Christiano Farina Haesbaert

OpenMdns

Porto Alegre - Rio Grande do Sul, Brasil

8 de julho de 2011

Christiano Farina Haesbaert

OpenMdns

Orientador:
Ana Cristina Benso da Silva

Pontifícia Universidade Católica do Rio Grande do Sul

Porto Alegre - Rio Grande do Sul, Brasil

8 de julho de 2011

Sumário

Lista de abreviaturas e siglas

IPv4	Internet Protocol version 4,	p. 9
GNU	GNU is Not UNIX,	p. 11
ISC	Internet Systems Consortium,	p. 12
DHCP	Dynamic Host Configuration Protocol,	p. 13
NAT	Network Address Translation,	p. 17
IPC	Inter Process Communication,	p. 35
IP	Internet Protocol,	p. 9
DNS	Domain Name System,	p. 9
TCP/IP	Internet Protocol Suite,	p. 9
MDNS	Multicast DNS,	p. 9
DNS-SD	DNS Service Discovery,	p. 9
IETF	Internet Engineering Task Force,	p. 10
FSF	Free Software Foundation,	p. 10
LGPL	GNU Lesser General Public License,	p. 11

Lista de Tabelas

Lista de Figuras

Listings

1 Introdução

Computadores e dispositivos de redes estão cada vez menores e mais portáveis, e as redes de computadores estão presentes em praticamente todo lugar, de residências a grandes corporações. Gradualmente a idéia de que cada dispositivo é um elemento estático em uma rede previamente configurada vem se tornando cada vez mais ultrapassada. Sendo assim, deseja-se comunicação entre dispositivos com a menor infra-estrutura possível, em especial, em ambientes nos quais não há um contingente técnico, como por exemplo uma residência familiar.

Por exemplo, uma funcionalidade desejável em uma rede IPv4 (*Internet Protocol version 4*) é a de mapear *hostnames* em endereços IP (*Internet Protocol*) sem a necessidade de um servidor de DNS (*Domain Name System*) (3) local ter sido previamente configurado nos *hosts* participantes da rede. Uma segunda funcionalidade que revela-se interessante é a de realizar enumeração de serviços (*Network Browsing*), para que um *host* possa pesquisar quais serviços são oferecidos na rede local. Estas são funcionalidades desejáveis em qualquer rede IPv4, seja em uma residência, escola ou corporação, portanto é importante tê-las com o mínimo esforço possível.

Pode-se enumerar então duas funcionalidades desejáveis na rede local:

1. Resolver nomes de DNS sem configuração prévia e/ou servidor central.
2. Realizar enumeração de serviços.

Ambas as funcionalidades não são fornecidas pela suite TCP/IP (*Internet Protocol Suite*), e portanto foram propostos dois protocolos que geralmente operam em conjunto: MDNS (*Multicast DNS*)(1) e DNS-SD (*DNS Service Discovery*)(2).

O **MDNS** fornece a funcionalidade 1, que é basicamente uma forma de realizar requisições de DNS via *multicast*, obtendo uma resposta também via *multicast*. Neste cenário cada *host* é um agente de MDNS autonômo não existindo um servidor central como no DNS convencional.

O **DNS-SD** fornece a funcionalidade 2 e permite que através de requisições de DNS seja possível realizar a enumeração de serviços. Ele é compatível com DNS convencional (*unicast*) e MDNS, sua popularização no entanto, deu-se pelo uso em conjunto com o MDNS.

Os protocolos MDNS/DNS-SD geralmente operam como protocolos complementares, com eles é possível por exemplo perguntar quais impressoras estão *online* ou qual o endereço IP de um *host* qualquer, ou até mesmo perguntar quais computadores possuem um serviço de compartilhamento de músicas.

Apesar de ambos os protocolos ainda possuirem estado de *Internet Drafts* junto a IETF (*Internet Engineering Task Force*), estes já possuem duas implementações bastante estáveis e utilizadas atualmente:

- **Bonjour**(4), implementação original dos protocolos, criado pela empresa Apple. Seu principal desenvolvedor foi o próprio criador dos protocolos MDNS/DNS-SD Stuart Cheshire(5), é amplamente utilizado na maioria dos produtos da empresa e distribuído juntamente com o MacOS-X. É liberado sob a licença Apache-2(6) e funciona em diversos sistemas operacionais (inclusive no Windows), possui cerca de 60000 linhas de código.

- **Avahi**(7), alternativa fornecida pela FSF (*Free Software Foundation*), fornece compatibilidade binária[1] com o Bonjour. É reconhecido como uma excelente implementação por Stuart Cheshire (criador do protocolo), funciona em diversos sistemas operacionais e possui cerca de 45000 linhas de código.

Apesar do estado maduro das implementações atuais, há três problemas significativos que atingem a ambas:

1. *Tamanho*: ambas são implementações grandes. Apesar da complexidade do problema, acreditamos que é possível desenvolver uma implementação menor, mais compacta e mais simples. Em outras palavras, acreditamos que ambos projetos possam sofrer de um sintoma conhecido como *bloat*(8)[2], porém, só poderemos concluir a existência do mesmo ao término deste trabalho.

2. *Complexidade*: acreditamos que ambas são projetos desnecessariamente complicados, embora isto só poderá ser confirmado ou não ao término deste trabalho.

[1] Compatibilidade em que uma biblioteca implementa as mesmas funções C de outra biblioteca, sendo assim trasnparente para o programa.

[2] excesso de código devido a má codificação ou excesso de funcionalidades

3. *Licença*: as licenças Apache-2(6) e LPGL (*GNU Lesser General Public License*), que licenciam o Bonjour e o Avahi respectivamente, possuem certas implicações(9) que não são aceitas em alguns sistemas operacionais, como é o caso do OpenBSD(10).

Além disto, ao analisar as alternativas existentes conclui-se que nenhuma das duas era compatível com as características e filosofias do sistema operacional OpenBSD.

O OpenBSD é um sistema operacional UNIX multi-plataforma aberto e livre. Podemos resumir e enumerar suas principais características e filosofias(10) como:

1. *Segurança*: mais que em outros sistemas o OpenBSD possue um forte foco em segurança. Muitas vezes outros atributos, como *performance* são sacrificados em prol de segurança. É reconhecido hoje como um dos sistema mais seguros do mundo.

2. *Licença*: é aceito apenas código com licenças BSD ou mais livre(9), o que não é o caso de nenhuma das implementações existentes.

3. *Código Aberto*: não são aceitos *blobs*[3] ou qualquer outro tipo de código fechado. Esta não é uma questão política e sim técnica, pois código fechado é igual a código não depurável e é visto como uma fonte de *bugs*.

4. *Simplicidade e Design*: muita atenção é dada para a simplicidade e o *design* do sistema, muitas coisas são reescritas pelo simples fato de não serem "simples e bem desenhadas" o suficiente, a crença do projeto é que simplicidade, segurança, portabilidade e robustez andam em conjunto.

5. *Portabilidade*: o sistema roda em diversas arquiteturas, portanto grande atenção é dada à portabilidade do código, fato que colabora também para a a correção dos programas, visto que nem sempre os *bugs* se manifestam igualmente em diferentes arquiteturas.

6. *Documentação*: o sistema é declaradamente *developer-friendly*. O alvo são usuários avançados e desenvolvedores, isto também colabora para que o sistema seja extensivamente bem documentado. A falta de documentação ou documentações imprecisas são considerados como *bugs*.

Sendo assim, acredita-se que ambas as alternativas de MDNS/DNS-SD existentes não atendem as expectativas do sistema OpenBSD.

[3]Binary Long Object

O objetivo deste trabalho foi fornecer livremente à comunidade do OpenBSD uma implementação de MDNS/DNS-SD sob a licença ISC (*Internet Systems Consortium*)(11). Esperava-se que futuramente, o fruto deste trabalho poderia ser incorporado ao sistema base do OpenBSD, sendo assim instalado juntamente com o sistema.

Esta implementação fornece meios para que:

1. Um programa qualquer possa realizar consultas de MDNS;
2. O *host* possa responder as consultas de MDNS feitas na rede local;
3. Um programa qualquer possa anunciar seus serviços na rede local via MDNS/DNS-SD.

Os capítulos deste texto são organizados da seguinte forma:

1. *Introdução*, esboço do problema, cenário atual e motivação do trabalho;
2. *Os Protocolos*, apresentação e detalhamento dos protocolos envolvidos, assim como uma apresentação mais detalhada de quais problemas o protocolo tenta endereçar;
3. *Projeto OpenMDNS*, apresentação do projeto realizado neste trabalho;
4. *Conclusão*, considerações finais.

2 Os Protocolos

Neste capítulo serão abordados os problemas que os protocolos MDNS e DNS-SD procuram tratar, assim como a funcionalidade oferecida pelos protocolos em acordo com os *drafts* MDNS(1) e DNS-SD(2) respectivamente.

2.1 Problema

Na atual versão do IP (IPv4), a comunicação entre *hosts* em uma rede local necessita de intervenção manual significativa, que nem sempre é viável pois necessita de uma infra-estrutura previamente configurada ou de usuários com nível técnico elevado. O protocolo IPv4 não possui nenhuma ou possui poucas funcionalidades/facilidades de autoconfiguração, e é o objetivo do MDNS/DNS-SD permitir que a comunicação em uma rede local IPv4 seja alcançada com menos configuração e maior automatização(1).

2.1.1 Endereçamento

O *Endereçamento* na rede IPv4 é numérico e não adequado para humanos, um IP do tipo 192.168.8.1 não é algo interessante de se decorar.

O problema aumenta quando este endereço é dinâmico (obtido via DHCP), pois o que se sabia antes agora pode não ser verdade. A inclusão de DHCP também introduz um outro problema: é necessário que alguém tenha configurado previamente o serviço de DHCP.

O problema do endereçamento numérico é mitigado com o DNS, que oferece uma maneira de mapear nomes em endereços, assim como outros tipos de registro, porém não se pode esperar que toda rede possua um servidor central de DNS previamente configurado e preparado para aceitar requisições por *hostnames* locais.

Em um cenário ideal, cada *host* ao ser ligado teria um nome único sem nenhum tipo

de configuração prévia ou servidor central. Com MDNS, o seguinte cenário seria possível:

- O *host frodo.local* foi ligado.
- Os outros *hosts* da rede podem agora efetuar o comando:

  ping frodo.local

  E este responde, independentemente de seu endereçamento numérico. Ou seja, o usuário não mais depende do conhecimento do IP numérico, todo *host* passa a ser endereçado por um nome único.

2.1.2 Network Browsing ou Network Discovery

Network Browsing ou *Network Discovery* é a funcionalidade que permite que serviços sejam oferecidos dinâmicamente na rede Um exemplo clássico de serviço é o de uma impressora ou servidor de arquivos. Na rede IPv4 atual, não há um mecanismo nativo para a publicação e resolução de serviços da rede local, o usuário depende do conhecimento de como acessar o serviço ou da funcionalidade de *Network Discovery*, que está embutida no serviço em questão, como por exemplo o NetBIOS(12).

Em um cenário ideal, o usuário ao ligar seu computador poderia acessar uma lista de serviços oferecidos na rede local, como por exemplo impressoras, compartilhamento de músicas e compartilhamento de arquivos entre outros. Caso este usuário tenha também serviços em seu computador, existiria um mecanismo simples que permitiria que este publicasse o serviço na rede local, seja qual for a natureza do mesmo.

2.1.3 Cenários

A seguir são apresentados três cenários hipotéticos que se beneficiariam caso as funcionalidades descritas anteriormente estivessem presentes na rede local.

2.1.3.1 IP Dinâmico

O objetivo do usuário é acessar via SSH um computador que estava desligado. O usuário sabe que este computador obtém seu IP na rede local via DHCP e que seu nome é *frodo.local*. Este computador não possui monitor e todo seu uso é feito remotamente. A rede local não possui um servidor de DNS interno para responder pelos nomes dos *hosts* locais. Assim, o cenário seria como descrito a seguir:

1. O usuário liga o computador, espera alguns segundos até que este inicie;
2. Ele tenta acessar o computador pelo IP que ele *geralmente* assume, no caso 192.168.8.25;
3. O computador não responde, o usuário assume que ele adquiriu outro IP, mas a pergunta é "qual ?". O fato de ele não saber fará com que ele tenha que conectar um cabo serial no computador ou que acesse os *logs* do servidor de DHCP, e nada garante que ele possua permissão de acesso ao mesmo.

No cenário ideal, o usuário não se importaria com o endereço, simplesmente faria o acesso pelo nome *frodo.local*, que não necessitaria ter sido previamente configurado em nenhum lugar.

2.1.3.2 Músicas na Rede Local

Usuário está no trabalho e gostaria de compartilhar suas músicas com seus colegas, portanto quer publicar um serviço do tipo *músicas* na rede local, a empresa é grande e não quer ter que avisar colega-a-colega que ele agora está disponibilizando suas músicas, o usuário quer que estes possam (ao procurar por músicas na rede) visualizar e usufruir de suas músicas.

Para que isto seja possível no cenário atual, a única esperança do usuário é que o protocolo no qual ele está disponibilizando suas músicas possua algum mecanismo de *Network Browsing* embutido, isto é, que todo a inteligência da publicação e enumeração dos serviços de músicas seja um artefato do protocolo em si.

É fácil imaginar como isto rapidamente se torna inadequado, à medida que cada tipo de serviço diferente possuiria um protocolo *específico* para prover *Network Browsing*. Por exemplo, o programa que compartilha músicas implementaria o seu protocolo, as impressoras implementariam outro, os servidores de arquivos outro e etc.

O importante é notar que o que se deseja é o *Network Browsing*, ou seja, a habilidade de enumerar serviços de diferentes tipos, esta é uma funcionalidade que pode ser genérica e não dependente de cada serviço sendo oferecido.

2.1.3.3 Storage Dinâmico

Uma empresa possui um grande *array*[1] de discos em seu *storage*[2] principal, estes discos estão localizados em diferentes dispositivos conectados por uma rede IP. A adição e remoção de discos do *array* é uma tarefa frequente, seja pela remoção de discos em falha ou pelo aumento do *array*.

Seria interessante ter uma forma de monitorar o estado e a presença dos discos, ou seja, seria interessante que cada disco assim como seu estado atual (ok, em falha etc) fosse publicado como um serviço, no caso um serviço que expressa o estado de um disco do *array*.

Este monitoramento dos recursos instáveis (no caso dos discos, que são inseridos e removidos) poderia ser feito via MDNS/DNS-SD com *Network Browsing*. Poderiam também ser tomadas atitudes sempre que um evento destes ocorre.

A idéia deste cenário foi motivado pelo *feedback* fornecido por Marco Peereboom, integrante da equipe do OpenBSD, como descrito no final deste texto.

2.2 MDNS

2.2.1 Origem

O desenvolvimento do protocolo teve origem quando a empresa Apple realizou sua transição(13) de *layer 3* de AppleTalk(14) para IPv4. Uma funcionalidade presente no AppleTalk e ausente no IPv4 era a de *Network Browsing* ou *Network Discovery*, era possível por exemplo, enumerar todos os *hosts* da rede assim como seus serviços, cada *host* era identificado por um nome, assim como seus serviços. Esta *auto-configuração* era automática e de fácil utilização.

Na tentativa de não desenvolver um protocolo totalmente novo para o IPv4, os engenheiros Cheshire e Krochmal optaram por adaptar um protocolo conhecido, no caso o DNS. É importante perceber que o DNS não é apenas um protocolo de mapeamento entre *hostnames* e IPs, ele também pode ser visto como um banco de dados distribuído armazenando virtualmente qualquer tipo de informação.

O protocolo MDNS, como proposto por Stuart Cheshire, encontra-se ainda em estado

[1] Diversos discos de armazenamento em um storage
[2] Dispositivo de grande armazenamento de dados

de *Internet Draft* e não é portanto um *RFC*. Porém pode ser considerado como funcional tendo em vista que as implementações existentes são estáveis e amplamente utilizadas.

2.2.2 Introdução

O *Multicast DNS* descrito no *draft* (1) é uma forma de se realizar *queries* de DNS via *multicast* no enlace local, de forma que não exista um servidor central previamente configurado. Os *hosts* da rede local cooperam entre si sem configuração prévia através de mensagens de *multicast* com o intuito de manter uma base de dados de DNS.

Esta base de dados pode possuir diversos tipos de registro (*Resource Records*)(3), cada uma com seu propósito específico, sendo o mais comum o *Resource Record* tipo *A*, que mapeia um *hostname* em um endereço IPv4. Assim como no DNS convencional, os mesmos *Resource Records* são utilizados e possuem o mesmo significado, isto reforça novamente a idéia de que o MDNS é apenas uma forma de se realizar *queries* de DNS via *multicast*.

