UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Elvis Pfützenreuter

Migração para IPv6 de aplicações usuárias da interface de programação Sockets BSD

Monografia submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Especialista em Ciência da Computação.

Luis Fernando Friedrich

Joinville, março de 2003.

Migração para IPv6 de aplicações usuárias da interface de programação Sockets BSD

Elvis Pfützenreuter

Esta Monografia foi julgada adequada para a obtenção do título de Especialista em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____
VITTORIO BRUNO MAZZOLA

Banca Examinadora

_____
LUIS FERNANDO FRIEDRICH

_____

_____

Dedicado a Richard W. Stevens (*in memoriam*) cujos livros são a grande fonte de informação e inspiração dos desenvolvedores Unix.

SUMÁRIO

RESUMO

Este trabalho relaciona, discute e exemplifica as principais extensões da interface de programação de aplicativos *Sockets BSD* para suporte ao protocolo de rede IPv6, com o objetivo de demonstrar a relativa facilidade de conversão para IPv6 de aplicativos baseados em IPv4. Também são relacionadas as diferenças entre os protocolos IPv4 e IPv6 que afetam diretamente o desenvolvimento de aplicativos.

ABSTRACT

This paper lists, discusses and exemplifies the IPv6 protocol support extensions of *Sockets BSD* application programming interface, aiming to demonstrate the relative easiness of application conversion from IPv4 to IPv6. Also, this paper shows the differences between protocols IPv4 and IPv6 that affect application development directly.

OBJETIVO DA MONOGRAFIA

O objetivo deste trabalho é demonstrar o uso da interface de programação Sockets BSD com o protocolo de rede IPv6, bem como as implicações da conversão de programas usuários desta interface para IPv6.

A interface de programação de aplicações Sockets BSD é o padrão de fato no acesso a recursos de rede. Essa interface foi desenvolvida na universidade de Berkeley como parte da implementação-modelo do TCP/IP para Unix.

A interface prevê a fácil inclusão de suporte a novos protocolos. Isso afeta ligeiramente a curva de aprendizado da interface, mas a longo prazo a extensibilidade revelou-se crucial para o sucesso do Sockets BSD e do próprio Unix. Os Sockets fazem parte da especificação POSIX, que todos as implementações Unix relevantes seguem; e todos os protocolos de rede suportados pelo sistema operacional subjacente são acessíveis via Sockets, seja IPX, Appletalk, NetBEUI, OSI, IP ou IPv6.

Outros sistemas operacionais não-Unix imitam em maior ou menor grau a interface Sockets BSD, por ser simples, eficiente, familiar aos desenvolvedores, e poupar o trabalho de criar-se uma nova interface. Um exemplo notório é o PalmOS, que oferece compatibilidade limitada do código-fonte C. O desenvolvedor pode testar os módulos de *networking* num computador de uso geral e depois recompilá-lo para Palm com pouca ou nenhuma modificação (RHODES, 1999).

Ao migrar um programa qualquer de um protocolo de rede para outro, e.g. de TCP/IP para IPX, pouca coisa muda no código-fonte. Esse é o objetivo do Sockets BSD. Mas, não basta ao desenvolvedor conhecer duas ou três macros `IPX_*` para usar o protocolo IPX; ele deve conhecer o novo protocolo um pouco mais profundamente para usá-lo de forma adequada. Afinal, tem de existir uma infra-estrutura de rede IPX funcional para que os soquetes sejam úteis.

O mesmo acontece com o IPv6. É fácil migrar um programa IPv4 para IPv6, conforme será demonstrado neste trabalho. Porém, é necessário saber algo mais sobre as

mudanças de protocolo interagem com os programas:

- Os programas poderiam continuar usando o protocolo IPv4 ? Ao mesmo tempo que IPv6?
- Um programa IPv6 pode comunicar-se com um servidor IPv4 ? Se sim, como fazer isso ?
- Um programa oferece determinado serviço à rede. Esse serviço ainda faz sentido em IPv6?
- Como um programa que se utiliza de um serviço IPv4 não mais existente em IPv6 (e.g. broadcast) poderia ainda ser utilizado ?

Para atingir o objetivo de responder tais perguntas, este trabalho aborda alguns aspectos do protocolo IPv6 em si, que apesar de estarem muito bem documentados em diversos livros e trabalhos, por vezes se utilizam de linguagem hermética e/ou abordam aspectos muito específicos do protocolo. Além disso, a suprema maioria desses materiais não aborda programação em Sockets BSD.

Após trazer informação suficiente sobre IPv6 para o interesse deste trabalho, aborda-se o desenvolvimento de programas em si, sempre tentando dar um enfoque prático, utilizando código real, prontamente utilizável e demonstrável.

O sistema operacional de referência para o estudo é o Linux, por ser o mais familiar ao autor. De qualquer forma, os exemplos devem ser 100% compatíveis com outros sistemas que seguem o padrão POSIX.

A interface de rede do Microsoft Windows difere consideravelmente do Sockets BSD (embora imite-a claramente em alguns aspectos), portanto infelizmente nada será falado sobre programação IPv6 para Windows. O produto CygWin, que permite rodar programas POSIX sob Windows, promete implementar suporte a IPv6 num futuro próximo.

1. IPv6: principais mudanças em relação a IPv4

As mudanças descritas neste capítulo estão baseadas nos trabalhos de HINDEN & DEERING (1998), MILLER (2000) e HAGEN (2002).

1.1. Espaço de endereçamento

O espaço de endereçamento do IPv6 é de 128 bits, contra os 32 bits do IPv4.

Esta é a mudança mais visível do IPv6 em relação ao IPv4. Algumas das primeiras propostas de evolução do IPv4 – vide CALLON (1992), PISTICELLO (1993) e BRADNER & MANKIN (1993) - propunham espaços de endereçamento de 64 ou 96 bits, perfeitamente suficientes para um prazo razoavelmente longo.

A proposta mais interessante, denominada TUBA (TCP and UDP with Bigger Addresses) propunha a substituição do IP pelo CNLP da pilha OSI. O CNLP é bem documentado e tem um espaço de endereçamento de até 20 octetos (160 bits). A indisposição generalizada da comunidade Internet com o protocolo OSI, constatada no trabalho de DIXON (1993), acabou sepultando a idéia. Textos a favor e contra o TUBA e o OSI podem ser encontrados facilmente na Internet.

O endereçamento finalmente adotado visa, principalmente:

a) abrir espaço à criação de tantas classes de endereços quantas forem necessárias, e ainda ter espaço de sobra para um número virtualmente inesgotável de endereços dentro de cada classe;

b) utilização massiva de roteamento por agregação, onde todas as sub-redes de uma mesma rede apresentam o mesmo prefixo de rede. Isto diminui drasticamente o número de rotas que cada roteador tem de conhecer, em todos os níveis.

Embora o roteamento por agregação seja padrão para IPv4 desde 1995 com a implementação da CIDR (FULLER, 1993), nem todas as redes classe A, B ou C podem

ser renumeradas, e os roteadores da espinha dorsal da Internet têm de conhecer rotas específicas para inúmeras redes não agregadas.

O tamanho do endereço IPv6 comporta tanto profundas hierarquias de endereçamento por agregação bem como um grande número de nós por sub-rede. Isso permite:

a) liberal distribuição de faixas de endereçamento a usuários finais, tornando desnecessários, por exemplo, os complexos roteadores NAT (*Network Address Translation* – tradução de endereço de rede) para compartilhamento de um IP por vários usuários. O IPv6 acaba com os cidadãos de segunda classe da Internet;

b) Com o desuso do NAT, ocorre uma grande simplificação na configuração de servidores e dispositivos de rede, o que contribui para o barateamento do acesso à Internet. Evita todos os problemas citados por PEÑA (2001) e permite floresçam protocolos mais sofisticados e.g. voz sobre IP.

Nada impede de um sistema operacional ou dispositivo de rede implementar NAT para IPv6, e de fato é implementado no Linux. Alguns administradores de rede têm a sensação subjetiva de que NAT aumenta a segurança, embora isso seja muito discutível.

1.1.1. Notação de endereços IPv6

Devido a seu tamanho, os endereços IPv6 são textualizados em hexadecimal, 8 palavras de 16 bits cada. Exemplo:

```
2002:0000:0000:0015:0000:0000:0000:0001
```

Algumas simplificações são permitidas, de modo a encurtar graficamente o endereço IPv6:

a) Seqüências de palavras `0000` podem ser omitidas; a posição da seqüência é

marcada pela cadeia "`::`". Exemplos equivalentes:

```
2002::0015:0000:0000:0000:0001
2002:0000:0000:0015::0001
```

b) Apenas uma seqüência de zeros pode ser contraída, pois se contraíssemos mais de uma, o endereço tornar-se-ia ambíguo:

`2002::0015::0001` (inválido, qual a posição da palavra `0015` dentro do endereço?)

c) Os zeros não significantes dentro de cada palavra podem ser omitidos:

```
2002::15:0:0:0:1
```

Os endereços IPv6 compatíveis com IPv4, que serão explanados mais adiante, admitem uma sintaxe parcialmente compatível com IPv4:

```
xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:a.b.c.d
```

onde (`a.b.c.d`) é o endereço IPv4 expresso na tradicional notação decimal pontuada, que ocupa 32 bits. Os demais 96 bits de tais endereços são expressos em hexadecimal, da forma usual IPv6, com as mesmas possibilidades de simplificação.

1.1.2. Máscara de rede

Não existem classes como A, B e C. O IPv6 utiliza o conceito de CIDR (FULLER, 1993), onde um determinado número de bits corresponde ao prefixo da rede, e os bits restantes identificam o nó.

Exemplo:
`FFFF:FFFF:FFFF:FFFF:0000:0000:0000:0000`
ou

```
FFFF:FFFF:FFFF:FFFF::
```

expressa uma máscara de rede de 64 bits.

As máscaras de rede IP na notação acima são pouco práticas, e aparecem apenas em textos didáticos. A notação usual é a "notação de barra", com o número de bits 1 da máscara sufixando o endereço IP. Exemplo:

```
2002:1/64
```

1.2. Simplificações do protocolo

O aumento do tamanho do endereço IP faz o cabeçalho do pacote crescer, o que aumenta o consumo de largura de banda, dado o mesmo *payload* (conteúdo útil do pacote). Também o processamento desses pacotes seria, em princípio, um pouco mais pesado para os roteadores.

Para mitigar o aumento de *overhead*, a camada de rede IPv6 e o respectivo cabeçalho foram simplificados:

a) Pacotes IPv6 não podem ser fragmentados por roteadores intermediários. Fragmentação é um grande complicador da pilha IPv4, bem como fonte abundante de problemas de implementação e brechas de segurança.

Ao receber um pacote IPv6 grande demais para ser repassado, o roteador deve descartá-lo e notificar o remetente via ICMPv6. É o mecanismo de *Path MTU discovery*, opcional em IPv4 e *mandatório* em IPv6.

O nó pode remeter pacotes fragmentados, valendo-se do cabeçalho opcional de fragmentação, mas normalmente não o fará pois já conhece o MTU correto, via *Path MTU discovery*.

b) IPv6 exige um protocolo de enlace com MTU mínimo de 1280 bytes; é

recomendado um MTU mínimo de 1500 bytes, padrão da Ethernet e da maioria das redes ligadas à Internet. Isso também ajuda a diminuir o uso de pacotes fragmentados.

c) Não há *checksum* no cabeçalho IPv6, por dois motivos:

c.1) praticamente todos os protocolos de enlace em uso corrente são muito confiáveis e implementam por conta própria checagem mais robusta (e.g. CRC da Ethernet);

c.2) A maioria dos protocolos de quarta camada (TCP, UDP, ICMPv6) têm um campo de *checksum*. Em IPv6, esse *checksum* inclui e protege também os endereços do cabeçalho de terceira camada, bem como os números de porta presentes no cabeçalho de quarta camada.

Esta mudança é extremamente bem-vinda por diminuir o tempo de processamento nos roteadores bem como a latência. Além disso, isenta o roteador de recalcular o *checksum* ao alterar algum campo variável do cabeçalho IP.

d) O cabeçalho IPv6 tem tamanho fixo, o mínimo necessário de campos para o uso regular;

e) Dados adicionais de camada de rede, quando necessários, são alocados em cabeçalhos opcionais, cada um com o tamanho necessário à sua tarefa;

f) Cada cabeçalho opcional tem um *flag hop-by-hop*, que sinaliza se os roteadores intermediários devem analisar o cabeçalho. Também colabora para diminuir a latência, pois a maioria dos cabeçalhos opcionais interessa apenas ao destinatário;

g) Os dados de todos os cabeçalhos são alinhados em 64 bits, que agiliza o processamento em computadores de 32 e 64 bits.

Estas simplificações permitem tornar a implementação do IPv6 no mínimo tão rápida quanto IPv4, e também facilitam sua implementação em hardware Pilha de rede

em hardware ou assistida por hardware é comum em roteadores de alto desempenho, já é oferecida em algumas placas de rede e deve ser algo comum no futuro mesmo em computadores de uso geral.

1.3. Mudanças estruturais

O IPv6, além da mudança do tamanho de endereço, possui algumas mudanças estruturais em relação ao IPv4.

1.3.1. Sem broadcast

Não existe broadcast em IPv6. As tarefas antes exercidas pelo broadcast são despachadas via multicast.

1.3.2. Uso extensivo de multicast

A implementação de multicast é obrigatória nas pilhas IPv6, e é utilizado em diversas tarefas de sistema, em particular aquelas onde o IPv4 emprega broadcast.

O multicast é inerentemente mais poderoso que o broadcast, mas boa parte desse poder depende de que os roteadores da Internet implementem os protocolos de roteamento de multicast. Especificamente em substituição ao broadcast na rede local, as vantagens do multicast são:

a) Filtragem por hardware na própria placa de rede. Apenas as máquinas interessadas em um grupo multicast arcam com os custos de processamento;

A rigor, esta filtragem é imperfeita, pois apenas os 24 bits finais do grupo multicast são mapeados no endereço de enlace. Além disso, cada placa de rede pode filtrar um número máximo de grupos multicast; ultrapassado esse limite, a filtragem é inteiramente delegada ao software. (Felizmente, as boas placas de rede costumam

suportar um número de grupos muito superior à demanda atual.)

b) Possibilidade de criar inúmeros grupos de multicast, enquanto broadcast é apenas um.

Em *hubs* e *switches* de baixo custo, pacotes multicast são efetivamente transmitidos via broadcast, portanto substituir broadcast por multicast pode não trazer vantagens na primeira camada. Alguns switches mais elaborados monitoram as mensagens IGMP e transmitem pacotes multicast apenas a quem interessa.

1.3.3. Autoconfiguração extensiva

Toda interface IPv6 tem um endereço definido de forma automática, mesmo que o nó esteja completamente isolado, ou ligado a uma rede local isolada.

O endereço automático, denominado *stateless*, sempre possui o prefixo `FE80::/10`. Devido a isto, ele é válido apenas na rede local, e não é roteável. Nesse último aspecto, ele é análogo a um endereço `10.0.0.0/8` do IPv4.

A garantia da existência deste endereço simplifica consideravelmente protocolos como o DHCPv6 e evita o uso de broadcast. Lembrar que, em IPv4, uma máquina em processo de configuração via DHCP ainda não tem endereço IP e toda a comunicação tem de ser feita por broadcast.

Uma interface IPv6 pode responder por múltiplos endereços - as implementações são obrigadas a suportar essa multiplicidade. O endereço *stateless* tipicamente será acumulado com outros definidos pelo administrador da rede ou atribuídos via DHCPv6.

Em IPv4, uma interface de rede pode ter mais de um IP (artifício conhecido como *IP alias*) porém o suporte a *IP alias* é opcional, e em algumas implementações seu uso causa problemas com a resolução ARP.

1.3.3.1 Autoconfiguração de prefixo de rede

Todo roteador IPv6 deve juntar-se a determinados grupos multicast e noticiar o prefixo de rede.

