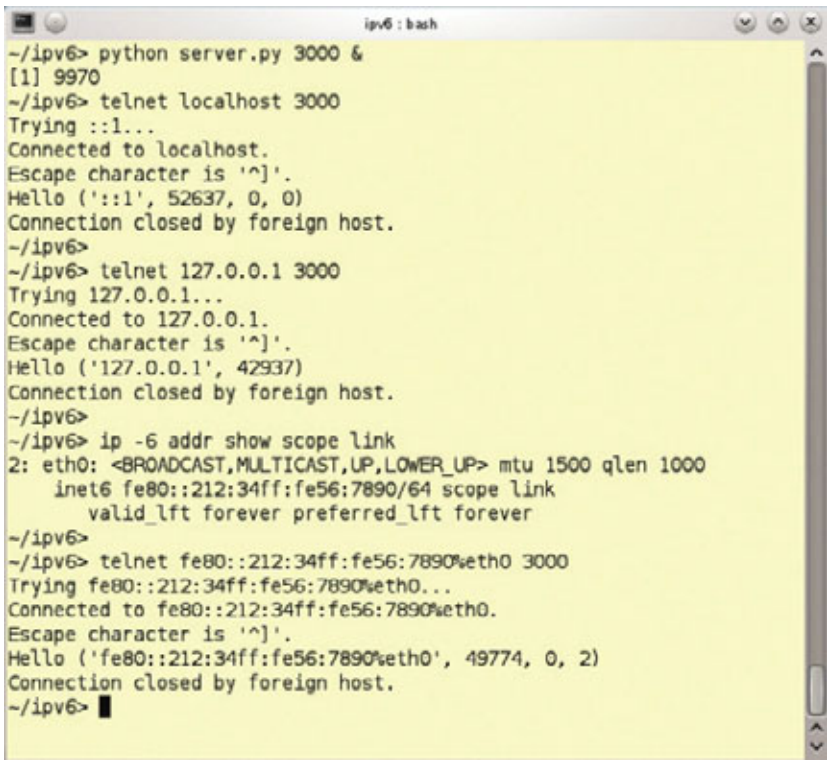

Figura 2 O servidor programado em Python responde às conexões telnet em IPv6 e IPv4.

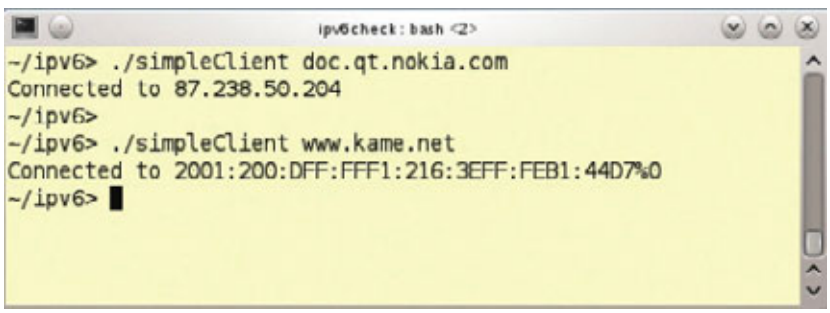

Figura 3 Um pequeno programa em Qt para a linha de comando descobre os endereços IP dos hostnames.

Listagem 6: server.py

```
01 #!/usr/bin/env python
02 import sys
03 from socket import *
04 from select import select
05
06 host  = None
07 port  = sys.argv[1]
08 flags = AI_PASSIVE
09
10 serverSockets = []
11
12 for addrinfo in
getaddrinfo(host, port, AF_
UNSPEC, SOCK_STREAM, 0, flags):
13     family, socktype, proto,
canonname, sockaddr = addrinfo
14     serverSocket =
socket(family, socktype, proto)
15     if family==AF_INET6:
16         serverSocket.
setsockopt(IPPROTO_IPV6, IPV6_
V6ONLY, 1)
17     serverSocket.
bind(sockaddr)
18     serverSocket.listen(1)
19     serverSockets.
append(serverSocket)
20
21
22 while True:
23     readable, writable,
special = select(
serverSockets, [], [])
24     for readSocket in
readable:
25         connectionSocket,
connectionAddress = readSocket.
accept()
26         connectionSocket.
send("Hello {0}\n".
format(connectionAddress).
encode("ascii"))
27         connectionSocket.
close()
```

IP em notação numérica; a segunda é sempre o número da porta. Para o IPv6, há entradas adicionais para `flow info` e `scope id`. Em ambos os casos, o ajuste pode ser usado como parâmetro para os métodos `socket bind()` e `connect()`.