A maioria dos usuários não possui um domínio em seu nome na Internet, tampouco possui a maioria dos *hosts* um IP roteável na Internet. Geralmente os *hosts* estão em uma rede privada, tendo acesso à Internet através de um único roteador fornecendo acesso através de NAT, e, portanto, mesmo que estes possuíssem um domínio para registrar seus endereços, o mesmo seria inválido, pois teria sentido apenas na rede privada local. É evidente que, é possível ter um servidor de DNS configurado na rede local e ter configurarado todos os *hosts* para adicionarem seus registros de DNS neste mesmo servidor, porém esta infra-estrutura é cara para a maioria das redes e exige configuração não trivial, sendo inviável para a grande maioria dos usuários. O MDNS fornece esta funcionalidade de forma automática e não exige nenhum tipo de configuração, pois os *hosts* são livres para adquirir um nome único válido apenas para o enlace local. Os usuário podem acessar os *hosts* da rede pelos seus nomes sem nenhum tipo de configuração prévia.

No capítulo seguinte será visto que esta automatização pode ser estendida para os serviços oferecidos de cada *host*, e este mecanismo será discutido mais detalhadamente quando for abordado o protocolo DNS-SD.

2.2.3 Domínio Multicast

O MDNS é um protocolo atuante apenas no enlace local, seu grupo *multicast* 224.0.0.251 está na faixa denominada pela IANA de *Link-Local* não sendo, portanto, roteado entre

redes IP. O protocolo atua sob a porta UDP 5353, diferentemente do DNS *unicast*, não existem diversos domínios, portanto também não há hierarquia entre os mesmos, há apenas um domínio, o *.local*. Sendo este o domínio *multicast* e possuidor de semântica especial. Portanto nenhum *host* é detentor de domínio algum, no entanto, os *hosts* podem ser detentores de nomes únicos, que serão abordados ao longo do texto. Quando um *host* que suporta MDNS realiza uma *query* para o domínio *.local*, esta *query* deve ser feita via MDNS, pois este é o domínio *multicast*. Um *host* implementando apenas DNS convencional enviaria a *query* para seu servidor de DNS previamente configurado, provavelmente não retornando resposta.

É importante ressaltar que o MDNS pode ser utilizado para resolver nomes globais (como no DNS *unicast*). Para isto, todas as requisições de DNS, independente do domínio, precisam ser enviadas via *multicast*. Neste caso, pelo menos um *host* no enlace local precisa fazer a ponte[3] entre o DNS *unicast* (consultando os *Root Servers*) e o MDNS. Este não é o funcionamento natural e seu tratamento é melhor descrito no *draft* MDNS(1).

2.2.4 Resource Records

Assim como no DNS convencional, o MDNS utiliza *Resource Records*, que são os registros de DNS e são descritos no RFC1035(3). Outra diferença bastante significativa para o DNS convencional é a classificação dos *Resource Records* em dois tipos:

- *Unique*: *Resource Record* onde a tripla DNAME/TYPE/CLASS(3) é única e apenas um *host* responde por este *Resource Record*.
- *Shared*: *Resource Record* onde uma *query* pode elicitar múltiplas respostas. E cada *host* pode responder por um ou mais *Resource Record* do tipo *shared*.

Um *Resource Record unique* é desejável por exemplo no mapeamento de *nome IP*, onde deseja-se que uma *query* pelo nome *foobar.local* do tipo *A* (IPv4 Address) retorne apenas uma resposta. Seria estranho se um nome mapeasse para dois *hosts* distintos. Outro exemplo bastante comum é o *Resource Record* de tipo *HINFO* (Host Information), usado para mapear um nome para um par de *strings* CPU/Arquitetura.

Para que um *host* possua um *Resource Record unique*, ele precisa primeiro verificar a unicidade do nome em questão, para isto ele precisa realizar uma etapa chamada de *Probing*, em que um *host* certifica-se que o nome em questão não está sendo utilizado por

[3]Replicar os dados os pacotes *multicast* entre duas interfaces

nenhum outro *host*. Depois deste passo, o *host* pode anunciar qualquer *Resource Record* sob este nome, podendo por exemplo anunciar *Resource Records* de tipo *A* e *HINFO*. A etapa de *Probing* será abordada em mais detalhe ao longo do texto.

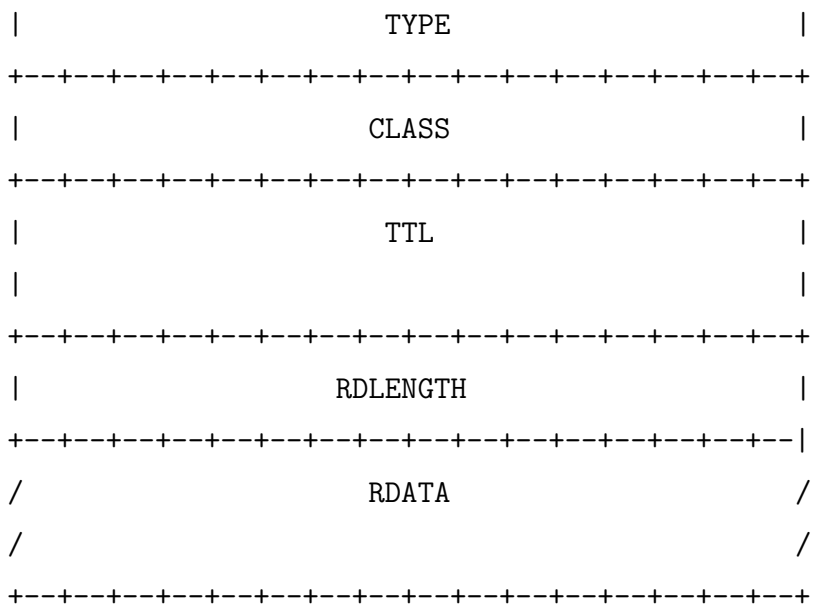
No caso do *Resource Record* do tipo *shared*, não é verdade que um *host* é o único possuidor do mesmo, isto é, cada *host* pode possuir zero ou mais instâncias do mesmo, cada um com seus respectivos RDATA(3). Este tem sido utilizado apenas para o DNS-SD, onde por exemplo pode-se requisitar uma lista de todos os servidores de *HTTP*.

2.2.5 Queries

No MDNS existem três tipos de *query*. Estas, assim como suas respectivas respostas possuem o mesmo formato do DNS *unicast* convencional como descrito no RFC 1035(3):

```
Question contida em uma Query
    1  1  1  1  1  1
    0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                                               |
    /                     QNAME                     /
    /                                               /
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     QTYPE                     |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     QCLASS                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+


Resource Record contido em uma resposta (Answer).
    1  1  1  1  1  1
    0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                                               |
    /                                               /
    /                      NAME                     /
    |                                               |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```



Apesar do mesmo formato, um *Querier* de MDNS deve sempre enviar suas requisições com a porta origem UDP 5353, ao contrário dos clientes *unicast* onde a porta origem é efêmera.

2.2.5.1 One-Shot

Neste tipo de *query* é enviada apenas uma pergunta e esperado apenas uma resposta. Caso sejam elicitadas múltiplas respostas, o cliente leva em conta apenas a primeira, desconsiderando as outras.

Este é o único tipo de *query* que pode ser utilizado por um cliente DNS *unicast* convencional e é chamada de *Legacy Query* pois pode ser feita por clientes legados (cliente DNS convencional). Todas as *queries* no MDNS devem ser feitas com a porta origem UDP 5353, mas como os clientes legados utilizam uma porta origem UDP efêmera, um MDNS *Responder* é capaz de identificar tal condição e enviar uma resposta *unicast* ao invés da resposta *multicast* padrão. Desta forma, o cliente consegue processar a resposta, o que não conseguiria caso esta tivesse sido enviada para o endereço *multicast*.

O suporte a *Legacy Queries* é interessante à medida que permite interoperabilidade com clientes legados com o mínimo de esforço.

A seguir é apresentado um exemplo de *One-Shot query* convencional (não legada), onde supõe-se que o *host bilbo.local* deseja descobrir o endereço IPv4 de *frodo.local*:

1. O *host bilbo.local* envia uma *query* com uma *question* com os atributos {QNAME=*frodo.local*,

QTYPE=A, QCLASS=IN}. Ele decide esperar por uma resposta em até 3 segundos, caso contrário *bilbo.local* assume que não existe resposta para o *Resource Record* desejado na rede.

2. O *host frodo.local* recebe a *query*, analisa a porta origem e determina que esta irá elicitar uma resposta *multicast*. Visto que a porta origem é a porta MDNS (5353), *frodo.local* verifica que possui o *Resource Record* em questão e envia uma resposta com os atributos {NAME=*frodo.local*, TYPE=A, CLASS=IN, TTL=120, RDLENGTH=4, RDATA=192.168.8.21}.

3. A resposta chega no *host bilbo.local* antes de seu *timeout* de 3 segundos, este processa a resposta e dá como encerrada a requisição.

O atributo TTL[4] escolhido como 120 segundos na resposta da *query* é o recomendado no *draft* MDNS para *Resource Record* de tipo A, ele especifica que a asserção é válida por este tempo, todos os *hosts* que receberam a resposta devem armazenar em seu *cache* o *Resource Record* por 120 segundos. Sempre que um *host* considera realizar uma *query* ele primeiro consulta seu *cache* a fim de evitar tráfego desnecessário na rede. No caso de uma *One-Shot query*, se a informação já existe no *cache*é suficiente para que nenhuma *query* seja enviada.

O RDLENGTH corresponde ao tamanho de um A record (4 bytes) e o RDATA é o endereço IP em questão(3).

2.2.5.2 One-Shot, Accumulating Multiple Responses

Este tipo de *query* é semelhante ao anterior, porém, o cliente aguarda alguns segundos e acumula possíveis múltiplas respostas. É importante notar que só haverão múltiplas respostas caso o *Resource Record* seja *shared*, pois do contrário só pode existir necessariamente um *Resource Record*.

Este tipo de *query* é raramente utilizada e só faz sentido no uso de DNS-SD, sendo sua funcionalidade suprida pelo tipo *Continuous Querying* descrito a seguir.

2.2.5.3 Continuous Querying

São enviadas *queries* com intervalos crescentes. No *draft* MDNS é recomendado que se dobre o intervalo a cada *query* para evitar o excesso de mensagens na rede. Sendo o

[4]Time To Live

primeiro intervalo em 1 segundo. Todas as respostas recebidas são acumuladas. Assim como no tipo anterior, este tipo de *query* é relevante quando se deseja (no DNS-SD) enumerar os serviços de um determinado tipo disponibilizados na rede.

A seguir um exemplo onde o *host bilbo.local* deseja enumerar todos os servidores de NTP da rede local:

1. O *host bilbo.local* envia uma *query* com os seguintes atributos {QNAME=_ntp._udp.local, QTYPE=PTR, QCLASS=IN}. Ele irá reenviar esta *query* dobrando o intervalo de tempo a cada envio, sendo o primeiro intervalo de 1 segundo e considerando T0 como o tempo inicial, será enviada uma *query* em T0, T0 + 1, T0 + 2, T0 + 4, ..., T0 + n, até o limite de uma hora.

2. Na rede local existem 2 serviços de NTP, um no *host gandalf.local* e outro no *host frodo.local*. Os nomes do serviço NTP são *The Valar Clock NTP Server* e *The Shire Clock NTP Server* respectivamente. Ao receber a *query* enviada por *bilbo.local* *gandalf.local* responde com {NAME=_ntp._udp.local, TYPE=PTR, CLASS=IN, RDATA=*The Valar Clock NTP Server*, ...} e o *host frodo.local* com {NAME=_ntp._udp.local, TYPE=PTR, CLASS=IN, RDATA=*The Shire Clock NTP Server*, ...}. Portanto é notar que são elicitadas duas respostas do tipo PTR, e eles apontam para o nome de seus respectivos serviços. O funcionamento da enumeração e resolução de serviços será melhor detalhado no capítulo que aborda o protocolo DNS-SD.

3. O *host bilbo.local* recebe as duas respostas e agora sabe que existem dois serviços do tipo NTP na rede. A razão para o envio de múltiplas *queries* deve-se ao fato de que o pacote pode não ter atingido todos os *hosts* da rede, um *switch* pode ter descartado o pacote devido à sobrecarga, ou pode ter ocorrido uma colisão no caso de uma rede *half-duplex*. Assim os envios subsequentes diminuem a chance de falha.

2.2.6 Cache

Outro aspecto importante do MDNS é a necessidade de se manter o estado de *queries* passadas, ao contrário do cliente *unicast* padrão (toma-se de exemplo a implementação da LIBC do OpenBSD), onde é possível a cada chamada de *getaddrinfo(3)*(15) ou *gethostbyname(3)*(16) criar um novo *socket*, enviar a requisição e aguardar por um *timeout*.

No MDNS o estado das *queries* passadas deve ser armazenado em um *cache*, que deve ser consultado por futuras respostas. A simples presença de um *cache* complica bastante

o protocolo como um todo, pois ele adiciona diversos *timers* e um protocolo de coêrencia de *cache*.

Cada *Resource Record* possui um TTL (*Time To Live*) que indica por quantos segundos aquela entrada deve ser considerada válida. No caso de *shared records* o *host* que está realizando a *query* deve primeiro consultar seu *cache*, procurando por possíveis respostas, e então enviar sua *query*. Com a *query* são incluidas todas as respostas previamente conhecidas, como descrito adiante na seção *Known Answer Supression*.

Este *cache* deve conter apenas os *Resource Records* propagados pela rede, ou seja, os registros vistos pelo *host* ao longo do tempo, e devem ser deletados do mesmo a medida que seu TTL expira.

O *draft* recomenda que um *host* tente renovar seus registros quando estes estão perto de expirar (ao término do TTL). Para isto, é recomendado que se realize uma *query* pelo *Resource Record* aos 80%, 90% e 95% do tempo de vida. Esta renovação deve ser evitada caso não exista interesse no *Resource Record*.

Os requisitos completos para coerência de *cache* e mecanismos para evitar um tráfego desnecessário são bastante complexos e serão abordados ao longo do texto.

2.2.7 Known Answer Suppression

O *Known Answer Suppression* (KNA) é um mecanismo para evitar a elicitação de respostas a *queries* nas quais já se sabe algumas das respostas, portanto, o KNA é aplicável apenas a *queries* feitas por *shared Resource Records*, visto que apenas eles podem elicitar mais de uma resposta para a mesma *query*.

Um *host* ao realizar uma *query* das quais já possui algumas respostas, deve incluír-las na *Answer Section*(3) do pacote. Um outro *host*, ao receber a *query*, não a responderá se as respostas que seriam feitas já estão listadas no *Answer Section* da *query*.

Esta funcionalidade é bastante útil quando se realizam *Continuous Queries*, onde uma *query* é enviada periodicamente, pois seria um tráfego desnecessário e inconveniente se a cada reenvio fossem enviadas as mesmas respostas.

2.2.8 Probing & Announcing

Quando um *host* deseja publicar um de seus *Resource Record* na rede, caso este seja do tipo *unique* ele precisa primeiro realizar uma etapa denominada de *Probing*. Caso o

Resource Record seja *shared*, esta etapa pode ser ignorada e passa-se para outra etapa denominada de *Announcing*.

No *Probing* o *host* adquire um nome único para um conjunto de *Resource Records* sob os quais ele deseja publicar. Para que isto ocorra, o *host* primeiro verifica que nenhum outro *host* da rede é detentor do nome desejado, podendo então partir para a etapa de *Announcing*.

2.2.9 Resolução de Conflitos

Um conflito surge quando dois *hosts* desejam publicar algum *Resource Record* único sob o mesmo nome. Quando um *host* recebe uma resposta com um *Resource Record* conflitante, este deve reiniciar o processo de *Probing*, o algoritmo de resolução de conflitos é acionado na fase de *Probing* como descrito no *draft*(1).

Ao fim da resolução de conflito, o *host* perdedor deve escolher um outro nome para o seu *Resource Record* e reiniciar o processo de *Probing*. Caso exista um novo conflito os passos são repetidos.

2.3 DNS-SD

O *DNS Service Discovery* descrito no *draft*(2) é um protocolo baseado no DNS que fornece a funcionalidade de *Network Discovery* ou *Network Browsing*. Apesar de o DNS-SD ser compatível com o DNS convencional (*unicast*), sua popularidade e uso deu-se em conjunto com o MDNS, pois o DNS-SD foi projetado em paralelo ao MDNS e pelos mesmos criadores.

O protocolo consiste em uma convenção de nomes e registros de DNS que devem ser utilizados de forma que seja possível realizar duas operações básicas:

1. *Enumeração de Serviços*
2. *Resolução de Serviços*

Na **Enumeração de Serviços**[5] utilizam-se *Resource Records* do tipo *PTR*(3) de forma análoga aos diretórios de um sistema de arquivos. Para cada serviço da rede local é publicado um registro do tipo *PTR* que aponta para a instância do serviço. O nome do

[5]também chamado de *browsing*

registro *PTR* é o protocolo do serviço. Supõe-se que existam dois serviços do protocolo HTTP, portanto haveria dois registros do tipo *PTR* na seguinte disposição:

