Universidade de Aveiro
2008
Departamento de Electrónica, Telecomunicações e Informática

Luís André Vela dos Reis

Empacotamento, Multiplexagem e Encriptação para TV

Universidade de Aveiro
2008
Departamento de Electrónica, Telecomunicações e Informática

Luís André Vela dos Reis

Empacotamento, Multiplexagem e Encriptação para TV

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor António José Nunes Navarro Rodrigues, Professor Auxiliar do Departamento de Engenharia Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri

presidente

Professor Doutor Rui Jorge Morais Tomaz Valadas
Professor Associado com Agregação da Universidade de Aveiro

vogais

Professor Doutor António José Nunes Navarro Rodrigues
Professor Auxiliar da Universidade de Aveiro

Professor Doutor Fernando José Pimentel Lopes
Professor Coordenador do Departamento de Engenharia Electrotécnica do Instituto Superior de Engenharia de Coimbra

agradecimentos

Este espaço é dedicado àqueles que deram a sua contribuição para que esta dissertação fosse realizada. A todos eles deixo aqui o meu agradecimento sincero.

Desejo expressar o meu reconhecido agradecimento ao Prof. Doutor António Navarro que propôs e orientou esta dissertação de Mestrado. As notas dominantes da sua orientação foram a utilidade das suas recomendações e a cordialidade com que sempre me recebeu.

Quero também deixar uma palavra de agradecimento aos meus colegas de grupo pelo ambiente de camaradagem e pelo companheirismo que demonstraram, em especial ao Tiago Caçoilo, Nuno Coelho e Nélson Cabral pela ajuda prestada no decorrer do desenvolvimento do trabalho.

Ao Instituto de Telecomunicações – Pólo de Aveiro pelas condições de trabalho que me proporcionaram.

De uma forma muito especial aos meus pais e irmãos pelo apoio e motivação.

palavras-chave

Sistema de Comunicação, Interface *Sockets,* Encriptação, Transmissão de vídeo, modelo TCP/IP, IPv4, IPv6, modelo OSI, RTP, DES, RC6, AES.

resumo

O presente trabalho propõe-se a estudar e implementar um sistema de comunicação e encriptação para TV, ou seja transmissão de vídeo através da Internet de forma segura.

Para tal, realiza-se inicialmente uma descrição da evolução da Internet, realiza-se uma descrição do modelo TCP/IP, ao nível da camada Rede, o protocolo IP (*Internet Protoco*l) é utilizado não apenas para o endereçamento mas também para a interligação de redes de telecomunicações, com tecnologias heterogéneas nas camadas inferiores (ligação de dados e física). Acima da camada de rede, a informação é encapsulada e transportada pelos protocolos UDP (*User Datagram Protocol*) e RTP (*Real Time Transport Protocol*). O principal objectivo destes protocolos é identificar a existência de erros nos dados e associar informação de sincronismo e sequência. Esta informação permite ao receptor minimizar os efeitos dos erros de comunicação e sincronizar diferentes conteúdos de vídeo.

Em seguida, introduz-se e define-se o conceito de interface *socket* em sistemas de comunicações. Descrevem-se os princípios e as funções desta interface necessária para a transmissão poder ocorrer. Logo de seguida são apresentados alguns protocolos de encriptação.

Com base nestes princípios, propõe-se e descreve-se um algoritmo para a transmissão de vídeo, bem como a sua encriptação, de modo a oferecer algum tipo de segurança durante a transmissão.

Finalmente, apresentam-se os resultados relativos de uma comunicação multimédia multiplexada em pacotes.

keywords

Communication System, *Sockets* Interface, Encryption, Video Transmission, TCP/IP Model, IPv4, IPv6, OSI Model, RTP, DES, RC6, AES.

abstract

This thesis explores and implements a communication system and encryption for TV, i.e. video transmission over the Internet in safe way.

To achieve that goal, we first review the evolution of the Internet, then we made a description of the TCP/IP model, at the network layer, the IP (Internet Protocol) is used not only for addressing purposes, but also for the interconnection of diverse telecommunications networks, with heterogeneous technologies at lower layers (data link and physical). Above the network layer, the information is encapsulated and transported by UDP (User Datagram Protocol) and RTP (Real Time Transport Protocol) protocols. The main objective of these protocols is to signal any communications errors and to include synchronization information as well as inserting packet sequence numbers.

Then, we introduced and defined the concept of *socket* interface in communications systems. We outline the basic principles and functions of this interface needed to support the transmission. Then, there are presented some encryption protocols.

Based on these principles, it is proposed and described an algorithm for the video transmission and its encryption in order to provide some type of security during the transmission.

Finally, we present the results for a multimedia communication encapsulated in packets.

"Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, while imagination embraces the entire world, and all there ever will be to know and understand."

Albert Einstein (1879-1955)

Índice

Índice de Figuras

Índice de Tabelas

Lista de Acrónimos

AES	*Advanced Encryption Standard*
AKC	*Asymmetric Key Cryptography*
ANSI	*American National Standards Institute*
API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
ARPA	*Advanced Research Projects Agency*
ASCII	*American Standard Code for Information Interchange*
ATM	*Asynchronous Transfer Mode*
BGP	*Border Gateway Protocol*
BSD	*Berkeley Software Distribution*
CIDR	*Classless Inter-Domain Routing*
CIF	*Common Intermediate Format*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
CSRC	*Contributing Source*
DARPA	*Defense Advanced Research Projects Agency*
DCA	*Defense Communication Agency*
DCE	*Data Circuit-terminating Equipment*
DEC	*Digital Equipment Corporation*
DES	*Data Encryption Standard*
DNS	*Domain Name Service*
DoD	*Department of Defense*
DTE	*Data Terminal Equipment*
FCS	*Frame check sequence*
FDDI	*Fiber Distributed Data Interface*
FIPS	*Federal Information Processing Standards*

FTP	*File Transfer Protocol*
GF	*Galois Field*
HD	*High Definition video*
HDLC	*High-Level Data Link Control*
HDTV	*High Definition Television*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
I/O	*Input/Output*
IAB	*Internet Architecture Board*
IANA	*Internet Assigned Numbers Authority*
ICCB	*Internet Configuration Control Board*
ICMP	*Internet Control Message Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
IESG	*Internet Engineering Steering Group*
IETF	*Internet Engineering Task Force*
IGMP	*Internet Group Message Protocol*
IMAP	*Internet Mail Access Protocol*
IP	*Internet Protocol*
IPng	*Internet Protocol next generation*
IPv4	*IP Version 4*
IPv6	*IP Version 6*
IRSG	*Internet Research Steering Group*
IRTF	*Internet Research Task Force*
ISDN	*Integrated Services Digital Network*
ISN	*Initial Sequence Number*
ISO	*International Standards Organization*
ISOC	*Internet Society*
ISP	*Internet Service Provider*
ITU-T	*International Telecommunications Union - Telecommunication Standardization Sector*
LAN	*Local Area Networks*
LLC	*Logical Link Control*
LSB	*Least Significant Byte*
MAC	*Medium Access Control*

MAC	*Message Authentication Code*
MAN	*Metropolitan Area Network*
MBZ	*Must be Zero*
MD5	*Message-Digest algorithm 5*
MIME	*Multipurpose Internet Mail Extension*
MSB	*Most Significant Byte*
MSS	*Maximum Segment Size*
NBS	*National Bureau of Standards*
NCP	*Network Control Protocol*
NFS	*Network File System*
NIST	*National Institute of Standards and Technology*
NNI	*Network-Network Interface*
NSA	*National Security Agency*
NTP	*Network Time Protocol*
OSI	*Open Systems Interconnection*
OSPF	*Open Shortest Path First*
PDN	*Public Data Networks*
PEM	*Privacy Enhanced Mail*
PGP	*Pretty Good Privacy*
PKC	*Public Key Cryptography*
POP	*Post Office Protocol*
PPP	*Point-to-Point Protocol*
PSE	*Packet-Switching Exchange*
PSN	*Packet-Switched Networks*
QoS	*Quality of Service*
RAM	*Random Access Memory*
RARP	*Reverse Address Resolution Protocol*
RC6	*Rivest Cipher ou Ron's Code*
RFC	*Requests For Comments*
RIP	*Routing Information Protocol*
RPC	*Remote Procedure Call*
RSA	*Rivest, Shamir and Adleman*
RTCP	*Real-time Transport Control Protocol*

RTP	*Real-Time Transport Protocol*
S/MIME	*Secure/Multipurpose Internet Mail Extension*
SACK	*Selective Acknowlegment*
SD	*Standard Definition video*
SIP	*Session Initiation Protocol*
SKC	*Symmetric Key Cryptography*
SMTP	*Simple Mail Transport Protocol*
SNMP	*Simple Network Management Protocol*
SSL	*Secure Sockets Layer*
SSRC	*Synchronization Source*
SUIT	*Scalable, Ultra-fast and Interoperable Interactive Television*
TCP	*Transmission Control Protocol*
TFTP	*Trivial File Transfer Protocol*
TLS	*Transport Layer Security*
TOS	*Type of Service*
TSOPT	*Timestamp Option*
TTL	*Time to Live*
UDP	*User Datagram Protocol*
UNI	*User-Network Interface*
VCI	*Virtual Channel Identifier*
VoIP	*Voice over IP*
VPI	*Virtual Path Identifier*
WAN	*Wide Area Network*
WSOPT	*Window Scale Option*
WWW	*Worl Wide Web*
XOR	*eXclusive OR*

Capítulo 1

Introdução

A Internet é de âmbito mundial, mas este *Internetwork* global é um meio de transmissão inseguro. A Internet revolucionou o mundo da informática e das comunicações para fins de desenvolvimento e suporte de serviços entre clientes e servidores. A disponibilidade da Internet, aliada à computação e comunicações potentes, tornou-se possível um novo paradigma do mundo comercial. Este tem sido tremendamente acelerado pela adopção de *browsers* e tecnologia *World Wide Web*, permitindo aos utilizadores um acesso fácil a informações ligadas por todo o globo. A Internet realmente provou ser um veículo essencial para a troca de informações nos dias de hoje.

1.1 Enquadramento e Motivação

A Internet é hoje uma infra-estrutura de informação, um mecanismo para a difusão de informação, e um meio de colaboração e de interacção entre os indivíduos, instituições académicas, instituições financeiras, e as empresas, sem preocupação da localização geográfica.

As pessoas estão cada vez mais dependentes da Internet para uso pessoal e profissional, com o aumento da consciência e popularidade da Internet, problemas de segurança da Internet, começaram a aparecer. A segurança da Internet é não só extremamente importante, mas tecnicamente mais complexa do que no passado, o simples facto de negócios e serviços on-line serem realizados através de um meio de transmissão inseguro é suficiente para estimular a actividade criminosa na Internet. O acesso à Internet muitas vezes cria

uma ameaça como uma falha de segurança. Para proteger os utilizadores de ataques baseados na Internet e para fornecer soluções adequadas quando a segurança é necessária, técnicas de encriptação devem ser utilizadas para resolver esses problemas.

O mundo depende cada vez mais de informações sobre sistemas de comunicações, que são neste momento baseados principalmente na Internet ou pelo menos sobre o IP (*Internet Protocol*). A solução para todos os tipos de ameaças criadas por actividades criminosas deve basear-se na utilização de operações de criptografia. A autenticação, encriptação e integridade da mensagem são muito importantes para melhorar e promover a segurança na Internet. Sem procedimentos de autenticação, um invasor poderia personificar qualquer pessoa e, depois, obter acesso à rede. A integridade da mensagem é necessária, uma vez que é transmitida através da Internet, os dados podem vir a ser alterados, tornando assim a informação enganosa ou mesmo inútil. Sem confidencialidade através da encriptação, a informação pode tornar-se pública.

De modo a reflectir a papel da interface *sockets,* bem como as operações criptográficas, princípios, algoritmos e protocolos na Internet, será desenvolvido um sistema de comunicação para a transmissão de vídeo, motivação suficiente para dedicarmos neste trabalho atenção aos princípios, desafios e benefícios proporcionados por uma concepção de rede através da interface *socket*.

1.2 Objectivos e Estrutura da Dissertação

O objectivo concreto desta dissertação é o desenvolvimento de um sistema de comunicação para a transmissão de vídeo, utilizando a interface *socket* e utilizando protocolos de encriptação para efectuar uma transmissão segura.

A dissertação está organizada em 7 capítulos e 2 anexos, que se descrevem em seguida.

O primeiro capítulo é dedicado às notas introdutórias e motivações que norteiam este trabalho.

No segundo capítulo direcciona-se a atenção para uma descrição da evolução das redes de comunicação. Além disso, apresenta-se também, o modelo de referência OSI e o modelo TCP/IP.

O terceiro capítulo é dedicado a uma descrição mais aprofundada do modelo TCP/IP, não só na abordagem e desenvolvimento de um sistema de comunicações, mas fundamentalmente, para o devido enquadramento do trabalho realizado no âmbito desta dissertação.

No quarto capítulo é apresentada uma descrição detalhada, quer da constituição, quer o funcionamento da Interface *Socket*.

No quinto capítulo abordam-se os conceitos relacionados com os protocolos de encriptação, bem como a sua implementação.

No sexto capítulo, apresenta-se o sistema de comunicação e os algoritmos de encriptação desenvolvidos.

Finalmente, o sétimo capítulo dá conta das principais conclusões, tecendo considerações de vária ordem sobre o trabalho realizado e aponta caminhos para investigações futuras.

No Anexo A, é feita uma demonstração dos resultados do trabalho implementado, servindo de complemento ao Capitulo 6.

O Anexo B apresenta a plataforma e as ferramentas utilizadas para o desenvolvimento do trabalho.

1.3 Principais Contribuições

As principais contribuições do trabalho descrito nesta dissertação podem-se resumir da seguinte forma:

- Estudo e abordagem dos conceitos da interface *socket*, para a transmissão de vídeo através da Internet.
- Estudo das principais técnicas de encriptação, de modo a fornecer segurança durante a transmissão de vídeo através da Internet.

1.4 Notação Utilizada

Nesta dissertação são utilizados alguns termos em inglês, apresentados em caracteres itálicos, cuja tradução para português não existe ou não reflecte o seu real significado. Tal

deve-se ao facto de esses termos terem origem na língua inglesa, e de serem universalmente utilizados na literatura existente nesta área. Sempre que possível são também utilizadas algumas traduções apropriadas ou que já se encontram enraizadas no Português.

Para evitar a repetição de longas expressões técnicas, são também utilizados acrónimos ao longo do texto. Para além de serem apresentados no início deste documento, a correspondência entre os termos técnicos e os respectivos acrónimos é normalmente feita na primeira ocorrência do acrónimo no texto.

As referências bibliográficas utilizadas são evocadas entre parênteses recto e apresentadas no final desta dissertação.

Capítulo 2

Redes de Comunicação (*Internetworking*)

A Internet tornou-se numa parte fundamental da vida, hoje em dia quase toda a gente possui um correio electrónico, acede diariamente ao *World Wide Web* que contém informações sobre os mais diversos temas, tais como condições atmosféricas, produção agrícola, os preços das acções, e as condições do trânsito.

A Internet é a principal das novas tecnologias de informação e comunicação (NTICs). De acordo com dados de Março de 2008, a Internet é usada por 21,1% da população mundial (cerca de 1,408 bilião de pessoas) [1].

Uma vez, que a maior parte das tecnologias de rede são concebidas para uma finalidade específica, torna-se impossível de gerir uma rede universal a partir de uma única tecnologia de rede, porque uma única tecnologia de rede não é suficiente para todas as finalidades pretendidas. Tem então, evoluído ao longo de mais de duas décadas uma nova tecnologia que torna possível interligar diferentes redes físicas, chamada de *Internetworking*. Esta tecnologia acomoda múltiplos e diversos tipos de hardware, proporcionando assim uma forma de interligar redes heterogéneas [2].

Internetworking é uma colecção de redes individuais, ligadas por intermédio de dispositivos de rede, que funcionam como uma única grande rede. A Figura 2.1 ilustra alguns dos diferentes tipos de tecnologias de rede que podem ser interligadas por *routers* e outros dispositivos de rede de modo a ser criada uma rede única [3].

O exemplo mais notável de *internetworking*, na prática, é a Internet, uma rede de redes a correr diferentes protocolos de baixo nível, interligados pelo Protocolo de Internet (IP –

Internet Protocol). A Internet esconde os detalhes de hardware da rede e permite a comunicação entre computadores independentemente das suas ligações físicas.

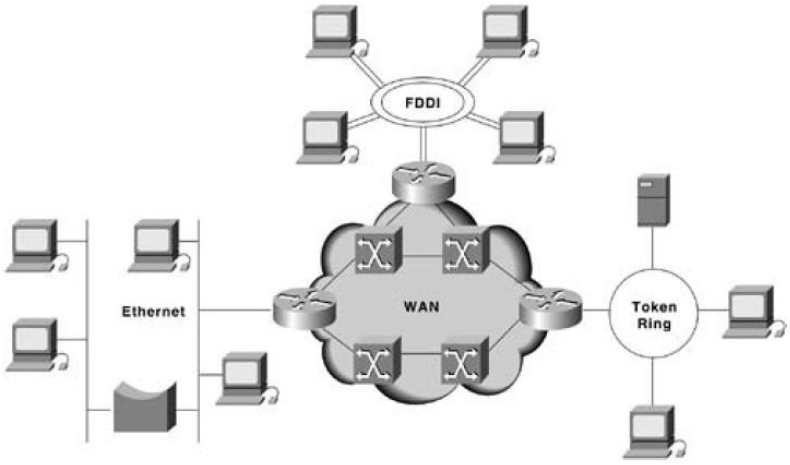

Figura 2.1 – Conexão de diferentes tecnologias de rede

A Internet é um exemplo do modelo *Open Systems Interconnection* (OSI), tendo sido concebido para promover a comunicação entre máquinas com diversas arquitecturas, permitindo assim usar praticamente qualquer hardware de redes de comutação de pacotes, utilizar uma maior variedade de aplicações e permitir a interligação de vários computadores com diferentes sistemas operativos [2].

O Protocolo de Internet (IP) sozinho não fornece um serviço fidedigno através da Internet, isto significa que o IP não efectua o contacto e a conexão com o destino antes de iniciar a transferência, para transferir dados com fiabilidade, deve ser utilizado um protocolo da camada de transporte do modelo OSI, como o *Transmission Control Protocol* (TCP).

Uma vez que o TCP é o protocolo de transporte mais utilizado, geralmente o TCP e IP são referidos em conjunto, como TCP/IP. Mas, algumas aplicações usam o *User Datagram Protocol* (UDP) como protocolo de transporte, para funções que não exigem um serviço absolutamente fidedigno, tal como o *streaming* de vídeo [3].

Neste capítulo, pretende-se fazer uma breve descrição da evolução da Internet ao longo dos tempos, bem como, apresentar o modelo OSI e o modelo TCP/IP.

2.1 Perspectiva Histórica

O antecessor da Internet foi o ARPANET, criado pela *Advanced Research Projects Agency* (ARPA), ou Agência de Pesquisa de Projectos Avançados, uma subdivisão do Departamento de Defesa dos Estados Unidos e que foi lançada em 1969. Esta rede foi criada em resposta à potencial ameaça de ataque nuclear por parte da União Soviética. Um dos objectivos primordiais da ARPA era conceber uma rede tolerante a falhas que permitiria aos líderes militares dos Estados Unidos ficarem em contacto em caso de guerra nuclear.

O protocolo usado na ARPANET foi designado de *Network Control Protocol* (NCP), o TCP/IP ainda não tinha sido desenvolvido. Com o crescimento da ARPANET, verificou-se a necessidade de um novo protocolo, dado que o NCP não conseguia cumprir todas as necessidades de uma rede maior. O protocolo NCP possuía uma linguagem muito semelhante à linguagem humana, mas que continha apenas algumas palavras [4, 5].

Nos finais dos anos 70, com o crescimento da Internet o interesse da comunidade científica aumentou, levando à necessidade da criação de um mecanismo de coordenação. A *Defense Advanced Research Projects Agency* (DARPA), decidiu então formar o *Internet Configuration Control Board* (ICCB) para coordenar e orientar a concepção dos protocolos e arquitectura da Internet, referimo-nos logicamente ao protocolo TCP/IP, oficialmente chamado de TCP/IP *Internet Protocol Suite*, desenvolvido, entre outros, por Robert Kahn, Vinton Cerf, e que rapidamente se tornou o protocolo de rede mais utilizado no mundo [6].

Para melhorar a tecnologia que estava a ser usada na ARPANET, foi desenvolvido um sistema de modo a incentivar e facilitar a correspondência entre os engenheiros que participavam no desenvolvimento desta nova rede. Este sistema, que ainda hoje em dia está em uso, assenta sobre os *Requests For Comments* (RFCs) no sentido de fornecer *feedback* e colaboração entre engenheiros. Um RFC é um documento escrito por um engenheiro, uma equipa de engenheiros, ou apenas alguém com uma ideia, para definir uma nova tecnologia ou reforçar uma tecnologia existente [4]. Durante muitos anos, apenas Jon Postel, foi editor dos RFCs, tornando-se assim num dos pioneiros e contribuinte mais significativo para o desenvolvimento do TCP/IP e Internet. A tarefa de edição dos RFCs recai agora para os gestores da área do *Internet Engineering Task Force* (IETF), enquanto o *Internet Engineering Steering Group* (IESG), como um todo, aprova os novos RFCs.

A Internet teve o seu início por volta do ano de 1980, altura em que a ARPA começou a converter as máquinas que utilizava nas suas redes de investigação para o novo protocolo TCP/IP. A ARPANET, rapidamente se tornou a *backbone* da nova Internet. A transição para a Internet ficou concluída em Janeiro de 1983, quando o Gabinete do Secretário de Defesa mandatou que todos os computadores conectados em redes de longa distância usassem TCP/IP. Ao mesmo tempo, a *Defense Communication Agency* (DCA) dividiu a ARPANET em duas redes, uma para investigação e outra para ser utilizada em comunicações militares. A rede para investigação manteve o nome de ARPANET, enquanto a outra foi renomeada para *military Network*, MILNET [2].

Com o constante crescimento da comunidade da Internet, a DARPA, em 1983 procedeu a uma nova reestruturação dos mecanismos de coordenação. O ICCB foi dissolvido e no seu lugar foi criado o *Internet Architecture Board* (IAB). O IAB era composto principalmente por dois grupos o *Internet Engineering Task Force* (IETF) e o *Internet Research Task Force* (IRTF). O IETF foi dividido em cerca de dez áreas, todas elas geridas pelo *Internet Engineering Steering Group* (IESG), tal como o IETF, o IRTF possui também o grupo, *Internet Research Steering Group* (IRSG), este foi criado para definir e coordenar as actividades de investigação. Ao contrário do que se possa pensar, a maior parte da investigação era feita dentro do IETF, o IRTF era de facto uma organização mais pequena e pouco activa [2, 5, 6].

Em 1992, à medida que a Internet se afastava do governo dos Estados Unidos, foi formada uma sociedade para incentivar a participação na Internet. Foi designada de *Internet Society* (ISOC), o grupo é uma organização internacional inspirada pela *National Geographic Society*. O IAB funcionaria assim dentro da *Internet Society,* mas continuaria a ajudar as pessoas a entrar e a utilizar a Internet no mundo inteiro [2].

2.2 Tecnologias de Rede

A transmissão de dados de um dispositivo para outro ocorre utilizando meios de transmissão, incluindo, entre outros, o cabo coaxial e a fibra óptica. Uma mensagem a transmitir representa a unidade básica de uma rede de comunicações, uma mensagem pode ser constituída por uma ou mais células, *frames* ou pacotes que são as unidades elementares de uma rede de comunicações. Das tecnologias de rede destacam-se as redes

limitadas geograficamente, como um único edifício, departamento ou campus, denominadas de *Local Area Networks* (LANs), bem como as redes *Wide Area Networks* (WANs) que abrangem grandes áreas geográficas, que podem compreender um país, um continente ou mesmo o mundo inteiro, interligando LANs de modo a obter conectividade [3, 6].

Nos dias de hoje, as LANs de alta velocidade estão a tornar-se largamente utilizadas, em grande parte porque operam em velocidades muito altas e suportam aplicações multimédia e de vídeo-conferência, que requerem uma largura de banda elevada.

O *Internetworking* evoluiu como forma de solucionar três problemas essenciais: LANs isoladas, duplicação de recursos e a falta de gestão das redes [3].

2.2.1 *Local Area Networks* (LANs)

As redes *Local Area Networks* (LANs) foram originalmente concebidas como *shared media* (camada Ligação de Dados, camada 2 do modelo OSI) e são ideais para distâncias relativamente curtas, uma área geográfica limitada, tais como um edifício, um armazém ou um campus. Ao sacrificar conexões de longas distâncias permitindo-nos velocidades de transmissão mais elevadas, tornaram-se no alicerce para os escritórios modernos - interligando servidores, processadores de texto, impressoras, sistemas de correio electrónico, servidores Web e por aí fora. Os vários tipos de LANs podem ser caracterizados pelas suas topologias distintas, mas todos partilham o mesmo método de transmissão, ou seja, interligam todos os dispositivos através dos mesmos protocolos, o *logical link control* (LLC) e o *medium access control* (MAC). As topologias mais comuns das LANs são as topologias estrela, anel e barramento, ilustradas na Figura 2.2.