Assim, os nós IPv6 podem obter o prefixo de rede e realizar a auto-configuração de endereços roteáveis na Internet. Se o prefixo de rede for mudado, apenas o roteador precisa ser reconfigurado; os demais nós vão conhecer a mudança dinamicamente.

1.3.4. Uso de endereço de segunda camada como sufixo de endereço IPv6

O tópico 1.3.3 não esclareceu de onde vem o sufixo dos endereços IPv6 automáticos. Esse sufixo está no padrão EUI-64, descrito por CRAWFORD (1998) e por HINDEN & DEERING (1998), e que contém várias informações importantes sobre o endereço:

a) fabricante do equipamento de rede (bits 0 a 5 MSB), tipicamente baseado no endereço de enlace;

b) se o código EUI-64 foi gerado localmente ou é baseado no endereço de enlace (bit 6 MSB);

c) se o endereço representa um endereço unicast ou multicast (bit 7 MSB);

d) identificador unívoco (demais bits), tipicamente baseado no endereço de enlace. O mapeamento do endereço de enlace para este identificador é ligeiramente diferente para cada arquitetura de enlace.

Por exemplo, o endereço *stateless* é calculado assim:

- prefixo `FE80::/10`
- sufixo de 64 bits EUI-64 baseado no endereço de enlace

Esta solução já foi adotada com sucesso por outros protocolos de rede, notoriamente o IPX. Ela funciona porque o endereço de enlace é tipicamente unívoco.

Em Ethernet e FDDI, a unicidade do endereço de enlace é garantida pelo instituto IEEE que fornece faixas de endereçamento aos diversos fabricantes, que por sua vez atribuem um endereço não reaproveitável a cada placa de rede.

Algumas arquiteturas de enlace não garantem unicidade. Em particular o ARCNet tem um endereço de 8 bits, fixado pelo administrador, suficiente apenas para distinguir o nó num segmento de rede. Nelas, a parte final do código EUI-64 é gerada aleatoriamente.

Outras arquiteturas, como ATM e Frame Relay, além de possuirem endereços de segunda camada pequenos, não suportam broadcast nem multicast. Para estes, deve haver um servidor ARP responsável pela tradução de endereços.

Algumas placas de rede permitem que seja mudado o endereço de enlace, e nada impede que alguém atribua o mesmo endereço a duas ou mais placas. Isto abre a possibilidade de coincidência ou “colisão” de endereços IPv6. A palavra "colisão" transmite melhor a idéia de que endereços coincidentes vão interferir e falhar ao comunicar-se com outros nós da rede.

Para evitar os dissabores de uma eventual colisão, o IPv6 realiza uma busca automática do endereço na rede local antes de adotá-lo. No improvável caso de outro nó estar usando o mesmo endereço, ocorre o *fallback* para a geração aleatória.

1.3.4.1 Sufixo de endereços IPv6 públicos

Também no caso dos endereços roteáveis, o sufixo do endereço pode ser preenchido com base no endereço de enlace.

Como vimos, os nós podem obter automaticamente o prefixo de rede do roteador, via multicast. (O roteador pode ele mesmo ter obtido o prefixo de forma automática junto ao provedor de acesso.) Portanto, os nós podem obter seu endereço completo (prefixo+sufixo) por meios dinâmicos.

Isso traz quatro problemas, todos solucionáveis:

a) O registro DNS referente ao nó teria de ser atualizado se/quando sua placa de rede fosse substituída, o que não é nada prático;

b) Um nó que ofereça serviços à Internet pública tornar-se-ia inacessível quando tiver a placa de rede substituída, mesmo que o registro DNS fosse consertado, pois essa mudança demoraria a propagar-se pelos servidores DNS;

c) Os endereços assim criados são difíceis de memorizar e administrar, e isso acabaria encorajando os administradores a modificar endereços de enlace, o que é totalmente indesejável;

d) Como os endereços de enlace são unícos em todo o mundo e rotulam o *hardware*, os endereços IPv6 baseados neles trazem um problema de privacidade – seria possível rastrear a localização e os hábitos de computadores e dispositivos portáteis.

As respectivas soluções são:

a) Em IPv6, a utilização de DNS dinâmico é, na prática, indispensável;

b) Nós acessíveis a partir da Internet pública (e.g. servidores Web) podem ter endereços IP adicionais, estáveis, não atrelados à placa de rede, atribuídos manualmente pelo administrador;

c) Tais endereços criados manualmente podem usar sufixos simples com 0, 1, 2 ... o que torna-os tão simples de memorizar e administrar quanto endereços IPv4;

d) Nós e aplicações que exijam precaução redobrada com privacidade podem gerar um sufixo de 64 bits aleatório ao invés de utilizar o endereço de enlace. Tal geração tem baixa probabilidade de colisão e de qualquer forma o protocolo IPv6 fiscaliza a ocorrência de colisões, conforme descrito em 1.3.4.

A chance de colisão é em torno de $2^{-0.5n}$, onde *n* é o número de bits do número aleatório. Para um número aleatório de 64 bits, a chance de colisão é 2^{-32}. Essa chance aparentemente alta de colisão é conhecida como Paradoxo do Aniversário: dadas 23 pessoas, existe uma chance de 50% de haver duas com a mesma data de aniversário. Referências em HOWSTUFFWORKS e BURTLE.

1.3.5 Faixas de endereçamento

Conforme foi dito em 1.1, o grande espaço de endereçamento visa a criação facilitada de classes de endereçamento. Tais classes, mais apropriadamente denominadas de *faixas de endereçamento*, são registradas junto à IETF.

Segue uma lista das principais faixas e os respectivos prefixos IPv6.

`0000::/8`	Reservado
`0000::/96`	Endereços IPv6 compatíveis com IPv4 (vide 2.1.2)
`::FFFF:0:0/96`	Endereços IPv4 mapeados em IPv6 (vide 2.1.1)
`0200::/8`	NSAP (obsoletado)
`0400::/8`	IPX (obsoletado)
`2000::/3`	Endereços roteáveis na Internet (prefixos `2xxx` e `3xxx`)
`FE80::/10`	Endereços da rede local (automáticos, estáticos ou *stateless*)
`FEC0::/19`	Endereços do sítio local
`FF00::/8`	Multicast

Aproximadamente 15% do espaço de endereçamento IPv6 foi alocado. Restam ainda 85%.

A presença de faixas para IPX e NSAP é resultado de uma diretiva de *design* do IPv6: permitir que outros protocolos trafeguem na Internet sem necessidade de

tunelamento explícito, bem como permitir que os aplicativos possam comunicar-se com nós não-IP de forma transparente (SKELTON, 1994; CLARK et al., 1994). Hoje, a utlização dos protocolos não-IP é tão restrita, e sem perspectiva de ressurgência, que essas faixas cairam em obsolescência.

Os endereços roteáveis na Internet têm mascara de rede de 64 bits, o que permite uma hierarquia de roteamento relativamente profunda (muito mais profunda que a proporcionada por qualquer classe IPv4) e um número virtualmente ilimitado de nós por rede.

Também existem alguns endereços especiais:

`0::0/128` ou `::/128`	Endereço não atribuído / inexistente
`0::1/128` ou `::1/128`	*Loopback*, análogo a `127.0.0.1`

1.3.6. Multicast

Formato do endereço:

```
FFFF:xyee:eeee:eeee:eeee:eeee:eeee:eeee
```

x = flags
y = escopo
e = restante do endereço

Os endereços multicast funcionam de forma muito semelhante ao IPv4, com as seguintes diferenças:

a) O espaço de endereçamento é muitíssimo maior;

b) Os octetos 5 e 6 têm propósito especial: flags e escopo, respectivamente;

c) No IPv4, o valor do campo TTL é sobrecarregado com o escopo do multicast. No IPv6, essa sobrecarga não existe; o sexto octeto do endereço de multicast é que informa o escopo.

1.3.6.1 Campo de *flags* (4 bits)

Bit	*Significado*
MSB 0,1 e 2	Reservados, devem ser 0
LSB 0	0 = endereço predefinido, registrado junto ao IANA
	1 = endereço temporário, definido por alguma aplicaçao local

1.3.6.2. Campo de escopo (4 bits)

Valor (decimal)	*Significado*
0	Reservado
1	Interface local
2	Rede local
5	Sítio local
8	Organização local
14	Global
15	Reservado

Um roteador IPv6 ligado a uma rede local nunca aceitará rotear pacotes multicast com escopo menor que 5 (sítio local); um roteador ligado à espinha dorsal da Internet não aceita multicast com escopo menor que 14, e assim por diante.

1.3.6.3. Endereços multicast predefinidos

O IANA predefine a aplicação de alguns de endereços de multicast IPv6. Exemplos:

`FF02::1` Todas as máquinas da rede local – propósito geral
`FF02::2` Todos os roteadores presentes na rede local – propósito geral

Esses dois grupos são mandatórios; todos os nós têm de se associar ao grupo `FF02::1`, e todos os roteadores têm de se associar ao grupo `FF02::2`. Por exemplo, para comunicar-se com todas as máquinas da rede local via multicast, basta usar o grupo predefinido `FF02::1`. Para procurar por um roteador, basta mandar pacotes para `FF02::2`.

Note que estes endereços têm o escopo igual a 2 (rede local). Outros exemplos de endereços predefinidos:

`FF02::1:2`	Agentes DHCP
`FF02::1:FFxx:xxxx`	Resolução de endereços (abordado em tópico próprio)

O fato de um nó estar no grupo `FF02::1` não quer dizer que ele processe todo o tráfego multicast direcionado a esse grupo. Se não houver nenhuma aplicação interessada em multicast, o tráfego é desprezado.

Os grupos citados são mandatórios porque são freqüentemente demandados; sua predefinição evita que cada aplicação utilize um grupo multicast diferente para o mesmíssimo fim.

1.3.7. Anycast

Um endereço anycast, assim como um endereço multicast, representa um agrupamento de nós de rede. Porém, é diferente em três aspectos cruciais:

a) apenas um dos nós do grupo receberá cada pacote cujo destino é um endereço anycast;

b) o prefixo de rede de um endereço anycast não é predefinido como no multicast; já o sufixo de rede é predefinido como zero;

c) o pacote anycast é roteável, segundo seu prefixo de rede; pode portanto trafegar na Internet sem necessidade de um protocolo de registro como o IGMP, nem suporte especial nos roteadores intermediários. Apenas o destinatário tem de dar tratamento diferenciado ao pacote anycast. Nesse aspecto, anycast é algo semelhante ao broadcast remoto IPv4.

O anycast é em princípio uma novidade muito promissora, embora sua natureza orientada a pacotes impeça seu uso em protocolos de transporte orientados a conexão (a mesma limitação do multicast).

Um dos primeiros usos atribuídos ao anycast foi a autoconfiguração dos nós. Posteriormente, tais funções acabaram delegadas ao multicast. O uso prático do anycast é atualmente muito restrito. Uma possível aplicação futura é o Mobile IP.

1.3.8 ICMPv6

Uma das metas do IPv6 é não forçar alterações em protocolos de camadas superiores. De fato, o protocolo TCP roda sem qualquer modificação sobre IPv6. Porém, o protocolo ICMP foi alterado, por boas razões.

O ICMP continua exercendo as funções que tinha no IPv4 (ping, MTU discovery, notificações de erros etc.), mas foi estendido para absorver funções de outros protocolos. Isso é bom pois evita a multiplicidade de protocolos, o que aumenta a coerência e torna as implementações mais enxutas.

1.3.8.1 Resolução de endereços de enlace

O protocolo ARP não é utilizado em IPv6, pois não existe broadcast em IPv6 e o ARP baseia-se em broadcast. Em seu lugar, é utilizado o protocolo ICMPv6 e

transmissão multicast.

Referimo-nos aqui ao protocolo ARP utilizado em arquiteturas de rede baseadas em difusão ou anel, nas quais é fácil e eficiente fazer a procura direta por um endereço de enlace mediante broadcast ou multicast.

Em arquiteturas hierárquicas como Frame Relay ou ATM, a procura exaustiva por um nó é inviável, nem é suportado broadcast e/ou multicast. Em tais arquiteturas, existe um servidor ARP com endereço de enlace conhecido, onde o administrador da rede cadastra os pares endereço de enlace:endereço IP. O protocolo ARP utilizado na comunicação com esse servidor, que não é abordado neste trabalho, é bem diferente do ARP tradicional (BRAZDZIUNAS, 1994).

O multicast é utilizado na procura de endereços IPv6 da seguinte forma:

a) É calculado um endereço multicast a partir de cada endereço IPv6 do nó, da seguinte forma:

- prefixo `FF02:0:0:0:0:1:FF00::/104` (citado em 1.3.6.3)
- sufixo: os últimos 24 bits do endereço IPv6

b) Todo nó IPv6 é obrigado a permanecer na escuta de todos os endereços multicast calculados por esse método.

Neste caso, não é necessário que haja uma aplicação interessada; a própria implementação (que tipicamente reside no kernel) deve interpretar o tráfego ICMP recebido por essa via.

c) Quando outro nó quiser descobrir onde está um determinado IP, emite um pacote ICMP multicast formado pelo método descrito acima.

Exemplo: se se deseja descobrir onde está o endereço `3FFF::DEAD::BEEF`, o endereço de remessa do pacote ICMP será `FF02::1:FFAD:BEEF`.

d) Os nós cujos endereços coincidam com o solicitado nos 24 bits finais receberão o pacote, e apenas o nó possuidor do endereço requisitado responderá.

Existe, é claro, a possibilidade de haver dois endereços IPv6 numa mesma rede que coincidam nos 24 bits finais. No entanto, esta chance é pequena – em torno de 2^{-12} (BURTLE) - se os sufixos IPv6 forem obtidos dos endereços de enlace ou forem aleatórios. Portanto, este método é muito mais leve que o broadcast, pois pouquíssimos nós (geralmente um) efetivamente analisam e respondem a cada solicitação.

1.3.8.2. IGMP

O protocolo IGMP também foi absorvido pelo ICMPv6. A mecânica do IGMP continua a mesma, exceto pelo tamanho dos endereços.

1.3.8.3. Detecção de colisões de endereços IPv6

Através do mesmo mecanismo descrito em 1.3.8.1, o ICMPv6 detecta colisões de endereços auto-configurados. Antes de adotar um endereço, procura-se por ele na rede; se ninguém reclamar sua posse, então pode ser usado.

Todas as implementações IPv6 são obrigadas a fiscalizar colisões desse gênero.

1.3.9. IP móvel

Há duas situações principais onde o IP móvel é útil:

a) um nó numa uma rede wireless, que troca de ponto de acesso na medida em que se movimenta;

b) um nó que é desligado de uma LAN e ligado à outra.

Nas duas situações, o nó móvel recebe um IP novo (via DHCP ou protocolo equivalente) e as conexões em andamento são perdidas. Para evitar isso, existe o IP móvel.

O funcionamento do IP móvel é relativamente simples. Ao contectar-se à uma LAN ou ponto de acesso, o nó móvel liga-se um agente de IP móvel. Toda a comunicação desse nó com a Internet é feita através do agente.

O endereço IP dos pacotes que trafegam na Internet será o do agente. O IP do nó móvel é anexado num cabeçalho opcional, denominado *"care of"* ("aos cuidados de"). Isso faculta aos nós remotos saber que estão comunicando-se com um nó IP móvel.

Em IPv4, devido à inexistência de cabeçalhos opcionais nos moldes do IPv6, o IP móvel é implementado como um túnel IP tradicional, o que torna problemáticos a instalação e o suporte. Em IPv6, a implementação é transparente aos aplicativos e à rede.

Segundo GAST(2002), é possível evitar a troca de IP numa rede *wireless* mesmo sem o uso de IP móvel. O padrão 802.12 prevê um endereço *care of* na camada de enlace, o que permite a um nó conservar o mesmo IP mesmo trocando de ponto de acesso (desde que é claro todos os pontos de acesso utilizados pertençam à mesma rede *wireless*). É uma solução interessante, complementar porém não substituta ao IP móvel.

1.3.10. Segurança

O protocolo padrão de segurança para IPv6 é o IPSEC, cuja implementação é mandatória em todos os nós IPv6.