```
_http._tcp.local --> paginaslegais._http._tcp.local
_http._tcp.local --> paginaschatas._http._tcp.local
```

Pode ser feita uma analogia com a organização dos arquivos em um diretório, onde o diretório seria o protocolo de serviço (*_http._tcp.local*) e os arquivos seriam as instâncias do serviço. O caracter "_" é usado como separador entre nomedoserviço_aplicação_protocolo.

É importante perceber que uma *query* pelo nome do protocolo de serviço elicita múltiplas respostas, este é o caso clássico de *shared Resource Records* utilizados no MDNS.

Na enumeração de serviços há também um nome especial, no caso o *_services._dns-sd._udp.local*, que funciona como um diretório superior ao dos protocolos de serviço e fornece uma lista dos protocolos de serviço em questão, portanto, é possível fazer uma enumeração recursiva da seguinte forma:

1. Faz-se uma *query* do tipo *PTR* pelo nome *_services._dns-sd._udp.local*. O resultado serão diversos *Resource Records* que apontam para os nomes de protocolo de serviço.

2. Faz-se então uma nova *query* do tipo *PTR* para cada protocolo de serviço recebido na *query* anterior.

A partir da *Enumeração de Serviços* é possível obter uma lista dos serviços ativos, e é possível que se realize a **Resolução de Serviços**. Para que isto ocorra, cada serviço publica, em adição ao registro *PTR*, mais dois registros adicionais: *TXT*(3) e *SRV*(3).

Estes registros são publicados sob o nome para o qual o registro *PTR* aponta (o nome do serviço em questão), no caso do exemplo anterior o nome de um dos serviços seria *paginaslegais._http._tcp.local*. Portanto, para realizar a resolução do serviço *paginaslegais._http._tcp.local* seria feita uma requisição pelos registros de tipo *SRV* e *TXT*.

O registro de tipo *SRV* especifica os seguintes atributos:

- *Priority e Weight*, a prioridade e peso do serviço, que possibilitam que seja possível publicar diversos serviços semelhantes e escolher apenas aquele com maior prioridade (menor valor). É pouco ou quase nunca utilizado no MDNS.

- *Port*, a porta do serviço em questão. No caso do HTTP, provavelmente seria 80. O protocolo de transporte está implícito no nome, no caso TCP para *paginaslegais._http._tcp.local*.

- *Target*, o nome do *host* que detém o serviço, o que possibilita que um *host* publique serviços que estão situados em outros *hosts*.

O registro de tipo *TXT* especifica atributos adicionais dependentes do protocolo do serviço. No caso do HTTP poderia ser por exemplo um nome de usuário para *login*, a versão do protocolo HTTP utilizada ou qualquer outra informação pertinente ao protocolo. Esta informação é dada como uma lista de tuplas que mapeiam *chave:valor*.

3 Projeto OpenMDNS

O nome escolhido para o projeto foi OpenMDNS seguindo a linha de outros projetos parceiros do OpenBSD, como OpenBGP(17), OpenOSPF(18) e OpenSSH(19).

Neste capítulo será abordado o projeto do OpenMDNS assim como serão explicados e apresentados alguns conceitos importantes para a compreensão do projeto, tais como *ZeroConf*, Licenças, OpenBSD e um breve resumo das implementações existentes. Logo em seguida serão discutidos os requisitos do projeto assim como a arquitetura proposta.

3.1 Introdução e Conceitos

Neste trabalho é proposto o desenvolvimento de uma implementação dos protocolos MDNS/DNS-SD para o sistema operacional OpenBSD. Atualmente o sistema possui um *port*[1] do Avahi, e com algum esforço seria possível portar o Bonjour para o sistema, porém é de nosso interesse criar uma nova implementação.

3.1.1 ZeroConf Networking

ZeroConf é o nome dado a uma série de técnicas e funcionalidades que fornecem um maior nível de auto-configuração para uma rede IPv4, técnicas que permitem que os usuários finais possam interconectar seus computadores e seus dispositivos de rede, tais como impressoras e afins. Sem ZeroConf, uma rede IPv4 necessita de considerável intervenção manual para que possa operar razoavelmente bem, sendo necessários diversos serviços que exigem uma configuração não trivial, como por exemplo:

- DHCP: sem um servidor de DHCP na rede, os usuários devem configurar seus IPs manualmente. Com DHCP, deve ser feita configuração a do servidor DHCP.

[1] Distribuicao de um programa UNIX

- DNS: para que um computador possa endereçar a outro em uma rede local por um nome simbólico (*hostname*), é necessário que seja configurado um servidor de DNS local, o que também não é trivial. Sem isto, cada usuário precisa comunicar pessoalmente seu endereço IP aos outros usuários que desejam utilizar algum tipo de serviço de seu computador.

- Impressoras e afins: a configuração de recursos como impressoras em uma rede IPv4 é bastante trabalhosa, existem muitas variáveis envolvidas tais como o endereço da impressora, o driver, o protocolo de comunicação utilizado, etc.

O ZeroConf tenta abordar as questões acima com três mecanismos distintos:

1. *Link-Local Address Autoconfiguration*: protocolo bastante simples que permite a um *host* assumir um IP reservado qualquer na rede sem que seja necessário nenhum tipo de configuração. A seguir é apresentado o funcionamento básico deste protocolo:

   (a) Seleciona um IP randômico em um intervalo reservado pela IANA para ser usado em redes locais.

   (b) Verifica através de *ARP Requests* que nenhum outro *host* responde por este endereço, caso sim, volta para *(a)* e seleciona outro endereço.

   É importante ressaltar que no RFC 3927 há um requisito que este protocolo só entre em funcionamento após uma falha confirmada da obtenção de um IP via DHCP, portanto este é um protocolo que funciona sem nenhuma configuração prévia (sem servidor de DHCP).

2. Resolução de *hostnames* automática na rede local, ou seja, fornecer as funcionalidades tradicionais do DNS, porém sem nenhuma configuração prévia e/ou servidor central. Esta funcionalidade é fornecida pelo MDNS.

3. *Network Discovery/Network Browsing*: possibilitar a realização da enumeração de serviços e recursos na rede, sem que seja necessário uma prévia configuração do mesmo. O exemplo clássico é de uma impressora. No caso ideal, o usuário ligaria seu computador e automaticamente teria acesso a todas as impressoras da rede, assim como a todos os outros serviços fornecidos pelos computadores da rede, tais como: servidores de arquivo (NFS/CIFS/FTP...) ou sincronizadores de relógio (NTP) e etc. Esta funcionalidade é fornecida pelo DNS-SD.

O ZeroConf é, portanto, a união destes três mecanismos. Neste trabalho propõe-se tratar dos últimos dois, o MDNS e o DNS-SD. O suporte ao primeiro foi rejeitado pois este seria desconexo do restante do trabalho, apesar destes protocolos fornecerem uma suite de serviço, estes não tem muito em comum. No futuro deseja-se implementar a primeira funcionalidade no cliente *dhclient*(20) do OpenBSD.

3.1.2 OpenBSD

O OpenBSD é um sistema operacional UNIX multi-plataforma oriundo de uma ramificação do NetBSD(21) em 1995 feito pelo seu fundador Theo De Raadt. O sistema é um direto descendente do BSD 4.4 desenvolvido em Berkeley na Califórnia, uma das variantes mais populares do UNIX de 1968.

O sistema é inteiramente grátis e o código é aberto, sendo licenciado em maior parte sob as licenças BSD e ISC e não possui fins lucrativos, todo o desenvolvimento é feito por voluntários e entusiastas. O projeto se mantém através da venda de CDs (estes que também são disponibilizados gratuitamente), o comprador paga na verdade pelos diversos adesivos e pelo encarte do CD. Este dinheiro é então utilizado para financiar eventos e o custo dos desenvolvedores oficiais do projeto, tais como as passagens para eventos, etc.

Como informado no site(10) do projeto os ideais do projeto são:

> *Our efforts emphasize portability, standardization, correctness, proactive security and integrated cryptography.*

Todo o código do sistema é extensivamente auditado e revisado por falhas mesmo que mínimas ou até mesmo de documentação, esta revisão é feita por diversos membros do projeto, lendo o código e efetuando testes. Todo código deve ser portável e correto, mesmo que isto tenha impacto na performance do mesmo. O OpenBSD é reconhecido hoje como possivelmente o sistema operacional mais seguro no mundo, e seu fundador, Theo De Raadt é tido como um dos maiores *experts* de segurança no mundo. Outro ponto bastante relevante do projeto é sua seriedade quanto a documentação. Mais que em outros sistemas, é uma obrigação possuir sempre uma documentação acessível e atualizada.

Os desenvolvedores do OpenBSD prezam por implementações compactas e seguras, rejeitando aplicações desnecessariamente grandes e poluídas com funcionalidades não expressivas. É da crença do autor que tanto Avahi e Bonjour se enquadram nestas definições, tendo o primeiro cerca de 60 mil linhas de código e o segundo algo com pouco menos de

50 mil linhas.

O OpenBSD trouxe algumas implementações utilizadas no mundo inteiro, tais como o OpenSSH, PF, OpenBGPD, OpenOSPFD, OpenRIPD. Todas estas partilham dos princípios descritos acima.

3.1.3 Licenças

O OpenBSD é bastante restrito quanto as licenças do código presente no sistema base, para que isto seja compreendido precisa-se entender como é feita a divisão entre sistema base e *third-party* (aplicativos disponibilizados via *ports* ou pacotes).

O sistema base (ou apenas base) compreende as aplicações base do sistema, desde os programas e utilitários em espaço de usuário ao *kernel* e *drivers*. É o sistema base que define na verdade o que é o OpenBSD, toda a base é auditada e mantida em grande parte pelos desenvolvedores do OpenBSD, é apenas nela que atuam as restrições de qualidade e política (licenças). É política do OpenBSD aceitar apenas código na base que possua licença equivalente ou mais-livre-que a licença BSD (salvo raras exceções(9)), sendo a licença ISC a preferida em código novo.

Os *ports (third-party)*(22) são pacotes disponibilizados de fontes diferentes. Ao contrário do sistema base, eles não possuem nenhum tipo de restrição, pois estes não fazem parte do OpenBSD e são mantidos por diversas pessoas ao redor do mundo que não possuem vínculo algum com as filosofias e políticas do OpenBSD. O ponto importante a ser compreendido é que o *ports* são apenas um mecanismo com o fim de facilitar a instalação de programas de terceiros.

O Avahi possui licença LGPL e o Bonjour possui licença Apache-2, ambas **não** são aceitas no OpenBSD. Sendo assim, o fruto deste trabalho deverá ser livremente distribuído sob a licença ISC, pois espera-se que este possa, ao término de sua implementação, ser aceito no sistema base do OpenBSD.

A seguir é apresentada uma cópia da licença BSD e ISC. Em 22 de julho de 1999 a Universidade de Berkeley rescindiu a terceira cláusula da licença BSD.

Licença BSD original (4 cláusulas):

Licença ISC