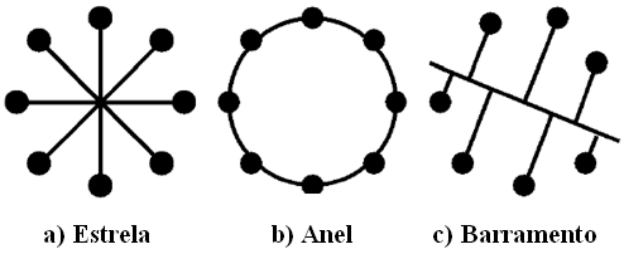

Figura 2.2 – Topologias LAN

A norma mais conhecida de uma rede LAN é a *Ethernet* que usa o protocolo IEEE 802.3 (ISO 8802.3) que define na camada física um protocolo denominado de CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) que normalmente é utilizado com uma topologia do tipo barramento. Temos também o *Token Bus*, IEEE 802.4 (ISO 8802.4) que define um protocolo alternativo na camada física (camada 1 do modelo OSI) que usa a topologia barramento e que também pode usar a topologia estrela. Temos ainda o *Token Ring*, IEEE 802.5 que define um protocolo adequado na camada física de modo a se usar uma topologia do tipo anel. Por fim, temos o *Fiber Distributed Data Interface* (FDDI), que usa uma topologia do tipo anel, e que utiliza o protocolo IEEE 802.2. Por sua vez, o protocolo IEEE 802.2 (ISO 8802.2) da camada *logical link control* (LLC) (equivalente à camada Ligação de Dados, camada 2 do modelo OSI) pode ser usado com qualquer um dos protocolos da camada física, referidos acima. A Figura 2.3 mostra os diferentes protocolos e sua relação com as camadas do modelo OSI. A principal diferença entre as diferentes tecnologias LAN reside no modo como previnem que mais do que um terminal utilize o barramento ou anel ao mesmo tempo [2, 6-8].

Camada Ligação de Dados (Camada 2 do OSI)	*Logical Link Control* (LLC)	IEEE 802.2		
	Medium Access Control (MAC)	IEEE 802.3 *Ethernet*	IEEE 802.4 *Token bus*	IEEE 802.5 *Token ring*
Camada Física (Camada 1 do OSI)	*Protocolo da camada Física* (PHY)			
	Physical medium dependet (PMD)			

Figura 2.3 – Normas IEEE para LANs

2.2.1.1 *Ethernet*

A *Ethernet* é uma norma de rede LAN originalmente desenvolvida pela *Xerox*, à qual, posteriormente se juntaram a *Digital Equipment Corporation* (DEC) e a *Intel Corporation*. A Ethernet tem as suas raízes numa rede *Packet Radio Nertwork*, chamada *Aloha*, que foi desenvolvida pela Universidade do Havai com o objectivo de suportar a comunicação entre computadores sobre as Ilhas Havaianas. Tal como numa rede baseada em *Aloha*, o problema fundamental com que a *Ethernet* se deparou foi a forma de mediar o acesso de uma forma justa e eficiente ao meio utilizado para efectuar a transmissão (no caso particular de uma rede *Aloha*, o meio utilizado foi a atmosfera, enquanto que na *Ethernet* o

meio utilizado é o cabo coaxial). A ideia fulcral de ambas as redes, *Aloha* e *Ethernet*, é um algoritmo que controla quando cada nó pode transmitir [9].

Como foi referido a *Ethernet* utiliza o protocolo IEEE 802.3 (ISO 8802.3), também conhecido como *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD). No CSMA/CD, as estações transmitem e recebem no mesmo canal, antes de uma estação transmitir, esta deve escutar o meio para verificar que nenhuma outra estação já não o está a utilizar, se nenhuma outra estação está a transmitir, então esta pode começar a fazê-lo.

Com este método o número de colisões é minimizado, podem, no entanto ocorrer colisões porque as estações estão a uma distância considerável umas das outras. Caso duas ou mais estações comecem a transmitir ao mesmo tempo, isso pode resultar numa colisão, por isso, todas as estações devem continuamente escutar o meio para detectar qualquer colisão que possa ocorrer. Se uma colisão ocorrer, todas as estações devem ignorar os dados recebidos, e as estações que fizeram a transmissão esperam um período de tempo antes de reenviarem os dados. Para reduzir a possibilidade de uma segunda colisão, as estações individualmente geram um número aleatório que determina quanto tempo a estação deve esperar antes de reenviar os dados [2, 3, 6, 7].

Actualmente estão definidas três taxas de transmissão para operar em fibra óptica e cabo de pares entrelaçados (*twisted-pair*):

- 10 Mbps – 10BASE-T Ethernet
- 100 Mbps – Fast Ethernet
- 1000 Mbps – Gigabit Ethernet

A norma 10 Gigabit Ethernet foi publicada em 2002, sendo designada de IEEE 802.3ae, esta não suporta o CSMA/CD, uma vez que opera apenas ponto a ponto, nem o *Half-Duplex*, apenas suporta o *Full-Duplex*. Uma desvantagem da rede 10 Gigabit Ethernet é que ela pode somente ser ponto a ponto, sendo utilizada principalmente em *backbones* estando já implantada no projecto *Internet2*.

Outras tecnologias e protocolos têm sido apresentados como prováveis substitutos, mas a *Ethernet* tem sobrevivido como a principal tecnologia LAN, porque o seu protocolo tem as seguintes características:

- É fácil de compreender, implementar, gerir e manter
- Permite implementações de redes de baixo custo
- Fornece flexibilidade na instalação
- Garantia de interligação bem sucedida e funcionamento em produtos implementados com as normas, independentemente do fabricante [3].

2.2.1.2 *Token Ring*

O *Token Ring* é uma norma LAN (definida pelo protocolo IEEE 802.5) em que utiliza um *token* para atribuir o direito de transmitir os dados na LAN. Originalmente desenvolvido pela IBM, usa uma topologia do tipo anel, e o método utilizado para o acesso pelo CSMA/CD pode resultar em colisões. Por isso, as estações podem tentar enviar dados muitas vezes antes de uma transmissão captar uma ligação perfeita. Não há nenhuma maneira, quer para prever a ocorrência de colisões ou os atrasos produzidos por várias estações na tentativa de capturar a ligação ao mesmo tempo. O *Token Ring* resolve essa incerteza ao fazer a estações alternarem no envio de dados.

Um pacote especial, chamado de *token* circula pela rede, sendo transmitido de estação para estação. Quando uma estação precisa de transmitir dados, ela espera até que o *token* chegue e, em seguida, começa a transmitir os seus dados. A transmissão de dados em redes *token* é diferente, em vez de serem propagados para toda a rede, os pacotes são transmitidos de estação para estação (daí a topologia lógica de anel). A primeira estação transmite para a segunda, que transmite para a terceira, etc. Quando os dados chegam à estação de destino, ela faz uma cópia dos dados para si, porém, continua a transmissão dos dados. A estação emissora continuará a enviar pacotes, até que o primeiro pacote enviado dê uma volta completa no anel e volte para ela, quando isto acontece, a estação pára de transmitir e envia o *token*, voltando a transmitir apenas quando receber novamente o *token.*

O sistema de *token* é mais eficiente em redes grandes e congestionadas, onde a diminuição do número de colisões resulta num maior desempenho em comparação com redes *Ethernet* semelhantes, dado que o anel funciona segundo uma disciplina do tipo *round-robin*, garantindo eficiência e equidade. Porém, em redes pequenas e médias, o sistema de *token* é

menos eficiente do que o sistema de barramento lógico das redes *Ethernet*, pois as estações têm de esperar mais tempo antes de poder transmitir [6, 7].

2.2.1.3 *Fiber Distributed Data Interface* (FDDI)

FDDI é um protocolo LAN normalizado pela ANSI (*American National Standards Institute*) e ITU-T (*International Telecommunications Union - Telecommunication Standardization Sector)*, sendo capaz de suportar taxas de transmissão de 100 Mbps e fornecer uma alternativa para a alta velocidade em relação ao *Ethernet* e ao *token ring*. Quando a tecnologia FDDI foi concebida, a taxa de transmissão de 100 Mbps exigia a utilização de fibra óptica.

A FDDI é uma *Metropolitan Area Network* (MAN) que pode ser utilizada para interligar LANs abrangendo mais de 100 km, permitindo alta velocidade de transferência. O método de acesso em FDDI também usa um *token*. Em uma rede *token ring*, uma estação pode enviar apenas uma *frame* cada vez que capta o token. Em FDDI, o acesso é limitado pelo tempo, cada estação tem um temporizador que mostra quando o *token* deve sair da estação. Se uma estação recebe o *token* mais cedo do que o tempo designado, é possível manter o *token* e enviar os dados até o tempo agendado terminar. Por outro lado, se uma estação recebe o *token* depois do tempo designado, esta deve deixar passar o *token* para a próxima estação e aguardar pelo próximo intervalo de transmissão.

Devido ao seu elevado custo e ao rápido desenvolvimento de tecnologias alternativas (incluindo o *Asynchronous Transfer Mode* (ATM) e, mais tarde, 100 Mbit/s *Fast Ethernet*), o FDDI caiu em declínio, já não sendo recomendado ou desenvolvido pela maioria dos fabricantes de LANs e de computadores. No entanto, alguns dos padrões desenvolvidos para a camada física do FDDI estão a ser utilizados no *Fast Ethernet*.

Em muitos aspectos, o FDDI é semelhante ao 802.5 e ao IBM *Token Ring*. No entanto, existem algumas diferenças significativas, umas resultantes do facto de utilizar fibra e não cobre, e alguns resultantes de inovações que foram feitas posteriormente à invenção do IBM *Token Ring*.

Ao contrário do 802.5 o FDDI é implementado como um anel duplo, dois anéis independentes que transmitem dados em direcções opostas, como ilustrado na Figura 2.4.a) Na maioria dos casos, a transmissão de dados limita-se ao anel principal. O anel secundário

é fornecido no caso de falha por parte do primeiro anel como ilustrado na Figura 2.4 b). Quando ocorre algum problema com o primeiro anel, o segundo anel pode ser activado de modo a manter e concluir o serviço, ou seja, usa o segundo anel para dar a volta de modo a formar um anel completo e, como consequência, uma rede FDDI é capaz de tolerar uma única quebra no cabo ou a falha de uma estação [3, 6, 7, 9].

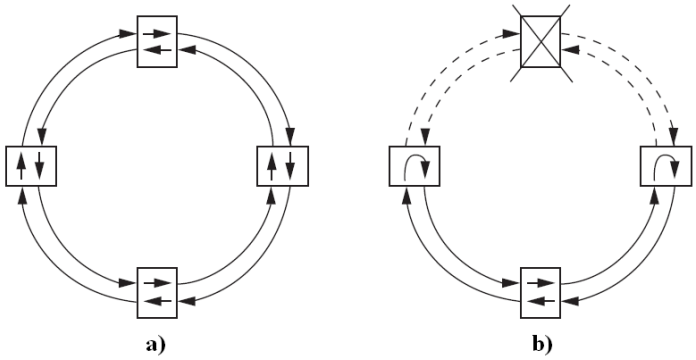

Figura 2.4 – Anel duplo de fibra

2.2.2 *Wide Area Networks* (WANs)

As WANs não têm limitação ao nível da distância, estas permitem transmissão de dados, voz e vídeo em grandes áreas geográficas que podem abranger um país, um continente ou mesmo o mundo. Ao contrário das LANs (que dependem do seu próprio hardware para a transmissão), as WANs podem utilizar dispositivos de comunicação públicos, privados ou mesmo alugados, normalmente em conjunto. As principais tecnologias WAN são *o Point-to-Point Protocol* (PPP), o X.25, o *Frame Relay* e o *Asynchronous Transfer Mode* (ATM).

Normalmente, as WANs operaram em velocidades mais lentas do que as LANs, e têm atrasos mais elevados entre conexões, devido às elevadas distâncias. As velocidades típicas de uma WAN andam no intervalo de 1,5 Mbps a 155 Mbps e os atrasos podem variar entre apenas alguns milisegundos e vários décimos de segundo.

Em tecnologias WAN, uma rede é constituída geralmente por uma série de computadores complexos denominados de *packet switches* interligados por longas linhas de comunicação. A dimensão das redes pode ser aumentada simplesmente pela adição de um *switch* ou outra linha de comunicação. A conexão de um computador a uma WAN, significa efectuar uma ligação do computador a um dos *packet switches*. Cada *switch* ao longo de um percurso na

WAN introduz um atraso na recepção e encaminhamento do pacote para o próximo *switch*, assim, quanto maior for a WAN maior é o tempo necessário para a percorrer [2, 6-8].

2.2.2.1 *Point-to-Point Protocol* (PPP)

O *Point-to-Point Protocol* (PPP) surgiu inicialmente como um protocolo de encapsulamento de modo a transportar tráfego IP sob ligações ponto-a-ponto. O PPP também estabeleceu uma norma para a atribuição e gestão de endereços IP, o encapsulamento assíncrono (*start*/*stop*) e síncrono orientado ao bit, a multiplexagem do protocolo de rede, a configuração da ligação, o teste de qualidade da ligação e a detecção de erros. O PPP suporta estas funções, proporcionando uma extensão do *Link Control Protocol* (LCP) e uma família de *Network Control Protocols* (NCP) para negociar parâmetros de configuração e funções.

O PPP contém três componentes principais, um método para encapsular datagramas ao longo das ligações, para tal o PPP utiliza o protocolo *High-Level Data Link Control* (HDLC) como base para encapsular os datagramas em ligações ponto-a-ponto. Usa uma extensão do LCP para estabelecer, configurar e testar a ligação, e por fim, possui uma família de NCPs para estabelecer e configurar diferentes protocolos da camada Rede, o PPP é concebido de forma a permitir a utilização simultânea de múltiplos protocolos da camada Rede. [2, 3, 6, 10, 11].

A *frame* do PPP possui o seguinte formato.

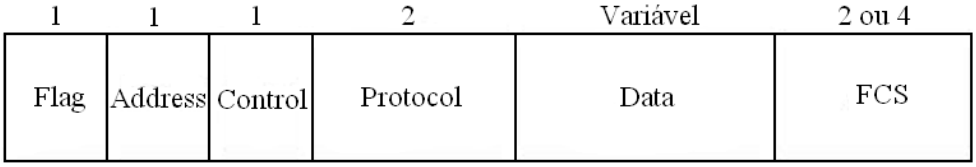

Figura 2.5 – Formato da *frame* PPP

Os campos da *frame* do PPP são descritos em seguida.

Flag: Um byte a indicar o início ou fim de uma *frame.* O campo *Flag* tem o valor de 7E (0111 1110).

Address: Este campo tem o valor de FF (1111 1111), é o endereço de *broadcast*.

Control: Este campo tem o valor de 03 (0000 0011).

Protocol: Este é um campo de dois bytes que identificam o protocolo encapsulado no campo da informação da frame e que cujo valor é 0021 (0000 0000 0010 0001) para o TCP/IP.

Data: Zero ou mais bytes que contém o datagrama para o protocolo especificado no campo *Protocol*. O fim do campo encontra-se ao localizar a sequência da *Flag* e permitindo 2 bytes para o campo *Frame check sequence* (FCS). O tamanho máximo por defeito é de 1500 bytes. Com acordo prévio, o PPP pode usar outros valores para o tamanho máximo do campo de informação.

FCS: A *Frame check sequence* (FCS) é composta normalmente por 16 bits (2 bytes). Com acordo prévio, o PPP pode usar 32 bits (4-byte) para melhorar a detecção de erros. A sequência é colocada no fim do campo de informação de modo a que os dados obtidos se tornem exactamente divisíveis por um número binário predeterminado. No seu destino, os dados recebidos são divididos pelo mesmo número. Se não houver um remanescente, o campo de informação é aceite. Se existe um resto, é possível que os dados tenham sido danificados, por conseguinte, devem ser rejeitados.

2.2.2.2 X.25

O X.25 é um protocolo da *International Telecommunication Union – Telecommunication Standardization Sector* (ITU-T) para comunicações WAN que define o modo como as ligações entre os dispositivos dos utilizadores e os dispositivos de rede são criadas e mantidas. O protocolo X.25 foi projectado para operar eficazmente independentemente do tipo de sistemas ligados à rede. É tipicamente utilizado nas *Packet Switched Networks* (PSNs) dos operadores públicos, tais como as companhias telefónicas, onde os assinantes são cobrados com base na sua utilização da rede. O desenvolvimento da norma X.25 foi iniciado pelos operadores públicos na década de 70. Nessa altura, havia a necessidade de protocolos WAN capazes de fornecer conectividade entre *Public Data Networks* (PDNs).

Os dispositivos de rede do X.25 podem ser divididos em três categorias: *Data Terminal Equipment* (DTE), *Data Circuit-terminating Equipment* (DCE) e *Packet-Switching Exchange* (PSE). Os *Data Terminal Equipment* (DTE) são sistemas finais que comunicam através da rede X.25. Estes são normalmente terminais, computadores pessoais, ou *network hosts*. Os DCE são dispositivos de comunicação, tais como modems e *packet switches*, que

fornecem a interface entre dispositivos DTE e PSE, e estão geralmente localizados nas instalações da operadora. Os PSEs são *switches* que compõem a maior parte da rede da operadora, eles transferem os dados de um DTE para outro através do X.25 PSN. A Figura 2.6 ilustra o relacionamento entre os três tipos de dispositivos de uma rede X.25.

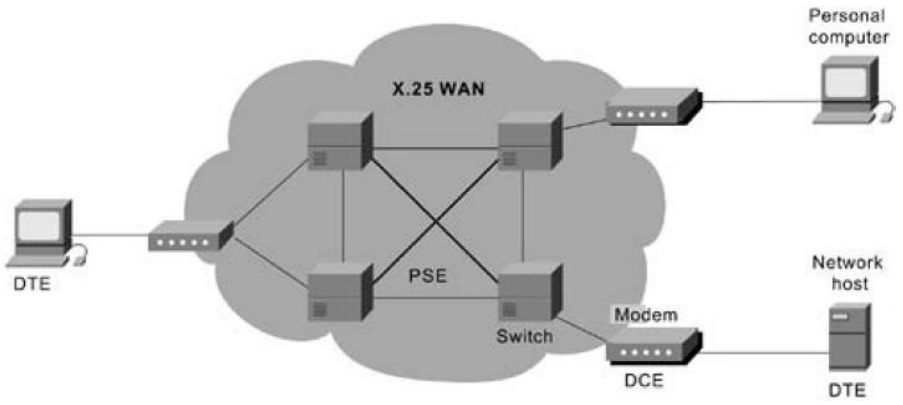

Figura 2.6 – Estrutura e dispositivos de uma rede X.25

A norma X.25 descreve os procedimentos necessários para o estabelecimento de conexão, troca de dados, *acknowledgement*, controlo de fluxo e controlo de dados. A norma X.25 é agora administrada pela ITU-T. [2, 3, 6].

2.2.2.3 *Frame Relay*

O *Frame Relay* é um protocolo WAN que opera na camada Física e na camada Ligação de Dados do modelo de referência OSI. O *Frame Relay* foi criado para ser usado através de redes digitais com integração e serviços (*Integrated Services Digital Network* (ISDN)). O *Frame Relay* é um protocolo WAN concebido em resposta às deficiências do X.25, o X.25 apenas verifica a existência de erros a partir da fonte e do destino, fazendo com que a maior parte do tráfego seja dedicado à verificação de erros para garantir a fiabilidade do serviço. O *Frame Relay* como opera na camada Física e na camada Ligação de Dados do modelo de referência OSI, permite que cada estação tenha uma cópia da *frame* original até que esta receba confirmação da próxima estação que a *frame* chegou intacta. Assim, todo o controlo de erro é efectuado pelas camadas Rede e Transporte, que usam o serviço de *Frame Relay* [3, 6].

2.2.2.4 *Asynchronous Transfer Mode* (ATM)

O *Asynchronous Transfer Mode* (ATM) é uma ideia revolucionária para reestruturar a infra-estrutura da comunicação de dados. Foi concebido para suportar a transmissão de dados, voz e vídeo através de uma elevada taxa de transmissão dos dados através do meio, como cabos de fibra óptica, sobre redes públicas e privadas usando a tecnologia *cell relay*. *Cell relay* refere-se a um método estatístico de multiplexagem do comprimento fixo dos pacotes, ou seja, as células, para o transporte de dados entre computadores ou tipos de equipamentos de rede.

O ATM é uma norma do *International Telecommunication Union Telecommunication Standardization Sector* (ITU-T). Actuais desenvolvimentos sobre as normas ATM estão a ser feitos principalmente pelo ATM *Forum*, que foi fundado em conjunto pela Cisco Systems, a NET/ADAPTATIVE, a Northern Telecom e Sprint, em 1991.

A unidade básica de informação usada pelo ATM é uma célula de tamanho fixo, constituído por 53 octetos, ou bytes. Os primeiros 5 bytes correspondem ao cabeçalho que contém informações, tais como o identificador da ligação, enquanto os restantes 48 bytes contêm os dados, ou *payload* (ver Figura 2.7). Como o *switch* do ATM não tem que detectar o tamanho da unidade de dados, a comutação pode ser realizada de uma forma eficiente. O pequeno tamanho da célula torna-a bem adaptada para a transferência de dados em tempo real, tais como voz e vídeo.

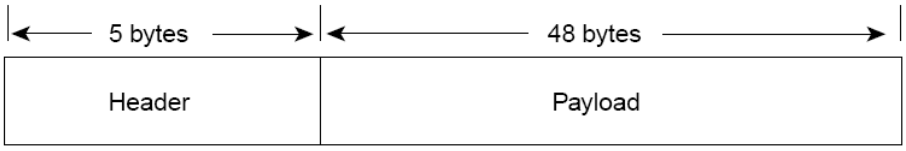

Figura 2.7 – Formato de uma célula ATM

Existem dois tipos de interfaces que interligam dispositivos ATM em ligações ponto-a-ponto: o *User-Network Interface* (UNI) e a *Network-Network Interface* (NNI), às vezes chamado de *Network-Node Interface*. A ligação UNI conecta uma ligação ATM *end-system* (do lado do utilizador) com um *switch* ATM (do lado da rede). Uma ligação NNI conecta dois *switches* ATM, neste caso, ambas as partes são rede.

O cabeçalho contém um *Virtual Path Identifier* (VPI) e um *Virtual Channel Identifier* (VCI). Estes dois identificadores são usados para encaminhar a célula através da rede para o destino final. No entanto o ATM, tal como o X.25 e o *Frame Relay*, é uma rede orientada à conexão, o que significa que, antes de os dois sistemas começarem a comunicar entre si, eles devem estabelecer uma ligação lógica. Para iniciar uma ligação, o sistema utiliza um endereço de 20 bytes, após a conexão ser estabelecida, a combinação do VPI/VCI encaminha a célula a partir da sua origem até ao destino final [6, 12].

2.3 Modelo de Referência OSI

A gestão de uma rede de comunicação de dados é uma questão complexa, requer, em particular, a realização de um grande número de funções bem diferenciadas e com diversos graus de dificuldade. Acresce a isso, o facto de estas funções terem de se realizar num ambiente potencialmente adverso (erros, falhas), envolvendo sistemas heterogéneos.

Para a rede de comunicações funcionar correctamente, centenas de perguntas devem ser respondidas por um conjunto de protocolos. Avaliar e trabalhar com estas centenas de perguntas tornar-se-ia impossível de gerir. Uma vez que a troca de informação entre as diferentes entidades de um sistema de comunicação é estruturalmente complexa e difícil de entender, a interacção entre os sistemas constituintes da rede requer, deste modo, mecanismos de comunicação, controlo e sincronismo, então, o processo de comunicação na sua globalidade foi universalmente organizado e baseado em camadas hierárquicas bem definidas.

Assim, a *International Standards Organization* (ISO), organismo de carácter geral que define padrões em praticamente todas as áreas da ciência e tecnologia, é responsável pela elaboração de um conhecido modelo de referência englobando todo o tipo de comunicações de dados. O modelo é designado por *Open Systems Interconnection* (OSI) e propõe uma arquitectura de 7 camadas para os sistemas de comunicação (Figura 2.8), dividindo assim modularmente as tarefas envolvidas no processo de comunicação de dados de modo a diminuir a sua complexidade e aumentar a sua compatibilidade.

O modelo de referência OSI especifica as sete camadas de funcionalidade, define as sete camadas desde a camada Física à camada Aplicação, no entanto, o modelo OSI não define os protocolos que implementam as funções em cada uma das camadas. O modelo OSI é

ainda importante para a compatibilidade, independência protocolar e para o crescimento da tecnologia de rede.