Devido à complexidade do protocolo IPSEC, sua implementação tanto em IPv4 quanto em IPv6 ainda está em andamento ou em estado imaturo em muitas plataformas. Então, apesar da obrigatoriedade teórica, um aplicativo IPv6 deve testar a presença de

IPSEC antes de contar com ele.

Características do protocolo IPSEC:

a) projetado para utilização em IPv4 e IPv6;

b) permite garantir integridade, confidencialidade e origem dos pacotes. Soluciona praticamente todos os problemas de segurança pertinentes às camadas de rede e transporte. *IPSEC não resolve, por exemplo, o problema da não repudiação de um e-mail. Tais problemas devem ser resolvidos pelos protocolos de aplicação, como o PGP*;

c) uso de certificados digitais, forma aceita como a mais adequada para identificação segura de uma entidade;

d) previsão de um conjunto mínimo de algoritmos de criptografia, que todos os nós IPSEC têm de implementar. Os nós podem implementar outros algoritmos, e negociam entre si para determinar qual o melhor algoritmo em comum que vai ser usado na conexão.

e) opera em dois modos: transporte e túnel;

No modo transporte, os endereços IP de origem e destino do cabeçalho não são criptografados, muito embora sua integridade possa ser garantida.

Este modo é mais prático, pois dois nós IPSEC quaisquer podem conectar-se sem previsão administrativa. Por outro lado, os hábitos de conexão dos nós ainda podem ser espionados. A utilização generalizada do modo transporte aumentará muito a segurança da Internet como um todo.

Já no modo túnel, os endereços IP também são criptografados. O pacote original é inteiramente reempacotado num pacote IPSEC. Os endereços IP visíveis são os endereços das extremidades do túnel, e não dos nós que estão realmente se comunicando.

Este modo exige providências administrativas nas duas extremidades, portanto apenas as conexões especialmente previstas serão seguras. Mas o túnel impede um espião de estudar os hábitos de conexão, o que aumenta a segurança. Também é possível, no modo túnel, interligar redes com endereçamento não roteável na Internet pública e/ou sem acesso direto à Internet.

O modo transporte é utilizado primariamente na construção de VPNs.

(O FreeS/WAN, implementação IPSEC para Linux, oferece um modo de "criptografia oportunista" para o modo túnel, sem configuração e sem comunicação prévia entre as duas extremidades; porém, é uma extensão própria do FreeS/WAN e implica em cadastrar a chave pública de pelo menos um dos sítios em um servidor DNS.)

f) Por ser um padrão, o IPSEC é o único protocolo de segurança de rede disponível em todos os sistemas operacionais de mercado, e devido a isso também é o único protocolo interoperável.

Infelizmente, alguns fatores têm retardado a adoção generalizada do IPSEC:

a) protocolo um tanto complexo;

b) as implementações existentes têm consideráveis desvios e apresentam problemas de interoperabilidade;

c) o algoritmo de criptografia DES, que todo nó IPSEC deveria implementar mandatoriamente, é notoriamente fraco para os padrões atuais. Segundo referência em FREESWAN, algumas implementações recusam-se a utilizá-lo (o que viola o protocolo), enquanto outras oferecem apenas esta opção (devido às restrições de exportação de criptografia do governo americano), o que elimina de plano a interoperabilidade (discussão completa a respeito em FREESWAN (a));

d) O uso do protocolo exige algumas providências administrativas, como a obtenção de certificados digitais, cuja existência e funcionalidade são ignoradas por muitos administradores de rede;

e) A existência de outros protocolos, que por incompletos e não interoperáveis são de utilização mais fácil, retarda a adoção generalizada do IPSEC.

Espera-se que com a pressão dos usuários, as diversas implementações IPSEC melhorem qualitativamente e uniformizem o conjunto mínimo de recursos oferecidos, bem como possibilitem criptografia automática sem qualquer intervenção administrativa.

1.3.11. Tráfego multimídia

O cabeçalho IPv6 possui campos de fluxo e prioridade, com uso potencial para aplicações que exijam fluxo com alguma garantia de qualidade (QoS). IPv4 possui apenas *flags* de prioridade.

A experiência com IPv4 revela, porém, que QoS na Internet é um problema por resolver, e a utilização desses campos é, hoje, quase nula. O principal problema é que implementar garantias numa Internet pública induziria todo fluxo a requisitar garantia, mesmo quando desnecessária, e como os recursos da Internet são limitados, a grande maioria das requisições seria rejeitada, invalidando completamente a idéia.

A saída é cobrar taxas extras dos requisitantes de fluxos com garantias de serviço, para o que seria necessário estabelecer acordos de cobrança inter-provedores (pois um fluxo pode percorrer diversos provedores, e a garantia de fluxo só é efetiva se todos os provedores intermediários honram a garantia solicitada), bem como preparar os equipamentos de rede para implementar as garantias bem como registrá-las e tarifar o solicitante. Essa infra-estrutura não existe hoje.

À época de projeto do IPv6, acreditava-se que todas estas questões logo estariam resolvidas, o que justificou a inclusão dos campos de fluxo cabeçalho principal. A visão

atual é que esses campos deveriam ter sido previstos em um cabeçalho opcional.

Presumindo que o futuro resolva tais problemas, o candidato mais promissor a protocolo de reserva de qualidade é o RSVP.

Finalmente, é prudente lembrar que, embora a garantia de qualidade de serviço não exista na Internet atual, ele pode ser e de fato é utilizada numa rede ou inter-rede privada, e nesse caso os campos do cabeçalho IPv6 podem ser úteis.

2. Convivência entre IPv4 e IPv6

O mais importante objetivo de projeto do IPv6 é a convivência pacífica com IPv4. Seria irreal esperar que todos os dispositivos ligados à Internet mudassem para outro protocolo num mesmo dia e hora. A transição só pode acontecer se for gradual.

As questões de transição e convivência estão intimamente ligadas, e ainda são objeto de estudo, portanto muito já mudou nesse tópico desde o projeto original do IPv6 até agora, e possivelmente soluções melhores que as atuais vão aparecer, na medida que mais pessoas tomam contato com IPv6.

2.1.Faixas de endereçamento IPv6 especiais para convivência com IPv4

2.1.1. Endereços IPv4 mapeados em IPv6

Faixa `0:0:0:0:0:FFFF/96` ou `::FFFF:0:0/96`

Os 32 bits do endereço IPv4 são mapeados nos últimos 32 bits do endereço IPv6; os primeiros 96 bits são preenchidos com o padrão descrito na máscara.

Esses endereços, para maior comodidade, podem ser expressos na forma `::FFFF:a.b.c.d`, onde `a.b.c.d` é o endereço IPv4 na familiar notação decimal pontuada. Eles servem para:

a) um nó IPv6 comunicar-se com outros que possuam apenas a pilha IPv4;

b) um nó servidor efetuar `bind()` apenas ao endereço IPv4, ou seja, ele aceitará apenas conexões IPv4.

2.1.2. Endereços IPv6 compatíveis com IPv4

Faixa `0:0:0:0:0:0/96`, ou `::/96`

Assim como em 2.1.1, os endereços IPv4 são transcritos nos últimos 32 bits do

endereço IPv6, e podem ser expressos usando a notação decimal pontuada.

Tais endereços servem para conexões entre nós IPv6 (ambos os nós têm de ter pilha IPv6), porém através de uma rede IPv4. O pacote IPv6 é reempacotado automaticamente num pacote IPv4 e enviado. Por este motivo, essa faixa de endereços também é denominada de “tunelamento automático", pois a única providência administrativa necessária é cadastrar o endereço compatível com IPv4 num registro DNS tipo AAAA.

2.1.3. Endereços *6to4*

Faixa `2002::/112`

O endereçamento descrito no tópico 2.1.2 não é muito flexível, pois permite apenas a conexão entre máquinas IPv6 que também tenham um endereço IPv4 roteável na Internet, com o competente registro AAAA no DNS. Ele não serve, por exemplo, para fazer tunelamento automático entre nós IPv6 puros (sem endereço IPv4). Isso exclui, por exemplo, os computadores ligados à Internet por roteadores NAT.

Os endereços *6to4* foram definidos para suprir essa deficiência, e tornaram os endereços compatíveis com IPv4 relativamente obsoletos (embora ainda seja cedo para afirmar que estes últimos não serão mais usados; apenas a adoção generalizada do IPv6 mostrará qual solução será a eleita pela comunidade).

Como se pode notar, o prefixo `2002::/112` faz parte dos endereços públicos IPv6. De fato, essa faixa foi alocada para o projeto *6to4*. É importante ressaltar que *6to4* não é uma mudança no protocolo IPv6 em si.

Nada impede haja outras iniciativas análogas no futuro, com prefixos diferentes. Na verdade, existem diversos projetos de conectividade IPv6, a maioria deles baseados em tunelamento manual; ninguém é obrigado a usar *6to4*.

As vantagens do *6to4* são:

a) livre para qualquer interessado em conectar-se à rede IPv6;

b) bem documentado;

c) possível adoção da tecnologia por parte dos roteadores IPv6;

d) provisão para configuração automática via multicast IPv4, portanto é o caminho mais suave para que provedores de acesso à Internet ofereçam IPv4 e IPv6 simultaneamente sem grande esforço adicional.

O formato do endereço *6to4* é:

Primeiros 16 bits: `0x2002`
Próximos 32 bits: endereço IPv4
Próximos 32 bits: endereço da sub-rede dentro da entidade
64 bits finais: endereço do nó dentro da entidade (atribuído ou deduzido do endereço de camada de enlace)

Normalmente, o endereço da sub-rede é `0` e o endereço do nó é `1`, pois a maior parte das máquinas que se conectam à rede IPv6 mundial é isolada e ligada à Internet por uma rede ponto-a-ponto.

2.2. Convivência na rede local

Todos os protocolos de camada de enlace em uso corrente permitem convivência entre IPv4, IPv6 e outros protocolos de rede. No cabeçalho de enlace, o valor do campo identificador do protocolo de rede é diferente para IPv4 e IPv6 (os valores absolutos variam de acordo com o protocolo de enlace).

2.3. Conectando-se à rede mundial IPv6

Hoje, já é perfeitamente possível navegar na Internet em IPv6 puro.

A opção ideal sem dúvida seria um provedor capaz de trafegar IPv6 diretamente, mas apenas um punhado de provedores ao redor do mundo suporta IPv6.

A segunda opção desejável é o protocolo *6to4*, mas o suporte dos provedores a *6to4* também não é nem de longe disseminado, muito embora seja o passo intermediário "natural" em direção a uma Internet IPv6.

A solução mais viável hoje (Fevereiro/2003) é o uso de um esquema de tunelamento manual, como o projeto Freenet6 (Referência: FREENET6).

2.4. DNS

A resolução de nomes DNS para nós IPv6 funciona de forma praticamente idêntica a IPv4. Naturalmente existem algumas mudanças no *modus operandi* e na sintaxe dos arquivos de configuração, que não abordaremos em profundidade. Para os fins deste trabalho, basta dizer que:

a) os endereços IPv4 e IPv6 convivem no sistema DNS e podem conviver num mesmo domínio;

b) os endereços IPv4 são armazenados em registros A e os endereços IPv6 são armazenados em registros AAAA, conforme proposto originalmente por THOMSON & HUITEMA (1995), portanto uma consulta ao DNS tem de visar especificamente ou um ou outro tipo de registro;

c) Um mesmo nome de máquina pode ser sobrecarregado com registros A e AAAA, portanto uma máquina pode ter o mesmíssimo nome DNS na Internet IPv4 e na Internet IPv6. Isso facilita a vida dos usuários finais, que nem precisam (em tese) tomar conhecimento da mudança de protocolo de rede;

d) os endereços reversos são ambos armazenados em registros PTR. A distinção é

feita pela raiz das árvores de endereçamento reverso: `in-addr.arpa.net` para IPv4 e `ip6.int` para IPv6. Com raízes diferentes, as duas famílias de endereçamento não se confundem. Naturalmente, cada consulta PTR deve visar especificamente ou uma ou outra família;

e) Além do registro AAAA, também existe o registro A6, proposto por CRAWFORD & HUITEMA (2000). Nele, consta o sufixo do endereço IPv6, o tamanho do prefixo em bits, e um *nome* correspondente ao prefixo.

Esse nome, por sua vez, é outro registro DNS AAAA ou A6, que deve ser recursivamente consultado, até que se encontre um registro AAAA e seja possível montar o endereço IPv6 completo.

Como o prefixo de rede é representado por um nome, isso facilita muito a criação e atualização dos registros DNS, inclusive tornando-os mais claros.

Para atender a clientes de legado que não suportem ainda registros A6, BUSH (2002) propõe que o servidor DNS possa responder consultas AAAA interpretando os registros A6 de forma transparente.

f) O DNS dinâmico, que pouco é utilizado em IPv4, deve ser universalmente adotado em IPv6, devido aos endereços longos, difíceis de memorizar e eventualmente atrelados à placa de rede;

g) Como praticamente cada usuário da Internet poderá ter uma faixa de endereçamento gigantesca para si, será muito mais fácil para pequenos domínios manterem seu próprio servidor DNS. Também o velho problema da delegação do DNS reverso fica automaticamente sanado.

O BIND - servidor DNS tradicional para UNIX, suporta IPv4 e IPv6 simultaneamente na versão 9, portanto uma instalação com este servidor DNS pode adicionar suporte a IPv6 incrementalmente sem qualquer modificação em hardware ou software.

3. Usos práticos do IPv6, hoje

Muito embora HAGEN (2002) não recomende nenhum novo investimento de vulto em IPv4, a verdade é que as entidades (pessoas, empresas, instituições) via de regra utilizam a informática como meio de resolver problemas, e não como um fim em si, e não migrarão para IPv6 até que:

a) precisem acessar serviços disponíveis apenas em IPv6;

b) os endereços IPv4 esgotarem-se de todo.

Nenhuma das duas coisas deve acontecer no futuro breve. O mais provável é que os endereços IPv4 tornem-se assintoticamente mais escassos; os critérios cada vez mais rígidos de alocação impedirão seu esgotamento total.

É certo porém que, para o usuário final, tornar-se-à praticamente impossível obter um endereço IPv4 público e estático. Isso já é uma realidade, se consideramos que os endereços dinâmicos singelos atribuídos a linhas discadas e ADSL impedem o usuário de hospedar um domínio, tornando-o um cidadão de segunda classe na Internet. Outros truques que provedores usam (e.g. Redirecionamento de tráfego Web para enlaces baratos e roteadores NAT) tornam o usuário um net-cidadão de terceira classe, sem IP público, cuja percepção de Internet restringe-se ao protocolo HTTP.

Dito isso, existe algum uso prático do IPv6 nos dias de hoje?

Sim, existe. IPv6 tem características únicas que o tornam útil mesmo não houvesse perspectiva de uma Internet IPv6.

A principal delas é a auto-configuração. Como já foi dito, cada interface de um nó IPv6 tem um endereço auto-configurado, baseado no endereço de enlace. Muito embora esse endereço seja apenas local (não pode trafegar na Internet), costuma ser suficiente.

Isso pode ser útil num ambiente com dispositivos embarcados, onde não se pode

esperar haver um administrador de rede. Tanto os dispositivos como os eventuais servidores têm de ser totalmente *stateless*, o que exclui o uso de DHCP e outros protocolos afins.

Adicione-se redes *wireless* a este cenário; distribuição de enderecos IP é um problema sério em redes *wireless.* Todas as soluções (rodar DHCP em cada ponto de acesso, rodar DHCP num único servidor para toda uma área *wireless* e utilizar os recursos 802.11 para evitar a troca de IP) têm vantagens e desvantagens. Todas têm alguma necessidade de configuração.

Numa rede *wireless* IPv6, assumindo não haja divisões da rede em segmentos de terceira camada, o DHCP é completamente desnecessário. Dependendo da situação, pode-se optar mesmo por redes *ad-hoc*, sem pontos de acesso.

A auto-configuração não se restringe a uma rede local singela; pode-se utilizar os endereços *site-local* auto-configurados e atribuir o prefixo de rede de forma dinâmica, desde que haja ao menos um roteador configurado para isso. Numa aplicação isolada, pode-se despachar o dispositivo-roteador com números de rede fixos.