O primeiro programa exemplo é um pequeno script Python (**listagem 5**) que usa `getaddrinfo()` para exibir os endereços para um host e tenta conectar-se a cada um destes endereços. Você pode usar `getaddrinfo()`, por exemplo, para verificar se um determinado servidor web é acessível via IPv6. O código é compatível com ambas as versões do Python: 2.7 e 3.1.

O primeiro parâmetro que o script espera é o nome de um host (`hostname`) que ele passa, sem mudanças, para `getaddrinfo()` (**linha 8**). O padrão é 80.

O script, então, tenta fazer uma conexão para cada endereço encontrado, criando um objeto socket com `family`, `socktype` e com os parâmetros `proto` que foram retornados. Feito isto, tenta fazer uma conexão `connect()` em relação a `sockaddr` (**linha 15**). Se isso funcionar, será criada uma nota em `haveConnection`.

O Google só fornece endereços IPv6 a certos provedores que tenham passado no teste IPv6. É possível observar isso nesta saída que mostra que cada família de endereços pode conter diversas entradas (**figura 1**).

O nome de host `ipv6.google.com` sempre retorna um endereço IPv6. Tal como é possível ver do Facebook, você pode adicionar uma mensagem de alerta para um endereço IPv6 de forma criativa:

```
$
python getaddrinfo.py
➥ www.facebook.com
IPv6 address 2620:0:1c08:4000:
➥ face:b00c:0:1,connect = True
IPv4 address
➥ 69.171.224.41,connect = True
```

Qual a aparência do lado do servidor? Em uma aplicação servidora capacitada a IPv6, é necessário es-

Listagem 7: connectToHost

```
01 void    connectToHost ( const QString & hostName,
➥ quint16 port, OpenMode openMode = ReadWrite )
```

Listagem 8: simpleClient.cpp

```
01 #include <QtCore/QCoreApplication>
02 #include <QtCore/QStringList>
03 #include <QtNetwork/QHostAddress>
04 #include <QtNetwork/QTcpSocket>
05 #include <iostream>
06
07 int main(int argc, char *argv[])  {
08   QCoreApplication app(argc, argv);
09
10   QString host = app.arguments().at(1);
11   int port = 80;
12
13   QTcpSocket socket;
14   socket.connectToHost(host, port);
15
16   if (!socket.waitForConnected(1000)) {
17     std::cout << "Could not connect" << std::endl;
18     return 10;
19   }
20   QHostAddress peerAddress = socket.peerAddress();
21   QString address = peerAddress.toString();
22   std::cout << "Connected to "
23             << address.toAscii().constData() << std::endl;
24   socket.close();
25   return 0;
26 }
```

Listagem 9: greeter.h

```
01 #ifndef GREETER_H
02 #define GREETER_H
03 #include <QtCore/QObject>
04 #include <QtNetwork/QTcpServer>
05 #include <QtNetwork/QTcpSocket>
06
07 class Greeter : public QObject  {
08   Q_OBJECT
09
10 public:
11   Greeter(QObject *parent) : QObject(parent) {}
12
13 public slots:
14   void newConnection(QObject* serverObject) {
15     QTcpServer* server = static_cast<QTcpServer*>(serverObject);
16
17     QTcpSocket* connection = server->nextPendingConnection();
18     connect(connection, SIGNAL(disconnected()),
19          connection, SLOT(deleteLater()));
20
21     QHostAddress peerAddress = connection->peerAddress();
22     QString address = peerAddress.toString();
23
24     connection->write("Hello ");
25     connection->write(address.toAscii());
26     connection->write("\n");
27     connection->disconnectFromHost();
28   }
29 };
30 #endif
```

Quadro 1: Detalhes dos AI_FLAGS

AI_ADDRCONFIG: Se o programador fornecer apenas as especificações deste sinal (*flag*), os endereços só serão retornados se alguma interface de rede tiver configurado pelo menos um endereço não-loopback de tipo correspondente. Isso significa que um dispositivo sem conexão IPv6 só receberá endereços IPv4 e vice-versa.