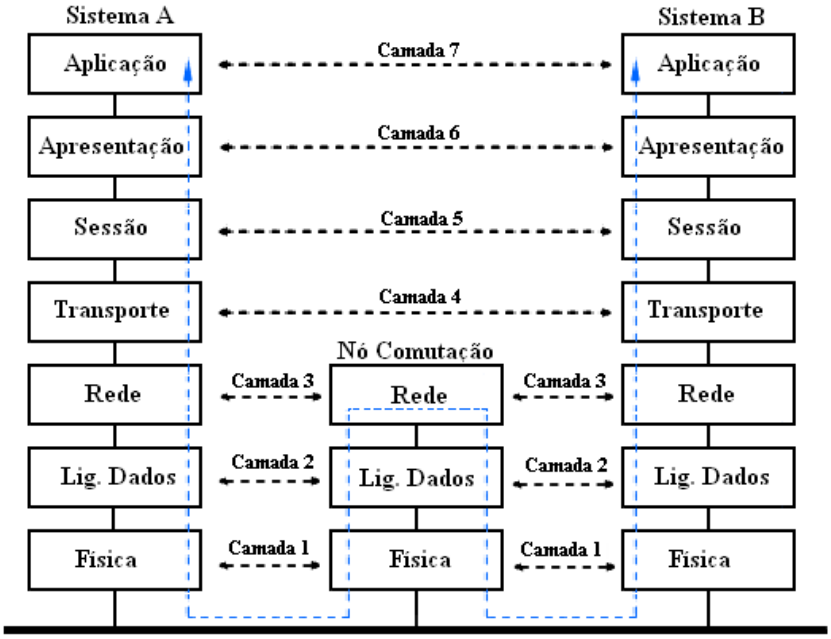

Figura 2.8 – Modelo de Referência ISO/OSI

As camadas Aplicação, Apresentação e Sessão tem mais em consideração os programas de aplicação executáveis nas máquinas, enquanto as camadas Rede, Ligação de Dados e Física lidam com a infra-estrutura de rede e de encaminhamento físico do processo de comunicação.

Seguidamente, apresenta-se uma breve descrição das características e funções desempenhadas pelas sete camadas do modelo OSI:

- **Camada Física** – Na parte inferior do modelo OSI temos a camada Física. Os tópicos desta camada determinam a forma de envio e recepção dos bits de dados que se deslocam ao longo da rede. Esta camada é responsável pela geração dos sinais eléctricos, ópticos ou electromagnéticos. Especifica a forma como os bits são codificados em sinais compatíveis com o meio físico de transmissão, bem como a especificação dos parâmetros e tolerâncias (tensões, correntes, frequências, …) dos sinais usados para transmitir a informação.
- **Camada Ligação de Dados** – Esta camada tem como preocupação oferecer sobre o nível físico uma transmissão de informação estruturada e fiável. Fornece

ainda o método de acesso à rede. É onde os dados são preparados para o envio para a rede, o pacote é encapsulado numa *frame*. Os protocolos desta camada ajudam no endereçamento e detecção de erros dos dados que estão a ser transferidos. Devido à sua complexidade, esta camada é usualmente, dividida em duas sub-camadas: *Medium Access Control* (MAC) e *Logical Link Control* (LLC). A sub-camada MAC define o controlo de acesso ao meio de transmissão, enquanto que a sub-camada LLC é responsável pelo controlo da ligação lógica, como a *Ethernet* ou o *Token Ring*.

- **Camada Rede** – A camada Rede tem inerentes as funções de encaminhamento e endereçamento da informação ao longo da rede. É também, nesta camada que é determinada a rota mais adequada, baseada no endereço dos pacotes e condições da rede.

- **Camada Transporte** – A camada Transporte pode garantir que os pacotes são recebidos, pode também estabelecer uma conexão e enviar *acknowledgments* em como os pacotes são recebidos. Os protocolos nesta camada proporcionam os meios para criar, manter e libertar conexões para os anfitriões envolvidos na comunicação. A camada Transporte efectua a comutação de informação num formato que recebe da camada Sessão, sequencia as mensagens e faz o seu endereçamento até ao destino e tem como objectivo principal, a preocupação em criar ambientes de comunicações para permitir às camadas superiores a troca de informação fiável e processos de comunicação independentes da rede.

- **Camada Sessão** – Esta camada assegura a resolução dos problemas intrínsecos à gestão do diálogo entre processos de comunicação (processos dependentes dos sistemas, não tanto da rede). Trata as funções necessárias à interacção relativa à troca de dados no sistema. Os protocolos da camada Sessão estabelecem sessões, ou conexões. Estes protocolos abrangem tópicos tais como a forma de estabelecer uma conexão, o modo de utilização de uma ligação, bem como a forma de terminar a ligação quando a sessão é concluída. Depois de que uma conexão é estabelecida, os protocolos da camada Sessão vão verificar a existência de erros na transmissão.

- **Camada Apresentação** – Esta camada tem em conta o problema de existirem modos diferentes de representar a informação. A camada Apresentação acrescenta estrutura aos pacotes de dados que estão a ser trocados. Efectua a conversão de dados para formatos padrão (ASCII), encriptação de dados, compressão de dados e tratamento dos terminais. O principal trabalho da camada Apresentação é o de assegurar que transmite a mensagem numa linguagem ou sintaxe que o computador receptor possa compreender. Os protocolos da camada Apresentação podem traduzir os dados para uma sintaxe compreensível e então comprimir e talvez encriptar os dados antes de passá-los para a camada Sessão.

- **Camada Aplicação** – Os programas de aplicação são exteriores ao modelo OSI. A camada Aplicação é o nível mais alto do modelo OSI, é totalmente dedicada e dependente das aplicações a fornecer ao utilizador específico. O objectivo da camada Aplicação é o de gerir as comunicações entre aplicações. Esta é a camada onde as aplicações recebem e pedem dados. Todas as outras camadas trabalham para esta camada.

O modelo OSI foi concebido para promover a interoperabilidade através da criação uma linha de orientação para a rede de transmissão de dados entre computadores que têm diferentes hardware, software, sistemas operativos e protocolos, e para contemplar e privilegiar o modo de comunicação orientada à conexão (duas entidades antes de efectuarem a comunicação associam-se de forma lógica). Por exemplo, no caso do processo de transferência de um simples arquivo. Do ponto de vista do utilizador, uma única operação foi realizada para a transferência do arquivo, mas na realidade, muitos processos diferentes ocorreram nos bastidores para realizar esta tarefa aparentemente simples.

O modelo hierárquico e modular facilita a comunicação entre aqueles que desenvolvem, fornecem e usam o sistema de comunicações, de tal forma, que qualquer alteração numa dada camada não afecta as outras camadas. Desta forma, pode-se desenvolver um subsistema de uma determinada camada sem que para isso haja necessidade de conhecer o sistema na sua globalidade. A consequência directa desta facilidade é a possibilidade de

uma alteração individual das camadas sem a necessidade de redesenhar todo o sistema de comunicações [4, 6-9, 13, 14].

2.4 Modelo TCP/IP

Um protocolo é um conjunto de regras que regem a forma como os dados serão transmitidos e recebidos em redes de comunicação. Os protocolos são, então, as regras que determinam a forma de uma rede actuar.

O modelo OSI e o modelo TCP/IP possuem muitas semelhanças, mas existem diferenças entre os dois modelos. No entanto, ambos lidam com comunicações entre computadores heterogéneos.

A diferença entre os dois modelos deve-se ao facto do protocolo TCP/IP ter sido desenvolvido antes do modelo OSI ter sido publicado. Como resultado, este não utiliza o modelo OSI como referência, o TCP/IP foi desenvolvido utilizando o modelo do *Department of Defense* (DoD) como referência. Assim sendo o modelo TCP/IP possui quatro camadas: camada Aplicação, camada Transporte, camada Internet e camada Interface de Rede. (Tabela 2.1)

Tabela 2.1 – Modelo TCP/IP

Modelo OSI	***Modelo TCP/IP***	***Protocolo Internet***
Camada Aplicação *Camada Apresentação* *Camada Sessão*	*Camada Aplicação*	*HTTP, FTP, TFTP, NFS, RPC, XDR, SMTP, POP, IMAP, MIME, SNMP, DNS, RIP, OSPF, BGP, TELNET, Rlogin*
Camada Transporte	*Camada Transporte*	*TCP, UDP*
Camada Rede	*Camada Internet*	*IP, ICMP, IGMP, ARP, RARP*
Camada Ligação de dados *Camada Física*	*Camada Interface Rede*	*Ethernet, Token Ring, FDDI, PPP, X.25, Frame Relay, ATM*

Seguidamente, apresenta-se uma breve descrição das características e funções desempenhadas pelas quatro camadas do modelo TCP/IP [4, 6, 15].

2.4.1 Camada Interface de Rede

A camada mais baixa do TCP/IP é a Camada Interface de Rede. A principal responsabilidade desta camada é a de definir a forma como um computador se conecta a uma rede. Esta é uma parte importante do processo de entrega de dados, pois os dados devem ser entregues a um determinado utente através de uma ligação a uma rede, e os dados que saem de um utente têm que seguir as regras da rede em que se encontram.

Uma função da camada é o encaminhamento de dados entre utentes ligados à mesma rede. Os serviços a serem fornecidos são o controlo de fluxo e controlo de erros entre utentes. A camada Interface de Rede pode ser invocada quer através da camada Internet quer através da camada Aplicação. Esta camada fornece os controladores que suportam as interacções entre diferentes tipos de rede, como *Ethernet*, *Token Ring*, ou *Fiber Distributed Data Interface* (FDDI). A camada Interface de Rede não regula o tipo de rede a que o utente está ligado, mas a rede a que o utente está ligado indica qual o controlador a ser usado pela camada Interface de Rede [4, 6].

2.4.2 Camada Internet

A camada Internet oferece uma função de endereçamento e encaminhamento, ou seja a camada Internet contém protocolos que são responsáveis pelo endereçamento e encaminhamento dos pacotes. Esta camada contém vários protocolos, incluindo o *Internet Protocol* (IP), *Address Resolution Protocol* (ARP), *Internet Control Message Protocol* (ICMP) e *Internet Group Message Protocol* (IGMP) [4, 6].

2.4.3 Camada Transporte

A camada Transporte entrega os dados entre dois processos de computadores diferentes. A camada Transporte determina se o emissor e o receptor irão criar uma conexão antes de começar a comunicação e com que frequência *acknowledgments* da ligação serão enviados para cada um. A camada Transporte possui apenas dois protocolos: o *Transmission Control Protocol* (TCP) e o *User Datagram Protocol* (UDP). Onde o TCP cria uma conexão e envia *acknowledgments* da mesma, enquanto o UDP não. O UDP pode transmitir os dados mais rápidamente, mas o TCP possui garantias de entrega [4, 6].

2.4.4 Camada Aplicação

A camada Aplicação representa o mais elevado nível de protocolos que são usados para fornecer uma interface directa com utentes ou aplicações. Esta camada contém protocolos para a partilha de recursos e para o acesso remoto. Alguns dos protocolos são o *File Transfer Protocol* (FTP) para a transferência de arquivos, o *HyperText Transfer Protocol* (HTTP) para a *World Wide Web*, e o *Simple Network Management Protocol* (SNMP) para controlar dispositivos de rede.

Possui ainda o *Domain Name Service* (DNS), que é responsável por converter endereços IP em nomes que podem ser mais facilmente lembrados pelos utilizadores. Muitos outros protocolos que lidam com os pequenos detalhes das aplicações estão também incluídos nesta camada. Estes incluem o *Simple Mail Transport Protocol* (SMTP), *Post Office Protocol* (POP), *Internet Mail Access Protocol* (IMAP), para o correio electrónico, o *Privacy Enhanced Mail* (PEM), *Pretty Good Privacy* (PGP) e *Secure Multipurpose Internet Mail Extension* (S/MIME) para a segurança do correio electrónico. Todos os protocolos contidos no TCP/IP *suite* serão aprofundados no Capítulo 3 [4, 6].

Capítulo 3

Modelo TCP/IP

Os protocolos de Internet são constituídos por um conjunto de protocolos de comunicação, dos quais os dois mais conhecidos são o *Transmission Control Protocol* (TCP) e o *Internet Protocol* (IP). O TCP/IP *suite* inclui não só protocolos de camadas inferiores (TCP, UDP, IP, ARP, RARP, ICMP e IGMP), mas também especifica aplicações comuns, como a *World Wide Web,* o correio electrónico, o *Domain Name Service* (DNS) e transferência de arquivos. A Tabela 2.1 no Capítulo 2 apresenta muitos dos protocolos do TCP/IP *suite* e camadas correspondentes ao modelo OSI.

Este capítulo dedica-se a uma descrição dos protocolos do TCP/IP que se consideram importantes e determinantes não só na abordagem e desenvolvimento de um sistema de comunicação, mas também necessários, para o devido enquadramento do trabalho realizado no âmbito desta dissertação.

3.1 Protocolos da Camada Rede

Na camada Rede do modelo OSI, o TCP/IP suporta o *Internet Protocol* (IP). O IP contém quatro protocolos: *Address Resolution Protocol* (ARP), *Reverse Address Resolution Protocol* (RARP), *Internet Control Message Protocol* (ICMP) e *Internet Group Message Protocol* (IGMP). Cada um destes protocolos é descrito em seguida.

3.1.1 *Internet Protocol* (IP)

O *Internet Protocol* (IP) é um protocolo da camada Rede (camada 3 do modelo OSI ou a camada Internet do modelo TCP/IP) que é responsável por determinar o endereço IP da

origem e do destino de todos os pacotes. O IP está bem documentado na RFC 791. O protocolo IP define a unidade básica para a transferência de dados e especifica o formato exacto de todos os dados que passam através da Internet. O IP é o protocolo que fornece um serviço não fiável e *connectionless*.

Por não fiável quer dizer que não há garantias de que um datagrama IP chega com sucesso ao seu destino. Para que haja fiabilidade, esta deve ser fornecida por camadas superiores (por exemplo, utilizando o TCP).

O termo *connectionless* significa que uma conexão lógica não é estabelecida entre os dispositivos. Isto significa que cada datagrama é tratado independentemente dos outros, cada datagrama é separado em pacotes e é enviado independentemente de seguir um caminho diferente para o seu destino. Isto implica que, se uma fonte envia vários datagramas para o mesmo destino, eles podem chegar fora de ordem.

A Figura 3.1 ilustra o formato de um datagrama IP. Dado que o processamento é feito por software, o conteúdo de um datagrama IP não é restringido por qualquer hardware [2-4, 6, 7, 15-17].

3.1.1.1. Datagramas IP

Pacotes na camada IP são chamados de datagramas. Cada datagrama IP é constituído por um cabeçalho (*Header*) (20 a 60 bytes) e dados (*Data*). O cabeçalho do datagrama IP consiste numa secção fixa de 20 bytes e uma secção de opções com tamanho variável com um máximo de 40 bytes. O comprimento máximo permitido de um datagrama IP é de 65536 bytes. No entanto, pacotes deste tamanho não seriam práticos, especialmente na Internet onde teriam que ser fragmentados muitas vezes. O RFC 791, tal como a maioria dos documentos de outros protocolos do IP-*suite*, especifica o formato do datagrama, em termos de *words*.

A Figura 3.1 ilustra o datagrama IPv4 (*Internet Protocol version* 4) segundo o RFC 791.Um Datagrama IPv4 é constituído principalmente por três componentes. O *Header* tem um tamanho de 20 bytes e contém uma série de campos. O campo *Options* é um conjunto de campos de tamanho variável, que pode ou não estar presente. No campo *Data* encontram-se os dados encapsulados, o *payload* de um nível superior, geralmente um segmento TCP ou datagrama UDP.

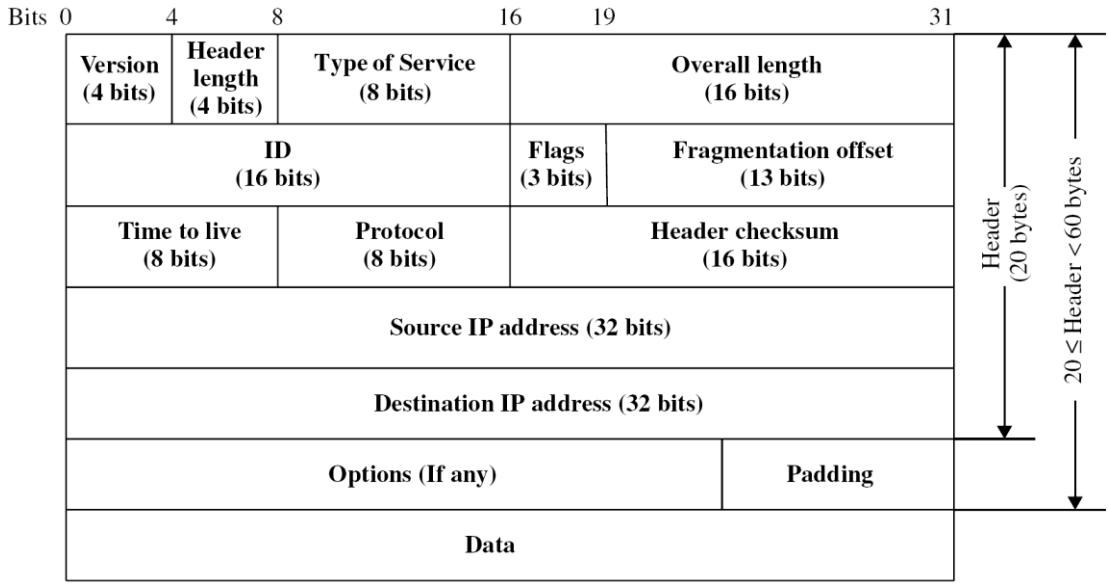

Figura 3.1 – Formato do datagrama IP

Uma breve descrição de cada campo de um datagrama IP é feita a seguir.

- ***Version*** (VER, 4 bits) – O primeiro campo de quatro bits do datagrama contém a versão do protocolo IP que foi usado para criar o datagrama. É utilizado para verificar se o emissor, receptor e qualquer *router* entre eles estejam de acordo no formato do datagrama. A versão 4 do protocolo de Internet (IPv4) tem sido utilizada desde 1981, mas a versão 6 (IPv6 ou IPng) irá substituí-lo em breve.

- ***Header length*** (HLEN, 4 bits) – Este campo define o tamanho total do cabeçalho do datagrama IPv4 medido em *words* de 32 bits. Este campo é necessário porque o tamanho do cabeçalho varia entre 20 e 60 bytes. Todos os campos no cabeçalho têm tamanhos fixos excepto o campo *Options* e o correspondente campo de *Padding*.

- ***Type of Service*** (TOS, 8 bits) – Este campo de oito bits específica como o datagrama deve ser processado pelos *routers*. Segundo o RFC 1349 [17], o campo TOS é constituído pelo campo *Precedence* (3 bits), o campo TOS (4 bits) e ainda um campo MBZ (1 bit) que não é utilizado, mas como o nome indica *Must be Zero* (MBZ) tem que ser zero como indica a Figura 3.2. O campo *Precedence* varia de 0 (000 em binário, precedência normal) a 7 (111 em binário, rede de controlo), permitindo aos emissores indicarem a importância de cada

datagrama. O campo *Precedence* define a prioridade do datagrama em questões de congestionamento da rede.

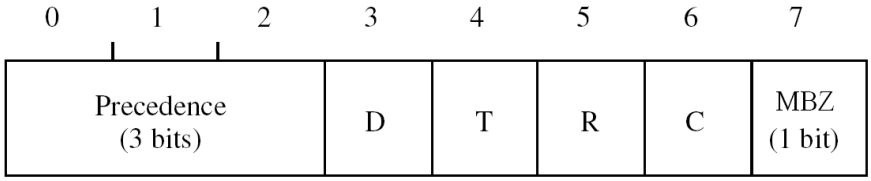

Figura 3.2 – Formato do campo *Type of Service* (TOS)

Actualmente, o campo *Precedence* não é utilizado na versão 4, mas prevê-se que venha a ser funcional nas versões futuras. O campo TOS é constituído por quatro bits, *Delay* (D), *Throughput* (T), *Reliability* (R) e *Cost* (C) que especificam o tipo de transporte desejado para o datagrama. Na Tabela 3.1 estão os valores e descrição do serviço definidos pelo RFC 1349.

Tabela 3.1 – Codificação do campo TOS

TOS Bit	***Descrição***
0000	*Normal (por defeito)*
0001	*Minimiza o custo*
0010	*Maximiza a fiabilidade*
0100	*Maximiza a taxa de transferência*
1000	*Minimiza o atraso*

- ***Overall length*** (16 bits) – Este campo corresponde ao tamanho total do datagrama IP, como o IPv4 atribui 16 bits para o campo, o tamanho total do datagrama IP é limitado a 2^{16} - 1 = 65 535 bytes. Para saber o tamanho dos dados provenientes da camada superior, é só subtrair o tamanho do cabeçalho ao tamanho total. Na prática, algumas redes não são capazes de encapsular um datagrama de 65 535 bytes através do processo de fragmentação.
- ***Identification*** (ID, 16 bits) – Este campo é usado para identificar um datagrama originário da fonte, de modo a ajudar o destino a fazer o *reassemble* de um pacote fragmentado. O campo ID é escolhido pelo emissor e identifica exclusivamente um datagrama IP enviado por uma fonte. A combinação entre a

Identification e a *source* IP *address* deve definir unicamente o mesmo datagrama, que está a deixar a fonte. Para garantir a singularidade, o protocolo IP utiliza um contador para identificar os datagramas. Quando um datagrama é fragmentado, o valor no campo *Identification* é copiado em todos os fragmentos. Assim, todos os fragmentos têm o mesmo número de identificação, correspondente ao mesmo datagrama original. O RFC 791 sugere que o número de identificação seja seleccionado pelo protocolo da camada superior, mas, na prática, ele tende a ser seleccionado pelo protocolo IP.

- ***Flags*** (3bits) – Este campo é usado na fragmentação. O Bit 0 está Reservado, o Bit 1 indica se o datagrama pode ser fragmentado ou não, se o seu valor for "1", significa que não pode fragmentar o datagrama. Se não for possível enviar o datagrama através da rede, então este é descartado e é enviada uma mensagem de erro ICMP para a fonte, o Bit 2 indica se é o último fragmento ou se existem mais fragmentos, se o seu valor for "1", significa que o datagrama não é o último fragmento, há mais fragmentos transmitidos, se o seu valor for "0", significa que este é o último ou único fragmento.

- ***Fragmentation offset*** (13 bits) – Este campo é utilizado durante o *reassemble* de um datagrama que foi fragmentado. Este campo indica a posição relativa de cada fragmento em relação ao pacote original. O primeiro fragmento tem o valor de *offset* "0", os fragmentos seguintes indicam a sua posição em múltiplos de 8 octetos (64 bits) a partir do início do pacote original. Dado que o tamanho do campo é de 13 bits, ele não pode representar um sequência de bytes maior que $2^{13} - 1 = 8191$. Para o *reassemble* do datagrama, o destino deve obter todos os fragmentos, a começar com o fragmento com *offset* "0" até ao fragmento com o maior *offset*.

- ***Time to live*** (TTL, 8 bits) – Este campo contém um valor inteiro binário correspondente à duração do tempo em segundos a que o pacote é autorizado a permanecer na Internet. Quando processam um datagrama, os *routers* devem decrementar esse tempo em pelo menos 1 segundo, sempre que esse valor chegar a zero depois de ser decrementado, o *router* descarta o datagrama e envia uma mensagem de erro para a fonte. Na prática, apesar do campo TTL ser o tempo de

vida em segundos, na realidade reflecte o número máximo de *hops* (pontos de passagem) antes de um pacote ser considerado como perdido e *undeliverable*. Um valor padrão comum da configuração inicial para o campo TTL é 64 (os valores possíveis vão desde 0 a 255 segundos).

- ***Protocol*** (8 bits) – Este campo de oito bits define o protocolo do nível superior que usa os serviços da camada IP. Um datagrama IP pode encapsular dados de vários protocolos da camada superior, tais como TCP, UDP, ICMP e IGMP. Este campo especifica o protocolo usado na próxima camada superior da comunicação, a próxima camada superior será tipicamente o protocolo TCP (*Transmission Control Protocol*) (*Protocol* = 6) ou UDP (*User Datagram Protocol*) (*Protocol* = 10).

- ***Header checksum*** (16 bits) – O método de detecção de erros utilizado pela maioria dos protocolos TCP/IP é o *checksum*. Este campo de 16 bits assegura a integridade dos valores do cabeçalho. O *checksum* (bits redundantes adicionados aos pacotes) protege contra erros que podem ocorrer durante a transmissão de um pacote. Na fonte, o *checksum* é calculado e o resultado obtido é enviado com o pacote. No receptor, o mesmo cálculo é repetido, se o resultado final for zero, então não existem erros nos dados, e o pacote é aceite, caso contrário o pacote é rejeitado. É importante referir que o *checksum* apenas se aplica aos valores do cabeçalho IP, e não aos dados. Os protocolos TCP, UDP, ICMP e IGMP possuem um *checksum* próprio que cobre o cabeçalho e os dados.