4. Interface Sockets BSD para programação em IPv6

Este capítulo lista as principais extensões da interface Sockets BSD para programação IPv6. São os elementos que devem ser revistos na migração de um programa de IPv4 para IPv6.

Boa parte dos elementos não sofre qualquer alteração. Exemplos: funções como `read()`, `write()`, `bind()`, `select()`, `poll()`. Eles não serão abordados neste trabalho.

4.1. Estruturas de endereçamento e soquetes

```
struct in6_addr {
        union {
                uint8_t u6_addr8[16];
                uint16_t u6_addr16[8];
                uint32_t u6_addr32[4];
        } in6_u;

#define s6_addr                 in6_u.u6_addr8
#define s6_addr16               in6_u.u6_addr16
#define s6_addr32               in6_u.u6_addr32

};
```

Esta estrutura é análoga a `in_addr` do IPv4, e contém o endereço IPv6 de 128 bits. Graças aos artifícios `union` e `#define`, é possível lidar com o endereço de diversas formas: como 16 bytes, 8 palavras ou 4 palavras duplas.

Unions e macros são artifícios próprios da linguagem C; outras linguagens como C++ permitem uma manipulação mais civilizada do endereço IPv6. Do ponto de vista do kernel, `in6_addr` é uma seqüência de 128 bits de comprimento, sem qualquer estrutura interna.

Conforme OPENGROUP, o membro `s6_addr` é obrigatoriamente definido; portanto é o mais portável.

A variável `in6addr_any` contém o endereço IPv6 "indefinido" ou nulo (completamente zerado). Esta constante é uma variável, não uma macro. O sistema encarrega-se de inicializá-la.

A variável `in6addr_loopback` contém o endereço IPv6 `::1`, local da máquina, equivalente ao endereço `127.0.0.1` do IPv4.

Também é definida a macro `IN6ADDR_ANY_INIT`, que pode ser usada para inicializar uma estrutura `in6_addr` com o endereço nulo.

Assim como em IPv4, o endereço nulo IPv6 é passado ao `bind()` em programas de servidor que desejem ouvir todas as interfaces existentes.

```
struct sockaddr_in6 {
        uint8_t          sin6_len;
        sa_family_t      sin6_family;
        in_port_t        sin6_port;
        uint32_t sin6_flowinfo;
        struct in6_addr sin6_addr;
        uint32_t sin6_scope_id;
};
```

Estrutura análoga ao `sockaddr_in` do IPv4. Há dois membros completamente novos na estrutura:

a) `sin6_flowinfo`: os primeiros 4 bits são reservados, os 4 bits seguintes são os bits de prioridade, os 24 bits finais correspondem aos bits da etiqueta de fluxo do cabeçalho IPv6.

Estes campos servem, em tese, para controle de prioridade e garantias de fluxo, e provavelmente vai interagir com protocolos de reserva de banda e.g. RSVP no futuro. Conforme 1.3.11, ainda é assunto em discussão.

b) `sin6_scope_id`: Identifica a interface pela qual deve trafegar o pacote. As

interfaces são identificadas por um número inteiro positivo; o número zero delega a escolha ao algoritmo de roteamento.

Este campo ainda não existia à época da obra de STEVENS (1998), nem tampouco é ali mencionado.

É mandatório especificar a interface se endereço de destino é *stateless* de rede local (`FE80::/10`), pois todas as redes LAN diretamente conectadas possuem esse mesmo prefixo de rede, e não há como o algoritmo de roteamento decidir por uma delas.

Interessante notar que esse campo é obrigatório para endereços locais mesmo que haja uma única interface de rede. Exemplo prático:

```
# ping6 fe80::2e0:18ff:fe1c:62b0
connect: Invalid argument

# ping6 -I eth0 fe80::2e0:18ff:fe1c:62b0
PING fe80::2e0:18ff:fe1c:62b0(fe80::2e0:18ff:fe1c:62b0) from ::1 eth0: 56 data bytes
64 bytes from fe80::2e0:18ff:fe1c:62b0: icmp_seq=1 ttl=64 time=59 usec
64 bytes from fe80::2e0:18ff:fe1c:62b0: icmp_seq=2 ttl=64 time=58 usec
```

O comando *ping6* funciona apenas quando especificamos a inteface de rede. Isto é um resultado direto da existência do campo `sin6_scope_id`.

As implementações mais antigas (sem `sin6_scope_id`) optavam pela primeira interface de rede não-local, o que via de regra funcionava (pois a maioria dos computadores têm apenas uma placa de rede) mas cria incerteza em computadores *multihomed*. É possível que alguma implementação moderna ainda siga esta regra se o campo `sin6_scope_id` for passado como zero.

Segundo USAGI, em versões futuras o utilitário *ping6* para Linux pretende suportar a sintaxe `endereço%interface`, como no exemplo:

```
# ping6 fe80::2e0:18ff:fe1c:62b0%eth0
```

Junto com o endereço *stateless*, codifica-se a interface de saída. Porém essa sintaxe não é suportada hoje, e também não há garantia que seja portável no futuro.

Segundo o documento de OPENGROUP, pode haver mais campos dependentes de plataforma nas estruturas de endereço e soquete IPv6. O programador deve sempre preencher inteiramente as estruturas com zero antes de utilizá-las, pois zerar cada campo individualmente pode deixar algum campo não-padrão com conteúdo aleatório.

Analogamente a IPv4, o campo `sin6_family` deve ser preenchido pelo programador com a constante `AF_INET6`. O campo `sin6_len`, se existir, deve ser preenchido com a constante `SIN6_LEN`.

A existência do campo `sin6_len` pode ser determinada em C pela presença das macros `HAVE_SA_LEN` ou `SIN6_LEN`. A título de exemplo, as diversas versões de BSD apresentam esse campo, porém o Linux não o apresenta. Todo programa com pretensões de portabilidade deve fazer o teste de macro, e, caso este resulte positivo, preencher o campo, como no exemplo a seguir:

```
struct sockaddr_in6 endereco;
#ifdef HAVE_SA_LEN
         endereco.sin6_len = SIN6_LEN;
#endif
```

O mesmo vale para IPv4, em relação ao campo `sin_len` e sua constante `SIN_LEN`.

O número de porta continua com 16 bits, como em IPv4, pois os protocolos TCP e UDP não mudaram.

Todos os campos apresentam os bytes em *network byte order*, ou seja, o primeiro byte (com endereço mais baixo) é o mais significativo.

A título de exemplo: na arquitetura *Intel 80386* e no *Alpha*, o primeiro byte de um inteiro é o menos significativo. Nas arquiteturas *PowerPC* e *Sun Sparc*, o primeiro byte

é o mais significativo, exatamente como na *network byte order*. Nestas últimas, as funções `htons()` etc. são inócuas.

Os campos `sin6_len` e `sin6_family` têm apenas um byte, logo a ordem dos bytes é irrelevante para eles. O padrão POSIX diz que os valores atribuídos a eles não devem ser processados pelas funções `htons()` etc.

Também é interessante observar que, ao contrário dos endereços e portas, esses valores nunca são transcritos para o pacote IP, portanto não há porque serem expressos em *network byte order*. Mesmo que esses campos venham a ter tamanho maior no futuro, devem continuar expressos na ordem de bytes nativa da máquina.

As funções para conversão de/para a ordem nativa da máquina continuam as mesmas: `htons()`, `htonl()`, `ntohs()`, `ntohl()`.

As funções para tratamento de números de 64 e 128 bits v.g. `htonll()`, `ntohll()`, `htonllll()` e `ntohllll()`, embora ainda não estejam disponíveis hoje (Fevereiro/2003) já aparecem em documentações on-line e devem materializar-se num futuro próximo.

No comando `socket()`, o primeiro parâmetro é o protocolo. A constante `AF_INET6` deve ser passada para comunicação IPv6, ao invés do tradicional `AF_INET` utilizado em IPv4.

4.2. Codificação/decodificação de endereços

As funções `inet_aton()`, `inet_ntoa()` e `inet_addr()` tratam apenas endereços IPv4 e são consideradas obsoletas.

As novas funções, que lidam tanto com IPv4 e IPv6, são:

a) `inet_pton`

```
int inet_pton(int familia, const char *origem,
              void *destino)
```

Converte um endereço em forma de texto (`origem`) para uma estrutura de endereçamento (apontada por `destino`). A `família` deve ser `AF_INET` ou `AF_INET6`, conforme o tipo de endereço desejado. A estrutura passada em `destino` deve ser compatível com a família (`in_addr` ou `in6_addr`).

Esta função retorna um valor negativo se a família for inválida, zero se o endereço for inválido, e positivo se a conversão foi bem-sucedida.

b) `inet_ntop`

```
const char* inet_ntop(int familia, const void* origem,
                      char* destino, size_t tamanho);
```

Converte uma estrutura de endereço para um endereço em forma textual. O tamanho do *buffer* de `destino` deve ser passado por `tamanho`. O parâmetro `origem` deve ser um ponteiro para uma estrutura compatível com a `familia` (`in_addr` ou `in6_addr`).

O *buffer* de destino deve no mínimo comportar o maior endereço textual IPv4 ou IPv6 possível (de acordo com a família com que está-se a lidar). Ao invés de estimar o tamanho, o programador deve alocar o *buffer* nos tamanhos `INET_ADDRSTRLEN` ou `INET6_ADDRSTRLEN`.

Esta função retorna o endereço de destino em caso de sucesso, ou `NULL` se a conversão não foi bem-sucedida.

Note que as funções descritas aceitam ponteiros `void*` como ponteiros para

estruturas de endereço, o que economiza uma coerção em ANSI C.

4.3. TCP e UDP básicos

A utilização de soquetes TCP e UDP sobre IPv6 é exatamente igual a IPv4. As constantes `SOCK_STREAM` e `SOCK_DGRAM` continuam sendo usadas como o segundo argumento de `socket()`.

As demais funções, como `bind()`, `listen()`, `connect()` etc. também continuam em uso da forma usual.

4.4. Cabeçalhos opcionais IPv6

Como foi dito, o IPv6 permite a adição de cabeçalhos adicionais em número e tamanho arbitrários. A descrição dos cabeçalhos presentemente definidos bem como as estruturas C correspondentes podem ser encontradas em STEVENS & THOMAS (1998), e as extensões de segurança podem ser encontradas em ATKINSON (1995) e ATKINSON (1995a).

Para remeter cabeçalhos adicionais, basta criá-los como dados auxiliares e remetê-los via `sendmsg()`. Por outro lado, para habilitar a recepção dos mesmos, deve-se habilitar as opções de soquete `IPV6_HOPOPTS` e `IPV6_DSTOPTS`, conforme se deseje receber cabeçalhos *hop-by-hop* e de destino, respectivamente. Tais opções também são abordadas brevemente na seção 4.11.

A mecânica de coleta de dados auxiliares é abordada na seção 4.7.

4.5. Conversão entre nomes e endereços IP

Estas funções permitem ao aplicativo determinar o endereço IP a partir de um nome e vice-versa. Têm como principal objetivo a simplicidade, e oferecer a aplicativos

comuns a funcionalidade de conversão de nomes.

O aplicativo não conhece e não controla quais bancos de dados são consultados na conversão do nome. Num sistema POSIX, os bancos típicos serão o arquivo `/etc/hosts`, o NIS e o DNS. No Windows, o WINS toma o lugar do NIS.

Em alguns sistemas POSIX, é possível ao administrador configurar quais bancos de dados serão consultados através do arquivo de configuração `/etc/nsswitch.conf`, bem como adicionar *plug-ins* de resolução de nomes.

Devido a complexidade da tarefa, a conversão de nomes é implementada fora do kernel, tipicamente como uma biblioteca.

Não há suporte a buscas mais avançadas, e.g. busca de registros MX do DNS que apenas servidores de e-mail utilizam. Tais buscas têm de ser implementadas pelo próprio aplicativo ou acessadas através de funções dependentes de plataforma.

A estrutura `hostent`, utilizada pelas funções de resolução de nomes, tem o seguinte formato:

```
struct hostent {
          char *h_name;        // nome oficial do host
          char **h_aliases;    // nomes alternativos
          int h_addrtype;      // tipo de endereço do host
          int h_length;        // comprimento do endereço do host
                               // (4 para IPv4, 16 para IPv6)
          char **h_addr_list;  // endereços do host,
                               // em forma binária
}
#define h_addr h_addr_list[0]  // primeiro endereço do host
```

Dada sua flexibilidade e inexistência de tamanhos fixos, seu formato não precisou ser alterado para suportar IPv6.

4.5.1. Conversão básica de nome para endereço

A função mais comumente usada, `gethostbyname()`, resolve apenas endereços IPv4. A função `gethostbyname2()` resolve tanto endereços IPv4 como IPv6 e deve ser utilizada em lugar da anterior por qualquer aplicativo novo.

```
struct hostent *gethostbyname2(const char *name, int af);
```

onde o segundo parâmetro `af` especifica a família de endereços desejada: `AF_INET` ou `AF_INET6`. Conforme a escolha, os registros DNS A ou AAAA serão consultados respectivamente, e a estrutura `hostent` será preenchida com o tipo de endereço desejado.

De acordo com GILLIGAN (1997), ao chamar `gethostbyname2()`, o aplicativo precisa saber de antemão (mas nem sempre tem como saber) se o endereço procurado é IPv4 ou IPv6. A procura simultânea por endereços IPv4 e IPv6 pode ser ativada por um *flag*.

Se for passado um endereço IP em forma de texto para `gethostbyname2()`, a chamada não falhará, mas simplesmente devolverá esse endereço como resultado. Portanto, o aplicativo não precisa prever explicitamente essa situação (e.g. se o usuário informa um endereço ao invés de um nome).

4.5.2. Conversão com busca simultânea a endereços IPv6 e IPv4

```
res_init();
_res.options |= RES_USE_INET6;
```

Com a ativação desse *flag*, a função `gethostbyname()`primeiro procurará por endereços IPv6; se não encontrar, procura por endereços IPv4.

Já a função `gethostbyname2()` não muda seu comportamento, mesmo com o *flag* ativado; procura apenas pela família de endereços solicitada através do segundo parâmetro.

Com o flag `RES_USE_INET6` ativado, *ambas* as funções retornam *apenas* endereços IPv6 em `hostent`; os endereços IPv4 são retornados mapeados em IPv6 (vide 2.1.1). O aplicativo deve estar preparado para isso.

4.5.3. Conversão de endereços IP para nomes

A função `gethostbyaddr()` sempre solicitou a família de endereços como argumento de chamada, portanto seu protótipo continua válido para IPv6:

```
struct hostent *gethostbyaddr(const char* src, int len, int af);
```

Segundo GILLIGAN (1997) "uma possível fonte de confusão é a manipulação de endereços IPv4 mapeados em IPv6, e endereços IPv4 compatíveis com IPv6." O mesmo documento indica as seguintes regras, avaliadas na ordem a seguir, para eliminar a confusão:

a) Se `af = AF_INET6` e `len = 16` (tamanho do endereço IPv6 na memória); e o endereço IPv6 é um endereço IPv4 mapeado ou compatível com IPv6: *faça a procura como IPv4.*

b) Se `af = AF_INET`, *faça a procura como IPv4.*

c) Se `af = AF_INET6`, *faça a procura como IPv6.*

d) Se a função retornou sucesso, e af = `AF_INET`, e o *flag* `RES_USE_INET6` estiver ativado (vide 4.5.2) então o endereço retornado em `hostent` será IPv6 (IPv4 mapeado em IPv6) e o membro `h_length` será modificado para 16.

4.5.4. Funções de resolução de nomes de protocolos e serviços

Como IPv4 e IPv6 têm as mesmas camadas de transporte, as demais funções e.g. `getservbyname()` permanecem inalteradas.

4.6. Macros de identificação de classes de endereçamento

Pelo menos as macros a seguir devem existir em um sistema compatível com soquetes IPv6. Todas têm sintaxe de função, aceitam o parâmetro `const struct in6_addr *`, e retornam um valor booleano.