Infelizmente, os desenvolvedores das atuais bibliotecas Glibc [10] e Eglibc [11] tomaram a decisão questionável de aceitar endereços para links locais IPv6 como configuráveis. Isso quer dizer que o `getaddrinfo()` apresenta endereços IPv6 inalcançáveis para os computadores que não tenham endereços IPv6 globais. Em tais ambientes as conexões normalmente falham logo, mas grandes timeouts ocorrem em alguns casos, e é justamente para evitá-lo que esta opção foi projetada. Fora isso, é aconselhável usar essa opção no lado do cliente para evitar tentativas de conexão e consultas DNS propensas a erros.

AI_PASSIVE: Retorna sockets apropriados às chamadas de sistema `bind()` e `accept()`.

AI_V4MAPPED: Retorna endereços IPv4 identificados por sockets `AF_INET6`, para os quais nenhum endereço IPv6 pôde ser verificado como endereço IPv6 do tipo mapeado em IPv4.

AI_ALL: Retorna todos os endereços IPv6 e IPv4 mapeados (apenas se for combinado com o `AI_V4MAPPED`).

AI_CANONNAME: Resolve um hostname canônico e o retorna dentro de uma lista `res` como primeiro item.

AI_NUMERICHOST: Evita consultas ao DNS. O `node` precisa ser especificado em notação numérica.

pecificar `None` como host, `AF_UNSPEC` como a família e `AI_PASSIVE` para os flags receberem os parâmetros e os endereços para sockets servidores que escutam as conexões em todas as interfaces de rede. Isso significa pelo menos dois sockets.

A primeira questão é quando um socket IPv4 ou IPv6 darão `bind` para a mesma porta, ao mesmo tempo. E a resposta é: depende da opção de socket `IPV6_V6ONLY`, já descrita no RFC3493.

No Python, você pode ajustar esse detalhe ao usar o método socket `sockopt()`.

```
socket.setsockopt
➥ (IPPROTO_IPV6, IPV6_V6ONLY, 1)
```

Infelizmente, a configuração padrão para essa opção depende do sistema operacional e até mesmo da distribuição. Alguns sistemas BSD a ativam por padrão e, no Linux, o administrador pode ativá-la durante a execução do sistema:

```
sysctl net.ipv6.bindv6only=1
```

É aconselhável que os programadores ajustem essa opção explicitamente e, neste caso, devem estar sempre prontos para ligar uma porta IPv6 mesmo que esta já esteja vinculada ao IPv4. Caso o `IPV6_V6ONLY` esteja desabilitado, as novas mensagens IPv4 são redirecionadas a um socket IPv6 por um socket V4-mapped.

As especificações IPv6 incluem uma técnica para mapeamento de endereços IPv4 para endereços IPv6 [3]. Apesar de os endereços IPv4 mapeados serem práticos, também possuem desvantagens. Uma delas é não ter representação textual, o que leva a strings modificadas, mais especificamente em saídas de log. Isso pode causar problemas para ferramentas de análise que usam, digamos, expressões regulares para buscar endereços IPv4 e não conseguem identificar novas strings. Por outro lado, uma série de problemas de segurança potenciais estão relacionados aos endereços mapeados em IPv4 [4]: o aplicativo não consegue distinguir se uma conexão IPv4 existe ou se está lidando com uma conexão IPv6 com um endereço manipulado.

O segundo problema com os dois server sockets é que a chamada `accept()` normalmente fica bloqueada enquanto espera por uma conexão. Para contornar esse problema, os desenvolvedores poderiam, teoricamente, ativar dois processadores no servidor ou trabalhar com várias threads. Contudo, isso não é necessário porque o `select` oferece uma solução bem mais elegante para o problema ao permitir que o programa espere por diversos sockets ao mesmo tempo. Essa abordagem pode parecer um pouco intrincada, mas os aplicativos mais complexos não conseguirão resolver o problema sem usar o `select` ou alternativas como `poll` ou `epoll`.

Em Python, você vai encontrar a função `select()` no módulo padrão Select [5].

```
readable, writable, special =
➥ select.select(rlist, wlist,
➥ xlist[, timeout])
```

Os parâmetros `rlist`, `wlist` e `xlist` são listas de arquivos ou objetos socket que reagem a leitura, escrita e eventos incomuns. Os três valores de retorno dados ao programa são listas de objetos que chegaram a um estado de espera. O valor para novas conexões é `readable`.