- ***Source IP address*** (32 bits) – Este campo de 32 bits especifica o endereço IP da origem do datagrama IP.

- ***Destination IP address*** (32 bits) – Este campo de 32 bits indica o endereço IP do destino para onde o datagrama tem de ser enviado.

- ***Options*** (tamanho variável) – Este campo tem um tamanho variável, que consiste de zero, uma ou mais opções individuais. Este campo especifica um conjunto de campos, que podem ou não estar presentes em qualquer datagrama, que descrevem um processamento específico que acontece a um pacote. O RFC 791 define um certo número opções, com opções adicionais definidas no RFC 3232. Nos casos em que o campo *Options* não é preenchido totalmente com a

opção, bytes adicionais de *padding* são adicionados ao final de qualquer opção específica para preencherem o espaço restante. As opções mais comuns incluem:

- *Security* esta opção é utilizada mais para aplicações militares e tende a não ser utilizada na maioria das redes comerciais. O RFC 1108 possui mais detalhes.
- *Record Route* é utilizada para registar os *routers* que lidam com o datagrama. Cada *router* grava o seu endereço IP no campo *Options*, o que permite detectar erros de encaminhamento.
- *Timestamp* esta opção é utilizada para registar o tempo de processamento de um datagrama no *router*. Esta opção pede a cada um dos *routers* para gravar tanto o endereço do *router* como o respectivo tempo de processamento. Esta opção é útil para fazer o *debug* dos *routers*.
- *Source routing*, o objectivo desta opção é permitir à fonte predeterminar uma rota para o datagrama. Esta opção permite que um utente defina quais os *routers* que o pacote vai usar durante a transmissão.

3.1.1.2. Endereçamento IP

Cada utente numa rede TCP/IP precisa de ter um endereço exclusivo, com este endereço é possível enviar dados de e para outros utentes. Cada pacote contém informações de endereçamento no cabeçalho, e o endereço IP no cabeçalho é usado para encaminhar os pacotes. O endereçamento IP consiste simplesmente, em configurar cada utente de TCP/IP com um endereço IP válido. Para aceder à Internet, um utente deve ter um endereço IP que identifica não apenas endereço do utente, mas também identifica o endereço da rede.

Endereços TCP/IP são baseados em endereços de 32 bits, mas em vez de trabalhar com 32 1s e 0s, as pessoas usam decimais para representar os endereços IP, especificamente, estes usam quatro números decimais separados por pontos. Estes quatro números decimais representam os 32 dígitos binários separados em quatro partes iguais, chamados octetos, em que cada octeto representa 8 bits. Um endereço IP, expresso em decimais é referido como notação decimal porque tem quatro números decimais, cada um separado por um ponto.

A Tabela 3.2 apresenta os endereços de três camadas diferentes do protocolo TCP/IP.

- **Endereço Físico** (local ou *link*) – Ao nível físico, os equipamentos são reconhecidos através do seu endereço físico. O endereço físico é o endereço de nível mais baixo que é especificado como o endereço local definido pela LAN ou WAN. Endereços físicos podem ser *unicast* (um único utente), *multicast* (um grupo de utentes), ou *broadcast* (todos os utentes da rede). O endereço físico será alterado à medida que o pacote se move de rede para rede.
- **Endereço IP** – Um endereço IP identifica um utente ou um *router* ao nível da rede. Endereços IP, podem ser *unicast*, *multicast* ou *broadcast*. Endereços IP foram concebidos para um sistema de endereçamento universal, em que cada utente pode ser inequivocamente identificado. Um endereço de Internet é actualmente um endereço de 32 bits que pode definir um único utente conectado à Internet.
- **Endereço de Porto** – A entrega de um pacote para um utente ou *router*, requer dois níveis de endereços, um físico e outro lógico. De modo a distinguir vários serviços, o TCP/IP, atribui a cada um desses serviços um endereço de porto, que tem um tamanho de 16 bits. A *Internet Assigned Numbers Authority* (IANA) gere os números dos portos entre 1 e 1023 para os serviços TCP/IP.

Tabela 3.2 – Arquitectura de endereçamento TCP/IP

Camada	***Protocolo TCP/IP***	***Endereço***
Aplicação	*HTTP, FTP, SMTP, DNS, e outros*	*Endereço do porto*
Transporte	*TCP e UDP*	----
Internet	*IP, ICMP, IGMP*	*Endereço IP*
Interface de Rede	*Rede Física*	*Endereço Físico*

Os servidores normalmente são conhecidos pelo número de porto. Um dos portos mais conhecidos é o porto do *File Transfer Protocol* (FTP) onde o seu serviço é fornecido através do porto TCP 21. Já o *Trivial File Transfer Protocol* (TFTP) utiliza o porto UDP 69. O número do porto para o *Domain Name System* (DNS) é o porto TCP 53.

Esquemas de Endereçamento

Cada endereço IP é composto por duas partes, o *netid* que define a rede e *hostid* que identifica o utente na rede. Os endereços IP são divididos em cinco classes diferentes: A, B, C, D, e E. As classes A, B e C diferem quanto ao número de utentes permitidos por rede. A classe D é utilizada para *multicast* e a classe E está reservada para uso futuro. A Tabela 3.3 mostra o número de redes e de utentes nas diferentes classes de endereços IP. De notar que os números binários entre parênteses indicam o prefixo da classe. A relação entre as classes de endereços IP e a notação decimal é resumida na Tabela 3.4, que indica o intervalo de valores para cada classe. Existem endereços IP que têm significados específicos. O endereço 0.0.0.0 é reservado, o 224.0.0.0 não é usado. O 255.255.255.255 é o endereço de *broadcast*, utilizado para chegar a todos os sistemas de uma ligação local. Nos endereços *multicast*, classe D, o endereço 224.0.0.0 nunca é utilizado e o 224.0.0.1 é atribuído ao grupo de todos os utentes IP, incluindo *gateways*.

Tabela 3.3 – Número de redes e utentes para cada clase de endereços IP

Classe de Endereço	*Netid*	*Hostid*	*Número de Redes e Utentes*	
			Netid	*Hostid*
A (0)	*Primeiro octeto (8 bits)*	*Três octetos (24 bits)*	$2^7 - 2 = 126$	$2^{24} - 2 = 16\,777\,214$
B (01)	*Dois octetos (16 bits)*	*Dois octetos (16 bits)*	$2^{14} = 16\,384$	$2^{16} - 2 = 65\,534$
C (110)	*Três octetos (24 bits)*	*Último octeto (8 bits)*	$2^{21} = 2\,097\,152$	$2^8 - 2 = 254$
D (1110)	----	----	*Sem Netid*	*Sem Hostid*
E (1111)	----	----	*Sem Netid*	*Sem Hostid*

Tabela 3.4 – Notação decimal para cada classe de endereços IP

Classe	*Prefixo*	*Intervalo de Endereços*	
A	*0*	*0.0.0.0*	*127.255.255.255*
B	*10*	*128.0.0.0*	*191.255.255.255*
C	*110*	*192.0.0.0*	*223.255.255.255*
D	*1110*	*224.0.0.0*	*239.255.255.255*
E	*1111*	*240.0.0.0*	*255.255.255.255*

Subnetting e *supernetting*

O contínuo aumento do número de utentes ligados à Internet e as restrições impostas pelo esquema de endereçamento da Internet levou à ideia de *subnetting* e *supernetting*. Em *subnetting*, uma grande rede é dividida em várias sub-redes mais pequenas, os endereços das classes A, B e C podem ser divididos. Em *supernetting*, várias redes são combinadas numa única rede, utilizando vários endereços da classe C de modo a criar uma ampla gama de endereços.

Subnetting é bem sucedido através da divisão de uma rede em sub-redes, sendo que agora o esquema do endereço IP é composto por *netid*, *subnetid* e *hostid*. *Subnetting* traz ao administrador de rede vários benefícios, incluindo extra flexibilidade, utilização mais eficiente da rede de endereços, e capacidade de conter o tráfego *broadcast*. As sub-redes estão sob a administração local, como tal, o resto do mundo vê uma organização como uma única rede e não tem conhecimento pormenorizado da organização interna da estrutura. Um determinado endereço de rede pode ser dividido em várias sub-redes. Por exemplo, 172.16.1.0, 172.16.2.0, 172.16.3.0, 172.16.4.0 são endereços de sub-redes dentro da rede 171.16.0.0 (Todos os 0s do campo *hostid* de um endereço especifica a rede inteira).

Supernetting permite atribuir endereços a uma única organização de modo a abranger múltiplos prefixos de classes. Um endereço classe C não pode acomodar mais de 254 utentes e um endereço de classe B não tem bits suficientes para fazer o *subnetting* conveniente.

Classless Interdomain Routing (CIDR)

Em Setembro de 1993, era evidente que com o crescimento de utilizadores de Internet era necessário uma solução, enquanto os detalhes do IPv6 não eram decididos. Isto levou a um novo método de endereçamento baseado em *Classless Interdomain Routing* (CIDR). Apesar do nome, o CIDR especifica tanto endereçamento como encaminhamento. O esquema de endereçamento introduzido pelo CIDR (RFC 1519) é um esquema mais flexível na atribuição de endereços, o que permite ao utente ou à sub-rede uma gama de endereços de qualquer tamanho.

Com a introdução do CIDR, os endereços não utilizados pelas classes B e C poderão ser partilhados entre mais organizações do que de outra forma não teria sido possível. A

principal característica do CIDR é a separação do endereço de rede e do endereço da sub-rede, por meio de uma máscara de sub-rede. É a máscara de sub-rede que revela onde o primeiro bit do endereço da sub-rede começa.

O CIDR não tem classes, o que representa um afastamento do modelo original de classes do IPv4. O CIDR preocupa-se com *interdomain routing* em vez da identificação do utente. O CIDR possui uma estratégia para a atribuição e utilização de endereços IPv4, em vez de uma nova proposta [18].

Mapeamento por Máscara

O Masking é um processo que extrai o endereço da rede física de um endereço IP. *Masking* pode ser realizado independentemente do facto de ter sido subdividido ou não. Sem sub-rede, o processo de *masking* extrai o endereço de rede a partir do endereço IP, enquanto com sub-redes, o *masking* também extrai o endereço da sub-rede do endereço IP. O *masking* pode ser conseguido realizando uma máscara de um endereço IP de 32 bits com outro endereço IP de 32 bits. Uma máscara consiste numa *string* contígua de 1s e 0s. A máscara contígua significa que uma *string* de 1s precede uma *string* de 0s. Para obter tanto o endereço de rede como o endereço da sub-rede, a operação lógica AND numa base de bit-a-bit deve ser aplicada sobre o endereço IP e sobre a máscara.

O mapeamento de um endereço lógico para um endereço físico pode ser estático ou dinâmico. Mapeamento estático envolve uma lista de correspondências entre o endereço físico e o endereço lógico, mas a manutenção da lista exige uma *overhead* elevado. O *Address Resolution Protocol* (ARP) é um método de mapeamento dinâmico que visa encontrar um endereço físico a partir de um endereço lógico. Um ARP *request* é *broadcast*, sendo difundido para todos os dispositivos na rede, enquanto que um ARP *reply* é *unicast* sendo enviado para o utente que solicitou o mapeamento. O *Reverse Address Resolution Protocol* (RARP) é uma forma de mapeamento dinâmico em que um determinado endereço físico está associado a um endereço lógico. O ARP e o RARP utilizam endereços físicos *unicast* e *broadcast*.

3.1.1.3. Encaminhamento IP

Conceptualmente, o encaminhamento IP é simples, especialmente para um utente, se o destino está directamente ligado ao utente (por exemplo, uma ligação ponto-a-ponto) ou

numa rede partilhada (por exemplo, *Ethernet* ou *Token Ring*), então o datagrama IP é enviado directamente para o destino. Caso contrário, o utente envia o datagrama para um *router*, e permite que o router entregue o datagrama ao seu destino. Este sistema lida com a maioria das configurações de rede.

Num sistema de comutação de pacotes, o encaminhamento refere-se ao processo de escolha de um caminho por onde se vai enviar os pacotes. Ao contrário do encaminhamento dentro de uma única rede, o encaminhamento IP deve escolher o algoritmo apropriado para decidir a melhor forma de enviar um datagrama através de várias redes físicas. Na verdade, o encaminhamento através da Internet é geralmente difícil porque muitos computadores têm múltiplas conexões de rede física.

A Internet é composta por várias redes físicas interligadas por *routers*. Cada *router* tem ligações directas a duas ou mais redes, enquanto que um utente normalmente se liga directamente a uma única rede física. A entrega de pacotes através de uma rede pode ser gerida em qualquer camada do modelo OSI. A camada Física é gerida pelo endereço *Media Access Control* (MAC), a camada Ligação de Dados inclui o *Logical Link Control* (LLC) e a camada Rede é onde a maior parte do encaminhamento ocorre.

3.1.2 *Address Resolution Protocol* (ARP)

Uma ligação de dados como a *Ethernet* ou o *Token Ring* tem seu próprio sistema de endereçamento (normalmente endereços de 48 bits), para o qual qualquer camada de rede utilizando a ligação de dados deve respeitar. O ARP fornece um mapeamento dinâmico entre dois tipos diferentes de endereços: endereços IP de 32-bits e qualquer tipo de endereço que a ligação de dados utiliza. No RFC 826 é feita a especificação do ARP.

O mapeamento de um endereço IP para um endereço físico pode ser feito por mapeamento estático ou dinâmico. Mapeamento estático significa criar uma tabela que associa um endereço IP com um endereço físico, no entanto, este mapeamento tem algumas limitações porque as pesquisas na tabela são ineficientes. Como consequência, o mapeamento estático cria uma enorme sobrecarga na rede. Mapeamento dinâmico pode empregar um protocolo para encontrar o outro. Dois protocolos (ARP, RARP) foram concebidos para realizar mapeamento dinâmico. Quando um utente precisa de encontrar o endereço físico de outro utente ou *router* na sua rede, envia um pacote ARP *query*. O destinatário reconhece o seu

endereço IP e envia de volta uma resposta ARP que contém os endereços IP e físico do destinatário.

O ARP é um protocolo que pode "resolver" um endereço IP para um endereço hardware. O primeiro local para onde o ARP olha para resolver um endereço IP para um endereço hardware é no ARP *cache*. O ARP *cache* é na área *random access memory* (RAM), onde o ARP mantém os endereços IP e hardware que foram resolvidos. Se o ARP conseguir encontrar os endereços IP e hardware no ARP *cache*, o pacote é dirigido ao endereço hardware sem mais nenhuma resolução. Se o endereço IP não está no ARP *cache*, ARP vai iniciar um ARP *request broadcast* na rede local. O ARP *broadcast* é endereçado a todos os utentes definindo o endereço hardware destino com FF:FF:FF:FF:FF:FF.

O ARP *broadcast* contém o endereço IP do destino de modo a que o destinatário seja identificado. O ARP *broacast* também contém o endereço hardware da fonte, para acelerar a resposta do destino. Após o destino receber e reconhecer que a ARP *broadcast* é destinado a ele, o destino coloca o endereço IP e o endereço hardware no seu próprio ARP *cache*, assim quando o ARP *reply* for enviado para a fonte, o endereço solicitado já é conhecido. Se o ARP *request* não for para este utente, o pacote é descartado. Quando o ARP *reply* é recebido, o endereço hardware e o endereço IP são colocados no ARP *cache* durante dois minutos.

Depois de um endereço IP ser resolvido para um endereço hardware, estes são armazenados no ARP *cache* durante dois minutos. Se o IP solicitar nova resolução para o mesmo endereço IP dentro desses dois minutos, a entrada fica no ARP *cache* mais dois minutos. Uma entrada pode permanecer na *cache* ARP por um máximo de 10 minutos, sendo removida do *cache* independentemente do facto de ter sido referenciada nos últimos dois minutos [2, 4, 6, 7, 15, 19].

3.1.3 *Reverse Address Resolution Protocol* (RARP)

Para criar um datagrama IP, um utente ou um *router* precisa de saber o seu próprio endereço IP, que é independente do endereço físico. O RARP foi concebido para "resolver" o mapeamento do endereço de uma máquina onde o seu endereço físico é conhecido, mas o seu endereço lógico (IP) é desconhecido. A máquina pode obter o seu endereço físico, que é único localmente, podendo desta forma usar o endereço físico para

obter o endereço IP lógico utilizando o protocolo RARP. Na realidade, o RARP é um protocolo de mapeamento dinâmico, em que um determinado endereço físico está associado a um endereço IP lógico. Para obter o endereço IP, um pacote RARP *request* é transmitido para todos os sistemas da rede, e todos os utentes e *routers* sobre a rede física vão receber o RARP *request*, mas apenas o servidor RARP irá responder. O servidor envia então, um pacote RARP *reply* incluindo o endereço IP do solicitador [4, 6, 7, 15, 20].

3.1.4 IP Versão 6 (IPv6, ou IPng)

O *Internet Protocol version* 6 (IPv6) é definido no RFC 2460. A IETF (*Internet Engineering Task Force*) decidiu manter a mesma arquitectura básica e o funcionamento do protocolo IPv4, de modo a que a migração do IPv4 para o IPv6 pudesse ser mais gradual, e que as duas versões diferentes do protocolo IP co-existissem. Uma das principais razões para o desenvolvimento de IPv6 foi, simplesmente, a necessidade de aumentar o número de endereços disponíveis.

O IP versão 6 (IPv6), também conhecido como *Internet Protocol next generation* (IPng), é a nova versão do protocolo Internet, projectado para ser um substituto do IPv4. No entanto, devido ao elevado número de sistemas na Internet, a transição do IPv4 para o IPv6 não pode ocorrer imediatamente, levando algum tempo até que todos os sistemas na Internet possam passar de IPv4 para o IPv6. As principais diferenças entre o IPv6 e o IPv4, são apresentadas a seguir.

- O tamanho do endereço IPv6 é aumentado de 32 para 128 bits.
- O IPv6 pode configurar automaticamente endereços locais e localizar *routers* IP de modo a reduzir problemas de configuração e de *setup*.
- O formato do cabeçalho IPv6 foi simplificado e alguns campos do cabeçalho foram retirados. Este novo formato do cabeçalho melhora o desempenho dos *routers* e tornar-se mais fácil de adicionar novos tipos de cabeçalho.
- A arquitectura do IPv6 suporta autenticação, a integridade e confidencialidade dos dados.
- Um novo conceito de fluxos (*flow*) foi adicionado para responder às crescentes necessidades do transporte de *streams* de comunicação em tempo real.

Apesar das estratégias de *subnetting* e *supernetting* terem atenuado alguns problemas de endereçamento, o encaminhamento tornou-se mais complicado. As opções de encriptação e autenticação do IPv6 fornecem confidencialidade e integridade do pacote [3, 4, 6, 7, 15, 21, 22].

3.1.4.1. Endereçamento IPv6

O RFC 2373 descreve o espaço do endereço associado ao IPv6. A maior preocupação será a migração de IPv4 para o IPv6. O endereçamento IPv4 tem as seguintes deficiências: o IPv4 foi definido quando a Internet era pequena e composta de redes de pequena dimensão e complexidade. Ele tem dois níveis na estrutura do endereço (*netid* e o *hostid*), com três formatos de endereço (classe A, B e C) para acomodar rede de diferentes tamanhos. Tanto o espaço limitado para o endereço e o tamanho de 32-bits do endereço IPv4 revelou-se insuficiente para processar, o aumento do tamanho da tabela de encaminhamento, causado pelo crescente aumento do número de utentes e servidores activos.

O IPv6 foi concebido para melhorar o IPv4 em cada uma destas áreas. O IPv6 atribui 128 bits para os endereços, e a análise mostra que este espaço para o endereço é suficiente para incorporar hierarquias flexíveis e para distribuir a responsabilidade da atribuição e gestão dos endereços IP.

Tal como no IPv4, os endereços IPv6 são representados como uma *string* de 128 bits ou 32 dígitos hexadecimal, os quais são subdivididos em oito inteiros de 16 bits separados por dois pontos (:). A representação básica assume a forma de oito secções, cada uma com um tamanho de dois bytes, (xx:xx:xx:xx:xx:xx:xx:xx) onde cada xx representa a forma hexadecimal de 16 bits do endereço.

3.1.4.2. Formato do Cabeçalho IPv6

O datagrama IPv6 é composto por um cabeçalho base (40 bytes) seguido pelo *payload*. O *payload* é constituído por duas partes: a opção *extension headers* e dados da camada superior. O *extension headers* e os pacotes da camada superior dados normalmente ocupam até 65 535 bytes de informação. A Figura 3.3 ilustra o formato do pacote IPv6. Cada datagrama IPv6 começa com um cabeçalho base. O cabeçalho IPv6 tem um tamanho fixo de 40 octetos, composto pelos seguintes campos:

- ***Version*** (4 bits) – Este campo define o número da versão do IP. Para o IPv6, o valor é logicamente 6.
- ***Priority*** (4 bits) – Este campo está disponível para os *routers* identificarem e distinguirem entre diferentes classes ou prioridades dos pacotes IPv6. Este campo é muito semelhante ao campo TOS do IPv4, de modo a proporcionar diferentes formas de "serviço diferenciado" para os pacotes IP.
- ***Flow label*** (24 bits) – Este campo foi projectado para fornecer tratamento especial a um determinado fluxo de dados. Este campo contém informação que os *routers* utilizam para associar um datagrama com um fluxo e prioridade específicos.

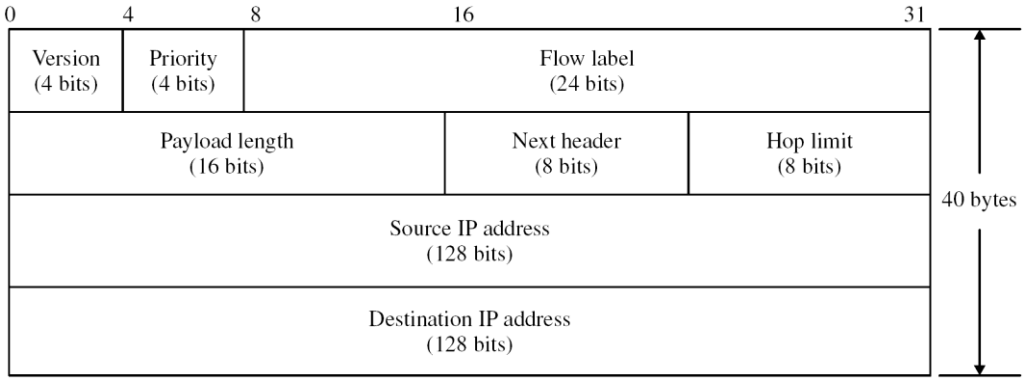

Figura 3.3 – Formato do cabeçalho IPv6

- ***Payload length*** (16 bits) – Este campo define o tamanho total do datagrama IP excluindo o cabeçalho base. O *payload* consiste de *extension headers* mais os dados da camada superior. Ocupa até 2^{16} - 1 = 65 535 bytes.
- ***Next header*** (8 bits) – Este campo define o cabeçalho que segue o cabeçalho base no datagrama. O próximo cabeçalho ou é um *extension header* opcional usado pelo IP ou um cabeçalho de um protocolo da camada superior, como o TCP ou UDP. *Extension headers* acrescentam funcionalidade ao datagrama IPv6.

  A Tabela 3.5 mostra os valores possíveis para o campo *Next header* (ou seja, *extension headers* IPv6). Seis tipos de *extension headers* foram definidos. Estes são o *hop-by-hop option*, *source routing*, *fragmentation*, *authentication*, *encrypted security payload*, e o *destination option*.

Tabela 3.5 – Opções para o *Next Header*

Código	*Next header*
0	*Hop-by-hop option*
2	*ICMP*
6	*TCP*
17	*UDP*
43	*Source routing*
44	*Fragmentation*
50	*Encrypted security payload*
51	*Authentication*
59	*Null (sem next header)*
60	*Destination option*

- ***Hop-limit*** (8 bits) – Este campo decrementa 1 valor por cada nó que encaminha o pacote. O pacote é descartado na situação em que o *hop limit* chegar até zero. Este campo serve o mesmo objectivo que o campo TTL em IPv4. O protocolo IPv6 interpreta o valor como sendo o número máximo de *hops* que um datagrama pode fazer antes a ser descartado.
- ***Source IP address*** (128 bits) – Este campo de 128 bits indica o endereço IP da fonte.
- ***Destination IP address*** (128 bits) – Este campo especifica um endereço de 128 bits que geralmente identifica o destino final do datagrama. No entanto, se *source routing* for usado, este campo contém o endereço do próximo *router*.