```
/*
* Copyright (c) CCYY YOUR NAME HERE <user@your.dom.ain>
*
* Permission to use, copy, modify, and distribute this software for any
* purpose with or without fee is hereby granted, provided that the above
* copyright notice and this permission notice appear in all copies.
*
* THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
* WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
* MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
* ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
* WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
* ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
* OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/
```

3.2 Projeto

Se o objetivo deste trabalho é fornecer uma implementação de MDNS/DNS-SD para o OpenBSD. Para que isto ocorra, deverão ser implementadas as funcionalidades pendentes do OpenMDNS. O projeto teve início antes deste trabalho, em Fevereiro de 2010.

Esta implementação será distribuida livremente para a comunidade e usuários do sistema OpenBSD, ficando assim disponível para que qualquer pessoa possa se beneficiar dele como bem entendê-lo. Por isto foi escolhida a licença ISC que impõe nenhuma ou quase nenhuma restrição ao uso do *software*.

Pode-se enumerar as funcionalidades básicas da implementação:

1. Fornecer meios para que um programa possa realizar consultas de MDNS, ou seja, deve existir alguma biblioteca ou integração com o sistema de modo que os aplicativos existentes se beneficiem do mesmo.

2. Fornecer meios para que o *host* responda as requisições de MDNS na rede.

3. Fornecer meios para que um programa possa publicar seus serviços via MDNS/DNS-

SD, ou seja, cada programa deve ter a opção de publicar algum serviço.

3.2.1 Requisitos

A seguir são apresentados os requisitos elaborados para atender a implementação. Estes requisitos surgiram a partir do estudo das implementações atuais, das características e expectativas do sistema OpenBSD, assim como da experiência do autor.

- *Tamanho*: a implementação deve ser menor e mais compacta que as existentes, logo espera-se atingir este requisito com um *design* adequado e com a não inclusão de funcionalidades supérfluas.
- *Simplicidade*: o código deve ser simples, pois a simplicidade deve ser favorecida sempre que possível, otimizações de performance só deverão ser consideradas quando provado que a solução mais simples não é suficiente.
- *Portabilidade*: a implementação deve funcionar em todas as arquiteturas suportadas pelo sistema OpenBSD. O autor não dispõe de acesso a todas as arquiteturas, portanto, testes serão feitos apenas nas arquiteturas: *i386, amd64, alpha, sparc64*. Espera-se que a comunidade realize testes nas arquiteturas restantes. O suporte a múltiplas arquiteturas colabora para a localização de *bugs*, assim como garante que o código está correto quanto a questoes como alinhamento e *endianness*[2].
- *Múltiplas Interfaces*: a implementação deve levar em conta o caso de um *host multihomed*[3], para isto, deve haver suporte para múltiplas interfaces de rede. Deve existir uma opção para habilitar e desabilitar estas interfaces.
- *Interoperabilidade*: deve existir interoperabilidade com Avahi e Bonjour. Para que isto ocorra deve-se atentar para a correta interpretação dos *drafts*.
- *Ferramenta de debug*: deve existir uma ferramenta básica de *debug* que permite operações simples como realizar consultas, publicar serviços e observar o estado interno do *software*.
- *Integração com LIBC*: deve existir uma opção para habilitar a resolução de nomes do domínio *.local* via MDNS, para isto é preciso integração com as chamadas da biblioteca *gethostbyname(3)(16)*, e *getaddrinfo(3)(15)*. Isto fará com que qualquer programa possa se beneficiar automaticamente da resolução de nomes via *multicast*.

[2]Ordem dos bytes em uma palavra.
[3]Um host com duas ou mais interfaces de rede

- *Segurança*: o código deve condizer com as políticas de segurança do sistema OpenBSD, tais como separação de privilégio(23) e *chroot*(23).

- *Dependências*: a implementação não deve depender de nenhuma biblioteca ou ferramenta que não esteja presente na base do sistema OpenBSD, fazendo com que caso o OpenMDNS seja aceito um dia como parte integrante do sistema, esta integração seja possível.

3.3 Arquitetura

Na *Figura 1* é apresentado um esboço da arquitetura do OpenMDNS, que é composta por três componentes base:

1. **mdnsd**
2. **libmdns**
3. **mdnsctl**

O **mdnsd** é um *daemon*, portanto um programa em espaço de usuário sem terminal controlador, responsável por toda a implementação dos protocolos MDNS/DNS-SD. Ele responde as requisições de MDNS oriundas da rede local assim como efetua todas as requisições feitas pelos outros programas do *host* local. As responsabilidades do **mdnsd** podem ser melhor visualizadas a partir de seus três pontos de entrada:

1. *Socket MDNS*: é o *socket* que opera na porta UDP 5353, ele escuta por requisições de MDNS da rede local e as responde de acordo.
2. *Socket de Controle*: é o socket que recebe as requisições realizadas pelos programas locais utilizando as funcionalidades de MDNS/DNS-SD.
3. *kernel Routing Socket*: é um *socket* da família AF_ROUTE para receber as notificações de alteração de estado das interfaces locais, tais como *link-up/link-down*, mudança de endereço e afins.

A **libmdns** é uma biblioteca (*shared library*[4]) que fornece a API para que os programas realizem requisições ao **mdnsd**. Esta API implementa uma série de funções para

[4]Biblioteca vinculada dinamicamente no momento da execução do programa

comunicação com **mdnsd** através de seu *socket* de controle. Quando um programa deseja resolver um nome via MDNS, por exemplo, ele utiliza funções da biblioteca que enviam uma mensagem para o *socket* de controle e aguarda por uma resposta, enquanto o **mdnsd** fará o que for necessário para atender a requisição e enviará uma resposta ao requerente.

O **mdnsctl** é um programa com interface em linha de comando para *debug* e utilização simples do MDNS/DNS-SD, este é basicamente uma interface para a utilização das funções da biblioteca, com ele pode se realizar uma consulta MDNS, enumerar serviços ou publicar um serviços na rede local, assim como outras funções como *dump* do *cache*.

A programação do **mdnsd** é orientada a eventos e não há nenhum tipo de concorrência, portanto não há múltiplas *threads* de controle. *Threads* são geralmente uma fonte de *bugs* e um complicador de depuração(24) e nesta implementação não há nada no problema que justifique a possível perda do determinismo pela adição de *threads*, sendo que todo processamento é baseado no tratamento de eventos, como por exemplo uma mensagem em algum *socket* ou a ocorrência de um *timeout*. É utilizada a biblioteca *libevent*(25) criada e desenvolvida por *Niels Provos*, que fornece toda a abstração necessária de eventos. No seu funcionamento normal, registram-se *callbacks* que deverão ser chamadas caso cada evento ocorra, não há concorrência e todo processamento é feito no processo corrente. A *libevent* é na verdade uma abstração dos mecanismos de *multiplexing* de entrada e saída do UNIX como por exemplo *poll(2)*(26) e *select(2)*(27).

A comunicação feita pelo *socket* de controle, ou seja, o *socket* utilizado pela biblioteca **libmdns** para comunicação com o **mdnsd** é feita através do *framework* de IPC (Inter Process Communication) IMSG(28), criada e desenvolvida por *Henning Brauer*. É um *framework* bastante simples sendo basicamente uma abstração para se enviar e receber estruturas de dados diretamente, todo o *buffering* das mensagens é feito pelo próprio *framework*.

Os eventos enviados pelo **mdnsd** são assíncronos. Isto é necessário por exemplo, quando se deseja realizar a enumeração de serviços de um determinado protocolo, quando um programa qualquer solicita a enumeração, e à medida que novos serviços vão sendo aprendidos pelo **mdnsd**e uma nova notificação é enviada ao programa, portanto, tal comunicação precisa ser assíncrona.

3.3.1 Implementação

A seguir descrevemos os pontos chave da implementação do OpenMDNS, começamos descrevendo o processo de inicialização do **mdnsd** e a implementação, tomando como ponto de partida os casos de uso e as mensagens recebidas do protocolo de outros equipamentos. Explicaremos as estruturas de dados do código à medida em que as encontramos.

O **mdnsd** é um *daemon*, ou seja, um programa sem terminal controlador que executa em modo *background*. Caso o programa seja iniciado em modo de *debug*, o **mdnsd** é executado em *foreground* e diversas mensagens de *debug* são impressas no terminal.

A seguir, enumeramos os principais passos executados pelo **mdnsd** em sua inicialização:

1. *Verificação dos argumentos.* Identifica as opções passadas na linha de comando via *getopt(3)*.

2. *Confirmação de privilégios.* Confirma que o programa foi iniciado pelo super usuário, aborta inicialização caso contrário.

3. *Confirmação de usuário.* Confirma que existe o usuário *_mdnsd* no sistema, aborta caso contrário.

4. *Abertura do socket de controle.* Inicializa uma lista de todos os controladores conectados e vincula o *socket* no caminho */var/run/mdnsd.sock*.

5. *Execução de chroot(2).* Altera o diretório raiz do processo com a chamada de sistema *chroot(2)*.

6. *Revogação (Drop) de privilégios.* Altera o *euid* (*Effective User Id*) do processo para o usuário *_mdnsd*, este que não possui nenhum privilégio.

7. *Inicialização da biblioteca libevent e dos tratadores de sinais (UNIX signals).*

8. *Abertura do* socket *com o kernel.* Utilizado para o recebimento de notificação de eventos das interfaces de rede.

9. *Inicialização de sub-sistemas.* Aloca e inicializa as estruturas do *cache*, *publisher* e *querier*.

10. *Abertura do socket MDNS.* Inicializa o *socket* UDP na porta 5353 para a comunicação com os outros *hosts* no grupo *multicast* de MDNS.

11. *Registro de eventos.* Registra os três principais eventos de leitura na *libevent*: a leitura do *socket* de controle, do *socket* com o kernel e do *socket* MDNS. Também são registradas funções que tratam estes eventos.

12. *Função principal event_dispatch().* O *mainloop* do **mdnsd** é uma função da *libevent* que não retorna nunca, é ela quem chama as funções registradas anteriormente para cada evento.

A revogação de privilégio altera o EUID e EGID (*Effective User ID, Effective Group ID*) do processo para um usuário desejado e só pode ser executada pelo super-usuário (por isso verificamos se fomos iniciados pelo super-usuário). Tal funcionalidade é atingida com três funções: *setgroups(2), setresgid(2), setresuid(2).* O usuário que utilizamos é o *_mdnsd*, que é um usuário sem *login* com seu *HOME* em um diretório vazio. A seguir demonstramos um exemplo deste usuário com sua entrada no */etc/passwd*:

```
_mdnsd:*:777:777:MDNS Daemon:/var/empty:/sbin/nologin
```

Esta funcionalidade é importante para a segurança do **mdnsd**. Com ela, garantimos que em um eventual ataque em que seja explorada alguma falha de segurança do programa, como um *buffer overflow* com inserção de *shell* na máquina atacada e chegando a um eventual *login*, o atacante possua o menor privilégio possível. Neste caso, o usuário *_mdnsd* não possui nenhum privilégio, ou seja, ele não pode fazer praticamente nada no sistema. Se isto não fosse feito, o atacante privilégios de super usuário no sistema, podendo realizar **qualquer** operação.

O *chroot(2)* é mais uma funcionalidade de segurança e amplia a segurança fornecida pela revogação de privilégios. Ela altera a visão de diretório raiz do sistema para o diretório especificado no processo corrente, ou seja, podemos alterar a raiz do sistema para um diretório qualquer, sem acesso a nada. No caso utilizaremos o *home* do usuário *_mdnsd* (*/var/empty*). No eventual acontecimento do ataque descrito anteriormente, não só o usuário não teria privilégio para alterar qualquer informação, como tampouco listar os diretórios do sistema, pois ele está *preso* a um diretório vazio. Ele não pode executar um programa simples, nem sair do diretório *HOME* (*/var/empty*). Ambas as funcionalidades foram criadas e popularizadas pela equipe do *OpenBSD*.

Ao término da inicialização temos alguns dos dados principais do programa na estrutura global *mdnsd_conf* 3.1:

Listing 3.1: struct mdnsd_conf

```
struct mdnsd_conf {
        LIST_HEAD(, iface)  iface_list; /* Our interface list */
        int                 mdns_sock; /* MDNS socket bound to udp 5353 */
        struct event        ev_mdns;   /* MDNS socket event */
        struct hinfo        hi;        /* MDNS Host Info */
        char                myname[MAXHOSTNAMELEN]; /* Hostname */
        int                 no_workstation; /* Don't publish workstation */
};
```

Neste ponto, o **mdnsd** está pronto para aceitar requisições de outros programas, atender requisições dos controladores e participar ativamente da rede MDNS.

3.3.2 Mapa de Conceitos, Funções e Estruturas

Segue aqui uma breve descrição de alguns conceitos que serão melhor explicados ao longo do texto:

- **mdnsd**: MDNS Daemon, processo principal do MDNS, responsável por atender os programas controladores e se comunicar com os outros *hosts* da rede MDNS.
- **Controlador**: qualquer programa utilizando a *libmdns* para realizar requisições e receber respostas do **mdnsd**.
- **libmdns**: biblioteca que os controladores utilizam para a comunicação com o **mdnsd**.
- **Querier**: sub-sistemaa responsável por realizar *queries* em nome dos controladores.
- **Publisher**: sub-sistemaa responsável por publicar serviços em nome dos controladores.
- **Cache**: estrutura de dados onde são armazenados *Resource Record*.
- **Mensagem**: as mensagens trocadas entre os controladores e o **mdnsd**, elas são identificadas por um nome de tipo IMSG_CTL_*.

A seguir apresentamos uma breve descrição da API da biblioteca *libmdns*:

- **struct mdns**: estrutura que possui todo o estado da conexão com o **mdnsd**, todas as operações da biblioteca operam nesta estrutura.
- **mdns_open()**: abre uma conexão com **mdnsd**.
- **mdns_close()**: fecha uma conexão com **mdnsd**.
- **mdns_lookup_TYPE()**: envia uma mensagem ao **mdnsd** solicitando a procura por um *Resource Record* com o TIPO especificado.
- **mdns_browse_add()**: envia uma mensagem ao **mdnsd** solicitando a realização da enumeração de serviços para um determinado protocolo, os serviços enumerados são notificados ao controlador.
- **mdns_browse_del()**: envia uma mensagem ao **mdnsd** solicitando o cancelamento da enumeração de serviços para um determinado protocolo.
- **mdns_resolve()**: envia uma mensagem ao **mdnsd** solicitando a resolução de um serviço previamente enumerado.

A seguir apresentamos uma breve descrição das principais estruturas de dado utilizadas pelo **mdnsd**:

- **mdnsd_conf**: estrutura global com opções de inicialização.
- **struct ctl_conn**: representa a conexão de um controlador com o **mdnsd**.
- **ctl_conns**: lista com todos os controladores conectados.
- **Question-Tree**: uma árvore de todas as requisições de *Resource Record* presentes no sistema.
- **struct iface**: representa uma interface de rede.
- **struct rr**: representa um *Resource Record*.
- **struct rrset**: representa o Nome/Tipo/Classe de um *Resource Record*.
- **struct pkt**: representa um pacote MDNS.

- **struct query**: representa uma *query*, é uma coleção de *question*. Pode ser criada a pedido de um controlador ou pode ser oriunda da rede MDNS, caso em que o **mdnsd** deve tentar responder.
- **struct question**: representa uma pergunta por *Resource Record*.
- **struct pg**: representa um grupo do *Publisher*. É uma coleção de *pge*. (Publish Group)
- **struct pge**: representa um serviço ou endereço a ser publicado pelo *Publisher*. possui uma coleção de *Resource Records* a serem publicados (Publish Group Entry).
- **pg_queue()**: lista encadeada de todos os *pg* presentes no sistema.
- **pge_queue()**: lista encadeada de todos os *pge* presentes no sistema.

A seguir apresentamos uma breve descrição das principais funções do **mdnsd**:

- **control_accept()**: aceita a conexão de um controlador e cria uma estrutura *struct ctl_conn*.
- **query_fsm()**: processa uma *query*, consulta o *cache* por possíveis respostas e/ou envia pacotes na rede solicitando respostas. Notifica controladores em caso de falha.
- **pge_fsm()**: processa uma *pge*, realiza as etapas de *probing* e *announcing* de um serviço ou endereço. Notifica o controlador a cada etapa.
- **rr_notify_in()**: consulta a *Question-Tree* para verificar quais controladores tem perguntas para um determinado *Resource Record* e os notifica.

3.3.3 Querier

A seguir descrevemos a trajetória de uma *query*, desde sua criação até sua resposta ou da confirmação de ausência da mesma. Chamaremos de *Querier* a seção do programa que lida com as *queries*. Este é o sub-sistemaa responsável por realizar *queries* de MDNS em nome dos controladores, em outras palavras, quando um programa cliente deseja realizar uma consulta MDNS (exclui-se aqui a publicação de serviços), é o *Querier* que se encarregará de gerar os pacotes, consultar o *cache* e esperar as respostas, assim como notificar o controlador do resultado.

Este sub-sistema, assim como os demais, é apenas uma divisão lógica do código, e serve apenas para facilitar o entendimento do processo como um todo.

As responsabilidades e deveres deste sub-sistema são:

- Validar as requisições dos controladores.
- Garantir que dois controladores não perguntem pelo mesmo *Resource Record* em *queries* distintas. Isto é necessário para que se respeite os *timers* de envio de *queries*. Caso um controlador A pergunte pelo *Resource Record* RA e um controlador B alguns segundos depois decide perguntar pelo mesmo *Resource Record*, é necessário que não ocorra 2 *timers* distintos para a pergunta. Portanto eles partilham da mesma *query*.
- Ser capaz de efetuar uma *query One-Shot*, *Continuous* e de resolver um serviço MDNS (uma coleção de *queries One-Shot*).
- Garantir respostas aos controladores que fizeram a requisição.
- Localizar rapidamente as *queries* à medida em que são recebidas diversas respostas. O número de respostas na rede pode ser muito alto, portanto não se pode, por exemplo, a cada resposta percorrer uma lista de *queries* a fim de verificar se a resposta atende a pergunta. Não é interessante que se faça isto em *O(n)* (sendo n o número de *queries*) pois em redes maiores, n pode ser consideravelmente grande.

Os **controladores** são programas que utilizam a biblioteca *libmdns*, eles executam funções da biblioteca que geram mensagens para o **mdnsd**. Quando um controlador deseja, por exemplo, consultar um registro do tipo T_A, ele utiliza a função *mdns_lookup()* que envia a mensagem IMSG_CTL_LOOKUP, o **mdnsd** tentará então responder com o registro solicitado.

A seguir apresentamos a interface das funções da *libmdns* relativas ao sub-sistema *Querier*, são estas funções que o controlador utiliza para realizar requisições ao **mdnsd**.

- *mdns_lookup_A(), mdns_lookup_PTR(), mdns_lookup_HINFO(), mdns_lookup_rev()*. Solicitam o *lookup* de um *Resource Record* ao **mdnsd**. É enviada a mensagem IMSG_CTL_LOOKUP ao **mdnsd**.
- *mdns_browse_add() e mdns_browse_del()*. Ativam ou desativam o *Browsing* de um determinado serviço. São enviadas as mensagens IMSG_CTL_BROWSE_[ADD—DEL] ao **mdnsd**.

- *mdns_resolve()*. Solicitam a resolução de um determinado serviço. É enviada a mensagem IMSG_CTL_RESOLVE ao **mdnsd**.

As mensagens acima são enviadas pelos controladores ao **mdnsd** e resultam em diferentes tipos de *query*. A seguir é apresentado um mapa de qual mensagem resulta em qual tipo de *query*:

- IMSG_CTL_LOOKUP: *query* do tipo *One-Shot*.
- IMSG_CTL_BROWSE_ADD: *query* do tipo *Continuous*.
- IMSG_CTL_RESOLVE: uma coleção de *queries One-Shot*.

Usaremos como exemplo uma *query* simples do tipo T_A pelo nome *foobar.local*, ou seja, um programa qualquer deseja descobrir o endereço IPV4 do nome *foobar.local* na rede MDNS.

O programa controlador deve portanto, *vincular* com a biblioteca *libmdns*, conectar no **mdnsd** através da biblioteca (*mdns_open()*) e realizar as requisições.

Na listagem 4.3 é apresentado um programa (**controlador**) realizando a *query* descrita acima e imprimindo o resultado na tela. Os pontos importantes a serem notados da listagem 4.3 são:

- O estado da conexão com o **mdnsd** está contido na estrutura *m* de tipo *struct mdns*.
- A *callback*[5] que será chamada quando o programa receber uma resposta é *my_lookup_A_hook()*.
- O programa espera por respostas em um *loop* infinito, bloqueando em *mdns_read()*.

Quando o usuário abre a conexão com *mdns_open()*, um evento de leitura é acionado no *socket* de controle do **mdnsd** (isto é, agora estamos falando do processo **mdnsd**). Este evento aceita a conexão e cria uma instância de *struct ctl_conn* que representa conexão de um controlador (função *control_accept()*). Esta estrutura é constituida de:

- Um *link* para uma lista encadeada global de todos os controladores (*entry*).
- *Buffers* de entrada e saída de dados (*iev*).
- Uma lista de todas as *queries* ativas (*qlist*).

[5]Função chamada indiretamente por um ponteiro

A *libmdns* funciona de forma assíncrona, isto é, o usuário faz requisições e cadastra *callbacks* que devem ser chamadas a medida que cada resposta é enviada. Isto é necessário pois o processo **não** pode bloquear esperando uma resposta do **mdnsd**, visto que ele pode ter outras tarefas que precisam ser feitas. Quando é aberta a conexão, é dado o descritor do socket para o usuário. Isto possibilita funcionalidades interessantes, como por exemplo, a integração com APIs existentes no sistema, como *select(2)* ou até mesmo com a *libevent(3)*, pois para todos os efeitos aquele é um descritor que representa a conexão com o **mdnsd**. Cabe ao programa decidir quando esperar e ler por respostas. É importante ressaltar que esta espera não é necessária pois existem diversas formas de verificar se existe algo a ser lido do *buffer* do *socket*(*select(2), kqueue(2), poll(2)*).

3.3.3.1 Estrutura ctl_conn

A seguir é apresentada a estrutura *struct ctl_conn*:

Listing 3.2: struct ctl_conn