`IN6_IS_ADDR_UNSPECIFIED`	Não especificado (totalmente zero)
`IN6_IS_ADDR_LOOPBACK`	Endereço `::1` (loopback)
`IN6_IS_ADDR_MULTICAST`	Endereço de multicast
`IN6_IS_ADDR_LINKLOCAL`	Endereço de rede local (*stateless*)
`IN6_IS_ADDR_SITELOCAL`	Endereço do sítio local
`IN6_IS_ADDR_V4MAPPED`	Endereço IPv4 mapeado em IPv6 (`::FFFF:0:0/96`)
`IN6_IS_ADDR_V4COMPAT`	Endereço IPv4 compatível com IPv6 (`::/96`)
`IN6_IS_ADDR_MC_NODELOCAL`	Multicast e local ao nó (simultaneamente)
`IN6_IS_ADDR_MC_LINKLOCAL`	Multicast e de rede local
`IN6_IS_ADDR_MC_SITELOCAL`	Multicast e de sítio local
`IN6_IS_ADDR_MC_ORGLOCAL`	Multicast e de organização local
`IN6_IS_ADDR_MC_GLOBAL`	Multicast global

4.7. Remessa e coleta de dados auxiliares

A interface de dados auxiliares permite remeter e coletar inúmeras informações que não cabem no paradigma padrão do Sockets BSD. Devido a sua flexibilidade, esta interface está progressivamente tomando o lugar de comandos `ioctl()` e soquetes crus.

O exemplo clássico é a obtenção do endereço de destino de uma conexão TCP ou UDP. Para a esmagadora maioria dos aplicativos, interessa apenas o endereço de origem, portanto comandos como `accept()` e `recvfrom()` apenas dão conhecimento deste último. Porém, quando o aplicativo precisa saber o endereço de destino do pacote, pode:

a) Utilizar um soquete de acesso direto à rede e analisar os pacotes de chegada por conta própria. Tem as sérias desvantagens da complexidade e de impor uma carga pesada ao computador;

b) Atrelar (`bind()`) um descritor de arquivo a cada interface de rede; se o pacote chegou por determinada interface, o endereço de destino era o endereço da interface. Esse raciocínio não funciona em interfaces que respondem por vários endereços IP (situação comum tanto em IPv4 como em IPv6).

c) Utilizar a interface de dados auxiliares, que entregará ao aplicativo apenas os dados que interessam, e apenas ao soquete interessado.

Em IPv6, essa interface tem ainda mais importância devido à presença dos cabeçalhos opcionais (brevemente abordados neste trabalho).

A mecânica da remessa e coleta de dados auxiliares é a mesma para todos os protocolos, inclusive IPv6. Revisando brevemente, o segundo parâmetro passado a `recvmsg()` e `sendmsg()` é um ponteiro para a estrutura `msghdr`:

```
struct msghdr {
        void *msg_name;
        socklen_t msg_namelen;
        struct iovec* msg_iov;
        SIze_t msg_iovlen;
        void *msg_control;
        socklen_t msg_controllen;
        int msg_flags;
};
```

O campo `msg_name` deve apontar para um *buffer* capaz de conter uma estrutura de endereço como `sockaddr_in6`; o campo `msg_namelen` deve conter o tamanho desse *buffer*.

O campo `msg_iov` deve apontar para uma matriz de estruturas `iovec` (descritas mais adiante); o campo `msg_iovlen` deve conter o número de elementos na matriz.

O campo `msg_control` deve apontar para um *buffer* que receberá uma ou mais estruturas `cmsghdr` (descrita mais adiante); o campo `msg_controllen` deve conter o tamanho desse *buffer*. Deve haver espaço suficiente para receber todos os dados auxiliares que o aplicativo solicitou via `setsockopt()`.

A estrutura *iovec* tem o seguinte formato:

```
struct iovec {
        void* iov_base;
        int iov_len;
};
```

A estrutura `iovec` contém ou conterá o *payload* do pacote IP transmitido ou recebido, tal qual os *buffers* utilizados em `read()`, `write()` etc. A grande diferença é que pode-se definir várias estruturas `iovec`, cada uma contendo uma fração do *payload*. Tipicamente, apenas um `iovec` grande é utilizado.

O valor de `iov_len` deve conter o tamanho do *buffer*. Esse valor não é alterado quando o sistema preenche o *buffer*; o programa deve obter o número de octetos recebidos pelo valor de retorno de `recvmsg()` (tal qual `read()`, `recv()` e `recvfrom()`).

Finalmente, a estrutura `cmsghdr`:

```
struct cmsghdr {
        int cmsg_len;
        int cmsg_level;
```

```
        int cmsg_type;
        char data[];
};
```

A estrutura `cmsghdr` recebe os dados auxiliares. A parte inicial fixa é seguida por um número variável de bytes, representados em C pela matriz indefinida `data[]`. Portanto, o tamanho total da estrutura é variável.

O campo `cmsg_len` contém o tamanho total da estrutura. Os campos `cmsg_level` e `cmsg_type` contém o tipo de dado auxiliar.

Adjacente a este cabeçalho fixo, estão contidos os dados auxiliares propriamente ditos, cujo tamanho e formato interno são dependentes do tipo de dado auxiliar.

Diversas estruturas `cmsghdr` podem estar encadeadas no *buffer* apontado por `msghdr.msg_control`; o aplicativo deve deduzir o início da próxima estrutura a partir do tamanho da anterior e das características de alinhamento da máquina. Para facilitar esta tarefa, as seguintes macros são definidas:

```
cmsghdr* CMSG_FIRSTHDR(msghdr *);
```

Determina a primeira estrutura `cmsghdr*` contida em `msghdr`.

```
cmsghdr* CMSG_NXTHDR(msghdr*, cmsghdr*);
```

Determina a próxima estrutura `cmsghdr*`. Retorna `NULL` se não há mais nenhuma.

```
unsigned char* CMSG_DATA(cmsghdr*);
```

Obtém o dado auxiliar contido em `cmsghdr()`, ou seja, `cmsghdr.data`.

```
unsigned int CMSG_SPACE(unsigned int);
```

Calcula o tamanho do *buffer* necessário para conter uma estrutura `cmsghdr`, levando em conta o alinhamento etc. Informa-se o tamanho do dado auxiliar passado ou esperado. Por exemplo, para obter o endereço de destino, o tamanho será `sizeof(in_addr)` ou `sizeof(in6_addr)` dependendo do protocolo).

Para calcular o tamanho total de um *buffer* que vá receber diversos dados

auxiliares, deve-se calcular assim:

```
CMSG_SPACE(sizeof(estrutura1)) + CMSG_SPACE(sizeof(estrutura2))
```

e *não* assim:

```
CMSG_SPACE(sizeof(estrutura1)+sizeof(estrutura2))
```

pois cada dado auxiliar tem de ser alinhado e completado com bytes de enchimento.

```
unsigned int CMSG_LEN(unsigned int);
```

Idem a `CMSG_SPACE()` porém retorna apenas o comprimento útil, sem os bytes de enchimento ao final.

O uso das três primeiras macros é revelado facilmente pelo seu protótipo. Quanto às duas últimas, servem para calcular o tamanho do *buffer* que receberá a estrutura `cmsghdr`, tamanho do dado auxiliar em questão.

As seguintes constantes são definidas para dados auxiliares IPV6:

cmsg_level	*cmsg_type*
IPPROTO_IPV6	IPV6_DSTOPTS
	IPV6_HOPLIMIT
	IPV6_HOPOPTS
	IPV6_NEXTHOP
	IPV6_PKTINFO
	IPV6_RECVDSTADDR (obsoleta, não suportada em Linux)
	IPV6_RECVIF (obsoleta, não suportada em Linux)
	IPV6_RTHDR

O formato interno do cabeçalho de extensão IPv6 permite identificar seu subtipo. As extensões definidas presentemente e seus códigos podem ser encontrados em STEVENS & THOMAS (1998).

4.8. Operações `ioctl()`

A chamada `ioctl()` é utilizada para operações que não cabem na metáfora abrir/ler/gravar/fechar arquivo, nem possuem uma chamada especial. (`fcntl` e `setsockopt` são exemplos de chamadas especiais, que originalmente eram supridas por `ioctl()`).

A tendência do padrão POSIX e dos sistemas operacionais é desencorajar o uso de `ioctl()` e oferecer cada vez mais chamadas especiais e expandir a interface `/proc`, porém `ioctl()` continua sendo útil - e muitas vezes as "chamadas especiais" são meras macros de acesso facilitado a `ioctl()`.

A maioria das operações `ioctl()` de rede atua apenas sobre o soquete passado como parâmetro. As operações de manipulação de interfaces e rotas, que atuam em características do sistema como um todo, usam um soquete qualquer como "trampolim", tradicionalmente UDP e da família de endereços sobre a qual se deseja atuar (e.g. família `AF_INET6` para operações IPv6).

4.8.1. Operações sobre soquetes

Estas operações podem ser usadas com qualquer protocolo de rede:

`SIOCSPGRP` configura o processo que deve receber sinais `SIGIO` e `SIGURG`.

`FIONBIO` liga/desliga o *flag* de I/O não-bloqueante.

4.8.2. ARP

Como o IPv6 não implementa o protocolo ARP, não existem as chamadas para manipulação da tabela ARP.

4.8.3. Manipulação de interfaces

Algumas operações disponíveis em IPv4 também estão disponíveis em IPv6. Conforme dito em 4.8, deve-se usar um soquete-trampolim da família `AF_INET6`.

Operação	*Comportamento em IPv6*
`SIOCGIFADDR`	Não suportada em IPv6
`SIOCSIFADDR`	Adiciona, ao invés de substituir, um endereço
`SIOCDIFADDR`	(nova) Elimina um endereço da interface
`SIOCGIFCONF`	Não suportada
outras	Não suportadas

Em IPv6, as chamadas utilizam a estrutura `in6_ifreq` ao invés de `ifreq`. A interface é identificada por um número positivo (`in6_ifreq.ifr6_ifindex`) ao invés de um nome como em IPv4 (`ifreq.ifr_name`).

Infelizmente, a chamada `SIOCGIFCONF`, que serve para obter a lista de todas as interfaces, não existe em IPv6. A tendência é oferecer esta informação via interface `/proc`.

Operações de broadcast são inválidas em IPv6 pois este não suporta broadcast.

4.8.4. Manipulação de rotas

As mesmas operações `SIOCADDRT` e `SIOCDELRT` são válidas para IPv4 e IPv6; o que define se a operação é IPv4 ou IPv6 é a família do soquete-trampolim utlizado.

Previsivelmente, as estruturas passadas via `ioctl()` são diferentes conforme a família. Ambas estão no arquivo de inclusão `<net/route.h>`.

Protocolo *Estrutura*

IPv4 `rtentry`
IPv6 `in6_rtmsg`

Um exemplo prático da manipulação de rotas via programa é o próprio utilitário *ip* do Linu (referência: IPROUTE). Segundo STEVENS (1998) e TORVALDS et al. (2002) não existe operação `ioctl()` para consulta da tabela de roteamento. A consulta tem de ser feita via interface `/proc`.

4.8.5. Conversões entre nome de interface e número de índice

A função `if_nametoindex(const char*)` executa esta tarefa. Verifique a seção 4.11.1 para mais detalhes.

4.9. Soquetes de roteamento

Soquetes de roteamento são criados com a família `AF_ROUTE`. São uma alternativa aos comandos `ioctl()` para manipulação da tabela de roteamento.

No Linux 2.4, a família `AF_ROUTE` é apresentada como sinônimo de `AF_DATALINK`. Embora esta última seja suportada pelo Linux, não são suportadas as funções de manipulação de rotas.

4.10. Sysctl

A função `sysctl()` permite acesso aos mesmos recursos da árvore `/proc/sys`, de forma mais apropriada a um programa em C, ou seja, sem necessidade de interprertar textos ASCII.

Internamente, esta função utiliza soquetes de roteamento ou netlink para comunicar-se com o kernel.

Os mesmos recursos disponíveis através dos arquivos em `/proc/sys` também são acessíveis e modificáveis via `sysctl()`. Porém, o identificador de cada recurso é uma constante inteira, e não um nome de arquivo. As constantes podem ser consultadas em `<sys/sysctl.h>` e `<linux/sysctl.h>`. Em outros sistemas operacionais, consulte `<sys/sysctl.h>`, que por sua vez inclui arquivos dependentes de plataforma.

A utilização dessa interface tem duas peculiaridades:

a) A interface `/proc` é ligeiramente diferente em cada sistema operacional, muito embora os Unixes tentem ser semelhantes nas operações acessíveis via `sysctl()`. Usar `/proc` ou `sysctl()` pode trazer problemas de portabilidade;

b) Em Linux, apenas números inteiros (e não *strings*, nem estruturas) podem ser acessados via `sysctl()`, portanto apenas parâmetros numéricos presentes em `/proc/sys` podem ser manipulados via `sysctl()`. Portanto, toda a discussão em STEVENS (1998) sobre o acesso às tabelas de roteamento via `sysctl()` é inválida para o Linux.

Esta é uma limitação e uma decisão de *design* específica do Linux. Ponderou TSO (2000) que prever manipulação de estruturas via `sysctl()` duplicaria esforços de programação sem que isso considerável vantagem aos aplicativos-usuários.

4.11. Multicast

Assim como em IPv4, o ingresso e saída de grupos multicast é feita através da função `setsockopt()`.

As macros de comando são todas análogas a IPv4:

`IPV6_ADD_MEMBERSHIP` ou `IPV6_JOIN_GROUP`

`IPV6_DROP_MEMBERSHIP` ou `IPV6_LEAVE_GROUP`

`IPV6_MULTICAST_IF`

`IPV6_MULTICAST_HOPS` (correspondente a `IP_MULTICAST_TTL`)

`IPV6_MULTICAST_LOOP`

As macros `IP*_MEMBERSHIP` são consideradas obsoletas.

A principal mudança é na forma de especificar uma interface. Em IPv4, a interface multicast é especificada através de um endereço IP. Em IPv6, a interface é especificada através de um número inteiro.

Estrutura de requisição de multicast, utilizada pelos comandos `IPV6_*_GROUP`:

```
struct ipv6_mreq {
        struct in6_addr ipv6mr_multiaddr;
        unsigned int ipv6mr_interface;
};
```

Em algumas implementações mais antigas, bem como na implementação de TORVALDS et al. (2002), o segundo campo é apresentado com o nome `ipv6mr_ifindex`.

O comando `IPV6_MULTICAST_IF` recebe um parâmetro `u_int` (número inteiro) ao invés de um endereço IP.

Os comandos `IPV6_MULTICAST_HOPS` e `IPV6_MULTICAST_LOOP` recebem, respectivamente, `int` e `u_int` como parâmetros, ao invés de `u_char`; na verdade, é uma simples mudança no tamanho do número.

4.11.1. Número da interface

Em IPv6, a interface é especificada pelo número, pois seria pouco prático

especificar a interface pelo seu endereço IP *stateless* (que pode mudar caso seja trocada a placa de rede).

Se este número de interface é passado como zero, a tabela de roteamento decidirá qual a interface de saída. No caso de endereços *stateless* `FE80::/10`, a passagem do número de interface é obrigatório (vide 4.1 item b).

Em geral, o programador dispõe do nome da interface por onde deve ser emitido o multicast, e precisa apenas descobrir o número. A função `if_nametoindex(const char*)` foi definida para facilitar esta tarefa.

4.11.1.1. Métodos diretos de descoberta do número da interface

Os métodos pelos quais a função `if_nametoindex()` descobre o número da interface são dependentes de implementação. No Linux, são dois:

a) Consultar o arquivo `/proc/net/if_inet6`;

b) Utilizar a chamada `ioctl(fd_trampolim, SIOGIFINDEX, &ifreq)`.

4.12. Opções de soquete

Segue a lista das opções disponíveis para IPv6, exceto as opções de multicast que já foram abordadas. As opções sem paralelo em IPv4 estão marcadas como "(nova)".