3.1.5 *Internet Control Message Protocol* (ICMP)

O protocolo ICMP é uma extensão para o Protocolo Internet, o qual é usado para comunicar entre uma *gateway* e uma fonte, para gerir erros e gerar mensagens de controlo. O Protocolo de Internet (IP) não foi concebido para ser absolutamente fiável. O objectivo das mensagens de controlo (ICMP) é o de fornecer *feedback* sobre problemas na comunicação, e não tornar o IP fiável.

Não há garantias de que um datagrama será entregue ou uma mensagem de controlo será enviada. Alguns datagramas podem não ser entregues sem que haja um relatório da sua

perda. O IP é um protocolo não fiável que não tem qualquer mecanismo para detecção de erros ou de controlo de erros.

O ICMP foi projectado para compensar esta deficiência do IP, no entanto, o ICMP não corrige os erros, apenas os reporta. O ICMP usa o endereço IP da fonte para enviar a mensagem de erro à fonte do datagrama, as mensagens ICMP consistem em mensagens *error-reporting* e mensagens *query*. As mensagens *error-reporting* reportam sobre problemas que um *router* ou um utente destino possa ter tido quando processam um pacote IP. Para além das mensagens *error-reporting* o ICMP pode diagnosticar alguns problemas da rede através de mensagens *query*.

Um exemplo da utilização do protocolo ICMP como ferramenta de diagnóstico, está na utilidade *Ping* (*Packet Internet Groper*). Um administrador usa o *Ping* para enviar quatro pacotes ICMP *echo request* para o destino e pedir que o destino responda a estes pacotes. O ICMP coloca uma pequena quantidade de dados e solicita que os dados sejam enviados de volta, se os dados retornarem, o administrador pode assumir que existe conectividade com o destino. Se os pacotes ICMP não retornarem, então existe um problema na ligação [4, 6, 15, 23].

3.1.6 *Internet Group Management Ptotocol* (IGMP)

O *Internet Group Management Protocol* (IGMP) é usado para facilitar a transmissão de uma mensagem para um grupo de utentes ao mesmo tempo. O IGMP é especificado no RFC 1112 (versão 1) e no RFC 2236 (versão 2). O IGMP, como o ICMP é parte integrante do protocolo Internet (IP). O IGMP contribui para que os *routers multicast* possam manter uma lista de endereços *multicast* de grupos. "*Multicasting*", significa o envio da mesma mensagem para mais do que um destinatário em simultâneo. Quando o *router* recebe uma mensagem com um endereço destino da lista, ele transmite a mensagem, convertendo o endereço IP *multicast* para um endereço físico *multicast*.

Para participar, numa rede local IP, o utente deve informar os *routers multicast* locais. Os *routers* locais contactam os outros *routers multicast*, passando informações de *membership* e estabelecendo um caminho. O IGMP tem apenas dois tipos de mensagens: *reports* e *query*. A mensagem *report* é enviada a partir do utente para o *router*. A mensagem *query* é enviada a partir do *router* para o utente. Um *router* envia um IGMP *query* para determinar

se um utente pretende continuar ligado a um grupo. A mensagem *query* é transmitida usando o endereço *multicast* 244.0.0.1. A mensagem *report* é transmitida usando um endereço destino igual ao endereço *multicast* reportado. Endereços IP que comecem com 1110 são endereços *multicast*, que são endereços classe D.

A mensagem IGMP é encapsulada num datagrama IP com o valor do protocolo de dois. Quando a mensagem é encapsulada no datagrama IP, o valor de TTL deve ser um, isto é necessário porque o domínio do IGMP é a LAN [4, 6, 7].

3.2 Protocolos da Camada Transporte

A camada Transporte determina se o emissor e o receptor vão estabelecer uma conexão antes de iniciarem a comunicação e com que frequência irão enviar *acknowledgments* da ligação entre eles. Existem dois protocolos para a camada Transporte: o *Transmission Control Protocol* (TCP) e o *User Datagram Protocol* (UDP). Enquanto o TCP estabelece uma conexão e envia *acknowledgments*, dando desta forma garantias de entrega, o UDP não, tornando por outro lado, a transmissão mais rápida. Tanto o TCP como o UDP estão entre a camada Aplicação e a camada Rede. Como um protocolo da camada Rede, o protocolo IP é responsável pela comunicação *host-to-host* ao nível do computador, enquanto que o TCP ou o UDP são responsáveis pela comunicação *process-by-process*, na camada Transporte.

3.2.1 *Transmission Control Protocol* (TCP)

O *Transmission Control Protocol* (TCP) é o protocolo de transporte do IP-*suite* que fornece o transporte de dados com garantia de entrega. O TCP é o protocolo que estabelece a ligação entre o emissor e o receptor. TCP é um protocolo orientado à conexão, o que significa que os dois utentes que estão em comunicação têm conhecimento um do outro. O TCP é definido no RFC 793.

O TCP fornece ligações entre clientes e servidores. Um cliente estabelece uma ligação TCP com um determinado servidor, troca dados com o servidor através da ligação e, em seguida, termina a ligação. O TCP é responsável pelo controlo do fluxo/erro e pela entrega do datagrama sem erros ao programa.

São necessários dois identificadores, o endereço IP e o número de porto, para uma ligação cliente/servidor oferecer um serviço *full-duplex*, ou seja, são estabelecidos dois canais lógicos independentes, um em cada sentido. A combinação do endereço IP e o número de porto, às vezes é referido como *socket*, que será abordado no Capitulo seguinte.

O TCP proporciona um processo de três etapas no transporte de dados. Na primeira fase, é criada uma ligação entre as duas estações que querem comunicar. Esta fase é chamada de sincronização. Após uma sincronização bem sucedida, há uma fase de transferência de dados, de modo a que a transferência de dados seja fiável (ou seja, que a entrega dos pacotes possa ser garantida), o TCP utiliza pacotes *acknowledgement*, bem como controlo de fluxo e controlo de congestionamento durante a fase de transferência dos dados. Após a transmissão de dados, qualquer uma das duas estações pode terminar a ligação. Se em algum momento, houver um grave erro na conexão da rede, a ligação é imediatamente fechada.

3.2.1.1. Cabeçalho TCP

Os dados TCP são encapsulados num datagrama IP como mostra a Figura 3.4. O pacote TCP é constituído por um cabeçalho de 20 a 60 bytes, seguido dos dados do programa. O cabeçalho é de 20 bytes, se não incluir nenhuma opção, e até 60 bytes, se utilizar alguma opção. A Figura 3.5 ilustra o formato do pacote TCP, cujo cabeçalho é explicado de seguida.

- ***Source* e *destination port numbers*** (16 bits cada) – Cada pacote TCP contém um campo de 16 bits, que define o número de porto da fonte e do destino para identificar a aplicação emissora e receptora. Estes dois números de porto, juntamente com os endereços IP da origem e do destino no cabeçalho IP, identificam unicamente cada ligação.
- ***Sequence number*** (32 bits) – Durante o estabelecimento da ligação, um gerador de números aleatórios cria o *Initial Sequence Number* (ISN), que é escolhido aleatoriamente pelo transmissor, e qualquer mensagem de dados posteriores tem um *sequence number* maior do que o anterior. O incremento no *sequence number* depende do número de octetos do pacote anterior. O *sequence number*

conta em octetos, mas o seu valor também identifica, exclusivamente, um pacote específico, permitindo assim que o receptor:

- Verifique se recebeu todos os octetos (e, portanto, todos os pacotes) dos dados do utilizador.
- Inequivocamente informar a recepção de todos os pacotes de volta para o transmissor.
- Ordenar a retransmissão de qualquer pacote em falta.
- Re-sequenciar os pacotes na ordem correcta, se eles forem recebidos fora de ordem.

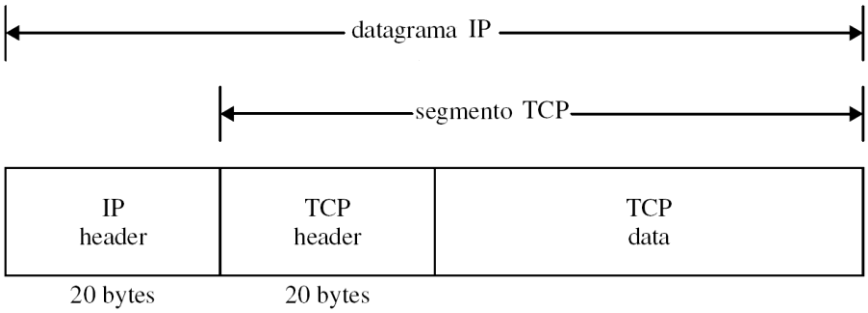

Figura 3.4 – Encapsulamento TCP

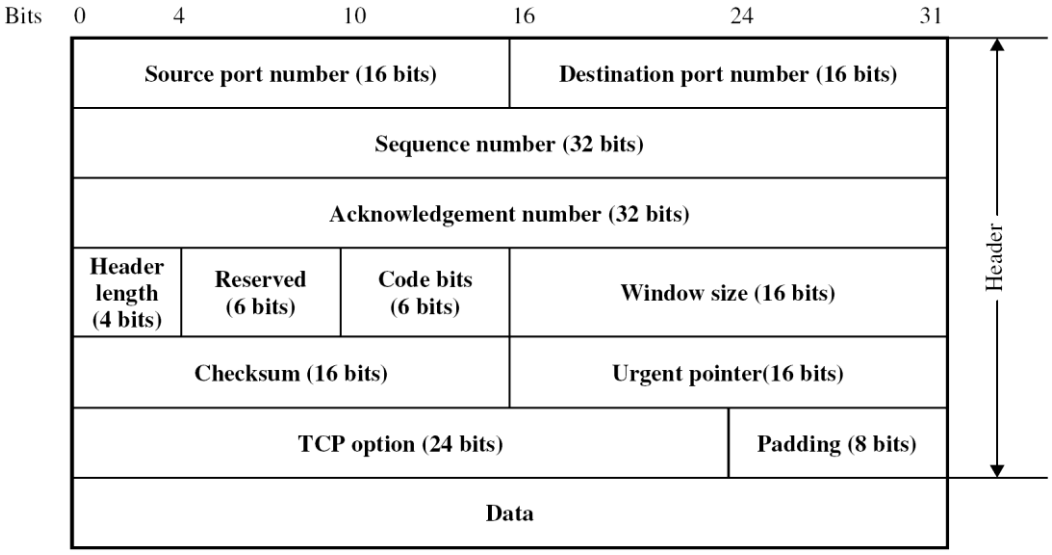

Figura 3.5 – Formato do pacote TCP

- ***Acknowledgement number*** (32 bits) – Este campo define o byte que o emissor do pacote está à espera de receber do receptor. Uma vez que o TCP fornece um serviço *full-duplex* para a camada Aplicação, os dados podem ser transmitidos nos dois sentidos simultaneamente. O *sequence number* refere-se ao *stream* que vai no mesmo sentido do pacote, enquanto o *acknowledgement number* refere-se ao *stream* que flui na direcção oposta ao do pacote. Por conseguinte, o *acknowledgement number* é o *sequence number* mais 1 relativamente ao último pacote de dados recebido com sucesso. Este campo só é válido se a *flag* ACK estiver activa.
- ***Header length*** (4 bits) – Este campo indica o tamanho (em *words* de 32 bits) do cabeçalho TCP (com efeito, se e quantas opções TCP estão a ser usadas). O valor também indica o início do campo TCP *data*. Uma vez, que o cabeçalho tem um tamanho entre 20 e 60 bytes, um valor inteiro deste campo poderá ser entre 5 e 15, pois $5 \times 4 = 20$ bytes e $15 \times 4 = 60$ bytes.
- ***Reserved*** (6 bits) – Este campo é reservado para uso futuro.
- ***Code bits*** (6 bits) – Há seis *flag* bits no cabeçalho TCP. As *flags* possíveis são SYN, FIN, RST, PSH, URG e ACK. A seguir apresenta-se uma breve descrição de cada uma.
  - URG – Esta *flag* significa que o pacote possui dados urgentes. Quando activo, o *urgent pointer* é válido, indicando onde começam os dados não urgentes do pacote.
  - ACK – O *acknowlegment number* é válido.
  - PSH – Esta *flag* indica que o emissor invocou a operação *push*, que é uma notificação do emissor para o receptor, para que este transmita todos os dados que ele tem ao processo do emissor.
  - RST – Faz o *reset* à ligação.
  - SYN – Sincroniza o *sequence number* para iniciar a ligação.
  - FIN – Significa que o emissor não tem mais dados para transmitir.

- ***Window size*** (16 bits) – Este campo de 16 bits define o tamanho da janela em bytes. Uma vez que o tamanho deste campo é de 16 bits, o tamanho máximo da janela é 2^{16} - 1 = 65 535 bytes. O controlo de fluxo do TCP é assegurado por cada estação, indicando o tamanho da janela. Este é o número de bytes que o receptor está disposto a aceitar.

- ***Checksum*** (16 bits) – Este campo fornece um meio de detectar erros no segmento TCP. O *checksum* abrange o segmento TCP, ou seja, o cabeçalho TCP H e o TCP *data*. Este é um campo obrigatório que deve ser calculado e armazenado pelo emissor e, em seguida, verificado pelo receptor, indicando se o cabeçalho foi danificado durante o transporte.

- ***Urgent pointer*** (16 bits) – Este campo só é válido se a *flag* URG estiver activa. O *urgent pointer* é usado quando o pacote contém dados urgentes. O valor contido neste campo indica o número do octeto do primeiro octeto de dados TCP "reais" do utilizador, após os dados urgentes do campo de dados TCP. Dados urgentes, normalmente, correspondem a um comando de controlo do computador tal como *abort*, *break*, *escape*, etc.

- ***TCP Options*** (24 bits) – Este campo tem um tamanho variável dependendo de que opções forem incluídas. O tamanho do cabeçalho TCP também varia dependendo das opções seleccionadas. O cabeçalho TCP pode ter até 40 bytes de informação opcional. As opções são usadas para transmitir informações adicionais para o destino ou para incluir outras opções. Cada *TCP option* inclui um campo *option kind*. O campo *option kind* é codificado de acordo com a Tabela 3.6, o que também enumera as principais opções.

O TCP é um protocolo orientado à conexão da camada Transporte do TCP/IP *suite*, que fornece uma ligação *Full-duplex* entre duas aplicações, permitindo trocar grandes volumes de dados de forma eficiente. Uma vez que o TCP fornece controlo de fluxo, permite que sistemas de diferentes velocidades possam comunicar. A detecção de erros é feita pelo *checksum*, *acknowledgment* e *timeout*. O TCP é usado por muitas aplicações, as mais conhecidas são o HTTP (*World Wide Web*), TELNET, Rlogin, FTP e SMTP para o correio electrónico [4, 6, 7, 13, 15, 24].

Tabela 3.6 – Opções do TCP

Valor do Option Kind	*Tamanho da opção em octetos*	*Opção*	*Definido no*
0	----	*Fim da lista de opções*	*RFC 0793*
1	----	*Nenhuma operação*	*RFC 0793*
2	*4*	*MSS (maximum segment size)*	*RFC 0793*
3	*3*	*WSOPT (window scale option)*	*RFC 1323*
4	*2*	*SACK (selective acknowlegment)*	*RFC 2018*
5	*Variável*	*SACK*	*RFC 2018*
8	*10*	*TSOPT (timestamp option)*	*RFC 1323*
9	*2*	*Partial order connection permitted*	*RFC 1693*
10	*3*	*Partial order service profile*	*RFC 1693*
14	*3*	*TCP alternate checksum request*	*RFC 1146*
15	*Variável*	*TCP alternate checksum data*	*RFC 1146*
19	*8*	*MD5(opção de encriptação)*	*RFC 2385*

3.2.2 *User Datagram Protocol* (UDP)

O *User Datagram Protocol* (UDP) é um protocolo da camada Transporte. Tal como TCP, o UDP serve de intermediário entre as aplicações e a rede. O UDP fornece às aplicações uma interface directa com o IP. Permite a troca de dados entre aplicações, e não apenas entre estações, através da introdução no seu cabeçalho de um campo identificador do porto. Este protocolo é descrito no RFC 768.

O UDP usa números de porto para realizar uma comunicação *process-to-process*, fornece controlo de fluxo ao nível do transporte, desempenha uma detecção de erros bastante limitada. UDP é um protocolo *connectionless* que não fornece serviços fiáveis como o TCP. É mais rápido e utiliza menos largura de banda, porque uma conexão UDP não está continuamente em manutenção para garantir que existe ligação, por isso é frequentemente utilizado para áudio ou vídeo. Ao contrário do TCP este protocolo não garante a entrega dos dados, nem repete a transmissão se uma transferência for corrompida.

3.2.2.1. Cabeçalho UDP

O UDP recebe os dados e adiciona o cabeçalho UDP, então passa o *user datagram* para o IP com os endereços *socket* (*source* e *destination port numbers*). O IP acrescenta o seu

próprio cabeçalho. O datagrama IP é, então, enviado para a camada Ligação de Dados, que acrescenta o seu próprio cabeçalho, e passa-o para a camada Física. A camada Física codifica os bits em sinais eléctricos ou ópticos e envia-os para a máquina remota. Não há nenhuma garantia de que um datagrama UDP irá chegar ao seu destino final, de que a ordem dos pacotes será preservada, ou que os datagramas chegam apenas uma vez.

A Figura 3.6 ilustra o encapsulamento de um datagrama UDP como um datagrama IP e a Figura 3.7 apresenta o cabeçalho UDP.

- ***Source port number*** (16 bits) – Este campo identifica o número de porto da aplicação origem. Uma vez que o *Source port number* é de 16 bits, ele pode variar de 0 a 65 656 bytes. Se a fonte é o cliente, o programa cliente atribui um número aleatório chamado de número de porto efémero solicitado pelo processo e escolhido pelo software UDP. Se a fonte é um servidor, o número de porto é um número de porto universal.

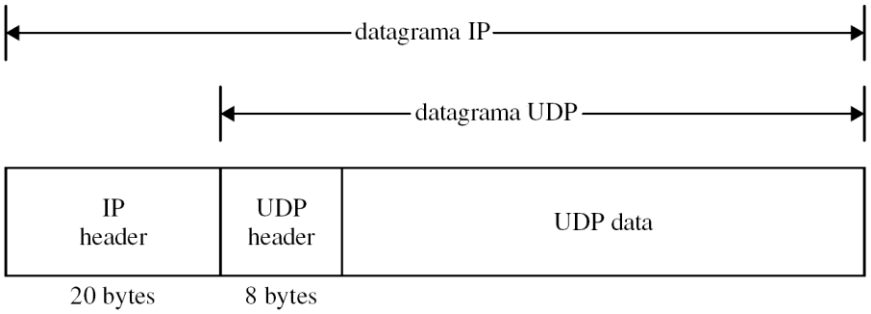

Figura 3.6 – Encapsulamento UDP

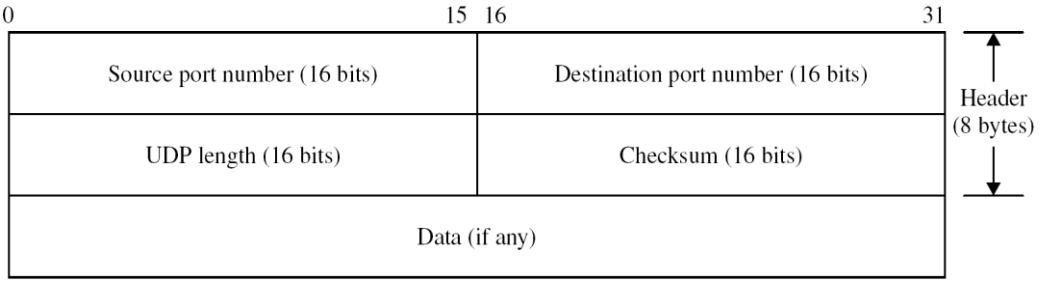

Figura 3.7 – Cabeçalho UDP

- ***Destination port number*** (16 bits) – Este campo identifica o número de porto da aplicação destino. Se o destino é um servidor, o número de porto é um número

de porto universal, mas se o destino é o cliente, o número de porto é o número de porto efémero.

- ***UDP Length*** (16 bits) – Este campo contém o tamanho de bytes do datagrama UDP, incluindo o cabeçalho UDP e os dados do utilizador. Este campo de 16 bits pode definir um tamanho total de 0 a 65 535 bytes. No entanto, o valor mínimo é de oito, o que indica um datagrama UDP apenas com cabeçalho e sem dados. Assim, o comprimento dos dados varia entre 0 e 65 507 bytes, subtraindo ao comprimento total de 65 535 bytes, os 20 bytes para um cabeçalho IP e os 8 bytes de um cabeçalho UDP. O datagrama IP contém o seu tamanho total em bytes, assim o tamanho do datagrama UDP é o tamanho total menos o tamanho do cabeçalho IP.
- ***Checksum*** (16 bits) – O UDP *checksum* é usado para detectar erros sobre todo o *user datagram* que abrange o cabeçalho UDP e os dados UDP. Os cálculos do UDP *checksum* incluem um pseudocabeçalho, o cabeçalho UDP e os dados provenientes da camada Aplicação. O valor do campo para o protocolo UDP é 17. Se este valor mudar durante a transmissão, o cálculo do *checksum* no receptor irá detectar a alteração e então o UDP descarta o pacote.

3.2.2.2. Multiplexagem e Desmultiplexagem

Numa máquina a utilizar o TCP/IP *suite*, existe apenas um UDP, mas pode haver vários processos que podem querer utilizar os serviços do UDP. Para lidar com esta situação, UDP necessita de multiplexagem e desmultiplexagem.

- **Multiplexagem**: Do lado do emissor, poderá haver vários processos que precisam de *user datagrams*, mas só há um UDP. Esta é uma relação de muitos-para-um que exige multiplexagem. O UDP aceita mensagens de diferentes processos, diferenciados pelos seus números de porto atribuídos. Depois de adicionar o cabeçalho, o UDP passa o *user datagram* para o IP.
- **Desmultiplexagem**: Do lado do receptor, existe também apenas um UDP. No entanto, pode acontecer haver muitos processos que podem receber *user datagram*. Esta é uma relação de um-para-muitos que exige desmultiplexagem. O protocolo UDP recebe *user datagrams* do IP e após a verificação da existência

de erros e retirar o cabeçalho, o UDP entrega cada mensagem para o processo adequado com base nos números de porto.

O UDP é um processo adequado para um processo que requer uma comunicação *request-response* sem grandes preocupações com o controlo de fluxo e de erro. Não é adequado para um processo que precisa de enviar os dados em massa, como o FTP. No entanto, o UDP é o protocolo de transporte para vários protocolos da camada Aplicação incluindo o *Network File System* (NFS), o *Simple Network Management Protocol* (SNMP), o *Domain Name System* (DNS), e o *Trivial File Transfer Protocol* (TFTP) [4, 6, 7, 13, 15, 25].

3.3 Protocolos da Camada Aplicação

A Camada Aplicação está associada com o maior número de protocolos que são identificados pelas suas aplicações. Temos o HTTP que é normalmente associado com a aplicação *Web browser*. Temos também protocolos de Transferência de Ficheiros, como o FTP e o TFTP que foram projectados para usar os serviços de transmissão fiável prestados pela combinação do TCP e do IP (TCP/IP). Os protocolos SMTP, POP3, IMAP e MIME estão relacionados com o Correio Electrónico. E as Aplicações Multimédia que utilizam os protocolos RTP e RTCP, para a transmissão de voz e vídeo sobre IP (VoIP).

A seguir é apresentada uma breve descrição dos protocolos utilizados nas Aplicações Multimédia, dado que os outros protocolos da camada Aplicação, apesar de importantes, não fazem parte do âmbito desta dissertação, logo não serão abordados.

3.3.1 Aplicações Multimédia

Aplicações multimédia, como aplicações de áudio e videoconferência necessitam de protocolos da camada Aplicação. Para além de um protocolo de sinalização QoS (*Quality of Service*), muitas aplicações multimédia também precisam de um protocolo de transporte com características bastante diferentes do TCP e com mais funcionalidades que o UDP. O protocolo que foi desenvolvido para satisfazer estas necessidades é chamado de *Real-Time Transport Protocol* (RTP).

Uma outra classe de protocolo que muitas aplicações multimédia precisam é de um protocolo de controlo de sessão. Por exemplo, se quisermos ter a possibilidade de fazer

chamadas telefónicas através da Internet. Teríamos que ter características como *call forwarding, three-way calling*. O SIP (*Session Initiation Protocol*) e o H.323 são exemplos de protocolos que abordam as questões do controlo de sessão [2, 7, 9, 15, 26].

3.3.1.1. *Real-Time Transport Protocol (RTP)*

O *Real-Time Transport Protocol* (RTP) foi definido pela *Audio-Video Transport Working Group* sob o RFC 1889. Como o próprio nome indica, este protocolo é usado para fornecer serviços de entrega *end-to-end* para dados com características de tempo real (tal como dados de áudio e vídeo).