Em Python, o `accept()` é suprido como método de objeto socket que retorna um 2-tuple,

```
(conn, address) = socket.accept()
```

em que `conn` é o novo socket na conexão estabelecida e `address` é o endereço da outra parte da notação tuple do Python.

O próximo exemplo de código (**listagem 6**) usa uma dessas técnicas IPv6 do Python na prática. O código encarregado de "escutar" ouve a porta, aceita as conexões que estão

chegando e transfere seu endereço IP. A **figura 2** mostra uma sessão Telnet em que o servidor usa IPv6 e Ipv4.

Kit de ferramentas Qt

O framework QT orientado a objetos C++ fornece diversas abstrações para programação voltada para redes dentro da biblioteca `QtNetwork` [6]. Em circunstâncias normais, os programas cliente que usam os métodos corretos serão conectados automaticamente via IPv6. O Qt oferece as classes `QTcpSocket` e `QudpSocket` para programação de redes. Ambas são derivadas da classe básica `QAbstract-Socket`.

Para estabelecer uma conexão TCP em um programa cliente, o desenvolvedor precisa primeiro criar uma `QTcpSocket` e, então, chamar o método `QAbstractSocket`, definido na classe base `connectToHost()` (**listagem 7**). Como exemplo, considere um programa mínimo de Qt (**listagem 8**) que tenta abrir uma conexão para um servidor web usando o nome passado pelo usuário como primeiro parâmetro na linha de comando (**figura 3**).

Quadro 2: IPv6 na programação Perl

Se você deseja escrever programas voltados para o IPv6 em Perl, sua única opção é o módulo CPAN `Socket6`, que oferece funções como `getaddrinfo()`. O Perl 5.14 (maio de 2011) foi a primeira versão a implementar a funcionalidade no módulo do `Socket` [12] no núcleo da linguagem. A **listagem 11** mostra uma exemplo de aplicativo nesse sentido.

A programação de Socket é mais conveniente com o `IO::Socket` [13]. O `IO::Socket::INET`, necessário para o IPv4, já faz parte do núcleo do Perl há muitos anos. O `IO::Socket::INET6` do CPAN fornece um duplo módulo para o IPv6.

```
use IO::Socket::INET6;
my $sock6 = IO::Socket::INET6->new( '[::1]:12345' );
my $sock4 = IO::Socket::INET6->new( '127.0.0.1:12345' );
```

O módulo `Socket6` é compatível retroativamente com o `IO:Socket::INET` e também pode criar conexões IPv4. O `Socket6` aceita endereços como nomes de host na notação IPv4 ou IPv6 porque o `IO::Socket::INET` suporta programação mais simplificada e sockets no estilo Libc. Muitos programas e módulos Perl o usam, com o sacrifício de suas capacidades IPv6 (por exemplo, módulos do núcleo, tais como o `Net::SMTP`, `Net::FTP` e módulos importantes do CPAN, tais como `LWP`).

Em muitos casos, é possível readequar programas ao IPv6 com a ajuda do `Net::INET6Glue::INET_is_INET6` [14], que substitui o `IO::Socket::INET` por `IO::Socket::INET6`:

```
use Net::INET6Glue::INET_is_INET6;
use LWP::Simple;
print get('http://ipv6.google.com/');
```

A seguinte linha de comando habilita ao IPv6 um programa já existente:

```
$ perl -Mnet::INET6Glue::INET_is_INET6 ipv4_programm.pl
```

`Net::INET6Glue::FTP` ainda estende o `Net::FTP`, adicionando os comandos essenciais `EPRT` e `EPSV` para IPv6. O `IO::Socket::SSL` fornece suporte simples ao SSL e, automaticamente, ao IPv6, desde que o `IO::Socket::INET6` esteja instalado.

Além disso, o Perl oferece diversas bibliotecas para manipulação não-bloqueável de sockets. Em muitos casos, essas soluções já suportam o IPv6, mas não usam o bloqueável `getaddrinfo()` para consultas de endereços, preferindo os seus próprios. Os exemplos incluem `AnyEvent` e POE. Contudo, o gerenciador deverá retornar os resultados em ordem diferente da que você poderia esperar com o `getaddrinfo()`.

O `QHostAddress` [7] é usado para saída: essa é a abstração Qt para endereços IPv4 e Ipv6. Entre outras coisas, ela fornece o método `QHostAddress::toString()`.