`IPV6_ADDRFORM`	Permite que um soquete seja convertido de/para IPv4. (nova)
`IPV6_CHECKSUM`	Especifica a localização do checksum dentro de um pacote IPv6 "cru". Para outros protocolos, inclusive ICMPv6, o kernel sempre encarrega-se de calcular o checksum. (nova)
`IPV6_DSTOPTS`	Cabeçalhos opcionais para o destino devem ser

	recebidos como dados auxiliares via `recvmsg()`. (nova)
`IPV6_HOPLIMIT`	Recebe o campo *hop count* como dado auxiliar. (nova)
`IPV6_HOPOPTS`	Cabeçalhos opcionais *hop-by-hop* devem ser passados como dados auxiliares. (nova)
`IPV6_NEXTHOP`	Permite especificar o endereço *next hop* via `sendmsg()` (nova)
`IPV6_PKTINFO`	Recebe endereço de destino e número da interface de chegada como dados auxiliares
`IPV6_RECVDSTADDR`	Recebe endereço de destino como dado auxiliar (obsoleto, não mais suportado pelo Linux)
`IPV6_RECVIF`	Recebe número da interface de chegada como dado auxiliar (obsoleto, não mais suportado em Linux)
`IPV6_PKTOPTIONS`	Permite que TCP troque dados auxiliares. Normalmente, dados auxiliares são tratados via `sendmsg()` e `recvmsg()` o que pressupõe comunicação em datagramas (UDP ou soquete "cru").
`IPV6_RTHDR`	Cabeçalho de roteamento IPv6 será recebido como dado auxiliar. (nova)
`IPV6_UNICAST_HOPS`	Permite especificar o limite de saltos (*hop limit*) (análoga a `IP_TTL` em IPv4)

As opções a seguir são mais recentes que STEVENS (1998):

`IPV6_AUTHHDR`	Cabeçalho IPSEC AH (será obsoletado)
`IPV6_ROUTER_ALERT`	Repassa todos os soquetes contendo um alerta de roteamento para o soquete
`IPV6_MTU_DISCOVER`	Controla o recurso de descoberta de MTU (*path MTU discovery*) no soquete.
`IPV6_MTU`	Obtém ou configura o MTU para o soquete. Limitado ao MTU da interface.

`IPV6_RECVERR`	Controla recebimento de erros assíncronos como dados auxiliares.

4.13. Soquetes crus (*raw*)

Soquetes crus permitem enviar e receber pacotes IP, sem imposição de um protocolo de quarta camada como UDP ou TCP. Por outro lado, a responsabilidade de criar pacotes válidos recai totalmente sobre o aplicativo.

A principal aplicação dos soquetes crus é a comunicação ICMP. Qualquer protocolo de transporte não suportado diretamente pelo kernel pode ser implementado diretamente pela aplicação através de soquetes crus.

Os soquetes crus diferem dos soquetes de acesso à camada de enlace pois enviam/recebem apenas pacotes IP, e recebem apenas pacotes que não puderam ser aproveitados por outra conexão. Por exemplo, os pacotes de uma conexão TCP em andamento não são ouvidos por nenhum soquete cru, portanto ele não se presta para *sniffing.*

Por outro lado, todos os pacotes IP que não tenham sido aproveitados por conexões TCP ou UDP serão repassados a *todos* os soquetes crus. Cada aplicativo usuário de soquetes crus deve, de alguma maneira, discriminar os pacotes desejados. (No protocolo ICMP, os campos *identificador* e *número seqüencial* prestam-se a isso). O uso excessivo de soquetes crus por muitos processos ao mesmo tempo pode sobrecarregar a máquina, pois o número de pacotes repassados aumenta exponencialmente.

Quando um soquete cru é criado, deve-se passar através do terceiro parâmetro de `socket()` uma constante `IPPROTO_x` onde `x` é o protocolo de quarta camada em que o programa está interessado.

As diferenças entre soquetes crus IPv4 e IPv6 são:

a) O soquete é criado com a família `AF_INET6` ao invés de `AF_INET`;

b) Os valores de `IPPROTO_ICMP` e `IPPROTO_ICMPV6` são diferentes, pois são protocolos diferentes (vide 1.3.8);

c) Em IPv4, o programa pode utilizar a opção `IP_HDRINCL`, que delega ao programa a geração e acesso ao cabeçalho de terceira camada (IP). Esta opção *não* é válida em IPv6. Programas que precisem lidar com o cabeçalho IP devem usar soquetes de acesso à camada de enlace;

d) Caso haja necessidade de ler/gravar dados do cabeçalho IP, inclusive cabeçalhos auxiliares, isso deve ser feito via opções IP ou acesso a dados auxiliares (vide respectivos tópicos);

e) Todos os campos nos cabeçalhos de protocolo apresentam-se em *network byte order*;

f) Como o *checksum* dos protocolos de quarta camada inclui dados da terceira camada em IPv6, o kernel oferece a comodidade em fazê-lo em lugar do programa;

Para os soquetes crus ICMPv6, esse cálculo sempre é feito pelo kernel. Para os demais protocolos, a opção `setsockopt() IPV6_CHECKSUM` permite comutar o recurso.

g) Como o protocolo ICMPv6 acumula muito mais funções que ICMPv4, existe uma opção de soquete que permite filtrar os pacotes por subtipo. A estrutura de filtragem é `icmp6_filter`. As macros que manipulam esta estrutura são

```
ICMP6_FILTER_SETPASSALL(struct icmp6_filter* filtro)
ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter* filtro)
ICMP6_FILTER_SETPASS(int subtipo, struct icmp6_filter* filtro)
```

```
ICMP6_FILTER_SETBLOCK(int subtipo, struct icmp6_filter* filtro)
int ICMP6_FILTER_WILLPASS(int subtipo,
                          const struct icmp6_filter *filtro)
int ICMP6_FILTER_WILLBLOCK(int subtipo,
                          const struct icmp6_filter *filtro)
```

Os valores aceitáveis para o subtipo podem ser consultados em `<netinet/icmp6.h>`. A estrutura é passada para o kernel via comando

```
setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER,
           filtro, sizeof(icmp6_filter))
```

4.14. Acesso à camada de enlace

A interface de acesso à camada de enlace não é um padrão do Sockets BSD; existem pelo menos três implementações bem difundidas, muito embora, segundo STEVENS (1998) a biblioteca *libpcap* seja o padrão de fato para ocultar essa multiplicidade.

A rigor o acesso à camada de enlace não sofre qualquer alteração pela inclusão do IPv6, que está numa camada superior. As possíveis mudanças num aplicativo que use esse recurso são:

a) Se o aplicativo está interessado em pacotes IPv6, deve utilizar a opção de filtragem adequada à interface.

b) Pacotes IPv6 devem ser enquadrados na estrutura `ip6_hdr` contida no arquivo de inclusão `<netinet/ip6.h>`.

4.15. Interoperabilidade entre IPv4 e IPv6

4.15.1. Convivência de aplicativos servidores IPv4 e IPv6

Um aplicativo pode permanecer na escuta de uma porta do protocolo de transporte em todas as interfaces, ou em apenas uma interface, ou ainda em algumas delas.

Quando o aplicativo deseja ouvir em todas as interfaces, sinaliza isso ao sistema operacional passando um pseudo-endereço a `bind()` (`INADDR_ANY` em IPv4, `in6addr_any` em IPv6). Apenas uma chamada a `bind()` é necessária.

É permitido que vários aplicativos ouçam a mesma porta, desde que cada um esteja atrelado a uma interface diferente. Não pode haver dois aplicativos ouvindo na mesma porta e na mesma interface. E naturalmente, se um aplicativo está ouvindo em todas as interfaces, nenhum outro aplicativo poderá atrelar-se à mesma porta.

Isso tudo é verdadeiro tanto para IPv4 quanto para IPv6, considerados isoladamente. Porém, IPv4 e IPv6 alimentam as mesmas camadas de transporte, e pode haver conversão entre endereços IPv4 e IPv6; portanto, é necessário estabelecer uma regra adicional de convivência.

Para o caso de dois aplicativos, um IPv4 e outro IPv6, ambos ouvindo na mesma porta e na(s) mesma(s) interface(s), a implementação pode escolher uma das regras a seguir:

a) Os servidores convivem; conexões IPv6 vão para o servidor IPv6, conexões IPv4 para o servidor IPv4; ou

b) Os servidores não podem conviver. O primeiro a ser iniciado tomará conta da porta, os próximos falharão ao tentar usar a mesma porta.

A maioria das implementações, Linux inclusive, segue o critério B por não ser ambíguo, conforme o item 4.15.2.

4.15.2 Mapeamento automático de IPv4 em IPv6

Os endereços IPv4 podem ser mapeados em IPv6 (`::FFFF:0:0/96`), e os protocolos de transporte são os mesmos para IPv4 e IPv6.

Graças a isso, um aplicativo-servidor IPv6 pode receber conexões IPv4 sem qualquer previsão ou tratamento especial. Basta que o sistema operacional subjacente remapeie as conexões IPv4 (esse é realmente o comportamento que se espera de qualquer implementação).

Se dois servidores IPv4 e IPv6 puderem conviver ouvindo a mesma porta, isso cria incerteza, pois uma conexão IPv4 poderia ser aceita por qualquer dos servidores.

Para eliminar essa ambigüidade, a maioria das implementações não permite que dois servidores, um IPv4 e outro IPv6, ouçam a mesma porta do protocolo de transporte (item 4.15.1 opção B).

O aplicativo pode optar em não receber conexões IPv4; basta testar se o endereço remoto é IPv4 mapeado em IPv6, com a macro `IN6_IS_ADDR_V4MAPPED()`.

4.15.3 Passagem de descritores de arquivo

Em programação POSIX, é bastante comum passar descritores de arquivo abertos para processos-filhos. Também é possível, embora menos comum, passar descritores entre processos não aparentados.

Normalmente o processo-escravo não se importa com o que o descritor realmente representa, e não precisa mesmo se importar se utilizar apenas as operações normais de arquivo, como `read()`, `write()` etc. sobre o descritor.

No entanto, há um caso especial criado pela convivência entre IPv4 e IPv6. Se um

processo-mestre com descritor IPv6 aberto repassar esse descritor a um processo-escravo que espera um soquete IPv4, e este último tentar executar alguma chamada como `setsockopt()` que se aplique apenas a IPv4, tal chamada falhará.

A solução definitiva é consertar o programa do processo-escravo; mas nem sempre isso é possível.

No caso de soquetes IPv6 que utilizem exclusivamente endereços IPv4 mapeados em IPv6 (`::FFFF:0:0/96`), a comunicação subjacente é IPv4, e esse soquete pode ser convertido para IPv4 durante seu uso. Se temos como modificar o programa-mestre mas não o programa-escravo, esta é a saída.

Para converter um soquete IPv6 para IPv4, GILLIGAN (1997) prescreve o seguinte comando `setsockopt()`:

```
int addrform = PF_INET;

setsockopt(descritor, IPPROTO_IPV6, IPV6_ADDRFORM,
           (char*) &addrform, sizeof(addrform));
```

A opção `IPV6_ADDRFORM` do comando `setsockopt()` presta-se a conversão de soquetes. O parâmetro passado (no exemplo, através da variável `addrform`) indica para que protocolo se quer converter o soquete - no caso, `PF_INET` (IPv4).

Uma vez convertido, o soquete pode sofrer qualquer operação IPv4 sem problemas. Isso permite inclusive que ele seja passado a outros processos que esperem exclusivamente soquetes IPv4.

É prudente frisar que essa conversão não faz nenhum milagre, e de forma nenhuma deve ser entendida como um proxy IPv4-IPv6. Um soquete IPv6 que utilize endereços nativos IPv6 não pode ser convertido. Soquetes crus IPv6 também não o podem, pois seu *modus operandi* é muito especializado.

Não se pode "pingar" um endereço IPv4 da seguinte forma:

```
# ping6 ::FFFF:200.215.89.83
```

pois não existe conversão subjacente de IPv6 para IPv4 para soquetes crus.

Por outro lado, para converter um soquete IPv4 para IPv6 (se o processo-escravo espera um descritor de arquivo IPv6), o comando é:

```
int addrform = PF_INET6;

setsockopt(descritor, IPPROTO_IP, IPV6_ADDRFORM,
           (char*) &addrform, sizeof(addrform));
```

5. A interface `/proc` do Linux

Na maioria dos sistemas operacionais baseados em Unix, existe um pseudo--sistema de arquivos montado em `/proc`, que permite obter e mudar configurações do sistema.

Como as configurações em `/proc` são lidas e gravadas como se fossem arquivos comuns, é muito mais facil para um programa interfacear com o sistema através desse caminho. Operações tipicamente muito complicadas de se realizar via soquetes, como por exemplo a configuração de interfaces, tornam-se triviais.

Infelizmente, há a grande desvantagem da perda da portabilidade, pois cada sistema operacional Unix estrutura o sistema de arquivos `/proc` de uma forma particular.

Este trabalho relaciona os principais pseudo-arquivos de configuração presentes no Linux e pertinentes a IPv6.

5.1. `/proc/net`

O diretório `/proc/net` apenas fornece dados, não permite alterá-los.

5.1.1. `/proc/net/if_inet6`

Este arquivo lista as interfaces IPv6 do sistema, conforme o exemplo abaixo:

```
$ cat /proc/net/if_inet6
00000000000000000000000000000001 01 80 10 80       lo
fe8000000000000002e07dfffe9c924e 02 0a 20 80     eth0
```

Os campos são, em ordem:

a) endereço IPv6 como um número hexadecimal de 128 bits;

b) número seqüêncial da interface (aquele que é necessário informar ao sistema em determinadas circunstância, vide 4.1);

c) comprimento em bits da máscara de rede;

d) escopo da interface;

e) estado da interface;

f) nome da interface no sistema, que é unívoco.

5.1.2. `/proc/net/igmp6`

Lista as interfaces e endereços que estão na escuta de multicast.

```
$ cat /proc/net/igmp6
1    lo              ff020000000000000000000000000001     1 00000004 0
2    eth0            ff020000000000000000000012341234     1 00000004 0
2    eth0            ff020000000000000000001ff9c924e     1 00000004 0
2    eth0            ff020000000000000000000000000001     2 00000004 0
```

Os primeiros quatro campos são os mais importantes:

a) número seqüencial da interface;

b) nome da interface;

c) endereço multicast IPv6, como um número hexadecimal de 128 bits;

d) contagem de aplicativos ouvindo o endereço.

No exemplo em particular, o endereço `FF02::1234:1234` foi escolhido por um aplicativo multicast, Esse mesmo aplicativo, bem como o próprio sistema operacional, ouvem ambos o endereço `FF02::1` - note que a contagem de uso deste último é igual a 2.

Os demais endereços de multicast são criados pelo sistema operacional e destinam-se à resolução entre endereços IPv6 e endereços de enlace (vide 1.3.8.1).

5.1.3. `/proc/net/tcp6`

Lista as conexões TCP/IPv6 ativas.

Seu formato é difícil de interpretar visualmente, e não serve para “consumo humano”. Aplicativos de baixo nível como *netstat* lêem esse arquivo (ao invés de usar chamadas `ioctl()`) para obter informações do sistema.

Nada impede, é claro, que o administrador do sistema interprete manualmente o arquivo e/ou escreva um aplicativo que o faça.

5.1.4. `/proc/net/ipv6_route`

Lista as rotas IPv6. Seu formato também não se destina à visualização direta.

5.2. `/proc/sys/net/ipv6`

Este diretório fornece configurações que podem ser tanto lidas quanto alteradas. (Possibilidade de alteração é uma característica de todos os arquivos em `/proc/sys`). O acesso a estas configurações pode ser feito tanto diretamente como por meio da função `sysctl()`.