A chave para o transporte de áudio/vídeo em redes IP é o *Real-time Transport Protocol* (RTP), junto com seus perfis e formatos de *payload*. O protocolo RTP visa prestar serviços úteis para o transporte de dados em tempo real, tal como áudio e vídeo, sobre redes IP. Estes serviços incluem *timing recovery*, *loss detection* e *correction*, *payload* e *source identification*, *reception quality feedback*, *media synchronization*, e *membership management*.

O protocolo RTP foi originalmente concebido como um protocolo para conferências multimédia *multicast*. O RTP não oferece garantias de qualidade de serviço (QoS), nem garante a ordem de entrega dos pacotes. Isto porque o RTP usa o UDP como protocolo de transporte. Desde essa altura, ele provou ser útil para uma série de outras aplicações: em videoconferência H.323, *webcasting*, distribuição de TV e em telefones fixos como em telemóveis. O protocolo já demonstrou usar comunicação ponto-a-ponto, como usar sessões *multicast* com milhares de utilizadores, pode ser usado por aplicações de baixa largura de banda, como também pode entregar sinais de *High-Definition Television* (HDTV) a taxas de gigabits.

Tal como o seu nome indica, o RTP é um protocolo de transporte. Com efeito, se este funcionasse como um protocolo de transporte convencional, o RTP exigiria que cada mensagem fosse directamente encapsulada num datagrama IP. Na verdade, o RTP não funciona como um protocolo transporte, embora seja permitido, o encapsulamento directo em IP não ocorre na prática. Em vez disso, o RTP é executado sobre o UDP, o que significa que cada mensagem RTP é encapsulada num datagrama UDP. A principal

vantagem de usar o UDP é o facto de um único computador poder ter várias aplicações a utilizarem o RTP sem interferência.

Detalhes do RTP

O Protocolo RTP realmente define dois protocolos, o RTP e o *Real-time Transport Control Protocol* (RTCP). O primeiro é utilizado para a troca de dados multimédia, enquanto o segundo é utilizado para enviar periodicamente informação de controlo associada a um determinado fluxo de dados. Ao ser executado sobre o UDP, o fluxo de dados RTP e a informação de controlo RTCP associada, utiliza portos consecutivos da camada Transporte. O fluxo de dados RTP utiliza um número de porto par enquanto a informação de controlo RTCP utiliza o número de porto impar mais próximo.

Como o protocolo RTP está projectado para suportar uma grande variedade de aplicações, este fornece um mecanismo flexível, pelo qual as novas aplicações podem ser desenvolvidas sem estarem repetidamente a rever o protocolo RTP. Para cada classe de aplicação (por exemplo, áudio), o RTP define um perfil e um ou vários formatos. O perfil fornece uma gama de informações que assegura um entendimento comum dos campos do cabeçalho RTP para a classe da aplicação, que será evidente quando se examina o cabeçalho em pormenor. A especificação do formato explica o modo como os dados que seguem o cabeçalho RTP estão a ser interpretados.

Formato do Cabeçalho RTP

Como o protocolo RTP foi concebido para o transporte de uma grande variedade de dados em tempo real, incluindo áudio e vídeo. Cada pacote RTP começa com um cabeçalho obrigatório, onde os campos do cabeçalho obrigatório especificam a forma de interpretar os restantes campos do cabeçalho e como interpretar o *payload*. A Figura 3.8 ilustra o formato do cabeçalho RTP. O pacote é divido em quatro partes, um cabeçalho RTP obrigatório, um *header extension* opcional, um *payload header* opcional (dependo do tipo de formato de *payload* usado) e o próprio *payload*.

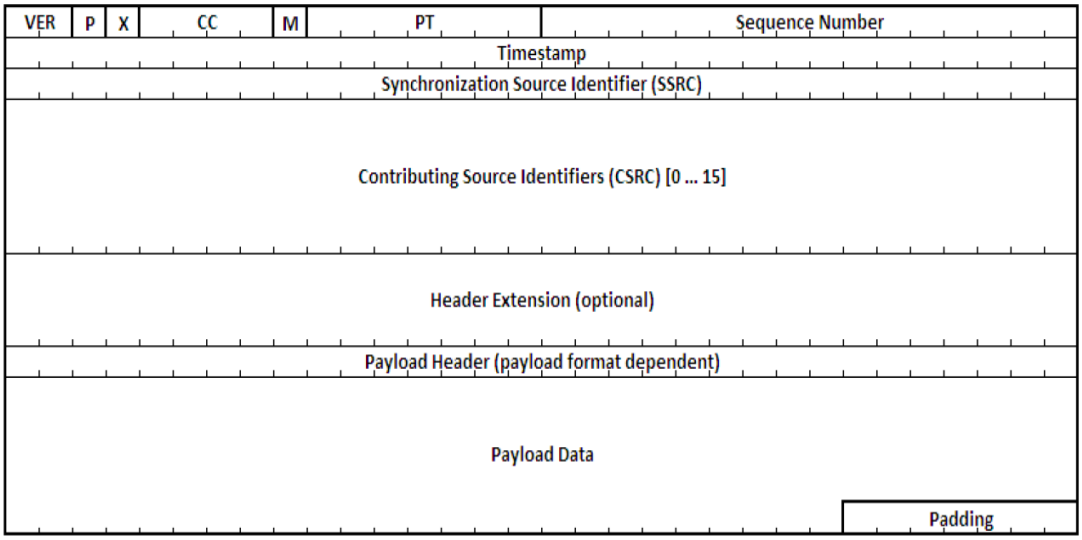

Figura 3.8 – Formato do cabeçalho RTP

O cabeçalho RTP obrigatório tem normalmente um tamanho de 12 bytes, mas opcionalmente pode conter o campo *Contributing Source* (CSRC) *identifiers*, que pode expandir o tamanho de 4 a 60 bytes. Os campos do cabeçalho obrigatório são apresentados a seguir.

- **VER** (2 bits) – Indica a versão do RTP, a versão actual (RFC 1889) é a versão 2.
- **P** (1 bit) – É o campo *padding bit*. Se o bit P está configurado com o valor "1", então o pacote inclui um ou mais octetos de *padding* no fim do *payload data*. Os octetos de *padding* não fazem parte do *payload* e devem ser removidos pelo receptor RTP antes de ser passado para protocolos da camada superior.
- **X** (1 bit) – É o *extension bit*. Especifica se um *header extension* está presente no pacote.
- **CC** (4 bits) – É o *CSRC count* (*contributing source count*). Este valor indica o número de identificadores CSRC que estão incluídos no cabeçalho.
- **M** (1 bit) – É o *marker bit*. Está destinado a ser utilizado pelas aplicações que necessitam de marcar eventos no fluxo de dados (por exemplo, no início de cada *frame* quando enviam vídeo), pode também ser usado como o marcador de reinício do FTP (*File Transfer Protocol*).
- **PT- *Payload Type*** (7 bits) – Identifica o perfil RTP que está a ser utilizado.

- ***Sequence number*** (16 bits) – É usado para detectar a perda de pacotes e restaurar a ordem dos pacotes. O valor inicial é um número escolhido aleatoriamente.
- ***Timestamp*** (4 bytes) – Reflecte o instante de tempo da amostragem do primeiro octeto no *payload*. É usado para reagrupar e sincronizar o sinal de tempo real na recepção.
- ***Synchronization Source* (SSRC) *identifiers*** (4 bytes) – Identifica os participantes de uma sessão RTP. É a fonte do fluxo de pacotes RTP. Os pacotes devem estar sincronizados para a reprodução, isso é feito por meio do *timestamp*, que normalmente, é codificado usando tempo *wallclock* i.e., tempo absoluto - representado, através do formato do *timestamp* do *Network Time Protocol (NTP)*.
- ***Contributing Source* (CSRC) *identifiers*** (4 a 60 bytes) – A lista de *contributing sources* (CSRCs) identifica os participantes que contribuíram com um pacote RTP mas que não foram responsáveis pela sincronização. Cada *contributing source identifier* é um inteiro de 32 bits, que corresponde ao SSRC do participante que contribuiu para o pacote.

Header Extension

O RTP prevê a possibilidade de *header extension*, que são apresentados após o cabeçalho RTP obrigatório, mas antes de qualquer cabeçalho *payload header* e *payload data*. Os *extension headers* são de tamanho variável, mas eles começam com um campo *type* de 16 bits seguido de um campo *length* de 16 bits.

Payload Header

O cabeçalho obrigatório RTP fornece informações que são comuns a todos os formatos de payload. Em muitos casos, um formato de *payload* irá precisar de mais informações para optimizar a operação, esta informação adicional cria um cabeçalho que é definido como parte da especificação do formato de *payload*. O *paylod header* está incluído num pacote RTP após o cabeçalho obrigatório e qualquer lista CSRC e *header extension*. Muitas vezes a definição de *payload header* constitui a maioria das especificações de *payload*.

A informação contida no *payload header* pode ser tanto estática - o mesmo para todas as sessões usando um determinado formato de *payload*, ou dinâmico. A especificação do formato de *payload* irá indicar quais as partes do *payload header* que são estáticas e quais as que são dinâmicas.

Payload Data

Uma ou mais *frames* de *payload*, directamente a seguir a qualquer *payload header*, compõem a parte final do pacote RTP (excepto o *padding*, se for necessário). O tamanho e o formato do *payload* dependem do formato de *payload* e do formato dos parâmetros escolhidos durante o *setup* da sessão.

3.3.1.2. *Real-Time Transport Control Protocol (RTCP)*

O *Real-time Transport Control Protocol* (RTCP), é usado para trocar informações sobre a qualidade da transmissão, a identificação dos participantes, notificações sobre a sessão, informações necessárias para a sincronização e outros tipos de informações. Este protocolo implementa principalmente as funções de *feedback* de qualidade de serviço (QoS) através do envio de determinados pacotes entre o servidor e o receptor onde são analisadas as condições da rede, no que diz respeito a percentagem de pacotes RTP perdidos entre relatórios, a percentagem de pacotes perdidos desde o início da sessão, a diferença de atrasos nos pacotes ou o atraso desde a recepção do último relatório. Estas informações são usadas pela fonte, pelos operadores ou pelos receptores para melhorar o desempenho da rede.

Os pacotes RTCP apresentam o formato a presentado pela Figura 3.9.

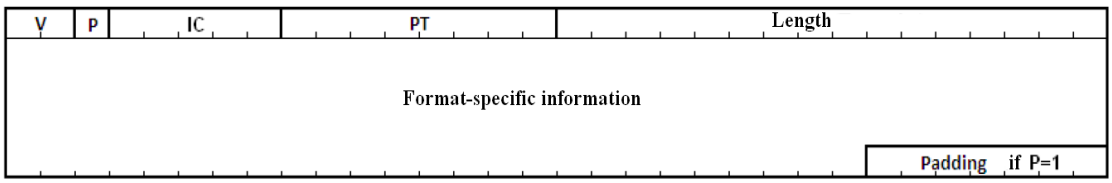

Figura 3.9 – Formato do cabeçalho RTCP

- **V** (2 bits) – Indica a versão do RTP, a versão actual (RFC 1889) é a versão 2.

- **P** (1 bit) – É o campo *padding bit*. Que é usado da mesma forma que no pacote RTP. Se o bit P está configurado com o valor "1", então o pacote inclui um ou mais octetos de *padding* no fim do pacote.
- **IC** (5 bits) – É o *Item Count*. Alguns tipos de pacotes contêm uma lista de itens, talvez em adição a algum tipo de informações específicas. O campo *Item Count* é utilizado por estes tipos de pacotes para indicar o número de itens incluídos no pacote.
- **PT - *Packet Type*** (8 bits) – Identifica o tipo de informação presente no pacote.
- ***Length*** (16 bits) – Indica o tamanho do pacote após o cabeçalho do pacote RTCP. É medido em unidades de 32 bits, pois todos os pacotes RTCP são múltiplos de 32 bits

A seguir ao cabeçalho RTCP, temos os dados do pacote (o formato de que depende do tipo de pacote – ***Format-specific information***) e o *padding* opcional. Existem cinco tipos de pacotes que se encontram definidos pela norma RTP, *Receiver Report* (RR), *Sender Report* (SR), *Source Description* (SDES), *Membership Management* (BYE) e *Application Defined* (APP).

- ***Receiver Report*** – O pacote *receiver report* é identificado com um *packet type* de 201. Este pacote contém o SSRC (*synchronization source*) do participante que está a enviar o relatório (o *reporter* SSRC), seguido de zero ou mais blocos de relatórios.
- ***Sender Report*** – O *payload* contém 24 octetos de blocos de informação do emissor, seguido de zero ou mais blocos de *receiver reports*, exactamente como se este fosse um pacote *receiver report*. Blocos *receiver report* estão presentes quando o emissor é também um receptor. O pacote *sender report* é identificado com um *packet type* de 200.
- ***Source Description Items*** – Contém informação relativa ao participante, e é identificado com um *packet type* de 202
- ***Membership Management* –** Indica quando um participante abandona a sessão ou muda de SSRC. Este pacote é identificado com um *packet type* de 203.

- ***Application Defined*** – Este pacote é identificado com um *packet type* de 204 e permite a extensão de funções específicas da aplicação.

Os pacotes RTCP nunca são transportados individualmente, são sempre agrupados para a transmissão, formando pacotes compostos. Cada pacote composto é encapsulado num pacote da camada mais baixa, normalmente um pacote UDP/IP para o transporte.

Capítulo 4

A Interface *Socket*

O *Berkeley Sockets Interface*, conhecido mundialmente como *sockets*, é a *Application Programming Interface* (API) padrão para a comunicação entre redes. Em 1969 começou o desenvolvimento do UNIX, mais tarde a *University of California, Berkeley* (UCB) desenvolveu a sua própria versão do UNIX, que era conhecido como *Berkeley Software Distribution* (BSD). A ARPA fundou um grupo na UCB, para transportarem o software do TCP/IP para sistemas UNIX, como parte do projecto foi criada uma interface que as aplicações usavam para comunicar. Decidiram usar as *system calls* do UNIX e criar novas para introduzir as funções do TCP/IP, o resultado ficou conhecido como interface *socket* e o sistema como *Berkely UNIX* ou *BSD UNIX*. Em 1982, a UCB lançou duas versões de UNIX, a 4.1BSD e a 4.2BSD que incluíam uma implementação da rede TCP/IP.

Apesar de cada sistema operativo estar livre para definir a sua própria rede API (e a maioria fê-lo), ao longo do tempo algumas destas APIs tornaram-se amplamente suportadas, isto é, foram adaptadas para sistemas operativos diferentes do seu sistema nativo. Isto foi o que aconteceu com a interface *socket*, inicialmente fornecida pela *Berkeley Distribution of Unix*, que agora é suportada em praticamente todos os sistemas operativos mais populares.

O *sockets* API foi projectado para proporcionar o acesso genérico aos serviços de comunicação inter-processos que poderiam ser implementados por qualquer um dos protocolos suportados por uma plataforma - IPX, AppleTalk, TCP/IP, e por aí fora. Embora a interface *socket* possa ser usada para comunicar usando vários protocolos de

rede, iremos apenas abordar a interface *socket* para o protocolo TCP/IP *suite*, dado que é o padrão, de facto, para a comunicação através da Internet.

O *socket* API conforme especificado pelo POSIX.1 é baseado no 4.4BSD *socket interface*. Embora tenham sido feitas pequenas alterações ao longo dos anos, a actual interface *socket* assemelha-se bastante com a interface originalmente introduzida no 4.2BSD, no início dos anos 80 [9, 27-30].

Neste capítulo, é feita uma descrição aprofundada da interface *socket* e apresentada, especificamente, uma implementação em Linux do BSD *socket*, utilizando os protocolos do modelo TCP/IP.

4.1 A Abstracção *Socket*

Um *socket* é uma generalização do mecanismo *file access* do UNIX, que fornece uma comunicação *endpoint*. É uma abstracção através da qual, uma aplicação pode enviar e receber dados, tal como o *file access*, a aplicação pede ao sistema operativo para criar um *socket* quando é necessário. Um *socket* permite que uma aplicação se ligue a uma rede e comunique com as outras aplicações que estão conectadas a essa mesma rede. A informação escrita para o *socket* por uma máquina pode ser lida por outra aplicação numa máquina diferente, e vice-versa.

Existem vários tipos de *sockets*, dependendo do tipo da família de protocolos, bem como dos diferentes protocolos no seio de uma família. Será apenas abordada a família do protocolo TCP/IP. Os principais tipos de *sockets*, no TCP/IP são o *stream sockets* e o *datagram sockets*. O *stream sockets* utiliza o TCP como protocolo *end-to-end* (com IP por baixo) e assim, consegue prestar um serviço de *byte-stream* fiável. Os *datagram sockets* usam o UDP (de novo, com IP por baixo) e assim, prestam um serviço *best-effort datagram*, que as aplicações podem utilizar para enviar mensagens individuais até cerca de 65500 bytes de tamanho. Os *stream* e *datagram sockets* também são suportados por outros protocolos, no entanto, lidamos apenas com *stream sockets* TCP e UDP *sockets datagram*.

Um *socket* que utilize o protocolo TCP/IP é identificado por um endereço de Internet único, um protocolo *end-to-end* (TCP ou UDP), e um número de porto. Quando um *socket* é criado, tem associado um protocolo, mas não tem um endereço Internet ou número de

porto. Até que um *socket* esteja associado a um número de porto, ele não pode receber mensagens a partir de uma aplicação remota.

Note-se que um único *socket* pode ser referenciado por múltiplas aplicações. Cada aplicação tem uma referência (designado de *descriptor*) para um determinado *socket* de modo a poder comunicar através desse mesmo *socket*. Anteriormente, dissemos que um porto identifica uma aplicação num utente, na verdade, um porto identifica um *socket* num utente [27].

4.2 *Socket Descriptors*

Em UNIX, a aplicação necessita de chamar a função *open* para criar um *file descriptor,* que é usado para aceder a um ficheiro, um *file descriptor* é implementado pelo sistema operativo como um *array* de ponteiros para estruturas internas de dados. O sistema mantém separada uma tabela de *file descriptors* para cada processo. Um *socket* é uma abstracção de uma comunicação *endpoint*, da mesma forma, que são usados os *file descriptors* para aceder a um ficheiro, as aplicações usam os *socket descriptors* para aceder a *sockets*. O UNIX guarda os *socket descriptors* na mesma tabela de *descriptors* que os *file descriptors,* mas a aplicação não pode ter ambos os *descriptors* com o mesmo valor.

O BSD UNIX possui a função *socket(),* que as aplicações usam para criar um *socket.* Uma aplicação só utiliza a função *open* para criar *file descriptors.* Tirando isto, os *socket descritors* são implementados tal como os *file descriptors* do sistema Unix, com efeito, muitas das funções que lidam com *file descriptors*, tais como *read()* e *write()*, funcionam num *socket descriptor* [27, 29].

4.2.1 Criação de um *Socket*

Para comunicar usando o TCP ou o UDP, um programa começa por pedir ao sistema operativo para criar uma instância da abstracção *socket*. A função que a concretiza é a *socket(),* e os seus parâmetros especificam o tipo do *socket* requerido pelo programa.

int socket**(int** *domain***, int** *type***, int** *protocol***);**

O primeiro parâmetro determina a família de protocolos do *socket*. Recordo, que a *sockets* API fornece uma interface genérica para um grande número de famílias de protocolos. A

Tabela 4.1 indica as constantes possíveis para este campo. A constante PF_INET especifica um *socket* que usa os protocolos da família do protocolo Internet. Uma vez que não será abordado outra família de protocolos, será sempre utilizada a constante PF_INET para a família do protocolo.

Tabela 4.1 – Famílias de Protocolo

Nome	***Propósito***
PF_UNIX, PF_LOCAL	*Comunicação Local*
PF_INET	*Protocolo Internet IPv4*
PF_INET6	*Protocolo Internet IPv6*
PF_IPX	*IPX – Protocolos Novell*
PF_NETLINK	*Kernel user interface device*
PF_X25	*Protocolo ITU-T X.25 / ISO-8208*
PF_AX25	*Protocolo AX.25(rádio amador)*
PF_ATMPVC	*Access to raw ATM PVCs*
PF_APPLETALK	*Aplletalk*
PF_PACKET	*Interface de pacotes de baixo nível*

O segundo parâmetro especifica o tipo do *socket*. O tipo determina as características da transmissão de dados com o *socket*. A constante SOCK_STREAM especifica um *socket* com uma característica de *byte-stream* fiável, já o SOCK_DGRAM especifica um *socket best-effort datagram*. A Tabela 4.2 apresenta uma breve descrição dos tipos de *sockets.*

O terceiro parâmetro especifica o protocolo *end-to-end* a ser usado. O argumento *protocol* é normalmente nulo, de modo a seleccionar o protocolo por defeito para o *domain* e *socket type* determinados anteriormente. Quando vários protocolos são suportados para o mesmo *domain* e *socket type*, podemos usar o argumento *protocol* para seleccionar um protocolo específico. Para a família do protocolo PF_INET, usamos o TCP (identificado pela constante IPPROTO_TCP) para um *stream socket* e o UDP (identificado pela constante IPPROTO_UDP) para um *socket datagram*. Como actualmente existe apenas uma escolha para *stream* e *datagram socket* na família do protocolo TCP/IP, poderíamos especificar o valor "0", mas para melhor compreensão do código iremos especificar o protocolo que pretendemos utilizar.

Tabela 4.2 – Tipos de *Sockets*

Tipo de Socket	*Descrição*
SOCK_STREAM	*Fornece byte streams sequenciais, fiáveis, bidireccionais, e orientados à conexão. Também pode suportar um mecanismo de transmissão de dados out-of-band*
SOCK_DGRAM	*Suporta datagramas (connectionless, mensagens não fiáveis com um tamanho máximo fixo)*
SOCK_SEQPACKET	*Fornece um caminho de transmissão de dados sequencial, fiável, bidireccional e orientado à conexão para datagramas de tamanho máximo fixo*
SOCK_RAW	*Fornece acesso ao protocolo de redes Raw*
SOCK_RDM	*Fornece uma camada de datagramas fiável que garante a ordenação*
SOCK_PACKET	*Obsoleto e não deve ser usado nos novos programas*

O valor devolvido pela função *socket()* é na verdade um inteiro, um valor não negativo para o sucesso e "-1" para o caso de falhar. O *socket descriptor*, é passado para outras funções API para identificar o *socket* em que a operação vai ser realizada. A função *socket()* é similar à função *open()*, em ambos os casos, recebemos um *file descriptor* que pode ser usado para I/O. Quando já não precisamos de utilizar o *file descriptor*, podemos fecha-lo, com a função *close()*, para terminar o acesso ao ficheiro ou ao *socket* e libertar o *file descriptor* para ser reutilizado.

Devido ao facto do *socket* ter sido desenvolvido, tendo como base o UNIX, foi necessário arranjar uma forma de simular o mecanismo *pipe* do UNIX. O mecanismo *pipe* difere das operações padrão da rede, porque cria ambos os terminais da comunicação simultaneamente. Foi então criada a função *sockerpair()* para fornecer esta funcionalidade, a função tem a seguinte forma:

```
int socketpair(int domain, int type, int protocol, int sv[2]);
```

A função *socketpair()* tem mais um argumento do que a função *socket()*, *sv*[2]. O argumento adicional dá o endereço de um *array* de inteiros de dois elementos. O s*ocketpair* cria dois *sockets* simultaneamente e coloca os dois *socket descripitors* nos dois elementos de *sv*. A função *socketpair()* não é significativa quando aplicada à família do protocolo TCP/IP (foi incluída, apenas para tornar a nossa descrição da interface mais completa) [2, 27-29].

4.2.2 Destruição de um *Socket*

Quando uma aplicação acaba de usar o *socket*, chama a função *close()*, onde o argumento *socket* especifica o *descriptor* do *socket* a fechar.

```
int close(int socket);
```

A função *close()* informa o protocolo subjacente para iniciar a acção necessária para encerrar as comunicações e libertar quaisquer recursos associados com o *socket*, a função *close()* devolve "0" ao sucesso ou "-1" no caso de falhar. Uma vez que a função *close()* foi chamada, já não é possível enviar ou receber dados através do *socket*. Internamente, a chamada da função *close()* decrementa a contagem de referências do *socket* e destrói o *socket* se a contagem chegar a zero.

A comunicação sobre um *socket* é bidireccional. Podemos desactivar o I/O num *socket* com a função *shutdown()*, que tem outras funcionalidades que a função *close()* não possui.

```
int shutdown(int socket, int how);
```

O argumento *socket* especifica o *descriptor* do *socket* a fechar, tal como na função *close()*. Se o argumento *how* for SHUT_RD, então a leitura do *socket* está desactivada, se for SHUT_WR, então não podemos usar o *socket* para a transmissão de dados. Podemos usar SHUT_RDWR para desactivar a transmissão e recepção dos dados.