```
struct ctl_conn {
        TAILQ_ENTRY(ctl_conn) entry; /* Controller queue entry */
        struct imsgev         iev;   /* Imsg event and buffers */
        LIST_HEAD(, query)    qlist; /* List of active lists */
};
```

O controle do programa ao receber um evento de conexão acaba na função *control_accept()*, que faz o seguinte:

1. Aloca uma nova instância de *struct ctl_conn* para a conexão;
2. Registra um evento de leitura para a conexão com a *callback control_dispatch_imsg()*, que é responsável por tratar as requisições enviadas pelos controladores;
3. Adiciona a instância na lista de controladores *ctl_conns*.

À medida que o controlador efetua novas requisições que resultam em novas *queries*, a lista *qlist* vai sendo populada. Quando uma *query* termina, ela é removida desta lista. Em outras palavras, cada vez que o controlador utiliza uma função do tipo *mdns_lookup_**, uma nova *query* é adicionada a *qlist*.

3.3.3.2 Estrutura query

A seguir é apresentada a estrutura que representa uma *query*, são instâncias desta que estarão em *qlist*.

Listing 3.3: struct query

```
enum query_style {
        QUERY_LOOKUP,          /* A simple single-shot query */
        QUERY_BROWSE,          /* A Continuous Querying query */
        QUERY_RESOLVE,         /* A service resolve query */
};
struct query {
        LIST_ENTRY(query)       entry; /* Query link */
        LIST_HEAD(, rrset)      rrslist;/* List of question tree keys */
        struct ctl_conn        *ctl;   /* Owner */
        enum query_style        style; /* Style */
        struct event            timer; /* query_fsm() timer */
        u_int                   count; /* Used in query_fsm() */
        struct rrset           *ms_srv; /* The SRV in QUERY_RESOLVE */
        struct rrset           *ms_a;  /* The A in QUERY_RESOLVE */
        struct rrset           *br_ptr; /* The PTR in QUERY_BROWSE */
};
```

Os pontos importantes a serem notados da *struct query* são:

- Toda *query* tem uma entrada em uma lista de *queries* (*entry*);
- Toda *query* possui uma lista de chaves de *question* (*rrslist*);
- Toda *query* possui um tipo/estilo (*style*);
- Toda *query* possui um controlador dono, ou seja, um controlador pode ter *n queries* e uma *query* pertence a *um* e somente *um* controlador;
- Os membros *timer* e *count* são utilizados na máquina de estado que realiza a *query* e será descrita mais adiante.

Outro ponto importante é que há 3 estilos de *query*, os quais são:

- *QUERY_LOOKUP*: uma *query One-Shot*. O controlador adiciona com as funções *mdns_lookup_**, e sua vida termina quando uma resposta é recebida, ou quando ocorre *timeout*.

- *QUERY_BROWSE*: uma *query Continuous*. O controlador adiciona com a função *mdns_browse_add()*, sua vida **só** termina quando o controlador a removê-la explicitamente com *mdns_browse_del()*. Esta *query*, como explicada anteriormente, é usada para a enumeração de serviços, podendo retornar diversas respostas, por isso, cabe ao controlador decidir quando parar de realizar a enumeração.

- *QUERY_RESOLVE*: uma coleção de *One-Shot*. O controlador adiciona com a função *mdns_resolve()*, depois da enumeração de serviços é esta a função que o controlador utiliza para resolver uma instância de serviço. Tal resolução exige o *lookup* de três registros (T_A, T_SRV, T_PTR). A *query* termina quando se tem as três respostas ou quando ocorre *timeout* de qualquer uma.

3.3.3.3 Resolução da Query

Neste momento temos o controlador conectado; uma query do tipo QUERY_LOOKUP pelo *Resource Record* de tipo T_A; e nome *foobar.local*. Estamos então, prontos para dar início à resolução da mesma.

Os passos para a resolução da *query* são os que seguem:

1. Verificar se o controlador já possui esta *query*, se sim, ignora;
2. Procurar no *cache* local se sabe a resposta desta *query*, se sim, responde e termina;
3. Criar uma instância uma chave para a *question*, adiciona a *query*, e agenda um evento de resolução da *query*, que é tratado pela função *query_fsm()*;

O *cache* é uma estrutura de dados que armazena todas as respostas aprendidas pela rede. Portanto, se já sabemos a resposta pra tal *query*, ela estaria no *cache*.

Como mencionado anteriormente, é necessário que ao obter uma resposta qualquer da rede (seja em virtude de uma *query* nossa, ou de outro host, ou até uma resposta gratuita), consigamos verificar se temos *questions* que são atendidas pela resposta em questão. Uma abordagem ingênua seria percorrer uma lista de *queries*, e então percorrer a lista de *questions* da mesma, porém isso seria muito custoso e pouco eficiente.

Para que isso seja compreendido, é preciso lembrar que uma *query* pode conter diversas *questions*. No caso da QUERY_LOOKUP, esta contém apenas uma *question*. Mais especificamente em nosso caso, a *question* é pelo nome *foobar.local* e tipo T_A.

3.3.3.4 Estrutura question

A seguir é apresentada a estrutura que expressa uma *question*:

Listing 3.4: struct question

```
struct question {
  LIST_ENTRY(question) entry;   /* Packet link */
  RB_ENTRY(question)  qst_entry; /* Question Tree link */
  struct rrset        rrs;      /* RR tripple */
  u_int               flags;    /* Question flags */
  #define QST_FLAG_UNIRESP 0x1  /* Accepts unicast response */
  int                 active;   /* Active controllers */
  u_int               sent;     /* Used in question_fsm */
  struct timespec     lastsent; /* Last time we sent this question */
  struct timespec     sched;    /* Next scheduled time to send */
};
```

Os pontos importantes a serem notados são:

- Toda *question* está inserida na *Question Tree* (*qst_entry*).
- O nome e tipo (em nosso caso *foobar.local* e T_A) estão contidos em *rrs*.
- Há uma *flag* para identificar que esta *question* aceita respostas *unicast*.
- Há um contador de referências de controladores, pois **não** podem existir duas *question* idênticas ao mesmo tempo, e portanto cada controlador tem sua instância de *query*, porém elas irão conter chaves para a **mesma** *question*. As chaves são nome/-tipo/classe (*foobar.local/T_A/C_IN*).
- Contadores de tempo para garantir as frequências máximas de envio do protocolo. (*lastsent, sched*).

3.3.3.5 Question Tree

Para garantir a eficiência, todas as *questions* estão em uma árvore *Red-And-Black*, que aqui chamamos de *Question Tree*. Nenhum controlador possui referências diretas para as *questions*, ele possui apenas chaves para as *questions*, e desta forma, o estado de uma *question* é compartilhado entre os diversos controladores que estão interessados. Caso, ao criar a chave, não exista nenhuma *question* que responda a mesma, a *question* é criada e inserida na árvore, e seu contador de referências passa a ter valor 1.

Caso não tenhamos obtido uma resposta de nosso *cache*, um evento de resolução é cadastrado para ocorrer entre $120ms$ e $140ms$ (exigido pelo protocolo).

3.3.3.6 Função query_fsm()

A função *query_fsm()* é a responsável pela resolução de uma *query*.

Esta função recebe como argumento a *query* a resolver e toda a informação que a função precisa para sua execução está contemplada neste único argumento (uma forma de *transparência-referencial*[6]).

A resolução de uma *query* necessita que sejam feitas tentativas de tempos em tempos, até se confirmar que não há resposta, e, como não temos *threads*, não podemos, obviamente, fazer algo como:

1. Mande pergunta.
2. Durma alguns segundos.
3. Perguntei demais ? Sim então Fim : Não então vai para 1.

Pois o programa não faria mais nada nestes segundos, não processaria mais requisições, não responderia a outros *hosts*.

Nossa solução é antiga, porém bastante elegante. A função *query_fsm()* se auto-reagenda após enviar uma pergunta, incrementa (*query.count*) a cada agendamento, e na próxima vez que for agendada, saberá se deve continuar tentando ou não (o máximo de tentativas para QUERY_LOOKUP é 3). Por isso é necessário que todo o estado da máquina esteja em seu único argumento. A função *query_fsm()* é apresentada na listagem 4.1.

[6]Processamento de uma função ou procedimento em que todo o estado da computação está presente em seus argumentos

Os pontos importantes a serem notados são:

- As requisições são feitas respeitando os intervalos do protocolo, 1 segundo, 2 segundos, 4 segundos...
- No caso de QUERY_LOOKUP, são feitas duas tentativas. No caso de QUERY_BROWSE, as requisições nunca terminam. Elas estabilizam, na frequência de uma requisição por hora. No caso de QUERY_RESOLVE, são feitas três tentativas.
- No caso de QUERY_BROWSE, é preciso incluir todas as respostas já conhecidas a cada reenvio, como definido no protocolo em *Known Answer Supression*, isto garante que não seja elicitadas respostas das quais já temos conhecimento.
- No caso de falha, ou seja, foram feitas todas as tentativas, é enviado uma mensagem para todos os controladores interessados notificando-os da mesma.
- No fim da função, ela é reagendada.

É importante notar que em nenhum momento é considerado um "sucesso", ou seja, são feitas notificações apenas das falhas. Isto ocorre pois o "sucesso" é dado à medida que se *recebe* uma resposta, e não é a *query_fsm()* quem percebe esta resposta. A *query_fsm()* é responsável **apenas** por realizar as requisições. Sabemos agora como as *queries* são realizadas, mas não respondidas.

Não iremos detalhar o processamento de um pacote MDNS, pois é algo inteiramente operacional. O importante é perceber que um pacote MDNS contendo alguma(s) resposta(s) que estamos interessados pode ser recebido a qualquer momento, é então que precisamos cruzar as respostas com nossas perguntas.

3.3.3.7 Processamento de Respostas

Durante o processamento de um pacote MDNS, cada resposta é processada no subsistema *cache* na função *cache_process()*. Esta função recebe como parâmetro um *Resource Record* (*struct rr*) contendo uma possível resposta, durante o processamento esse *Resource Record* é inserido no *cache* (*cache_insert()*) e **caso** seja um **novo** *Resource Record* (ou seja, não é um *update/refresh*) a função *rr_notify_in()* é chamada. É ela a responsável por cruzar as respostas com nossas requisições. Seu funcionamento é bastante simples, e descrito a seguir:

1. Verifica se há uma pergunta na *Question Tree* para a dada resposta. Se não, fim.

2. Varre todas as perguntas de todos os controladores e envia uma mensagem com a resposta de "sucesso" aos mesmos. Esta é a operação de "sucesso" comentada em *query_fsm()*. A varredura por todas as perguntas é necessária pois a partir de uma *question* não sabemos quais controladores estão interessados (evitamos referências cruzadas pois são uma fonte conhecida de *bugs*).

3. Caso seja do tipo QUERY_LOOKUP, termina a *query* e cancela o seu possivel futuro evento em *query_fsm*.

4. Caso seja do tipo QUERY_RESOLVE, guarda a resposta na propria *query* e caso se tenha **todas** as respostas da *query*, responde para o controlador. Isto é necessário pois só podemos responder ao controlador quando tivermos todas as três respostas comentadas anteriormente.

A função *rr_notify_in()* só é chamada dentro de *cache_insert()* quando este registro é considerado *published*, todos registros externos são considerados *published*. Veremos no próximo capítulo que quando nós inserimos nossos próprios registros no *cache* eles ainda não são considerados *published*.

Neste ponto o ciclo de vida de uma *query* está completo. É importante ressaltar que cobrimos apenas o tipo mais básico de *query* (*One-Shot*), tentando ao mesmo tempo ressaltar os fatos importantes dos tipos mais complexos.

3.3.4 Publisher

A seguir descrevemos a trajetória da publicação de um serviço DNS-SD, bem como o sub-sistema que a implementa. Chamaremos de *Publisher* a seção do programa que lida com a publicação de serviços, é ele o sub-sistema responsável por publicar um serviço DNS-SD em nome dos controladores. Em outras palavras, quando um programa cliente deseja publicar um serviço DNS-SD, é o *Publisher* que se encarregará de enviar os pacotes de *probing* e *announcing*, verificar a existência de conflitos e afins.

As responsabilidades e deveres deste sub-sistema são:

- Validar as solicitações dos controladores;
- Publicar um serviço, ou um grupo de serviços;
- Garantir que os controladores serão notificados, seja do sucesso ou da eventual falha na publicação;

- Garantir que não existão conflitos entre os serviços;
- Garantir que caso aconteça um evento de *link-down/link-up*, o estado dos serviços seja mantido, isto é, eles serão republicados, como especificado no *draft*MDNS;
- Garantir a defesa do serviço, caso outro *host* tente publicar um serviço conflitante.

A principal função do *publisher* é publicar um serviço de DNS-SD. Para que isto ocorra o controlador necessita realizar os seguintes passos:

1. Abrir uma conexão com o **mdnsd** com a função *mdns_open()*;
2. Criar um grupo de publicação com a função *mdns_group_add()*;
3. Adicionar um ou mais serviços ao grupo criado com a função *mdns_group_add_service()*;
4. Esperar pela confirmação de sucesso ou falha da publicação.

A seguir são descritas as funções da *libmdns* e as mensagens que os controladores enviam ao **mdnsd** para realizar a publicação de serviços. Estes são os pontos de entrada do sub-sistema, tendo como ponto de partida, os controladores:

- IMSG_CTL_GROUP_ADD, cria um grupo, especifica um nome para o mesmo. Controlador utiliza a função *mdns_group_add()*.
- IMSG_CTL_GROUP_RESET, reseta o estado de um grupo, isto é, retira todos os serviços do grupo. Controlador utiliza a função *mdns_group_reset()*.
- IMSG_CTL_GROUP_ADD_SERVICE, adiciona um serviço qualquer ao grupo. Controlador utiliza a função *mdns_group_add_service()*.
- IMSG_CTL_GROUP_COMMIT, realiza o *commit* do grupo, isto é, pública todos os serviços previamente adicionados com *mdns_group_add_service()*. Controlador utiliza a função *mdns_group_commit()*.

Para a publicação de um serviço, é necessário que ele pertença a um grupo de publicação. Um grupo de publicação é uma coleção de serviços, geralmente mas não necessariamente, relacionados. A razão de existirem grupos é que algumas vezes desejamos publicar mais de um serviço em que: ou todos serviços falham, ou todos obtém êxito.

Para a melhor compreensão dos grupos, consideremos o exemplo de um servidor de impressão que suporta vários protocolos de impressão (cada protocolo é um serviço diferente com o mesmo nome de serviço, mas protocolos distintos).

Suponhamos que o nome do serviço seja *impressora3205*, e que este é publicado em dois protocolos: HTTP (*impressora3205._http._tcp._.local* e IPP *impressora3205._ipp._tcp._.local*. Para fins de MDNS estes são nomes diferentes, porém não queremos que caso ocorra um conflito em um deles, outro nome seja escolhido enquanto o outro ficaria com o nome original. Portando um grupo é uma forma de garantir que "se existir algum conflito e/ou outro problema em pelo menos um dos serviços deste grupo, todos os serviços deste grupo devem ser desconsiderados".

O *commit* do grupo é necessário pois o **mdnsd** inicia a publicação dos serviços do grupo paralelamente. A comunicação e os eventos entre o controlador e o **mdnsd** sempre são relacionados a um **grupo** e não a algum **serviço** do grupo.

O processo de conexão com o **mdnsd** é igual ao da *query*, não existe distinção entre um controlador que realiza *queries* e do controlador que publica serviços, ou seja, todo o processo de conexão e instanciamento da estrutura *ctl_conn* é idêntico como o descrito no capítulo do caso de uso da *query*. Portanto, não iremos abordar novamente o processo de conexão. Um controlador pode realizar todas as operações (publicação e *queries*) na mesma conexão com o **mdnsd**.

Usaremos como exemplo um programa servidor de FTP que deseja publicar seu serviço na rede MDNS, lembrando que o programa precisa obviamente *vincular* com a biblioteca *libmdns*.

Na listagem 4.4 é apresentado um exemplo de publicação do serviço de FTP descrito acima. Os pontos importantes a serem notados da listagem são:

- A estrutura *struct mdns_service* expressa um serviço de MDNS do ponto de vista do controlador e contém o nome do serviço, atributos, porta e afins.
- Toda vez que ocorre um evento relativo a qualquer grupo, a *callback my_group_hook* é chamada com o identificador do grupo e a instância do serviço *mdns_service*.
- O grupo *ftp_group* só possui um serviço, *ftp_server_86*.
- O programa poderia, ao receber uma notificação de conflito, escolher outro nome para o serviço e tentar novamente, porém para fins de simplificação, ele não o faz.

- É apenas depois do *commit* que o **mdnsd** da início a publicação do grupo, como veremos adiante.

Neste momento o **mdnsd** está pronto para iniciar a publicação do grupo, para que isto seja entendido, precisamos primeiro explicar as estruturas envolvidas na publicação.

3.3.4.1 Estrutura pg

A *struct pg* apresentada a seguir é a estrutura que expressa um grupo do ponto de vista do **mdnsd**:

Listing 3.5: struct pg