A aplicação `QCoreApplication` dá aos desenvolvedores a opção de criar programas Qt sem uma GUI (**listagem 8**, **linha 10**). Após `connectToHost()` (**linha 16**), que imediatamente retorna as chamadas de código de `waitForConnected()` para esperar por uma conexão (**linha 18**). Uma outra alternativa é atribuir o sinal `connected()` a um slot apropriado. Uma vez estabelecida a conexão, o programa exibe o `peerAddress`.

Em um ambiente com suporte a IPv6, é possível ver que o programa estabelece automaticamente a conexão IPv6.

```
$ ./simpleClient www.google.com
Connected to
➥ 2A00:1450:4001:C01:0:0:0:68%0
```

Isso é prático, mas não lhe dá a possibilidade de intervenção que o `getaddrinfo()` dá aos programadores C. A função que chega mais perto de fazê-lo é `QHostInfo` [8], que oferece métodos para resolução de nomes. Infelizmente, `QHostInfo` não oferece toda a conveniência que `getaddrinfo` dá. Por exemplo, é impossível especificar quando você quer endereços IPv4 ou IPv6 e o desenvolvedor não pode especificar `flags`, tais como `AI_ADDRCONFIG`.

A implementação é simples: o método estático, `QHostInfo::fromName(const QString& name)` retorna uma instância `QHostInfo`, que usa `QHostInfo::addresses()` para retornar uma `QList` de instâncias `QHostaddress`. Uma variante do nome `lookupHost()` trabalha em outra direção, usando threads separadas para lidar com requisições feitas, as quais são chamadas dentro do slot especificado do objeto.

As aplicações de servidor não são automaticamente compatíveis com o

IPv6 no Qt. O `QTcpServer` que usa abstração para o socket servidor apenas escuta um endereço. Infelizmente, o argumento padrão é um endereço IPv4 `QHostAddress::Any` dentro do método `listen()`.

Listagem 10: server.cpp

```
01 #include <QtCore/QCoreApplication>
02 #include <QtCore/QSignalMapper>
03 #include <QtCore/QStringList>
04 #include <QtNetwork/QNetworkInterface>
05 #include <QtNetwork/QHostAddress>
06 #include <QtNetwork/QTcpServer>
07 #include <iostream>
08 #include "greeter.h"
09
10 int main(int argc, char *argv[])
11 {
12   QCoreApplication app(argc, argv);
13   int port = app.arguments().at(1).toInt();
14
15   Greeter* greeter = new Greeter(&app);
16
17   QSignalMapper* sigMap;
18   sigMap = new QSignalMapper(&app);
19   greeter->connect(sigMap,
20            SIGNAL(mapped(QObject *)),
21            SLOT(newConnection(QObject *)));
22
23   QList<QTcpServer> servers;
24
25   QList<QNetworkInterface> ifs;
26   ifs = QNetworkInterface::allInterfaces();
27
28   foreach(const QNetworkInterface& i, ifs) {
29     QList<QNetworkAddressEntry> entries;
30     entries = i.addressEntries();
31
32     foreach(const QNetworkAddressEntry& entry, entries) {
33       QHostAddress address = entry.ip();
34
35       // fix scope of link-local addresses
36       Q_IPV6ADDR addr6;// = address.toIPv6Address();
37       addr6 = address.toIPv6Address();
38       if (addr6[0] == 0xfe &&
39         addr6[1] == 0x80) {
40           QString name=i.humanReadableName();
41           address.setScopeId(name);
42         }
43
44       QTcpServer* server;
45       server = new QTcpServer(&app);
46       sigMap->setMapping(server, server);
47       sigMap->connect(server,
48   SIGNAL(newConnection()),
49  SLOT(map()));
50
51       server->listen(address, port);
52       if (!server->isListening()) {
53         std::cout << "Cannot listen on "
54             << address.toString().toAscii().constData() << std::endl;
55       }
56     }
57   }
58
59 return app.exec();
60 }
```

Para contornar essa situação, os desenvolvedores podem especificar o `QHostAdress::AnyIPv6` IPv6 para o método `listen()`. Essa solução vai funcionar em qualquer plataforma e distribuição onde a opção padrão do socket `IPV6_V6ONLY` for `0`. O programa resultante recebe conexões IPv4 via endereços mapeados em IPv4. No entanto, não há uma maneira simples de dizer ao `QTcpServer` para usar a opção socket `IPV6_V6ONLY`. Uma solução é injetar um descritor socket criado em C através do `QTcpServer::setSocketDescriptor`.