Há várias razões para usar a função *shutdown()* em detrimento da função *close()*. Em primeiro lugar, a função *close()* irá libertar o *endpoint* da rede apenas quando a última referência activa for fechada. A função *shutdown()* permite desactivar um *socket* independentemente do número de *file descriptors* activos que estão a referenciar o *socket*. Em segundo lugar, às vezes é conveniente fechar um *socket* apenas numa direcção. Por exemplo, quando terminámos a transmissão mas não a recepção de dados, é possível fechar o *socket* do lado da transmissão, permitindo-nos utilizar o *socket* para receber os dados enviados para nós pelo processo [2, 27, 28, 31].

4.3 Endereçamento

Na secção anterior, mostrámos como criar e destruir um *socket*. Antes de começarmos a utilizar o *socket*, temos de identificar o processo com que queremos comunicar. Identificar o processo tem duas componentes: o endereço de rede da máquina ajuda-nos a identificar o computador na rede, com que queremos comunicar, e o serviço ajuda-nos a identificar o processo específico no computador.

As aplicações que usam *sockets* precisam de ser capazes de especificar os endereços Internet e os portos ao *kernel*. Por exemplo, um cliente deve especificar o endereço do servidor com o qual necessita comunicar. Além disso, a camada *Sockets* às vezes tem de passar endereços para a aplicação.

4.3.1 Ordenação de Bytes

Quando a comunicação entre processos ocorre no mesmo computador, não temos que nos preocupar com a ordenação de bytes. A ordem do byte é uma característica da arquitectura do processador, ditando o modo como os bytes são ordenados dentro de maiores tipos de dados, tais como inteiros. A Figura 4.1 mostra como os bytes dentro de um número inteiro de 32 bits são numerados.

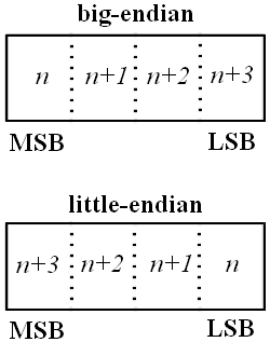

Figura 4.1 – Ordem do Byte

Se a arquitectura do processador suportar uma ordenação *big-endian*, então o byte mais alto do endereço ocorre no byte menos significativo (LSB - *least significant byte*). A ordenação no *little-endian* é o oposto, o byte mais significativo (MSB - *most significant byte*) é que contém o byte mais alto do endereço.

O protocolo TCP/IP *suite* usa uma ordenação *big-endian*. Com o TCP/IP, os endereços são apresentados com a ordem de byte da rede, por isso, às vezes as aplicações têm necessidade de os traduzir entre a ordem de byte usado pelo processador e a ordem de byte da rede. Isso é comum, por exemplo quando se imprime um endereço numa forma legível para humanos. O *socket* API oferece quatro funções que convertem entre a ordem de byte usada pela máquina local e a ordem de byte usada pela rede. De modo a se fazer com que os programas sejam portáveis, estas funções devem ser usadas sempre que haja troca de dados entre a máquina local e a rede, e vice-versa.

```
long int htonl(long int hostlong);
short int htons(short int hostshort);
long int ntohl(long int netlong);
short int ntohs(short int netshort);
```

A função *htonl()* (*host to network long*) devolve o resultado da conversão do valor de quatro bytes (*long*) da ordem de byte usada pela máquina local para a ordem de byte usada pela rede. (Se a ordem de byte usada pela máquina local é a mesma ordem de byte usada pela rede, a função de conversão é uma *no-op*) Da mesma forma, *htons()* (*host to network short*) converte um valor de dois bytes (*short*) para a ordem de byte usada pela rede e devolve o resultado, *ntohl()* (*network to host long*) e *ntohs()* (*network to host short*) convertem valores da ordem de byte usada pela rede para a ordem de byte usada pela máquina local [2, 13, 27, 28].

4.3.2 Funções de Manipulação de Bytes

Existem dois grupos de funções que operam em campos multibyte, sem interpretarem os dados, e sem assumir que os dados são uma *string* C terminada com zero. Precisamos deste tipo de funções quando se lida com estruturas de endereços de *sockets*, uma vez, que precisamos de manipular campos, tais como, endereços IP, que podem conter bytes de "0", mas não são *strings* de caracteres.

O primeiro grupo de funções, cujos nomes começam com *b* (para byte), é do 4.2BSD e ainda são fornecidas por praticamente qualquer sistema que suporte as funções do *socket*. O segundo grupo de funções, cujos nomes começam com *mem* (para memória), é do padrão ANSI C e são fornecidos com qualquer sistema que suporte uma biblioteca ANSI C.

Só este segundo grupo será abordado em maior pormenor, já de seguida, uma vez que foi o utilizado para a realização do trabalho. As funções do ANSI C são as seguintes:

void ***memset(void ****s***, int** *c***, size_t** *n***);**
void ***memcpy(void ****dest***, const void ****src***, size_t** *n***);**
int memcmp**(const void ****s1***, const void ****s2***, size_t** *n***);**

A função *memset()* preenche os primeiros *n* bytes da área de memória apontado pela constante *s* com o byte *c*. A *memcpy()* copia *n* bytes da área de memória *src* para a área de memória *dest*. A função *memcmp()* compara os primeiros *n* bytes das áreas de memória *s1* e *s2*, devolvendo um inteiro menor que, igual a, ou maior que zero se *s1* for, respectivamente, inferior, igual, ou maior do que *s2* [13].

4.3.3 Formatos de Endereços

Um endereço identifica um *socket endpoint* num domínio da comunicação particular. O formato do endereço é específico do domínio, para que os endereços com diferentes formatos possam ser passados para as funções do *socket*, os endereços são expressos através de uma estrutura genérica *sockaddr*:

```
struct sockaddr
{
   unsigned short sa_family;      /* Família de endereços */
   char sa_data[14];              /* Informação do endereço da família */
};
```

A primeira parte desta estrutura de endereço define a família de endereços - o espaço ao qual o endereço pertence. Para o nosso propósito, vamos usar sempre a constante AF_INET, que especifica a família de endereços da Internet. A segunda parte são bits cuja forma exacta depende da família de endereços, esta é uma forma típica de lidar com a heterogeneidade nos sistemas operativos e nas redes. Os criadores do *sockets* API quiseram proporcionar o máximo de flexibilidade, pois estes vislumbravam a possibilidade de que uma família de protocolos viesse a ter diferentes esquemas de endereçamento. Por isso, permitiram que a família de protocolos e a família de endereços fossem independentes uma da outra. Na prática, existe apenas uma família de endereços por família de protocolos, na realidade, o AF_xxx e PF_xxx têm historicamente, sido intercambiáveis (por exemplo, o

AF_INET tem o mesmo valor que PF_INET). Quando nos referimos a famílias de protocolos e endereços, usaremos PF_INET e AF_INET, respectivamente.

A forma particular da estrutura *sockaddr* que é usada para os endereços do *socket* TCP/IP no IPv4 é a estrutura *sockaddr_in*.

```
struct in_addr
{
   in_addr_t s_addr;              /* Endereço Internet IPv4 */
};

struct sockaddr_in
{
   unsigned short sin_family;     /* Família de endereços */
   unsigned short sin_port;       /* Número de porto */
   struct in_addr sin_addr;       /* Endereço Internet IPv4 */
   unsigned char sin_zero[8];     /* Filler (Não é usado) */
};
```

A estrutura *sockaddr_in* tem campos para o número de porto e o endereço de Internet, para além da família de endereços. A *sockaddr_in* é apenas uma outra visão dos dados na estrutura *sockaddr*, adaptada aos *sockets* para usar os protocolos Internet. O *sin_zero* é um campo de enchimento que deve ser preenchido com valores de zero.

O endereço de *socket* do IPv6 é representado pela estrutura *sockaddr_in6*.

```
struct in6_addr
{
   short s6_addr[16];                  /* Endereço Internet IPv6 */
};

struct sockaddr_in6
{
   short sin6_family;                  /* Família de endereços */
   unsigned short sin6_port;           /* Número de porto */
   unsigned long sin6_flowinfo;        /* Classe de tráfego e Info de fluxo */
   struct in6_addr sin6_addr;          /* Endereço Internet IPv6 */
   unsigned long sin6_scope_id;        /* Conjunto de interfaces para scope */
};
```

É de notar que, apesar de as estruturas *sockaddr_in* e *sockaddr_in6* serem bastante diferentes, ambas são passadas para rotinas do *socket* e expressas através de uma estrutura *sockaddr* [13, 27, 28].

4.3.4 Manipulação de Endereços IP

Por vezes é necessário imprimir um endereço num formato que seja compreensível por uma pessoa em vez de um computador, e vice-versa. O BSD incluiu funções que fazem esta conversão, *inet_ntoa, inet_addr, inet_network* e *inet_ntoa*. Estas funções, porém, só funcionam com endereços IPv4, existem novas funções que funcionam quer com IPv4, quer com IPv6, que serão apresentadas mais à frente. Muitas vezes, programas que manipulam os endereços IP têm que combinar um endereço de rede com o endereço local do utente, bem como o inverso, para tal existem as funções *inet_makeaddr()*, *inet_netof()* e *inet_lnaof()*. Uma breve descrição de cada uma das funções é feita de seguida.

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

A função *inet_aton()* converte o endereço Internet do utente *cp* da notação decimal para dados binários que armazena na estrutura apontada por *inp*. Devolve não-zero se o endereço é válido, e zero caso contrário.

A função *inet_addr()* converte o endereço Internet do utente *cp* da notação decimal para dados binários na ordenação de bytes usada pela rede. Se a entrada é inválida, INADDR_NONE (normalmente "-1") é devolvido. Esta é uma interface obsoleta para *inet_aton()*, imediatamente acima descrita, pois "-1" é um endereço válido (255.255.255.255), e *inet_aton()* fornece uma forma mais limpa para indicar um erro.

A função *inet_network()* extrai o número da rede na ordem de bytes do utente a partir do endereço *cp* que está na notação decimal. Se a entrada é inválida, "-1" é devolvido.

A função *inet_ntoa()* converte o endereço Internet do utente *in* dado na ordem de bytes da rede para uma *string* na notação decimal.

A função *inet_makeaddr()* cria um endereço Internet do utente na ordem de bytes da rede ao combinar o número da rede *net* com a rede local *host* na rede *net*, ambos na ordem de bytes do utente.

A função *inet_lnaof()* devolve o endereço local do utente, que faz parte do endereço Internet *in.* O endereço local do utente é devolvido na ordem de bytes do utente.

A função *inet_netof()* devolve o número da rede, que faz parte do endereço Internet *in.* O número da rede é devolvido na ordem de bytes do utente.

A estrutura *in_addr* apresentada na secção anterior é utilizada nas funções *inet_ntoa(), inet_makeaddr(), inet_lnoaf()* e *inet_netof()*.

As novas funções que oferecem a mesma funcionalidade de conversão de endereços IP e que funcionam com o IPv4 e IPv6 são a *inet_ntop()* e *inet_pton().*

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

A função *inet_ntop()* converte a estrutura do endereço da rede *src* da família de endereços *af* numa *string* de caracteres, a qual é copiada para o *buffer dst*, que tem um tamanho de *cnt* bytes. A função *inet_pton()* converte a *string* de caracteres *src* numa estrutura do endereço da rede da família de endereços *af* , depois copia a estrutura do endereço da rede para o *dst*.

As funções *inet_ntop()* e *inet_pton()* são uma extensão das função *inet_ntoa()* e *inet_addr(),* respectivamente, para suportarem múltiplas famílias de endereços, as famílias de endereços suportadas são a AF_INET e a AF_INET6 [2, 13, 28].

4.3.5 Associar Endereços com *Sockets*

O endereço associado a um *socket* do cliente, tem pouco interesse, e podemos deixar o sistema escolher um endereço. Para um servidor, porém, temos de associar um endereço bem conhecido com o *socket* do servidor, no qual os pedidos do cliente irão chegar. Os clientes precisam de uma maneira de descobrir o endereço a utilizar para contactar com o servidor.

A função *bind()* atribui o endereço local de Internet e o porto para um *socket*. O número de porto deve ser especificado. A chamada irá falhar (EADDRINUSE) se o número de porto especificado é o porto local de um outro *socket* e se a opção SO_REUSEADDR não foi definida.

int bind**(int** *socket***, struct sockaddr ***my_addr***, unsigned int** *addrlen***);**

Há, no entanto, várias restrições em relação ao endereço que podemos usar:

- O endereço especificado deve ser válido para a máquina na qual o processo está a ser executado, não podemos especificar um endereço pertencente a uma outra máquina.
- O endereço deve corresponder ao formato suportado pela família de endereços que foi usada para criar o *socket*.
- O número de porto, no endereço não pode ser inferior a 1024, a menos que o processo tenha um privilégio adequado (isto é, se for o *superuser*).
- Normalmente, apenas um *socket endpoint* pode estar associado a um determinado endereço, apesar de alguns protocolos permitirem duplicar *bindings*.

No domínio Internet, se especificarmos o endereço IP INADDR_ANY, o *socket endpoint* será associado a todas as interfaces da rede do sistema. Isto significa que podemos receber pacotes a partir de qualquer uma das placas de interface de rede instaladas no sistema.

Podemos usar a função *getsockname()* para descobrir o endereço associado a um *socket*.

int getsockname**(int** *socket***, struct sockaddr ***name***, socklen_t ***namelen***);**

A função devolve o *name* do *socket* especificado. O parâmetro *namelen* deve ser inicializado e indicar a quantidade de espaço necessário para o *name*. No regresso contém o tamanho real do *name* (em bytes).

Se o socket está ligado a um *peer*, nós podemos descobrir o endereço do *peer* ao chamar a função *getpeername()*.

int getpeername**(int** *socket***, struct sockaddr ***name***, socklen_t ***namelen***);**

À excepção de devolver o endereço do *peer*, a função *getpeername()* é idêntica à função *getsockname()* [27, 28].

4.4 Estabelecimento de uma Conexão

Um *socket* TCP necessita de estar conectado com outro *socket* antes de poder começar a transmissão de dados. Ao usar um serviço de rede orientado à conexão (SOCK_STREAM ou SOCK_SEQPACKET), temos de criar uma ligação entre o *socket* do processo que pediu o serviço (o cliente) e o *socket* do processo que fornece o serviço (o servidor). Usamos a função *connect()* para criar uma ligação.

int connect**(int** *socket***, const struct sockaddr ***_serv_addr_**, socklen_t** *addrlen***);**

O argumento *socket* é o *descriptor* criado pela função *socket().* O *serv_addr* é declarado para ser um ponteiro para uma estrutura *sockaddr,* dado os *sockets* API serem genéricos, para os nossos propósitos, este será sempre um ponteiro para um *sockaddr_in* contendo o endereço Internet e o porto do servidor. O *addrlen* especifica o tamanho da estrutura do endereço e é invariavelmente definido como *sizeof(struct sockaddr_in).*

Para o cliente entrar em contacto com o servidor, o *socket* do servidor deve ter um endereço e porto local atribuído, a função *bind()* é que faz esta atribuição. Apesar do cliente ter de fornecer o endereço do servidor à função *connect(),* o servidor tem de especificar seu próprio endereço à função *bind().*

Depois do *socket* ter um endereço, ou pelo menos um porto, o servidor pode começar a aceitar os pedidos para se conectar ao chamar a função *listen().*

int listen**(int** *socket***, int** *queueLimit***);**

O parâmetro *queueLimit* especifica um limite superior sobre o número de conexões recebidas que podem estar à espera. Se um pedido de conexão chega e a fila está cheia, o cliente pode receber um erro com uma indicação de ECONNREFUSED, ou, se o protocolo subjacente suportar retransmissão, o pedido pode ser ignorado, para que novas tentativas possam ser bem sucedidas. A função *listen()* devolve "0" no sucesso e "-1" em caso de falhar.

Como vimos, um servidor utiliza as funções *socket(), bind(),* e *listen()* para criar um *socket,* associa-lo a um porto, e especificar o tamanho da fila dos pedidos de conexão. De notar, que apesar de o *socket* estar associado a um porto, este não está ligado a nenhum endereço.

Uma vez que um *socket* foi criado, o servidor tem de esperar por uma ligação. Para fazer isso, este usa a função *accept()*.

int accept**(int** *socket***, struct sockaddr ***_clientAddress_**, socklen_t ***_addrlen_**);**

O *file descriptor* devolvido pela função *accept()* é um *socket descriptor* que está ligado ao cliente que solicitou a conexão. Este novo *socket descriptor* tem o mesmo tipo de família de endereços que o *socket* original. O *socket* original não está associado com a conexão criada, mas continua disponível para receber mais pedidos de conexão.

Se a fila estiver vazia, a função *accept()* bloqueia até que um pedido de conexão chegue. Quando bem sucedido, o *accept()* preenche a estrutura *sockaddr*, apontada pelo *clientAddress*, com o endereço do cliente do outro lado da ligação, *addrlen* especifica o tamanho máximo da estrutura *clientAddress* e contém o número de bytes, efectivamente, utilizado para o endereço no regresso [2, 13, 27-29].

4.5 Transferência de Dados

Uma vez que um *socket* é representado como um *file descriptor*, podemos utilizar as funções *read()* e *write()* para comunicar com um *socket*, desde que este esteja conectado. Usar estas funções com *socket descriptors* é importante, porque significa que podemos passar *socket descriptors* para funções que foram originalmente projectadas para trabalhar com ficheiros locais.

Embora possamos trocar dados usando as funções *read()* e *write()*, isso é basicamente tudo o que podemos fazer com estas duas funções. Se quisermos especificar opções, como receber pacotes de vários clientes, precisamos de utilizar uma das outras seis funções concebidas para a transferência de dados.

4.5.1 Envio de Dados

Depois de uma aplicação ter estabelecido uma ligação com o *socket*, pode usa-lo para transmitir dados. Existem cinco funções possíveis que podemos escolher: *write(), writev(), send(), sendto()* e *sendmsg()*. As funções *send(), write()* e *writev()* apenas funcionam com *sockets* conectados, porque a chamada não permite especificar um endereço destino. As diferenças entre estas três funções são mínimas.

Já as funções *sendto()* e *sendmsg()* permitem enviar uma mensagem através de um socket não conectado, porque precisam que o endereço destino seja especificado.

A função *write()* é composta por três argumentos.

ssize_t write**(int** *socket*, **const void*** *buffer*, **size_t** *length***);**

O argumento *socket* contém um *socket descriptor* inteiro (*write* também pode ser utilizado com outros tipos de *descriptors*). O argumento *buffer* contém o endereço dos dados a serem enviados, e o argumento *length* especifica o número de bytes a serem enviados. A função *write()* bloqueia até que os dados possam ser transmitidos.

A função *writev()* funciona de forma igual à função *write()*, mas deixa-nos ler um ou mais *buffers* com uma única chamada da função. Esta operação é designada de *gather write* (desde que múltiplos *buffers* estejam reunidos para uma única operação de saída). A função apresenta a seguinte forma:

ssize_t writev **(int** *socket*, **const struct iovec** **iov*, **int** *iovcnt***);**

O argumento *iov* aponta para uma estrutura do tipo *iovec* que contém uma sequência de ponteiros para os blocos de bytes que formam a mensagem., em que cada um é acompanhado pelo seu tamanho, como é possível ver na estrutura apresentada a seguir. O argumento *iovcnt* especifica o número de entradas na estrutura *iov*.

```
struct iovec {
   void *iov_base;      /* Endereço de partida do buffer */
   size_t iov_len;      /* Tamanho do buffer */
};
```

A função *send()* tem a seguinte forma:

int send**(int** *socket***, const void *****msg*, **unsigned int** *msgLength*, ***int*** *flags***);**

O argumento *socket* é o *descriptor* para o *socket* conectado através do qual os dados vão ser enviados. O argumento *msg* aponta para a mensagem a ser enviada, e o *msgLength* é o tamanho, em bytes, da mensagem. O comportamento normal da função *send()* é bloquear até que todos os dados sejam enviados. O argumento *flags* fornece uma maneira de mudar o comportamento padrão do *socket*, permite diversas características especiais, tais como, o

acesso a dados *out-of-band.* Definir *flags* para "0" especifica o comportamento padrão, a função *send()* devolve o número de bytes enviados ou "-1" no caso de falhar.

A função *sendto()*, que tem o endereço destino como um argumento, tem a seguinte forma:

int sendto**(int** *socket,* **const void** **msg,* **unsigned *int*** *msgLength,* ***int*** *flags,*
struct sockaddr **destAddr,* **unsigned *int*** *addrLen***);**

Os quatro primeiros argumentos são exactamente os mesmos que os utilizados com a função *send()*. Os dois últimos argumentos especificam o endereço destino e o tamanho desse mesmo endereço. O argumento *destAddr* especifica o endereço destino utilizando a estrutura *sockaddr_in*, e o *addrLen* será definido como *sizeof (struct sockaddr_in)*.

Um programador pode optar por utilizar a função *sendmsg()* nos casos em que a longa lista de argumentos necessários para o *sendto()* faz com que o programa seja ineficaz ou difícil de ler. A função *sendmsg()* apresenta a seguinte forma:

int sendmsg**(int** *socket*, **struct msghdr** **msg*, **int** *flags***);**

Onde o argumento *msg* aponta para a estrutura *msghdr* , apresentada a seguir

```
struct msghdr {
    void *msg_name;              /* Endereço opcional */
    socklen_t msg_namelen;       /* Tamanho do endereço em bytes */
    struct iovec  *msg_iov;      /* array de buffers I/O */
    int msg_iovlen;              /* Número de elementos no array */
    void *msg_control;           /* Dados adicionais */
    socklen_t msg_controllen;    /* Número de bytes adicionais */
    int msg_flags;               /* flags para a mensagem recebida */
};
```

A estrutura contém informações sobre a mensagem a ser enviada, o seu tamanho, o endereço destino e o tamanho do endereço. Esta função é especialmente útil, porque existe uma operação de recepção correspondente (descrito na secção seguinte) que produz a estrutura da mensagem exactamente no mesmo formato [2, 13, 27, 28].

4.5.2 Recepção de Dados

Como acontece com as cinco diferentes operações de saída, o *socket* API oferece cinco funções que um processo pode utilizar para receber dados através de um *socket*: *read(),*

readv(), recv(), recvfrom(), e *recvmsg()*. A operação de entrada convencional, *read()*, só pode ser utilizada quando o socket está conectado. Esta tem a seguinte forma:

ssize_t read **(int** *descriptor***, void*** *buffer*, **size_t** *length***);**

Onde o argumento *descriptor* é o *descriptor* do *socket* ou o *file descriptor* para a leitura dos dados, o *buffer* especifica o endereço da memória para armazenamento dos dados, e o *length* especifica o número máximo de bytes a ler.

Uma forma alternativa, é a função *readv()*, que permite escrever num ou mais *buffers* com uma única chamada da função. Esta operação é designada de *scatter read* (desde que a operação de entrada esteja dispersa em múltiplos *buffers* de aplicações). A função apresenta a seguinte forma:

ssize_t readv**(int** *descriptor*, **const struct iovec** **iov*, **int** *iovcnt***);**

O argumento *iov* aponta para uma estrutura do tipo *iovec* (apresentada anteriormente) que contém uma sequência de ponteiros para os blocos de memória, nos quais os dados a receber devem ser armazenados. O argumento *iovcnt* especifica o número de entradas na estrutura *iov*.

Para além das operações de entrada convencionais, existem três funções adicionais para a recepção de mensagens. A função *recv()* recebe dados de um *socket* conectado. Ele tem a seguinte forma:

int recv**(int** *socket*, **const void** **rcvBuffer*, **unsigned int** *bufferLength*, **int** *flags***);**

O argumento *socket* especifica um *socket descriptor* a partir do qual os dados devem ser recebidos. O argumento *rcvBuffer* especifica o endereço na memória onde a mensagem deverá ser colocada, e o argumento *bufferLength* especifica o tamanho do *rcvBuffer*.

Por último, o argumento *flags* fornece uma maneira de mudar o comportamento padrão do *socket*, permite diversas características especiais, tais como o acesso a dados *out-of-band*. Definir *flags* para "0" especifica o comportamento padrão, a função *recv()* devolve o número de bytes recebidos ou "-1" no caso de falhar.

A função *recvfrom()* permite especificar a entrada de um *socket* não conectado. Inclui argumentos adicionais que permitem especificar onde gravar o endereço do emissor. A sua forma é:

int recvfrom(int *socket***, void ****msg***, unsigned int** *msgLenth***, int** *flags***,**
struct sockaddr ****srcAddr*, unsigned int ****addrlen***);**

Os dois argumentos adicionais, *srcAddr* e *addrlen*, são ponteiros para uma estrutura de endereços *socket* e para um número inteiro, respectivamente. O sistema operativo utiliza *srcAddr* para gravar o endereço do emissor da mensagem e usa *addrlen* para gravar o tamanho do endereço do emissor.