```
enum pg_state {
        PG_STA_NEW,
        PG_STA_COMMITED,
        PG_STA_COLLISION
};
/* Publish Group */
struct pg {
        TAILQ_ENTRY(pg)       entry;        /* pg_queue link */
        LIST_HEAD(, pge)      pge_list;     /* List of pge */
        struct ctl_conn       *c;           /* Owner */
        char                  name[MAXHOSTNAMELEN]; /* Group id */
        u_int                 flags;        /* Misc flags */
        enum pg_state         state;        /* enum pg_state */
};
```

Os pontos importantes desta estrutura são:

- Todo grupo pertence a uma lista de grupos (*entry*). Chamamos esta lista de *pg_queue*.
- Todo grupo possui uma lista de serviços, aqui chamados de *publish group entries (pge)* (*pge_list*).
- Todo grupo possui um controlador dono, no caso quem publicou o grupo (**c*).
- Todo grupo tem uma *string* identificador (*name*), que é o nome do grupo especificado pelo controlador (*ftp_group*).

- Todo grupo possui um estado (*state*).

3.3.4.2 Estrutura pge

A estrutura *struct pge* expressa um serviço do ponto de vista do **mdnsd**. Esta é a estrutura chave da publicação de serviços, ela é **também** utilizada para publicar um endereço primário de MDNS. A seguir é apresentada a estrutura *struct pge*:

Listing 3.6: struct pge

```
/* Publish Group Entry types */
enum pge_type {
        PGE_TYPE_CUSTOM,    /* Unused */
        PGE_TYPE_SERVICE,   /* A DNS-SD Service */
        PGE_TYPE_ADDRESS    /* A Primary Address */
};
/* Publish Group Entry States */
enum pge_state {
        PGE_STA_UNPUBLISHED,         /* Initial state */
        PGE_STA_PROBING,             /* Probing state */
        PGE_STA_ANNOUNCING,          /* Considered announced */
        PGE_STA_PUBLISHED,           /* Finished announcing */
};

#define PGE_RR_MAX 8
struct pge {
        TAILQ_ENTRY(pge) entry;      /* pge_queue link */
        LIST_ENTRY(pge) pge_entry;   /* Group link */
        struct pg       *pg;         /* Parent Publish Group */
        enum pge_type    pge_type;   /* Type of this entry */
        u_int            pge_flags;  /* Misc flags */
#define PGE_FLAG_INC_A  0x01         /* Include primary T_A record */
#define PGE_FLAG_INTERNAL 0x02       /* TODO: kill and test for pg */
        struct question *pqst;       /* Probing Question, may be NULL */
        struct rr       *rr[PGE_RR_MAX]; /* Array of to publish rr */
        int              nrr;        /* Members in rr array */
        struct iface    *iface;      /* Iface to be published */
```

```
        struct event    timer;      /* FSM timer */
        enum pge_state  state;      /* FSM state */
        u_int           sent;       /* How many sent packets */
};
```

Os principais pontos a serem notados sobre a estrutura *struct pge* são:

- Toda *pge* está em uma lista de todas as *pge*, a chamamos de *pge_queue* (*entry*). É utilizada quando precisamos iteirar sobre todas as *pge*;
- Toda *pge* está também em uma lista de *pge* do grupo (*pg*), isto é, um *pg* pode ter *N pge* (*pge_entry*);
- Toda *pge* possui um tipo, PGE_TYPE_SERVICE ou PGE_TYPE_ADDRESS, este que será mais abordado adiante (*pge_type*);
- A *pge* pode ter uma *question*, esta que é a *question* utilizada na fase de *probing*, pois como o *probing* é, em alguns casos, opcional. Em nossa implementação, sempre há uma *question* (*pqst*);
- Um serviço, como comentado anteriormente, é constituído de três *Resource Record*, um T_A, um T_SRV e um T_TXT, estes *Resource Record* estão no vetor *rr[PGE_RR_MAX]*, o tamanho não é fixado em três pois quando a *pge* não se refere a um serviço, podem ser menos ou mais que três *Resource Record*;
- Toda *pge* possui um estado, este que é análogo ao estado da *query*, é o estado utilizado pela função de publicação *pge_fsm*, assim como o da *query* é usado em *query_fsm()*.

Como mencionado anteriormente, a estrutura *pge* pode representar tanto a publicação de um serviço (PGE_TYPE_SERVICE) como de um endereço primário (PGE_TYPE_ADDRESS).

Uma *pge* que expressa um endereço primário é constituida dos seguintes *Resource Records*:

- Endereço IPv4. *Resource Record* do tipo T_A, por exemplo: *foobar.local* = 192.168.8.1.
- Mapeamento reverso de IPv4. *Resource Record* do tipo T_PTR, por exemplo: 192.168.8.1 = *foobar.local*.

- Informações sobre o *host*. *Resource Record* do tipo T_HINFO, por exemplo: *foo-bar.local* = OpenBSD/sparc64.

A publicação do endereço primário é feita sempre que há um evento de *link-up*, este que é oriundo do *socket* AF_ROUTE mencionado anteriormente.

O endereço primário é necessário pois cada **serviço** inclui o *Resource Record* de tipo T_A na publicação, ou seja, não podemos publicar serviços sem antes publicarmos o endereço primário da interface. Esta dependência precisa ser tratada, e toda publicação de serviços é adiada caso seu endereço primário ainda não tenha sido publicado.

Explicadas as estruturas principais do *Publisher*, podemos agora seguir em diante com nosso exemplo.

3.3.4.3 Publicação

Quando o *mdnsd* recebe o comando de *commit* para um grupo, são feitos os seguintes passos:

1. Verifica se há um conflito interno, isto é, verifica se outro controlador já não publicou um serviço conflitante (chamamos um conflito interno de *colisão*). Se há colisão, envia a mensagem IMSG_CTL_GROUP_ERR_COLLISION e termina o grupo.

2. Chama a função *pge_from_ms()* que transforma uma *struct mdns_service* em uma *struct pge*.

A função *pge_from_ms()* faz mais uma operação que será crucial para a resolução de conflitos e a detecção de colisão: os *Resource Records* criados pela função, além de serem vinculados a *pge*, são também **inseridos no cache**, porém tais *Resource Records* possuem uma *flag* indicando que não estão publicados, ou seja, o **mdnsd** não deve **ainda** responder pelos mesmos.

Com esta inserção no *cache* a verificação de colisão fica bastante simples. Quando é feita verificação de colisão para um novo serviço, procura-se no *cache* por todos os *Resource Records* a serem publicados, caso já exista algum e este seja de algum outro controlador, considera-se uma colisão.

A detecção de conflitos também fica bastante simples, pois quando recebemos uma resposta externa, podemos averiguar se possuimos *Resource Record* conflitantes de forma eficiente consultando apenas o *cache*.

No início do desenvolvimento, o *cache* externo (*Resource Records* de outros *hosts*) era separado do interno, o que resultou em diversos *bugs* e dificuldades de implementação, aqui vale uma menção, a unificação do *cache* resultou em um decréscimo de cerca de 1000 linhas no código, além de simplificar muito as partes do código que interagiam com o mesmo.

Neste ponto o controlador efetuou *commit* do grupo *ftp_group*, o **mdnsd** possui então um grupo (*pg*) com apenas uma entrada (*pge*). A publicação do serviço é feita pela função *pge_fsm* (listagem 4.2.

3.3.4.4 Função pge_fsm()

A função *pge_fsm()* é a responsável por executar a publicação de um serviço ou endereço primário. Ela funciona de maneira análoga a *query_fsm()* e a mesma técnica de reagendamento do evento é utilizada. A seguir é apresentado o funcionamento da função *pge_fsm()*:

1. Se *pge* for um serviço, confirma que seu endereço primário já foi publicado, se não, tenta novamente em 1 segundo.

2. Inicia fase de *probing* como descrito no *draft*MDNS e notifica o controlador com mensagem IMSG_CTL_GROUP_PROBING.

3. Inicia fase de *announcing*, como descrito no *draft*MDNS e notifica o controlador com mensagem IMSG_CTL_GROUP_ANNOUNCING.

4. Terminada a fase de *announcing*, marca o grupo como **publicado**. Adiciona a *flag* de publicado em todos os *Resource Records* (que também estão vinculados no *cache*), a partir deste momento o **mdnsd** começará a responder aos outros *hosts* pelos *Resource Records* publicados.

A função *pge_fsm()* pode ser visualizada na listagem 4.2.

3.3.5 Cache

A seguir apresentamos o sub-sistema do *cache*, também será apresentada e detalhada a estrutura que representa um *Resource Record*.

O *cache* é uma das estruturas principais do **mdnsd**, sua principal função é armazenar os diversos *Resource Records* aprendidos na rede MDNS, assim como os *Resource Record* publicado pelos controladores.

A estrutura que representa um *Resource Record* é a *struct rr*, a seguir apresentamos sua definição.

Listing 3.7: struct rr

```
struct rr {
        LIST_ENTRY(rr)          centry; /* Cache entry */
        LIST_ENTRY(rr)          pentry; /* Packet entry */
        struct rrset            rrs;    /* RR tripple */
        u_int32_t               ttl;    /* DNS Time to live */
        union {
                struct in_addr A;       /* IPv4 Address */
                char            CNAME[MAXHOSTNAMELEN]; /* CNAME */
                char            PTR[MAXHOSTNAMELEN]; /* PTR */
                char            NS[MAXHOSTNAMELEN];  /* Name server */
                char            TXT[MAXCHARSTR];     /* Text */
                struct srv      SRV;                 /* Service */
                struct hinfo    HINFO;               /* Host Info */

        } rdata;
        struct iface            *iface; /* Published/Received interface */
        struct cache_node       *cn;    /* Cache parent node */
        int                      auth_refcount; /* Number of pges holding us */
        int                      revision; /* at 80% of ttl, then 90% and 95% */
        struct event             timer; /* revision timer */
        struct timespec          age;   /* rr age */
        u_int                    flags; /* RR Flags */
#define RR_FLAG_CACHEFLUSH    0x01    /* Unique record */
#define RR_FLAG_PUBLISHED     0x02    /* Published record */
};
```

Os principais pontos a serem notados são:

- Um *rr* pode estar inserido no *cache* (*centry*).

- Um *rr* pode ser oriunda da rede MDNS, neste caso *auth_refcount* é 0. ou oriunda de algum dos controladores, neste caso, *auth_refcount* é maior que 0.
- Um *rr* pode ser *Shared* ou *Unique*, neste caso, a *flag* RR_FLAG_CACHEFLUSH estará presente.
- Um *rr* pode estar publicado, caso em que a *flag* RR_FLAG_PUBLISHED estará presente.
- Todo *rr* possui uma tupla ternária que representam nome/tipo/classe (*rrs*).
- O tipo do *rr* especifica qual é o conteúdo do *rdata*, como descrito no RFC 1035.
- Todo *rr* possui um TTL (*Time-To-Live*).

O *cache* é implementado com uma *Red-And-Black-Tree* em que os nomes dos *Resource Records* são as chaves e os nodos são listas de *Resource Records*.

Portanto, se existir por exemplo, os *Resource Records* {frodo.local/T_A/C_IN}[7] e {frodo.local/T_HINFO/C_IN}, o *lookup* pelo nome *frodo.local* retornará uma lista com ambos os *Resource Records*. Vale ressaltar que podem haver *Resource Record* de mesmo nome/tipo/classe (*Shared Resource Records*), como descrito no *draft*.

Sejam n o número de nomes (chaves) no *cache* e nr o número de *Resource Record* na lista encadeada sob um dado nome, podemos concluir a notação O da seguinte forma:

- *Lookup, Delete*: $O(\log n) + O(nr)$. Procurar o nome na árvore + procurar o registro na lista.
- *Insert*: $O(\log n)$. Procurar o nome na árvore e inserir, como insere na cabeça da lista encadeada, não altera o custo.

A função *cache_insert()*, responsável por inserir um *Resource Record* no *cache*, é chamada em duas outras funções, são elas:

- *cache_process()*. Função que processa um *Resource Record* externo, isto é, um *Resource Record* oriundo do *socket* MDNS.
- *auth_get()*. Função utilizada pelo *Publisher* para inserir um *Resource Record* interno, ou seja, oriundo de um dos controladores.

[7] Nome/Tipo/Classe

O funcionamento da função *cache_process()* é apresentada a seguir:

1. Registra o *timestamp* de chegada do *Resource Record*.
2. Seta *flag* de publicado no *Resource Record*, pois todos *Resource Records* externos são considerados publicados.
3. Decide se este *Resource Record* é **novo**, se ele **substitui** outros, ou ainda se é apenas um **update** de um *Resource Record* já existente no *cache*.
4. Insere o *Resource Record* caso necessário.
5. Agenda a revisão do *Resource Record*com base no TTL. Aos 80%, 90% e 95% do tempo de vida do *Resource Record*, um evento de revisão será disparado. A renovação consiste em realizar uma *query* para receber novamente o registro, atualizando assim seu tempo de vida. O envio desta *query* **só** ocorre **caso** exista algum controlador interessado, isto é, se existe uma *question* para o *Resource Record* em questão na árvore de *question*.

3.4 mdnsctl

O programa **mdnsctl** implementa uma interface simples com o **mdnsd** via linha de comando.

As seguintes operações são suportadas:

1. *Lookup*: Procura um registro dado um nome e um tipo.
2. *Browse*: Realiza a enumeração de serviços dado um protocolo. Também é possível enumerar todos os serviços de todos os protocolos.
3. *Browse-And-Resolve*: Realiza o *Browse* e resolve todos os serviços enumerados.

A seguir apresentamos um exemplo de um *Lookup* pelo nome *axis-00408c7de04e.local* e tipo T_A (endereço IPv4).