Os programadores encontrarão uma solução alternativa ao mergulhar na biblioteca `QtNetwork`. Se você o fizer, encontrará a classe `QNetworkInterface`, que enumera todos os dispositivos da rede. Essa classe cria uma lista de instâncias `QNetworkAddressEntry` para cada dispositivo e cada um tem um `QHostAdress`. Assim é possível descobrir todos os endereços IPv4 e IPv6 existentes no sistema.

O desenvolvedor tem, primeiro, a opção de abrir uma porta no servidor para certas placas de rede

Listagem 11: Socket Perl

```
01 #!/usr/bin/perl
02
03 use Socket6; # ab 5.14
reicht 'use Socket'
04 my @res = getaddrinfo('ipv6.
google.com','http',AF_
UNSPEC,SOCK_STREAM );
05 my $sock;
06 while (! $sock and @res) {
07     my ($fam,$type,$proto,$s
addr,$cname) = splice(@
res,0,5);
08
socket($sock,$fam,$type,$proto)
or die $!;
09     connect($sock,$saddr)
and last;
10     undef $sock;
11 }
12 $sock or die $!;
```

apenas. O código para servidores QT (**listagens 9** e **10**) interage com todos os dispositivos e endereços da rede (**linha 29** na **listagem 10**). Você pode simplesmente usar `QNetworkInterface::allAddresses()` [9] para isso, mas infelizmente ele vai retornar 40 endereços locais de links.

Para cada endereço de rede, o programa cria um novo `QTcpServer` (**linha 46**, **listagem 10**) e o conecta usando um `QSignalMapper`, o que é necessário porque o `QTcpServer` apenas envia um sinal `newConnection()` vazio que não informa ao receptor qual dos muitos remetentes tem uma nova conexão. O `QSignalMapper` avalia um sinal com a informação do seu remetente, de forma que o programa possa dizer ao visitante qual IP eles estão usando.

Conclusão

Se você sabe o que está procurando, não será difícil fazer um programa compatível com o IPv6. Ainda que os programas clientes sejam razoavelmente simples, os programas servidores têm maior complexidade caso o desenvolvedor deseje abordar quaisquer eventualidades.

Os frameworks com alto nível de abstração podem virar um problema caso não tenham funcionalidade crítica em sua camada de abstração, tal como `IPV6_V6ONLY` dentro do kit de ferramentas QT. ■

Gostou do artigo?

Queremos ouvir sua opinião.
Fale conosco em
cartas@linuxmagazine.com.br

Este artigo no nosso site:
http://lnm.com.br/article/6106

Mais informações

[1] RFC 3493: http://tools.ietf.org/html/rfc3493

[2] getaddrinfo() em Python: http://docs.python.org / library/socket.html#socket.getaddrinfo

[3] Endereços IPv4-Mapped IPv5: http://www.tcpipguide.com/free/t_IPv6IPv4AddressEmbedding-2.htm

[4] Aspectos da segurança com endereços V4MAPPED: http://tools.ietf.org/html/draft-itojun-v6ops-v4mapped-harmful-02

[5] select() em Python: http://docs.python.org/library/select.html#select.select

[6] Biblioteca QtNetwork: http://doc.qt.nokia.com/4.6/network-programming.html

[7] QHostAddress: http://doc.qt.nokia.com/4.6/qhostaddress.html

[8] HostInfo: http://doc.qt.nokia.com/4.6/qhostinfo.html

[9] QNetworkInterface: http://doc.qt.nokia.com/4.6/qnetworkinterface.html

[10] Glibc bug: http://sourceware.org/bugzilla/show_bug.cgi?id=12377

[11] Bug em Ubuntu Eglibc: https://bugs.launchpad.net/ubuntu/+source/eglibc/+bug/762512

[12] Módulo Perl socket: http://perldoc.perl.org/Socket.html

[13] IO::Socket::INET: http:// perldoc.perl.org/IO/Socket/INET.html

[14] Net::INET6Glue : http://search.cpan.org/~sullr/Net-INET6Glue-0.5/

[15] Listagens do artigo, incluindo os arquivos para o Qmake: [LINK LINK LINK LINK LINK LINK LINK]