5.2.1. `/proc/sys/net/ipv6/conf` e `/proc/sys/net/ipv6/neigh`

Através destes diretórios, pode-se modificar parâmetros IPv6 individuais de cada interface de rede. Para permitir o acesso a cada interface, cada diretório apresenta subdiretórios com os nomes das interfaces.

```
$ find /proc/sys/net/ipv6/conf/
/proc/sys/net/ipv6/conf/
/proc/sys/net/ipv6/conf/default
/proc/sys/net/ipv6/conf/default/router_solicitation_delay
/proc/sys/net/ipv6/conf/default/router_solicitation_interval
/proc/sys/net/ipv6/conf/default/router_solicitations
...
/proc/sys/net/ipv6/conf/all/dad_transmits
/proc/sys/net/ipv6/conf/all/autoconf
```

```
/proc/sys/net/ipv6/conf/all/accept_redirects
/proc/sys/net/ipv6/conf/all/accept_ra
/proc/sys/net/ipv6/conf/all/mtu
...
/proc/sys/net/ipv6/conf/eth0/hop_limit
/proc/sys/net/ipv6/conf/eth0/forwarding
...
/proc/sys/net/ipv6/conf/lo/autoconf
/proc/sys/net/ipv6/conf/lo/accept_redirects
/proc/sys/net/ipv6/conf/lo/accept_ra
...

$ find /proc/sys/net/ipv6/neigh/
/proc/sys/net/ipv6/neigh/
/proc/sys/net/ipv6/neigh/eth0
/proc/sys/net/ipv6/neigh/eth0/locktime
/proc/sys/net/ipv6/neigh/eth0/proxy_delay
/proc/sys/net/ipv6/neigh/eth0/anycast_delay
/proc/sys/net/ipv6/neigh/eth0/proxy_qlen
...
/proc/sys/net/ipv6/neigh/lo/unres_qlen
/proc/sys/net/ipv6/neigh/lo/gc_stale_time
/proc/sys/net/ipv6/neigh/lo/delay_first_probe_time
/proc/sys/net/ipv6/neigh/lo/base_reachable_time
....
/proc/sys/net/ipv6/neigh/default/proxy_delay
/proc/sys/net/ipv6/neigh/default/anycast_delay
/proc/sys/net/ipv6/neigh/default/proxy_qlen
...
```

O diretório `conf/` contém configurações intrínsecas da interface, enquanto `neigh/` contém configurações que afetam o relacionamento da interface com os demais nós da rede.

Em ambos os diretórios há um subdiretório `default/`, uma espécie de modelo cujas configurações são copiadas no momento de ativação de alguma interface.

Dentro de `conf/` há ainda o subdiretório `all/`, que permite mudar determinada configuração de todas as interfaces ao mesmo tempo. Para os fins práticos, os arquivos ali contidos são somente de escrita, pois os valores lidos não representam o estado de nenhuma interface em particular.

Segue alguns exemplos de configurações por interface:

`/proc/sys/net/ipv6/conf/eth0/forwarding`

se a interface `eth0` pode servir para roteamento

`/proc/sys/net/ipv6/conf/eth0/hop_limit`

Limite de saltos em pacotes emitidos por `eth0`

`/proc/sys/net/ipv6/conf/eth0/mtu`

MTU dos pacotes emitidos por `eth0`

5.2.2. `/proc/sys/net/ipv6/route`

Contém diversos parâmetros de roteamento IPv6. Dificilmente tais valores precisam ser consultados ou alterados, a não ser possivelmente em roteadores comerciais (tipo "caixa preta") baseados em Linux.

6. Exemplos de código Sockets BSD para IPv6

Segue uma lista de exemplos de código funcional Sockets BSD para IPv6, em linguagem C++. Eles foram testados testados no sistema operacional Linux, kernel 2.4.19.

Os programas são padrão POSIX e devem funcionar em outros sistemas operacionais POSIX que suportem IPv6. Pode, no entanto, haver pequenas diferenças e incompatibilidades, visto que IPv6 evoluiu muito nos últimos anos e ainda evolui, portanto há mudanças na API que versões antigas de determinado sistema não implementavam (vide 4.1, campo `sin6_scope_id`).

Os exemplos, embora tenham sido retirados de programas funcionais, não executam tarefas úteis. Para torná-los funcionais, é necessário acrescentar a lógica da camada de aplicação.

Foi utilizado C++ particularmente na manipulação de *strings*; tais manipulações podem ser feitas em C porém são mais práticas e claramente demonstráveis em C++.

6.1. Comunicação TCP básica

6.1.1. Cliente

```
#define PORTA 1000
#define SERVIDOR "fe80::1234:4568"
#define INTERFACE "eth0"

sockaddr_in6 end_servidor;
bzero(&end_servidor, sizeof(end_servidor));
#ifdef SIN6_LEN
           // necessário em BSD, desnecessário em Linux
           end_servidor.sin6_len = SIN6_LEN;
#endif
end_servidor.sin6_family = AF_INET6;
end_servidor.sin6_port = htons(PORTA);
inet_pton(AF_INET6, SERVIDOR, &end_servidor.sin6_addr);
// se o endereço não fosse link-local, bastaria preencher
// sin6_scope_id com zero
end_servidor.sin6_scope_id = if_nametoindex(INTERFACE);
```

```
// descritor de arquivo
int fd;

if((fd = socket(AF_INET6, SOCK_STREAM, 0)) < 0) {
           // soquete inválido (IPv6 não suportado na máquina)
           exit(1);
}

if (connect(fd, (sockaddr*) &end_servidor, sizeof(end_servidor))) {
           // conexão falhou
           exit(2);
}
```

Daqui para diante, absolutamente nada muda seja qual for o protocolo de rede; as funções de comunicação como `read()`, `write()`, `send()`, `recv()`, `writev()`, `readv()`, `shutdown()` e `close()` funcionam da mesma forma.

6.1.2. Servidor

```
#define PORTA 1000

// criamos um endereço IPv6 "coringa" (0000::0000), para atrelar
// todas as interfaces com bind()

sockaddr_in6 ia6_any;
bzero(&ia6_any, sizeof(ia6_any));
#ifdef SIN6_LEN
           // necessário em BSD, desnecessário em Linux
           ia6_any.sin6_len = SIN6_LEN;
#endif
ia6_any.sin6_family = AF_INET6;
ia6_any.sin6_port = htons(PORTA);
ia6_any.sin6_addr = in6addr_any;
ia6_any.sin6_scope_id = 0;

// descritor de arquivo
int fd;

if((fd = socket(AF_INET6, SOCK_STREAM, 0)) < 0) {
           // soquete inválido (IPv6 não suportado na máquina)
           exit(1);
}

if (bind(fd, (sockaddr*) &ia6_any, sizeof(ia6_any))) {
           // bind() falhou, provavelmente outro já está bind()ado à porta
           exit(1);
}

listen(fd, 5);

for(;;) {
           sockaddr_in6 end_cliente;
           socklen_t len_cliente = sizeof(end_cliente);
           int fd_conexao = accept(fd, (sockaddr*) &end_cliente,
                                                   &len_cliente);
```

```
            // comunica-se com o cliente

            close(fd_conexao);
}
```

Assim como no código de exemplo para cliente TCP, a comunicação em si faz uso das mesmas diretivas (`send()`, `recv()` etc.) seja qual for o protocolo de rede.

6.2. Comunicação UDP básica

6.2.1. Criação do soquete e definições gerais

Em UDP, a criação do soquete é idêntica seja ele utilizado como cliente ou servidor, pois do ponto de vista do sistema operacional não existem conexões UDP.

```
#define PORTA 1000

// criamos um endereço IPv6 "coringa" (0000::0000), para atrelar
// todas as interfaces com bind()

sockaddr_in6 ia6_any;
bzero(&ia6_any, sizeof(ia6_any));
#ifdef SIN6_LEN
            // necessário em BSD, desnecessário em Linux
            ia6_any.sin6_len = SIN6_LEN;
#endif
ia6_any.sin6_family = AF_INET6;
ia6_any.sin6_port = htons(PORTA);
ia6_any.sin6_addr = in6addr_any;
ia6_any.sin6_scope_id = 0;

// descritor de arquivo
int fd;

if((fd = socket(AF_INET6, SOCK_DGRAM, 0)) < 0) {
            // soquete inválido (IPv6 não suportado na máquina)
            exit(1);
}

// Este segmento torna o soquete não bloqueante, ou seja, recv(), recvfrom()
// e recvmsg() retornam imediamente com erro se não houver dados a receber.
//
// A maioria dos programas usa select() ao invés de soquetes não bloqueantes.
//
int opcoes = fcntl(fd, F_GETFL, 0);
if (fcntl(fd, F_SETFL, opcoes | O_NONBLOCK) == -1) {
            // soquetes não bloqueantes não são suportados
            exit(1);
}

if (bind(fd, (sockaddr*) &ia6_any, sizeof(ia6_any))) {
            // bind() falhou, provavelmente outro já está bind()ado à porta
```

```
            exit(1);
}
```

6.2.2. Envio

```
#define INTERFACE "eth0"
#define DESTINO "fe80::1234:5678"

char buffer[1500]; // contém a mensagem
int len_buffer; // contém o comprimento da mensagem

sockaddr_in6 end_destino;

bzero(&end_destino, sizeof(end_destino));

#ifdef SIN6_LEN
            // necessário em BSD, desnecessário em Linux
            end_destino.sin6_len = SIN6_LEN;
#endif

end_destino.sin6_family = AF_INET6;

end_destino.sin6_port = htons(PORTA);

inet_pton(AF_INET6, DESTINO, &end_destino.sin6_addr);

// necessário se utilizarmos endereços link-local, como o exemplo.
// Do contrário, deve-se passar zero
end_destino.sin6_scope_id = if_nametoindex(INTERFACE);

int vol = sendto(fd, buffer, len_buffer, 0, (sockaddr*) &end_destino,
                 sizeof(end_destino));


if (vol < 0) {
            // falha
            if (errno != EAGAIN && errno != EINTR && errno != EWOULDBLOCK) {
                        // erro fatal
                        exit(1);
            } else {
                        // erro não fatal
            }
}
```

De acordo com a especificação do Sockets BSD e do POSIX, uma remessa ou recebimento de datagrama *nunca deve falhar* (exceto por `EWOULDBLOCK`, se o soquete for não-bloqueante), Mas é possível que haja alguma implementação imperfeita que não respeite essa regra; portanto é prudente prevermos a situação de falha mesmo em protocolos orientados a datagrama.

6.2.3. Recebimento

```
char buffer[1500];
int len_buffer;

sockaddr_in6 end_origem;
u_int socklen = sizeof(end_origem);

len_buffer = recvfrom(fd, buffer, sizeof(buffer), 0,
                              (sockaddr*) &end_origem, &socklen);

if (len_buffer < 0) {
           // falha
           if(errno != EAGAIN && errno != EINTR && errno != EWOULDBLOCK) {
                      // erro fatal
                      exit(1);
           } else {
                      // erro não fatal, pode tentar novamente mais tarde
           }
}
```

6.2.4. Coleta de dados auxiliares

Possivelmente o dado auxiliar mais comumente coletado é o endereço de destino de um pacote UDP, e é o que será utilizado como exemplo neste trabalho. A coleta de outros dados auxiliares é análoga.

A chamada `recvmsg()` entra em substituição a `recvfrom()`. O código a seguir substitui o exemplo 6.2.3.

```
char buffer[1500];
int len_buffer;

sockaddr_in6 end_origem;

// cria estrutura iovec, que deve apontar para um buffer que
// receberá o conteúdo do pacote em si
iovec iov;
iov.iov_base = buffer;
iov.iov_len = sizeof(buffer);

// buffer para receber o endereço de destino
char buf_end_destino[CMSG_SPACE(sizeof(in6_pktinfo))];

// finalmente cria a estrutura msghdr e relaciona seus componentes
// aos elementos criados mais acima
msghdr msg;

msg.msg_name = &end_origem;
msg.msg_namelen = sizeof(end_origem);
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

```
msg.msg_control = buf_end_destino;
msg.msg_controllen = sizeof(buf_end_destino);
msg.msg_flags = 0;

// configuramos o descritor de arquivo para fornecer o endereço de
// destino do pacote

// Linux *não* suporta mais as opções IP(V6)_RECVDSTADDR e IP(V6)_RECVIF
// porém suporta a opção IP(V6)_PKTINFO que fornece as duas informações
// ao mesmo tempo

int ligado = 1;
if (setsockopt(fd, IPPROTO_IPV6, IPV6_PKTINFO, &ligado, sizeof(ligado)) < 0)
{
            // setsockopt() falhou (opção não suportada pela implementação?)
            exit(1);
}

len_buffer = recvmsg(fd, &msg, 0);

if (len_buffer < 0) {
            // falha
            if(errno != EAGAIN && errno != EINTR && errno != EWOULDBLOCK) {
                        // erro fatal
                        exit(1);
            } else {
                        // erro não fatal, pode tentar novamente mais tarde
                        exit(1);
            }
}

// buffer[] conterá o payload do pacote, através de msg.msg_iov
// end_origem conterá o endereço de origem, através de msg.msg_name
// resta obter o endereço de destino e a interface de onde veio o pacote

in6_pktinfo info;
in6_addr end_destino;
int interface;

// obtém o dado auxiliar requisitado
for(cmsghdr* cmptr = CMSG_FIRSTHDR(&msg); cmptr != 0;
    cmptr = CMSG_NXTHDR(&msg, cmptr)) {
            if (cmptr->cmsg_level == IPPROTO_IPV6 &&
                cmptr->cmsg_type == IPV6_PKTINFO) {
                        memcpy(&info, CMSG_DATA(cmptr), sizeof(info));
            }
}

end_destino = info.ipi6_addr;
interface = info.ipi6_ifindex;
```

Como se pode notar por este código-fonte, o recebimento de dados auxiliares utiliza muitos ponteiros e é fácil cometer erros ou esquecer alguma atribuição.

6.3. Comunicação com nós IPv4

Valem as regras explanadas em 4.15. Em resumo:

a) Servidores IPv6 recebem conexões e pacotes IPv4 sem necessidade de qualquer previsão especial no código. O endereço de origem será um endereço IPv4 mapeado em IPv6 (`::FFFF:0:0/96`);

b) Se o servidor precisar discriminar clientes IPv4 de IPv6, pode usar a macro `IN6_IS_ADDR_V4MAPPED()` para testar se o endereço de origem é IPv4;

c) Clientes IPv6 devem utilizar os endereços IPv4 mapeados em IPv6 para conectar-se a servidores IPv4.

6.4. Multicast

A comunicação multicast só funciona com protocolos de transporte orientados a datagrama. Tipicamente é utilizado o UDP, e que também será usado no código de exemplo.

No Linux até a versão 2.4.19, os endereços de multicast provisórios (prefixo `FF1x::/24`) não funcionam; a função `sendto()` falha e o soquete é inutilizado após este erro.

O problema está no arquivo `net/ipv6/addrconf.c` do kernel, mais especificamente na função `ipv6_addr_type()`, que falha em identificar o escopo do endereço se o flag for diferente de zero.

Felizmente, os endereços `FF0x::/24` funcionam a contento.

6.4.1. Criação do soquete

Note que a parte inicial é virtualmente idêntica a 6.2.1. Neste exemplo, a aplicação entra em dois grupos de multicast: um predefinido (`FF02::1` – todos os nós da rede local) e outro criado pela aplicação (`FF02::1234:1234`). Ambos têm escopo de rede local.