```
elendil:tmp: mdnsctl lookup axis-00408c7de04e.local
Address: 10.32.170.243
```

A seguir apresentamos um exemplo de *Browse* por todos os serviços do protocolo HTTP:

```
elendil:tmp: mdnsctl browse http tcp
+++ AXIS 221 - 00408C7DE04B           http  tcp
+++ AXIS 221 - 00408C79DF88           http  tcp
+++ AXIS 221 - 00408C7DE04E           http  tcp
+++ AXIS 221 - 00408C7DE053           http  tcp
+++ AXIS 221 - 00408C7D905A           http  tcp
```

A seguir apresentamos um exemplo de *Browse-And-Resolve* por todos os serviços do protocolo HTTP:

```
elendil:tmp: mdnsctl browse -r http tcp
+++ AXIS 221 - 00408C7D905A           http  tcp
Name: AXIS 221 - 00408C7D905A
Port: 80
Target: CF33-00408c7d905a.local
Address: 169.254.240.201
Txt:
+++ AXIS 221 - 00408C7DE053           http  tcp
Name: AXIS 221 - 00408C7DE053
Port: 80
Target: axis-00408c7de053.local
Address: 169.254.194.30
Txt:
+++ AXIS 221 - 00408C7D9053           http  tcp
Name: AXIS 221 - 00408C7D9053
Port: 80
Target: CF34-00408c7d9053.local
Address: 169.254.236.249
Txt:
...
```

O manual completo e atualizado do programa pode ser encontrado em *http://haesbaert.org/openm*

3.5 Status

O OpenMDNS, proposto neste trabalho, encontra-se na versão 0.5, este que teve início em fevereiro de 2010 tendo em vista ser abordado no Trabalho de Conclusão.

Todas as operações propostas neste trabalho foram implementadas e estão em funcionamento estável, com a exceção do suporte a múltiplas interfaces que está sendo postergado.

Foram feitos testes nas arquiteturas *alpha, i386, amd64 e sparc64*. Foi utilizada uma máquina com o sistema operacional Linux e a implementação MDNS/DNS-SD Avahi para testes de interoperabilidade.

A biblioteca ainda não é uma *shared library* pois não escrevemos a documentação da mesma e não houve necessidade, pois não há outros programas que não o mdnsctl que a utilizem.

3.5.1 Recursos

O *site* do projeto está em *http://www.haesbaert.org/openmdns* e o código está disponível em *http://github.com/haesbaert/mdnsd*.

Os manuais dos programas estão em: *http://haesbaert.org/openmdns/mdnsd.8.html* e *http://haesbaert.org/openmdns/mdnsctl.8.html*.

3.5.2 Histórico

O primeiro *commit* do projeto ocorreu em 23 de janeiro as 21:52:33. Entre 23 de janeiro de 2010 e 26 de abril de 2011 ocorreram 391 *commits*.

Em Maio de 2010 o OpenMDNS foi apresentado à dois integrantes da equipe do OpenBSD: Marco Peereboom e Nicholas Marriot, e estes fizeram diversos comentários, críticas e realizaram revisões de código, apontando *bugs* e imperfeições. Em junho de 2010 o projeto foi levado a discussão entre a equipe do OpenBSD e outros membros demonstraram interesse. Acreditava-se trabalho possuia chances de ser aceito na base do OpenBSD. A inclusão do projeto representaria um marco importante, pois rapidamente milhares de

pessoas estariam indiretamente utilizando o projeto, reportando suas experiências e *bugs* encontrados.

A seguir são apresentadas os *release dates* de cada uma das versões estáveis:

- Versão 0.1 - 23 de fevereiro de 2011
- Versão 0.2 - 2 de março de 2011.
- Versão 0.3 - 2 de março de 2011 (*bug* de última hora).
- Versão 0.4 - 7 de março de 2011.
- Versão 0.5 - 26 de abril de 2011.

3.5.2.1 Patch para libc

Em fevereiro de 2011 foi enviado um *patch* para a *libc* do OpenBSD adicionando suporte ao *lookup* de nomes de *mdns* via *gethostbyname(3) e getaddrinfo(3)*. Este foi rejeitado por *Theo de Raadt* sob a premissa de que MDNS é um protocolo inseguro, e por mais que o usuário precisasse explicitamente habilitar o MDNS em */etc/resolv.conf*, a *libc* teria que manter um padrão mais alto. Segue a conversação:

```
deraadt: I am terrified of MDNS.
haesbaert: Could you elaborate on that ?
deraadt: I do not see the benefit, over the danger.
haesbaert:
   I can see the point in not having mdnsd in base, as it is an insecure protocol
   by default, I won't argue about that.

   But the libc patch I'd argue, it only adds a lookup discipline, independent of
   mdnsd code or anything else.

   What I do is to send the same query over multicast if it is for the .local
   domain *and* the user enabled mdns on resolv.conf.
   My point is, if the user explicitly enabled mdns on resolv.conf *and* is
   querying for foo.local, I think we could assume he knows the risks,
   couldn't we ?
deraadt: No, we can't assume that. libc has to meet a higher standard.
```

Este *patch* pode ser encontrado em:

http://marc.info/?l=openbsd-tech&m=130058498908291&w=2

Com isso também são muito remotas as chances de o **mdnsd** entrar na base do Sistema Operacional, porém isso não nos desmotivou e o programa é usado por diversas pessoas através de um *port*.

Alguns *developers* do OpenBSD se mostraram interessados como Henning Brauer, Marco Peereboom e Nicholas Marriot, porém quem tem a palavra final é o líder Theo De Raadt, e suas ações ao longo dos últimos 20 anos tem mostrado que suas escolhas tem embasamento e mérito.

3.5.2.2 O Port

Gonzalo Lionel Rodrigues (Buenos Aires, Argentina) criou um *port* do OpenMDNS para o sistema de pacotes do OpenBSD. Com isso, hoje é possível instalar o OpenMDNS com apenas dois comandos:

```
elendil:tmp: cd /usr/ports/net/openmdns/
elendil:openmdns: sudo make install
```

3.5.2.3 Patch Multicast

O desenvolvimento do OpenMDNS também originou um *patch* que corrige a semântica de *sockets multicast*.

Quando um programa especifica qual interface deseja utilizar para o envio do datagrama *multicast*, o sistema **não** pode levar em consideração a tabela de roteamento.

Existia um *bug* quando a rota fosse do tipo *reject*, isto é, o sistema levava em consideração a tabela de roteamento, o que não deveria acontecer.

Foi corrigido e o *patch* foi aceito.

Também foram enviados diversos outros pequenos *patches* para o OpenRIPD e OpenOSPFD.

3.5.2.4 Estado da Implementação

Na tabela 1 é apresentado o estado da implementação de cada seção descrita no *draft* MDNS(1) e na tabela 2 é apresentado o mesmo para o *draft* DNS-SD (2). O estado da implementação de cada seção é descrito como *I = "Implementado"*, *NI = "Não Implementado"*, *PI = "Parcialmente Implementado"* e *NS = "Não será implementado"*.

3.6 Feedback

Desde que o OpenMDNS tornou-se funcional, diversos foram os *feedbacks* recebidos por usuários da comunidade.

Estes relatos são importantes a medida que ajudam no teste do projeto em ambientes desconhecidos, facilitando muito a descoberta de *bugs* e afins.

O código do projeto foi sendo disponibilizado à medida em que foi escrito, apenas testes básicos eram feitos antes de submeter o código ao repositório. A idéia era receber o máximo possível de *feedback* no menor tempo possível.

Também é fato que tais relatos são motivantes à medida que percebe-se se o rumo do trabalho está correto ou não.

A seguir uma lista dos principais eventos:

- pjnouch@github reportou 2 bugs e testou suas correções.
- Peter J Phillip, Canada, escreveu um ataque que resultava em uma falha de segmentação, foi corrigido e testado.
- Gonzalo Lionel Rodriguez, Buenos Aires, escreveu um *port* do projeto, facilitando muito a sua instalação.
- Amador Pahim, Caxias, disponibilizou uma máquina na rede da UCS para testes.
- Tobias Ulmer, Alemanha, mostrou constante suporte.
- Diversos outras pessoas enviaram um email agradecendo ou apenas reportando que "estavam usando".
- Mauricio Mauad, Porto Alegre, doou um *Alpha Workstation 500au* para testes.

- Pedro Kiefer, Porto Alegre, começou a portar o OpenMDNS para o Linux, porém não foi finalizado.

Em especial agradou muito o relato enviado por Marco Peereboom, arquiteto *senior* de uma importante empresa que desenvolve dispositivos de *storage* e um dos membos mais respeitados da equipe do OpenBSD. Ele é o criador e arquiteto dos projetos *softraid(4)*(29) e *acpi(4)*(30). A seguir é apresentado uma cópia da conversa por *email*, esta conversa deu-se antes da primeira versão funcional:

```
On Tue, Nov 09, 2010 at 11:11:05PM -0200, Christiano F. Haesbaert wrote:
> On 9 November 2010 21:50, Marco Peereboom <slash@peereboom.us> wrote:
> M: > I cnahaz publish?
> >
>
H> I found out that my first publish design was plain flawed and I
H> decided to step back and rethink.
H> Avahi does some crazy <censurado>when it comes to publishing features, I'm
H> trying to match all those feature without compromising the hole
H> design.
H> I've a new design in a piece of paper that looks promising.
H> There's also no point in implementing publishing without the proper
H> support for multiple interfaces, so the design gets trickier.
H> Thing is I'll be busy until December (exams and thesis), so publishing
H> will have to wait a little :/.
H> As soon as I get the publishing done I'll spend some time fixing bugs
H> and then will move forward to make a proper libmdns.

M: Cool.  I appreciate good design.

>
> M> I love the way it works and want to use it badly.
> >
>
H> Great ! That makes panda real happy :-) !
H> Please feel free to tell me what you need and I can try to make it a priority.
```

M>
We are working on a product that requires backend infrastructure. I want to make the entire backend infrastructure use mdns to publish the services (maybe even users) on the network. I want zeroconf the whole thing.

I am by no means an expert on mdns but bare with me for a second.

We are going to for example have disks inside a machine and the idea is to advertise that disk using mdns. I know at least that is possible however I am wondering if I could reflect status in mdns. For example the disk went up in flames; can it say "don't talk to me"?

Another thing I though off is users. I am going to have multiple machines that host different users. Can I advertise on which machine marco has his account?

I have countless other questions but I think with these answered that I need to start reading the spec much more closely. Is there a quick and dirty guide you know of?

btw, what threw me for a loop is that publish is in the ui but didn't work ;-) A man page would have been awesome ;-)

Thanks a lot for this code it is super cool.

4 Conclusão

Os problemas endereçados pelos protocolos MDNS/DNS-SD estão presentes no dia a dia da maioria das pessoas, ao analisar os protocolos acredita-se que estes proporcionam uma solução viável aos mesmos, ressaltando ainda o fato que empresas como a Apple o vem utilizando a algum tempo com um moderado grau de sucesso.

Esperava-se que ao término deste trabalho, o MDNS tivesse obtido o *status* de RFC, porém o mesmo não ocorreu.

O nosso principal objetivo era construir uma solução segura, eficiente, simples e pequena, acreditamos que obtivemos êxito nestes pontos.

O OpenMDNS possui **6966** linhas incluindo os *headers*. Em momentos anteriores o OpenMDNS chegou a ter quase **10000** linhas. Porém, a medida que fomos prestando atenção em erros de *design* e optando por não remendar decisões ruins, e sim, refaze-las. Com isto conseguimos diminuir bastante o volume de código.

Vale ressaltar que o Avahi possui cerca de **40000** linhas e o Bonjour **60000** linhas. É evidente que o OpenMDNS não é tão maduro e não implementa algumas das funcionalidades das soluções existentes, mesmo assim, acreditamos porém, embora ainda não tenhamos feito uma comparação a fundo com o Avahi e Bonjour que é possível sim ter uma implementação menor e mais simples.

Conseguimos entregar um trabalho funcional no mundo real para a comunidade do OpenBSD, seguindo os princípios e filosofias desta comunidade. Acreditamos que mesmo o trabalho não sendo incluso na base do sistema, fornecemos uma alternativa viável às existentes (até certo ponto), e com uma licença mais livre.

O contato com *hackers* mais experientes nos trouxe **muito** conhecimento, as diversas opiniões e *feedbacks* nos trouxeram um conhecimento que é caro. Mais que isto, conhecemos de perto uma cultura que não faz questão de aparecer ou fazer sucesso, uma cultura que foi precursora no desenvolvimento de Sistemas Operacionais e se apega aos mesmos

valores e filosofias a mais de 40 anos.

Acreditamos que o *bloat* é um dos principais, se não o principal problema dos *software* nos dias de hoje. Hoje em dia, temos navegadores que alocam cerca de 1gb de memória, navegadores que deveriam apenas renderizar HTML e *javascript*. É de nossa **opinião** que a excelência em *design* e a busca pelo *minimalismo* precisa ser resgatada.

Chegamos ao fim deste texto com um sentimento de trabalho cumprido, de sucesso e vitória. A conclusão deste trabalho era um desafio pessoal na formação do autor, pois é o primeiro projeto deste porte que é finalizado.

"Small is beautiful."

Tabelas

Tabela 1: Estado da Implementação das Seções do Draft MDNS(1)

Seção	Item	Status
4	Reverse Address Mapping	I
5.1	One-Shot Multicast DNS Queries	NS
5.2	One-Shot Queries, Accumulating Multiple Responses	I
5.3	Continuous Multicast DNS Querying	I
5.4	Multiple Questions per Query	I
5.5	Questions Requesting Unicast Responses	I
5.6	Direct Unicast Queries to port 5353	I
6.1	Known Answer Suppression	I
6.2	Multi-Packet Known Answer Suppression	I
6.3	Duplicate Question Suppression	I
7	Responding	I
7.1	Negative Responses	NI
7.2	Responding to Address Queries	I
7.3	Responding to Multi-Question Queries	I
7.4	Response Aggregation	I
7.5	Wildcard Queries (qtype "ANY" and qclass "ANY")	I
7.6	Legacy Unicast Responses	I
8	Probing and Announcing on Startup	I
8.1	Probing	I
8.2	Simultaneous Probe Tie-Breaking	PI
8.2.1	Simultaneous Probe Tie-Breaking for Multiple Records	PI
8.3	Announcing	I
8.4	Updating	I
9	Conflict Resolution	I
10.2	Goodbye Packets	I
10.3	Announcements to Flush Outdated Cache Entries	I
10.4	Cache Flush on Topology change	I
10.5	Cache Flush on Failure Indication	NI
10.6	Passive Observation of Failures (POOF)	NI
11	Source Address Check	I

Tabela 2: Estado da Implementação das Seções do Draft DNS-SD(2)

Seção	Item	Status
4	Service Instance Enumeration (Browsing)	I
5	Service Name Resolution	I
8	Flagship Naming	NS
9	Service Type Enumeration	I
12	Discovery of Browsing and Registration Domains	NS

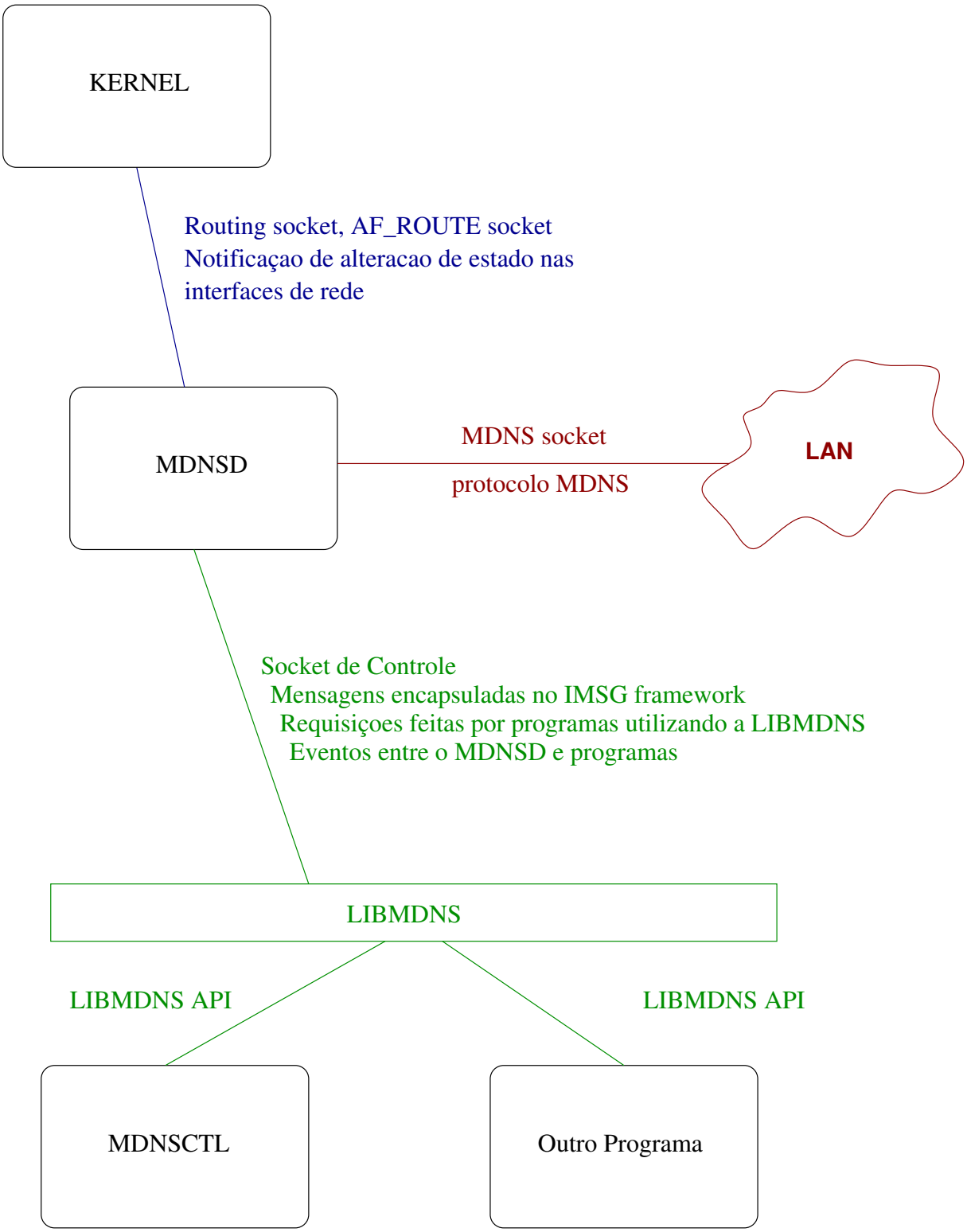


Figura 1: Arquitetura do OpenMDNS

Listagens

Listing 4.1: Função query_fsm()

```
void
query_fsm(int unused, short event, void *v_query)
{
        struct pkt             pkt;
        struct mdns_service    nullms;
        struct query          *q;
        struct question       *qst;
        struct rr             *rraux, nullrr;
        struct rrset          *rrs;
        struct timespec        tnow;
        struct timeval         tv;
        time_t                 tosleep;

        q = v_query;
        pkt_init(&pkt);
        pkt.h.qr = MDNS_QUERY;

        /* This will send at seconds 0, 1, 2, 4, 8, 16... */
        tosleep = (2 << q->count) - (1 << q->count);
        if (tosleep > MAXQUERYTIME)
                tosleep = MAXQUERYTIME;
        timerclear(&tv);
        tv.tv_sec = tosleep;

        if (clock_gettime(CLOCK_MONOTONIC,&tnow) == -1)
                fatal("clock_gettime");

        /*
```

```
 * If we're in our third call and we're still alive,
 * consider a failure.
 */
if (q->style == QUERY_LOOKUP && q->count == 2) {
        rrs = LIST_FIRST(&q->rrslist);
        bzero(&nullrr, sizeof(nullrr));
        nullrr.rrs = *rrs;
        control_send_rr(q->ctl, &nullrr,
    IMSG_CTL_LOOKUP_FAILURE);
        query_remove(q);
        return;
}

if (q->style == QUERY_RESOLVE && q->count == 3) {
        log_debug("query_resolve␣failed");
        bzero(&nullms, sizeof(nullms));
        strlcpy(nullms.name, q->ms_srv->dname,
    sizeof(nullms.name));
        control_send_ms(q->ctl, &nullms,
    IMSG_CTL_RESOLVE_FAILURE);
        query_remove(q);
        return;
}

LIST_FOREACH(rrs, &q->rrslist, entry) {
        if (q->style == QUERY_RESOLVE&&cache_lookup(rrs)) {
                log_debug("question␣for␣%s␣supressed␣"
                    "have␣answer", rrs_str(rrs));
                continue;
        }
        if ((qst = question_lookup(rrs)) == NULL) {
                log_warnx("Can't␣find␣question␣in␣query_fsm␣"
            "for␣%s", rrs_str(rrs));
                /* XXX: we leak memory */
                return;
```

```
                }

                /* Can't send question before schedule */
                if (timespeccmp(&tnow, &qst->sched, <)) {
                        log_debug("question␣for␣%s␣before␣schedule",
                            rrs_str(rrs));
                        continue;
                }

                pkt_add_question(&pkt, qst);
                qst->sent++;
                qst->lastsent = tnow;
                qst->sched = tnow;
                qst->sched.tv_sec += tosleep;
                if (q->style == QUERY_BROWSE) {
                        /* Known Answer Supression */
                        CACHE_FOREACH_RRS(rraux, rrs) {
                                /* Don't include rr if it's too old */
                                if (rr_ttl_left(rraux) < rraux->ttl / 2)
                                        continue;
                                pkt_add_anrr(&pkt, rraux);
                        }
                }
        }

        if (pkt.h.qdcount > 0)
                if (pkt_send_allif(&pkt) == -1)
                        log_warnx("can't␣send␣packet␣to␣all␣interfaces");
        q->count++;
        if (evtimer_pending(&q->timer, NULL))
                evtimer_del(&q->timer);
        evtimer_add(&q->timer,&tv);
}
```

Listing 4.2: Função pge_fsm()

```
void
```

```
pge_fsm(int unused, short event, void *v_pge)
{
        struct pg      *pg;
        struct pge     *pge, *pge_primary;
        struct iface   *iface;
        struct timeval  tv;
        struct pkt      pkt;
        int             inc_prim, i;

        pge = v_pge;
        pg  = pge->pg;

        /*
         * In order to publish services and addresses we
         * must first make sure our primary address has been
         * sucessfully published, if not, we delay
         * publication for a second.
         */
        if (pge->pge_type == PGE_TYPE_SERVICE) {
                LIST_FOREACH(iface, &conf->iface_list, entry) {
                        pge_primary = iface->pge_primary;
                        if (pge_primary->state < PGE_STA_ANNOUNCING) {
                                timerclear(&tv);
                                tv.tv_sec = 1;
                                pge_fsm_restart(pge, &tv);

                                return;
                        }
                }
        }

        switch (pge->state){
        case PGE_STA_UNPUBLISHED:
                pge->state = PGE_STA_PROBING;
                /* FALLTHROUGH */
```

```
case PGE_STA_PROBING:
        if ((pge->pge_flags & PGE_FLAG_INTERNAL) == 0 &&
            pge->sent == 0)
                control_notify_pg(pg->c, pg,
                    IMSG_CTL_GROUP_PROBING);
        /* Build up our probe packet */
        pkt_init(&pkt);
        pkt.h.qr = MDNS_QUERY;
        if (pge->pqst != NULL) {
                /* Unicast question ? */
                if (pge->sent < 2)
                        pge->pqst->flags |= QST_FLAG_UNIRESP;
                else
                        pge->pqst->flags &= ~QST_FLAG_UNIRESP;
                pkt_add_question(&pkt, pge->pqst);
        }
        /* Add the RRs in the NS section */
        for (i = 0; i < pge->nrr; i++)
                pkt_add_nsrr(&pkt, pge->rr[i]);
        if (pkt_send_allif(&pkt) == -1)
                log_warnx("can't␣send␣probe␣packet");
        /* Probing done, start announcing */
        if (++pge->sent == 3) {
                /* sent is re-used by PGE_STA_ANNOUNCING */
                pge->sent = 0;
                pge->state = PGE_STA_ANNOUNCING;
                /*
                 * Consider records published
                 */
                for (i = 0; i < pge->nrr; i++) {
                        if ((pge->rr[i]->flags & RR_FLAG_PUBLISHED)
                            == 0)
                                rr_notify_in(pge->rr[i]);
                        pge->rr[i]->flags |= RR_FLAG_PUBLISHED;
                }
```

```
        }
        timerclear(&tv);
        tv.tv_usec = INTERVAL_PROBETIME;
        pge_fsm_restart(pge, &tv);
        break;
case PGE_STA_ANNOUNCING:
        if ((pge->pge_flags & PGE_FLAG_INTERNAL) == 0 &&
            pge->sent == 0)
                control_notify_pg(pg->c, pg,
                    IMSG_CTL_GROUP_ANNOUNCING);
        /* Build up our announcing packet */
        pkt_init(&pkt);
        pkt.h.qr = MDNS_RESPONSE;
        /* Add the RRs in the AN secion */
        for (i = 0; i < pge->nrr; i++)
                pkt_add_anrr(&pkt, pge->rr[i]);
        /*
         * PGE_FLAG_INC_A, we should add our primary
         * A resource record to the packet.
         * We must look for the A record on our primary
         */
        inc_prim = !!(pge->pge_flags & PGE_FLAG_INC_A);
        if (pkt_send_allif_do(&pkt, inc_prim) == -1) {
                log_warnx("can't␣send␣probe␣packet");
                return;
        }
        if (++pge->sent < 3) {
                timerclear(&tv);
                tv.tv_sec = pge->sent;
                pge_fsm_restart(pge, &tv);
                break;
        }
        pge->state = PGE_STA_PUBLISHED;
        /* FALLTHROUGH */
case PGE_STA_PUBLISHED:
```

```
                if ((pge->pge_flags & PGE_FLAG_INTERNAL) == 0)
                        log_debug("group␣%s␣published", pg->name);
                /*
                 * Check if every pge in pg is published, if it is
                 * we'll consider the group as published, notify controller
                 */
                if ((pge->pge_flags & PGE_FLAG_INTERNAL) == 0 &&
                    pg_published(pg))
                        control_notify_pg(pg->c, pg, IMSG_CTL_GROUP_PUBLISHED);
                break;
        default:
                fatalx("invalid␣group␣state");
        }
}
```

Listing 4.3: Exemplo lookup

```
/*
 * We will be called everytime we have a success or failure
 * for any lookup request.
 */
void
my_lookup_A_hook(struct mdns *m, int ev, const char *host,
    struct in_addr a)
{
        switch (ev) {
        case MDNS_LOOKUP_SUCCESS:
                printf("Address:␣%s\n", inet_ntoa(a));
                break;
        case MDNS_LOOKUP_FAILURE:
                printf("Address␣not␣found\n");
                break;
        default:
                errx(1, "Unhandled␣event");
                break; /* NOTREACHED */
        }
}
```

```
int
main(void)
{
        struct mdns    m;
        int            mdns_sock;

        /* Open a connection to mdnsd, get the socket in mdns_sock. */
        mdns_sock = mdns_open(&m);
        if (mdns_sock == -1)
                err(1, "mdns");
        /* Set our lookup hook, will be called when we get an answer. */
        mdns_set_lookup_A_hook(&m, my_lookup_A_hook);
        /* Ask for foobar.local */
        if (mdns_lookup_A(&mdns, "foobar.local") == -1)
                err(1, "mdns_lookup_A");
        /* Wait for an answer. */
        for (; ;)
                if (mdns_read(&m) <= 0)
                        break;

        return (0);
}
```

Listing 4.4: Exemplo publicação

```
/*
 * We will be called everytime we have a group event.
 */
void
my_group_hook(struct mdns *m, int ev, const char *group)
{
        switch (ev) {
        case MDNS_GROUP_ERR_COLLISION:
                printf("Group␣%s␣got␣a␣collision,␣not␣published\n",
                    group);
                exit(1);
```

```
                break;
        case MDNS_GROUP_ERR_NOT_FOUND:
                printf("Group␣%s␣not␣found,␣this␣is␣an␣internal␣error,"
                    "␣please␣report\n", group);
                exit(1);
                break;
        case MDNS_GROUP_ERR_DOUBLE_ADD:
                printf("Group␣%s␣got␣a␣double␣add,␣ignore␣for␣now...\n",
                    group);
                exit(1);
                break;
        case MDNS_GROUP_PROBING:
                printf("Group␣%s␣is␣probing...\n", group);
                break;
        case MDNS_GROUP_ANNOUNCING:
                printf("Group␣%s␣is␣announcing...\n", group);
                break;
        case MDNS_GROUP_PUBLISHED:
                printf("Group␣%s␣published.\n", group);
                break;
        default:
                warnx("Unhandle␣group␣event");
                break;
        }
}
int
main(void)
{
        struct mdns             m;
        struct mdns_service     ms;
        int                     mdns_sock;

        /* Open a connection to mdnsd, get the socket in mdns_sock. */
        mdns_sock = mdns_open(&m);
        if (mdns_sock == -1)
```

```
                err(1, "mdns");
        /* Set our group hook, will be called when we a group event. */
        mdns_set_group_hook(&m, my_group_hook);
        /* We chose ftp_group as our group identifier. */
        group = "ftp_group";
        /* Create the group */
        if (mdns_group_add(&mdns, group) == -1)
                err(1, "mdns_group_add");
        /*
         * Set our service parameters in ms:
         * Name is ftp_server_86.
         * Protocol ftp/tcp
         * Port 21
         * Anonymous login is welcome.
         */
        if (mdns_service_init(&ms, "ftp_server_86", "ftp", "tcp",
           21, "anonymous=yes", NULL) == -1)
                errx(1, "mdns_service_init");
        /* Add the service to our group */
        if (mdns_group_add_service(&mdns, group, &ms) == -1)
                errx(1, "mdns_group_add_service");
        /* Commit the group and hope for the best */
        if (mdns_group_commit(&mdns, group) == -1)
                errx(1, "mdns_group_commit");
        /* Wait for an answer. */
        for (; ;)
                if (mdns_read(&m) <= 0)
                        break;

        return (0);
}
```

Referências

1 CHESHIRE, M. K. S. *Multicast DNS*. [S.l.], set. 2010. At http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt.

2 CHESHIRE, M. K. S. *DNS Service Discovery*. [S.l.], maio 2010. At http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt.

3 MOCKAPETRIS, P. V. *RFC 1035: Domain names — implementation and specification*. nov. 1987. Disponível em: <ftp://ftp.internic.net/rfc/rfc1035.txt>.

4 BONJOUR. Disponível em: <http://developer.apple.com/networking/bonjour/download/>.

5 STUART Cheshire. Disponível em: <http://www.stuartcheshire.org/>.

6 APACHE-2 License. Disponível em: <http://www.apache.org/licenses/LICENSE-2.0.html>.

7 AVAHI. Disponível em: <http://www.avahi.org>.

8 BLOAT. Disponível em: <http://en.wikipedia.org/wiki/Code_bloat>.

9 OPENBSD Policy. Disponível em: <http://www.openbsd.org/policy.html>.

10 OPENBSD, Operating System. Disponível em: <http://www.openbsd.org>.

11 ISC License. Disponível em: <http://opensource.org/licenses/isc-license.txt>.

12 NetBIOS Working Group. *RFC 1001: Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods*. mar. 1987. See also STD0019 . Status: STANDARD. Disponível em: <ftp://ftp.internic.net/rfc/rfc1001.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc1001.txt>.

13 CHESHIRE, M. K. S. *Requirements for a Protocol to Replace AppleTalk NBP*. [S.l.], set. 2010. At http://files.dns-sd.org/draft-cheshire-dnsext-nbp.txt.

14 CHESHIRE, M. K. S. *Requirements for a Protocol to Replace AppleTalk NBP*. [S.l.], mar. 2010. At http://files.multicastdns.org/draft-cheshire-dnsext-nbp-08.txt.

15 GETADDRINFO Library Call. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=getaddrinfo>.

16 GETHOSTBYNAME Library Call. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=gethostbyname>.

17 OPENBGP. Disponível em: <http://www.openbgp.org>.

18 OPENOSPF. Disponível em: <http://www.openbsd.org>.

19 OPENSSH. Disponível em: <http://www.openssh.com>.

20 DHCP Client. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=dhclient>.

21 NETBSD, Operating System. Disponível em: <http://www.netbsd.org>.

22 THE OpenBSD packages and ports system. Disponível em: <http://www.openbsd.org/faq/faq15.html>.

23 EXPLOIT Mitigation Techniques. Disponível em: <http://www.openbsd.org/papers/ven05-deraadt/>.

24 LEE. The problem with threads. *COMPUTER: IEEE Computer*, v. 39, 2006.

25 LIBEVENT. Disponível em: <http://monkey.org/ provos/libevent/>.

26 POLL System Call. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=poll>.

27 SELECT System Call. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=select>.

28 IMSG Framework. Disponível em: <"http://www.openbsd.org/cgi-bin/man.cgi?query=imsg_init">.

29 SOFTRAID. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=softraid>.

30 ACPI. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=acpi>.

31 MOCKAPETRIS, P. *Domain Names - Concepts and Facilities*. [S.l.], nov. 1987. At http://info.internet.isi.edu/in-notes/rfc/files/rfc1034.txt.

32 FORK System Call. Disponível em: <http://www.openbsd.org/cgi-bin/man.cgi?query=fork>.

33 THEO De Raadt Interview. Disponível em: <http://www.thejemreport.com/content/view/369/>.

34 BSD License. Disponível em: <http://www.opensource.org/licenses/bsd-license.php>.

35 Brooks, Jr., F. P. *The Mythical Man Month: Essays on Software Engineering*. first. Addison-Wesley, 1975. 200 p. Disponível em: <http://www.amazon.com/Mythical-Man-Month-Essays-Software-Engineering/dp/0201006502>.

36 RAYMOND, E. S. *The Art of UNIX Programming*. pub-AW:adr: Addison-Wesley, 2004. xxxii + 525 p. ISBN 0-13-124085-4.

37 STEVENS, R. W. *UNIX Network Programming*. [S.l.]: Prentice Hall PTR, 1990. (Software Series).

38 STEVENS, R. W. *TCP/IP Illustrated, Volume 1: The Protocols*. Reading: Addison Wesley, 1994. ISBN 0-201-63346-9.

39 STEVENS, W. R. *TCP/IP Illustrated, Volume 2*. [S.l.]: Addison Wesley, 1994.

40 STEVENS, W. R. *Advanced Programming in the UNIX Environment*. [S.l.]: Addison-Wesley, 1992.

41 HUSEMANN, M. Fighting the lemmings. Unspecified.