```
#define PORTA 1000
#define MULTICAST_LAN "FF02::1"
#define INTERFACE “eth0”
#define MULTICAST_PARTICULAR "FF02::1234:1234"

// criamos um endereço IPv6 "coringa" (0000::0000), para atrelar
// todas as interfaces com bind()

sockaddr_in6 ia6_any;
bzero(&ia6_any, sizeof(ia6_any));
#ifdef SIN6_LEN
            // necessário em BSD, desnecessário em Linux
            ia6_any.sin6_len = SIN6_LEN;
#endif
ia6_any.sin6_family = AF_INET6;
ia6_any.sin6_port = htons(PORTA);
ia6_any.sin6_addr = in6addr_any;
ia6_any.sin6_scope_id = 0;

// descritor de arquivo
int fd;

if((fd = socket(AF_INET6, SOCK_DGRAM, 0)) < 0) {
            // soquete inválido (IPv6 não suportado na máquina)
            exit(1);
}

// Este segmento torna o soquete não bloqueante, ou seja, recv(), recvfrom()
// e recvmsg() retornam imediamente com erro se não houver dados a receber.
//
// A maioria dos programas usa select() ao invés de soquetes não bloqueantes.
//
int opcoes = fcntl(fd, F_GETFL, 0);
if (fcntl(fd, F_SETFL, opcoes | O_NONBLOCK) == -1) {
            // soquetes não bloqueantes não são suportados
            exit(1);
}

// Passar o endereço de multicast ao bind() é uma forma simples
// de evitar que pacotes "normais" sejam recebidos. Infelizmente
// algumas implementações falham quando isso é feito; e mesmo que
// funcione, continua sendo necessário chamar setsockopt(IPV6_ADD_MEMBERSHIP)
//
// Assim, o procedimento portável é passar o endereço in6_addrany para
// o bind().

if (bind(fd, (sockaddr*) &ia6_any, sizeof(ia6_any))) {
            // bind() falhou, provavelmente outro já está bind()ado à porta
            exit(1);
}

// este segmento desliga o loopback de pacotes multicast, e costuma ser
// utilizada. Dificilmente um aplicativo tem interesse em ouvir pacotes
```

```
// multicast emitidos por ele mesmo.
u_int mcast_loop = 0;
if (setsockopt(fd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
               &mcast_loop, sizeof(mcast_loop))) {
           exit(1);
}

in6_addr mcast_geral, mcast_particular;

inet_pton(AF_INET6, MULTICAST_LAN, &mcast_geral);
inet_pton(AF_INET6, MULTICAST_PARTICULAR, &mcast_particular);

// entra no grupo multicast "todas as máquinas da LAN" (FF02::1)

ipv6_mreq mcast_req;

mcast_req.ipv6mr_multiaddr = mcast_geral;
mcast_req.ipv6mr_interface = if_nametoindex(INTERFACE);

if (setsockopt(fd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
               &mcast_req, sizeof(mcast_req))) {
           // não conseguiu entrar no grupo
           exit(1);
}

// entra num outro grupo de multicast arbitrariamente definido pela aplicação

mcast_req.ipv6mr_multiaddr = mcast_particular;
mcast_req.ipv6mr_interface = if_nametoindex(INTERFACE);

if (setsockopt(fd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
                                 &mcast_req, sizeof(mcast_req))) {
           // não conseguiu entrar no grupo
           exit(1);
}
```

6.4.2. Envio

Idêntico a 6.2.2, basta passar um endereço de multicast como endereço de destino.

O endereço de origem não pode ser um endereço de multicast. Se a atribuição do endereço de origem for deixada a cargo do sistema operacional (como tipicamente o é) será utilizado o endereço da interface de saída.

6.4.3. Recebimento

Se o aplicativo não precisa distinguir pacotes normais de pacotes multicast, vide item 6.2.3.

Se o aplicativo quiser discriminar pacotes de multicast, vide item 6.2.4. Para descobrir se um pacote é normal ou multicast, basta testar o endereço de destino com a macro `IN6_IS_ADDR_MULTICAST`:

```
if (IN6_IS_ADDR_MULTICAST(&end_origem.sin6_addr))) {
        // multicast
        ...
}
```

6.5. Obtenção dos endereços IPv6 da máquina

Infelizmente, não há uma função padrão para obter os endereços IPv6 da máquina. Deve-se utilizar métodos diretos (de baixo nível e dependentes de implementação).

6.5.1. Via `ioctl()`

Segundo USAGI, já mencionado neste trabalho, não é possível obter os endereços IPv6 via `ioctl(SIOCGIFADDR)`, como seria possível em IPv4, pois uma interface IPv6 pode ter múltiplos endereços.

Em IPv4, a sintaxe *interface:número-de-alias* (e.g. `eth0:2`), pode ser usada para especificar qual o endereço alternativo desejado na chamada a `ioctl(SIOCGIFADDR)`.

Tal sintaxe não é suportada no IPv6 do Linux, portanto realmente não há meio de utilizarmos `ioctl()` para obter os diversos endereços IPv6 de uma interface.

6.5.2. Via `/proc`, no Linux

```
static std::string HEXDIGITS = "0123456789abcdef";

int hex2bin(const std::string& s)
{
```

```
        int ret = 0;

        for(unsigned int i = 0; i < s.size(); ++i) {
                ret <<= 4;
                ret |= (unsigned char) HEXDIGITS.find(tolower(s[i]));
        }

        return ret;
}

int get_ifv6_info(const char *if_nome, in6_addr *addr)
{
        std::ifstream arq("/proc/net/if_inet6");
        std::string _bruto, _endereco, _indice, _netmask, _escopo,
                    _estado, _nome;
        std::string endereco;
        int indice;

        // formato: endereço (32 digitos hexa), indice, comprimento da
        // mascara,
        // escopo (0x20 = link-local)
        // estado (0x80=on), nome da interface

        while(getline(arq, _bruto, '\n') && _bruto.size() >= 53) {
                _endereco = _bruto.substr(0, 32);
                _indice = _bruto.substr(33, 2);
                _netmask = _bruto.substr(36, 2);
                _escopo = _bruto.substr(39, 2);
                _estado = _bruto.substr(42, 2);
                _nome = _bruto.substr(49, 4);
                if (if_nome == _nome) {
                        for(int n = 0; n < 32; n += 4) {
                                endereco += _endereco.substr(n, 4)+":";
                        }
                        endereco.erase(endereco.size()-1, 1);
                        inet_pton(AF_INET6, endereco.c_str(), addr);
                        if (IN6_IS_ADDR_LINKLOCAL(addr)) {
                                indice = hex2bin(_indice);
                                return indice;
                        }
                }
        }
        return -1;
}
```

Para obter o endereço IPv6, a função deve ser chamada da seguinte forma:

```
int iface_nr;
const char* iface_nome = "eth0";
sockaddr_in6 ia6_local;

iface_nr = get_ifv6_info(iface_nome, &ia6_local.sin6_addr);
```

A função retorna diretamente o número seqüencial da interface (que também poderia ser obtido pela função padrão `if_nametoindex()`) e indiretamente o endereço IPv6, preenchido em `ia6_local.sin6_addr` que foi passado como parâmetro.

6.6. Configuração de endereço IPv6 adicional

Para o administrador, a forma mais simples é utilizar o comando *ifconfig*, como no exemplo:

```
# ifconfig eth0 add 2001::1/64
```

Conforme pode ser visto em NETTOOLS, internamente o *ifconfig* preenche uma estrutura `in6_ifreq` com os dados da interface e do endereço, e utiliza a chamada `ioctl(fd, SIOCSIFADDR, in6_ifreq*)` para adicionar um endereço e `ioctl(fd, SIOCDIFADDR, in6_ifreq*)` para remover um endereço.

6.7. Resolução de nomes IPv6

6.7.1. Simples conversão de endereço textual para binário e vice-versa

```
// texto para endereço
const char endereco_texto = "FE80::1234:5678";
sockaddr_in6 endereco;
inet_pton(AF_INET6, endereco_texto, &endereco.sin6_addr);


// endereço para texto
char endereco_texto_2[INET6_ADDRSTRLEN];
inet_ntop(AF_INET6, &endereco.sin6_addr,
          endereco_texto_2, sizeof(endereco_texto_2));
```

6.7.2. Resolução de nomes

```
const char *nome = "www.dominio.com.br";

char buffer[INET6_ADDRSTRLEN];

hostent* h = gethostbyname2(nome, AF_INET6);
if (!h) {
            // falha na resolução do nome
            exit(1);
}
for(char** ppc = h->h_addr_list; (*ppc) != NULL; ++ppc) {
            inet_ntop(h->h_addrtype, *ppc, buffer, sizeof(buffer));
```

```
            printf("Endereço IPv6: %s\n", buffer);
}

hostent* h = gethostbyname2(nome, AF_INET);
if (!h) {
            // falha na resolução do nome
            exit(1);
}
for(char** ppc = h->h_addr_list; (*ppc) != NULL; ++ppc) {
            inet_ntop(h->h_addrtype, *ppc, buffer, sizeof(buffer));
            printf("Endereço IPv4: %s\n", buffer);
}
```

CONCLUSÃO

A necessidade do IPv6 é premente, e as ferramentas necessárias para sua implementação estão prontamente disponíveis. As iniciativas de utilização surgem a todo momento: vários provedores de serviços importantes já disponibilizam acesso via IPv6.

Alguns poucos provedores de acesso já oferecem acesso IPv6 nativo; não é surpresa isso ocorra em regiões como Japão, Austrália e Europa, menos privilegiadas na distribuição das faixas IPv4. A próxima grande frente de adoção de IPv6 é a América Latina.

O grande temor de muitos, e gancho para críticas severas ao IPv6, foi a dificuldade de obtenção de *momentum* para que a comunidade Internet começasse a efetivamente utilizar esse novo protocolo. Hoje, esse é um problema resolvido. Já existe uma boa massa crítica de documentação, software e administradores de rede capacitados em IPv6.

Resta agora que a necessidade dos recursos disponíveis unicamente em IPv6 aperte um pouco mais, o que vai ocorrer num futuro não muito distante.

E a necessidade não deve tardar. Segundo alguns, a próxima grande revolução é a substituição massiva do telefone por conexões rápidas à Internet; a comunicação de voz será absorvida pela voz-sobre-IP, que é uma tecnologia bem estabelecida. Mas não há como acomodar todos os terminais telefônicos no espaço de endereçamento IP atual.

E antes de os endereços IP estarem esgotados, os roteadores NAT já terão infernizado o suficiente os administradores de rede, a ponto de todos considerarem a migração para IPv6.

Quem já tentou implementar um esquema de voz-sobre-IP entre duas corporações, cada uma com seu firewall, conhece bem esses problemas. A única solução funcional hoje para H.323 e IPv4 é fornecida por um grande fabricante de equipamentos de rede –

cujo uso implica aquisição de hardware caro.

Das ferramentas de implementação IPv6, a mais importante é sem dúvida a interface de programação de aplicações (API). As duas famílias de sistemas operacionais mais importantes do mercado atual, Windows e UNIX, já possuem APIs implementadas e funcionais.

Conforme este trabalho pôde demonstrar, a API Sockets BSD para IPv6 é uma interface bem definida, bem resolvida, suficientemente independente de plataforma para que qualquer software de rede IPv6 (exceto talvez softwares de configuração de baixo nível e.g. *ifconfig* e *route*) seja portável entre os UNIX modernos.

Mais que isso, a sua concepção permite criar aplicativos prontos para IPv6, e que "pensam" unicamente em IPv6, mas funcionam corretamente numa rede puramente IPv4 ou mista IPv4/IPv6. Os softwares novos prevêem apenas IPv6, os softwares antigos podem ser modificados para isso; mas tudo continua funcionando na Internet atual.

Em particular no mundo do software livre, que reage mais rapida e positivamente à mudança, esse recurso já tem sido bastante explorado e vários softwares de usuário e de serviços já estão prontos para IPv6, e até mesmo mostram os endereços IPv4 como `::FFFF.a.b.c.d`.

Assim, é perfeitamente factível que, quando os provedores passarem a oferecer acesso IPv6 corriqueiramente, os usuários finais nem notem a diferença.

Concluímos assim afirmando que IPv6 é uma realidade inexorável, que ninguém pode mais dar-se ao luxo de ignorar. Porque as implementações e aplicações já existem; porque a necessidade é premente; porque bem ou mal já está sendo usado; porque é suportado pelos fabricantes; porque após um processo que já foi excessivamente demorado, foi a saída apontada pelo IETF, e não consta haja alternativa melhor; porque foi desenhado de modo a causar o menor impacto possível na Internet em funcionamento.

BIBLIOGRAFIA

CALLON, Ross. RFC 1347 - TCP e UDP com endereços maiores (TUBA). 1992

FULLER, V. et al. RFC 1519 - Roteamento inter-domínios sem classes (CIDR). 1993.

DIXON, T. RFC 1454 – Comparação de propostas da próxima versão de IP. 1993.

BRADNER, S; MANKIN, A. RFC 1550 - Solicitação de artigos sobre o IP Nova Geração (IPng). 1993.

PISTICELLO, D. RFC 1561 - Uso do ISO CLNP em ambientes TUBA. 1993.

SKELTON, R. RFC 1673 – Comentários da Electric Power Research Institute sobre IPng. 1994.

BRAZDZIUNAS, C. RFC 1680 – Suporte do IPng a serviços ATM. 1994.

CLARK, R.; AMMAR, M; CALVERT, K. RFC 1683 - Interoperabilidade multiprotocolos em IPng. 1994.

ATKINSON, R. RFC 1826 – Cabeçalho de autenticação IP. 1995.

ATKINSON, R. RFC 1827 – IP encapsulando payload de segurança (ESP). 1995.

THOMSON, S.; HUITEMA, C. RFC 1886 - Extensões ao DNS para suporte a endereços IPv6. 1995.

GILLIGAN, R. et al. RFC 2133 - Extensões IPv6 à Interface Sockets Básica. 1997.

STEVENS, Richard W.; THOMAS, M. RFC 2292 - API Sockets avançada para IPv6. 1998.

STEVENS, Richard W. Unix Network Programming. 2.ed. Prentice-Hall, 1998.

KENT, S. RFC 2401 - Arquitetura de segurança para o IP (IPSEC). 1998.

CRAWFORD, M. RFC 2464 - Transmissão de pacotes IPv6 sobre redes Ethernet. 1998.

ALLMAN, M; OSTERMANN, S.; METZ, C. RFC 2428 - Extensões do FTP para IPv6 e NATs. 1998.

HINDEN, R; DEERING, S. RFC 2373 – Arquitetura de endereçamento do IP versão 6. 1998.

RHODES, Neil; McKEEHAN, Julie. Palm Programming: The Developer's Guide. O'Reilly, 1999.

CRAWFORD, M.; HUITEMA, C. RFC 2874 - Extensões ao DNS para suporte a agregação e renumeração de endereços IPv6. 2000.

MILLER, Mark A. Implementing IPv6. 2.ed. M&T, 2000.

TSO, Theodore Y. *http://lists.insecure.org/lists/linux-kernel/2000/Feb/1140.html*. Linux Kernel: Re: What /proc should contain [was: /proc/driver/microcode]. 2000.

OPENGROUP. The Open Group Base Specifications Issue 6. IEEE Std 1003.1-2001. 2001.

PENÃ, Javier Fernández-Sanguino. *http://ipv6-gw.compendium.net.ar/6fevu/text/IPSEC-NAT.SGML.html*. Problems due to widespread use of NAT and IPSEC considerations. 2001

BUSH, R. Et al. RFC 3363 - Representação de endereços IPv6 no DNS. 2002.

GAST, Matthew S. 802.11 Wireless Networks. O'Reilly, 2002.

HAGEN, Silvia. IPv6 Essentials. O'Reilly, 2002.

IPROUTE. Código-fonte do pacote IPRoute2 para Linux. Versão 2.2.4. 2002.

TORVALDS, Linus et al. Código-fonte do kernel do Linux. Versão 2.4.19. 2002.

HOWSTUFFWORKS. *http://www.howstuffworks.com/question261.htm*. How does the birthday paradox work?

BURTLE. *http://burtleburtle.net/bob/crypto/exchange.html*. Secure key exchange with hashing and the birthday paradox.

FREES/WAN. *http://www.freeswan.org*. Site do projeto FreeS/WAN (IPSEC para Linux).

FREES/WAN. *http://www.freeswan.org/freeswan_snaps/CURRENT-SNAP/doc/politics.html#weak*

Government promotion of weak crypto (motivações do não suporte ao algoritmo DES, no FreeS/WAN).

FREENET6. *http://www.freenet6.net/*. Site do projeto Freenet6.

USAGI. *http://www.linux-ipv6.org/faq.html*. Site do projeto USAGI - implementação do IPv6 para Linux: perguntas mais freqüentes.

BLUNDELL, Phillip et al. Código-fonte do pacote NET-TOOLS - softwares de administração de rede de baixo nível para Linux.