A última função utilizada é, a *recvmsg()*, que é análoga à função *sendmsg()*. A função *recvmsg()* funciona como *recvfrom()*, mas exige menos argumentos. A sua forma é a seguinte:

int recvmsg**(int** *socket*, **struct msghdr** **msg*, **int** *flags***);**

Onde o argumento *msg* contém o endereço da estrutura que guarda o endereço de uma mensagem recebida, bem como locais para o endereço do emissor. A estrutura produzida pela *recvmsg()* é exactamente a mesma que a estrutura utilizada pelo *sendmsg*, fazendo com que funcionem bem como um par [2, 13, 27, 28].

4.6 Opções dos *Sockets*

Surgiu a necessidade de um mecanismo que permitisse às aplicações controlarem o *socket*. Por exemplo, quando se utilizam protocolos que usam *timeout* e retransmissão, a aplicação pode querer obter ou definir os parâmetros do *timeout*. Também pode querer controlar a atribuição do espaço do *buffer*, determinar se o *socket* permite transmissão *broadcast*, ou controlar o processamento de dados *out-of-band*. Em vez de se adicionar novas funções para cada nova operação de controlo, os criadores decidiram construir um único mecanismo. O mecanismo tem duas operações: *getsockopt()* e *setsockopt()*.

A função *getsockopt()* permite que a aplicação possa pedir informações sobre o *socket*. A função tem a seguinte forma:

```
int getsockopt(int socket, int level, int optName, void *optVal,
               unsigned int *optLen);
```

O argumento *socket* é um *socket descriptor* que especifica o *socket* para o qual são necessárias as informações. O argumento *level* identifica se a operação se aplica ao *socket* ou aos protocolos subjacentes que estão a ser utilizados. Algumas opções são independentes do protocolo e são, portanto, manipuladas pela própria *Socket Layer* (SOL_SOCKET), algumas são específicas para o protocolo de transporte (IPPROTO_TCP), e algumas são manipuladas através do protocolo Internet (IPPROTO_IP). A opção em si é especificada pelo número inteiro *optName*. O argumento *optVal* é um ponteiro para um *buffer* que fornece o endereço do *buffer* no qual o sistema coloca o valor solicitado. O *optLen* especifica o tamanho do *buffer*.

A função *setsockopt()* permite que uma aplicação defina uma opção do *socket* utilizando o conjunto de valores obtidos com o *getsockopt()*. O *setsockopt()* tem a seguinte forma:

```
int setsockopt (int socket, int level, int optName, const void *optVal,
                unsigned int *optLen);
```

Onde os argumentos são como os do *getsockopt()*, excepto que o argumento *optLen* contém o tamanho da opção que está a ser passada para o sistema. Naturalmente, nem todas as opções se aplicam a todos os *sockets*. A semântica dos pedidos individuais depende do estado actual do *socket* e dos protocolos que estão a ser usados no momento.

A Tabela 4.3 mostra algumas das opções mais usadas em cada nível, incluindo uma descrição e o tipo de dados do *buffer* apontado pelo *optVal*.

4.7 Programação dos *Sockets*

Nas secções anteriores foram abordadas as características básicas para a programação de *sockets*. O próximo passo é integrar estas ideias em vários modelos de programação, como multitarefa, sinalização, e de *broadcast*. Demonstraremos estes princípios no contexto da programação UNIX, no entanto, os sistemas operativos mais modernos suportam características semelhantes (por exemplo, processos e *threads*).

Tabela 4.3 – Opções dos *Sockets*

Nome da Opção	*Tipo*	*Valor*	*Descrição*
SOL_SOCKET Level			
SO_BROADCAST	***int***	*0,1*	*Broadcast autorizado*
SO_KEEPALIVE	***int***	*0,1*	*Menssagens keepalive activas*
SO_LINGER	***linger{}***	*Time*	*Tempo para atrasar o close() devolver à espera de confirmação*
SO_RCVBUF	***int***	*Bytes*	*Bytes no buffer do socket receive*
SO_RCVLOWAT	***int***	*Bytes*	*Número mínimo de bytes que faz com que o recv() devolva*
SO_REUSEADDR	***int***	*0,1*	*Binding permitido (sob certas regras) com um endereço ou porto já em uso*
SO_SNDLOWAT	***int***	*Bytes*	*Mínimo de bytes a enviar*
SO_SNDBUF	***int***	*Bytes*	*Bytes no buffer do socket send*
IPPROTO_TCP Level			
TCP_MAX	***int***	*Segundos*	*Segundos entre mensagens keepalive*
TCP_NODELAY	***int***	*0,1*	*Não permite atraso na união dos dados (algoritmo Nagle´s)*
IPPROTO_IP Level			
IP_TTL	***int***	*0-255*	*Time-to-live para pacotes IP unicast*
IP_MULTICAST_TTL	***unsigned char***	*0-255*	*Time-to-live para pacotes IP multicast*
IP_MULTICAST_LOOP	***int***	*0,1*	*Permite o sockect multicast receber pacotes que enviou*
IP_ADD_MEMBERSHIP	***ip_mreq{}***	*endereço de grupo*	*Permite a recepção de pacotes endereçados para um grupo multicast específico*
IP_DROP_MEMBERSHIP	***ip_mreq{}***	*endereço de grupo*	*Não permite a recepção de pacotes endereçados para um grupo multicast específico*

4.7.1 Sinais

Os sinais fornecem um mecanismo para notificar os programas de que determinados eventos ocorreram. Alguns dos eventos (e, por conseguinte, a notificação) podem ocorrer assincronamente, o que significa que a notificação é entregue ao programa, independentemente de onde, no código, o programa está a ser executado. Quando um sinal é emitido para um programa em execução, uma das quatro situações pode acontecer:

- O sinal é ignorado. O processo nunca toma conhecimento de que o sinal foi entregue.
- O programa é encerrado pelo sistema operativo.
- Uma rotina *signal-handling*, especificada pelo programa, é executada. Esta execução ocorre numa *thread* de controlo diferente da *thread* principal do programa, de modo a que o programa não seja necessária e imediatamente informado do mesmo.
- O sinal é bloqueado, ou seja, impedido de ter qualquer efeito até que o programa tome medidas para permitir a sua entrega. Cada processo tem uma máscara, indicando quais são os sinais que estão actualmente bloqueados nesse mesmo processo. (Na verdade, cada *thread* num programa pode ter a sua própria máscara de sinal).

O UNIX tem dezenas de diferentes sinais, cada um indica a ocorrência de diferentes tipos de eventos. Cada sinal tem definido um comportamento por defeito, que é uma das duas primeiras possibilidades listadas acima. Iremos apenas abordar alguns sinais que são frequentemente encontrados no contexto da programação de *sockets*. Por isso, apresentamos os conceitos básicos para lidar com sinais, incidindo sobre os cinco apresentados na Tabela 4.4.

Tabela 4.4 – Sinais usados nos *Sockets*

Sinal	***Evento desencadeado***	***Comportamento por defeito***
SIGALRM	*Expiração de um alarme de timer*	*Terminação*
SIGCHLD	*Sai do processo child*	*Ignora*
SIGINT	*Caracter de interrupção (Control-C)*	*Terminação*
SIGIO	*Socket pronto para I/O*	*Ignora*
SIGPIPE	*Tentativa de escrever num socket fechado*	*Terminação*

Um programa pode alterar o comportamento definido por defeito para um determinado sinal usando a função *sigaction()*.

```
int sigaction(int whichSignal, const struct sigaction * newAction,
              struct sigaction * oldAction);
```

A função *sigaction()* devolve "0", se for bem sucedida e "-1" em caso contrário. Cada sinal é identificado por um número inteiro constante, o *whichSignal* especifica o sinal para o qual pretendemos alterar o comportamento. O parâmetro *newAction* aponta para a estrutura *sigaction*, que define o novo comportamento para o dado sinal, se o ponteiro *oldAction* for não nulo, a estrutura *sigaction* que descreve o comportamento anterior para o dado sinal é copiada para este.

```
struct sigaction {
   void (*sa_handler)(int);    /* Handler do sinal */
   sigset_t sa_mask;           /* Sinais a bloquear quando o handler está em execução */
   int sa_flags;               /* Flags para modificarem o comportamento por defeito */
};
```

O campo *sa_handler* controla qual das três primeiras possibilidades ocorre quando um sinal é entregue. Se o seu valor for a constante SIG_IGN, o sinal será ignorado, se tiver o valor SIG_DFL, será usado o comportamento por defeito. Se o seu valor for o endereço de uma função, essa função será invocada com um parâmetro que indica o sinal que foi entregue.

O campo *sa_mask* especifica que sinais devem ser bloqueados enquanto se lida com o *whichSignal*, isto só faz sentido quando *sa_handler* não é SIG_IGN ou SIG_DFL. Por norma o *whichSignal* é sempre bloqueado, independentemente de ser reflectido no *sa_mask*. O campo *sa_flags* controla mais alguns pormenores sobre a forma como é tratado o *whichSignal*. O campo *sa_mask* é implementado como um conjunto de *flags* booleanas, uma para cada tipo de sinal. Este conjunto de *flags* pode ser manipulado com as quatro funções seguintes.

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int whichSignal);
int sigdelset(sigset_t *set, int whichSignal);
```

As funções *sigfillset()* e *sigemptyset()* fazem a activação e desactivação de todas as *flags* no conjunto dado. As funções *sigaddset()* e *sigdelset()* fazem a activação e desactivação de *flags* individuais, especificadas pelo número do sinal, dado pelo argumento *set*. Todas as quatro funções devolvem "0" para o sucesso e "-1" no caso de falhar.

Então, o que acontece quando um sinal, que de outra forma seria entregue, é bloqueado? A entrega é adiada até ser possível lidar com o sinal. A esse sinal é atribuído o estado pendente. Um sinal ou está pendente, ou não.

Um dos aspectos mais importantes de sinais diz respeito à interface *socket*. Se um sinal é entregue enquanto o programa está bloqueado com um *socket* (como um *recv()* ou *connect()*), e o *handler* para esse sinal já foi especificado, logo que o *handler* termine, o *socket* vai devolver "-1" com *errno* a ser atribuído com EINTR. Por isso, o programa deve estar preparado para lidar com esta situação errónea.

Aqui nós vamos descrever brevemente a semântica do SIGPIPE. Considere o seguinte cenário: um servidor (ou cliente) tem um *socket* TCP conectado, e no outro extremo inesperadamente a ligação é desligada. Quando tenta enviar o *socket*, nesse momento, acontece uma de duas coisas: ou uma mensagem de erro ("-1") é devolvida, ou SIGPIPE é entregue (o SIGPIPE é entregue com sincronismo). Este facto é especialmente significativo para os servidores, pois o comportamento por defeito para o SIGPIPE é a de encerrar o programa. Assim, se os servidores que não alterem esse tipo de comportamento podem ser encerrados por clientes mal intencionados. Os servidores devem sempre tratar o SIGPIPE, para que possam detectar o desaparecimento do cliente e reclamar eventuais recursos que estavam a ser usados com esse mesmo cliente [13, 27-29].

4.7.2 Nonblocking I/O

O comportamento padrão de um *socket* é o de bloquear até que a acção solicitada seja concluída. Naturalmente, um processo com uma função bloqueada é suspenso pelo sistema operativo.

Um *socket* pode bloquear por várias razões, as funções (*recv()* e *recvfrom()*) bloqueiam se os dados não estiverem disponíveis. A função *send()* num *socket* TCP pode bloquear se não existir espaço suficiente no *buffer* para transmitir os dados. Funções relacionadas com o *sockets* TCP bloqueiam até que uma ligação seja estabelecida. Em todos estes casos, a função devolve somente após o pedido ter sido satisfeito.

Se um datagrama enviado a partir do cliente ou o eco do datagrama a partir do servidor for perdido, o nosso cliente bloqueia indefinidamente. Neste caso, precisamos da função *recvfrom()* para desbloquear após algum tempo, para permitir que o cliente possa lidar com

a perda do datagrama. Felizmente, estão disponíveis mecanismos para controlar comportamentos de bloqueio indesejados. Tratamos de três aqui: *nonblocking sockets*, I/O assíncrono e *timeouts* [27-29].

4.7.2.1. *Nonblocking Sockets*

Uma solução óbvia para o problema do bloqueio indesejável é mudar o comportamento do *socket* para que todas as chamadas à função sejam *nonblocking*. Para um *socket* assim, se for solicitada uma operação que possa ser concluída imediatamente, é devolvido o valor que indica sucesso, caso contrário ele indica que falhou (normalmente "-1"). Em qualquer dos casos, a chamada não bloqueia indefinidamente. No caso de falha, necessitamos de ter a capacidade de distinguir entre uma falha devido ao bloqueio de outro tipo de falhas. Se a falha ocorreu devido a bloqueio, o sistema configura o *errno* para EWOULDBLOCK, excepto para a função *connect()*, que devolve um *errno* de EINPROGRESS.

Podemos alterar o comportamento padrão de bloqueio com a função *fcntl()* ("*file control*").

int fcntl **(int** *socket*, **int** *command*, **long** *argument***);**

A operação a ser realizada é dada pelo argumento *command*. O comportamento que queremos modificar é controlado por *flags* associadas ao *descriptor*, que podemos obter e definir com os comandos F_GETFL e F_SETFL, (com o *argument* igual 0). Ao configurar as *flags* do *socket*, temos que indicar as novas *flags* no *argument*. A *flag* O_NONBLOCK controla o comportamento *nonblocking*.

Existem algumas excepções a este modelo de *sockets nonblocking*. Nos *sockets* UDP, não há *buffers* para o envio, então as funções *send()* e *sendto()* não devolvem EWOULDBLOCK. Para todas as funções, tirando a *connect()*, a operação solicitada é realizada antes de devolver, ou então, nenhuma das operações é concluída. Para o UDP, a função *connect()* atribui, simplesmente, um endereço destino para futuras transmissões de dados, o que sinifica que nunca bloqueia. Para o TCP, *connect()* inicia a configuração de uma ligação. Se a ligação não for concluída sem bloquear, *connect()* devolve um erro, configurando *errno* com EINPROGRESS, que indica que o *socket* ainda está a tentar criar a ligação TCP.

4.7.2.2. I/O Assíncrono

A dificuldade com as chamadas do *socket nonblocking* é que não há nenhuma maneira de saber quando é bem sucedido, excepto fazendo o *polling*. O sistema operativo pode informar o programa quando uma chamada do *socket* é bem sucedida. Deste modo o programa pode desperdiçar tempo noutros trabalhos até receber a notificação de que o *socket* está pronto. Isto é designado de I/O assíncrono, e funciona fazendo com que o sinal SIGIO seja emitido para o processo quando ocorrem eventos de I/O no *socket*.

Utilizar o SIGIO envolve três etapas. Em primeiro lugar, temos que informar o sistema do sinal desejado usando *sigaction()*. Depois, garantimos que os sinais relacionados com o *socket* serão entregues a este processo, tornando-o proprietário do *socket*, com a função *fcntl()*. Por último, marcamos o *socket* como sendo I/O assíncrono através da definição da *flag* FASYNC, novamente via *fcntl ()*.

4.7.2.3. *Timeouts*

Na subsecção anterior, dependíamos do sistema para notificar o programa da ocorrência de um evento relacionado com I/O. Às vezes, porém, podemos realmente precisar de saber que alguns eventos I/O não ocorreram durante um certo período de tempo. Por exemplo, já referimos que as mensagens UDP podem-se perder, nesse caso o cliente nunca irá receber uma resposta do seu pedido. Evidentemente, o cliente não pode dizer directamente que uma perda ocorreu, de modo que estabelece um limite sobre quanto tempo irá esperar por uma resposta. A reacção do cliente para um determinado tempo limite poderia ser a desistência ou tentar novamente o reenvio do pedido.

O método para implementar *timeouts* é definir um alarme antes de chamar uma função que bloqueia.

unsigned int alarm**(unsigned int** *secs*)

A função *alarm()* inicia um temporizador, que expira após o número especificado de segundos (*secs*). Esta devolve o número de segundos restantes de qualquer alarme previamente agendado ou "0" se não houver alarmes programados. Quando o temporizador expira, um sinal SIGALRM é enviado para o processo, e a função *handler* para o SIGALRM, é executada.

4.7.3 Multitarefa

Se vários clientes desejam ligar-se a um mesmo servidor, quando este já está a ser usado, as suas ligações serão estabelecidas, mas o servidor só irá responder após ter acabado com o primeiro cliente. Este tipo de aplicação é designado de servidor iterativo. Servidores iterativos funcionam melhor para aplicações em que cada cliente necessita de uma pequena e delimitada quantidade de trabalho do servidor, no entanto, se o tempo necessário para tratar um cliente for longa, o tempo global de ligação experimentado por qualquer um dos clientes que estão à espera, pode tornar-se inaceitavelmente longo.

Sistemas operativos com multitarefa, como o UNIX, fornecem uma solução para este dilema. Usando processos ou *threads*, podemos dividir a responsabilidade de cada cliente para uma cópia independente do servidor. Existem vários modelos, incluindo o *per-client process*, o *per-client threads* e o *constrained multitasking* [13, 27-29].

4.7.4 Multiplexagem

É frequente o caso em que uma aplicação precisa da capacidade de fazer I/O em múltiplos canais simultaneamente. Por exemplo, podemos querer prestar um serviço em vários portos de uma só vez, mas o problema está após o servidor criar e ligar um *socket* para cada porto. O servidor está disposto a aceitar conexões, mas que *socket* escolher? A função *accept()* (ou *recv()*) num *socket* pode bloquear, causando com que as outras ligações estabelecidas para os outros *sockets* tenham que esperar sem necessidade. A melhor forma de resolver o problema é deixar que o servidor bloqueie até que algum *socket* esteja pronto para I/O.

O UNIX fornece uma maneira de fazer isso. Com a função *select()*, um programa pode especificar uma lista de *descriptors* para verificar se há I/O pendentes, a função *select()* suspende o programa até que um dos *descriptors* da lista esteja pronto para executar I/O e devolve uma indicação dos *descriptors* que estão prontos. Em seguida, o programa pode prosseguir com I/O no *descriptor* com a garantia de que a operação não irá bloquear.

```
int select(int maxDescPlus1, fd_set * readDescs, fd_set * writeDescs,
           fd_set * exceptionDescs, struct timeval * timeout);
```

A função *select()* monitoriza três listas de *descriptors* que, normalmente, são implementados como vectores de bits. Temos o *readDescs* onde os *descriptors* deste

vector são verificados para a disponibilidade imediata de entrada de dados. O *writeDescs* verifica a disponibilidade dos *descriptors* deste vector para gravar dados. Por fim os *descriptors* do vector *exceptionDescs* verificam a existência de excepções pendentes.

Passar NULL para qualquer um dos vectores *descriptors* faz a função *select()* ignorar esse tipo de I/O.

As listas do *descriptor fd_set* são manipuladas por quatro macros, fornecidas pelo sistema.

FD_ZERO(**fd_set** * *descriptorVector*)
FD_CLR(**int** *descriptor*, **fd_set** * *descriptorVector*)
FD_SET(**int** *descriptor*, **fd_set** * *descriptorVector*)
FD_ISSET(**int** *descriptor*, **fcl_set** * *descriptorVector*)

O FD_ZERO remove todos os *descriptors* do vector. O FD_CLR() e o FD_SET() removem e adicionam *descriptors* ao vector, respectivamente. A condição de membro do vector de um *descriptor* é verificada pelo FD_ISSET().

Para evitar que a *select()* pesquise todas as posições possíveis para todos os três vectores, especificamos no *maxDescPlus1* o número máximo de valores de *descriptors* a considerar em cada vector.

O último parâmetro (*timeout*) permite controlar quanto tempo a função *select()* irá esperar até que aconteça alguma coisa. O *timeout* é especificado com uma estrutura de dados *timeval*.

```
struct timeval{
   time_t tv_sec;       /* Segundos */
   time_t tv_usec;      /* Microsegundos */
};
```

Se o tempo especificado na estrutura *timeval* terminar antes de qualquer um dos *descriptors* especificados estar pronto para I/O, a função *select()* devolve o valor "0". Se *timeout* é NULL, a função *select()* não tem tempo limite e espera até que um *descriptor* esteja pronto. Configurar tanto o *tv_sec* como o *tv_usec* a "0", faz com que *select()* devolva imediatamente, permitindo o *polling* dos *descriptors* I/O. Caso não ocorram erros, o *select()* devolve o número total de *descriptors* preparados para I/O.

A função *select()* é uma função poderosa. Também pode ser utilizada para implementar uma versão com *timeouts* em qualquer função de bloqueio I/O (por exemplo, *recv(), accept()*) sem o uso de alarmes [13, 27-29, 31].

4.7.5 *Broadcast* e *Multicast*

Até agora, abordamos apenas *sockets* que tiveram de lidar com uma comunicação entre duas entidades, geralmente um servidor e um cliente. Tal comunicação é designada de *unicast*, porque só uma cópia dos dados é enviada. Em alguns casos, a informação é de interesse para vários destinatários. Poderíamos, simplesmente, fazer o *unicast* de uma cópia dos dados para cada um dos destinatários, no entanto, isso pode ser muito ineficiente. Na verdade, se cada conexão *unicast* através da rede exigir uma quantidade fixa de largura de banda, há um limite rígido para o número de receptores que pode suportar. Por exemplo, se um servidor de vídeo envia *streams* de 1 Mbps e a ligação de rede do servidor é apenas 3 Mbps, só pode suportar três utilizadores ao mesmo tempo.

Felizmente, a rede proporciona uma maneira mais eficiente de utilizar a largura de banda. Em vez de fazer o emissor responsável pela duplicação de pacotes, podemos dar esse trabalho à rede. No exemplo do servidor de vídeo, enviamos apenas uma única cópia do *stream* através da ligação à rede do servidor, o que duplica os dados apenas quando for necessário. Com este modelo de duplicação, o servidor usa apenas 1 Mbps na sua ligação à rede, independentemente do número de clientes.

Existem dois tipos de duplicação de rede: *broadcast* e *multicast*. Com *broadcast*, todos os utentes da rede receberão uma cópia da mensagem, que é enviada indiscriminadamente a todos os que estão na rede. A mensagem *multicast* é enviada para alguns subconjuntos de todos os utentes na rede. Obviamente, o *broadcast* é apenas um caso especial do *multicast*, onde o subconjunto de receptores contém todos os utentes da rede. Para o protocolo IP, só o *sockets* UDP está autorizado a usar *broadcast* e *multicast* [27-29, 31].

4.7.5.1. *Broadcast*

Fazer o *broadcast* de datagramas UDP é semelhante ao envio de datagramas *unicast*. A principal distinção entre o uso de *broadcast* e *unicast* é a forma do endereço. Na prática, existem dois tipos de *broadcast*: *broadcast* local e *broadcast* direccionado. Um endereço de *broadcast* local (255.255.255.255) envia a mensagem para todos os utentes na mesma

rede de *broadcast*. Mensagens de *broadcast* local nunca são enviadas pelos *routers*. O *Broadcast* direccionado permite difundir para todos os utentes numa rede específica.

Consideremos o impacto na rede de realizar um *broadcast* para cada um dos utentes da Internet. O envio de um único datagrama daria origem a um grande número de duplicação de pacotes pelos *routers*, e a largura de banda seria consumida em toda a rede. As consequências do mau uso (mal intencionado ou acidental) são muito grandes, por isso os criadores do IP deixaram de fora uma funcionalidade deste tipo, um endereço de *broadcast Internet-wide*, de propósito. Mesmo com estas restrições, o *broadcast* pode ser muito útil.

4.7.5.2. *Multicast*

Tal como o *broadcast*, o UDP *multicast* é muito semelhante ao UDP *unicast*. Novamente, a principal diferença é a forma do endereço. Um endereço *multicast* identifica um conjunto de receptores de mensagens *multicast*. Os criadores do IP atribuíram uma gama de espaço de endereços dedicado ao *multicast*. Estes são os endereços da classe D e variam de 224.0.0.0 a 239.255.255.255. Com a excepção de um pequeno número de endereços *multicast* reservados, um emissor pode enviar datagramas dirigidos a qualquer endereço da classe D.

Cada pacote IP contém um TTL (*time to live*), inicializado com um valor por defeito e decrementado por cada *router* que lida com o pacote. Quando o TTL chega a "0", o pacote é descartado. Ao definir o TTL, limitamos o número de *hops* que um pacote *multicast* pode realizar a partir do emissor. Podemos alterar o valor do TTL através da configuração de uma opção do *socket*. O TTL pode também ser utilizado para *broadcast*, no entanto, uma vez que os *routers* geralmente não transmitem pacotes *broadcast*, geralmente não tem nenhum efeito.

Ao contrário do *broadcast*, uma rede *multicast* duplica a mensagem apenas para um conjunto específico de receptores. Este conjunto de receptores, designado por grupo *multicast*, é identificado por um endereço partilhado *multicast.* Estes receptores precisam de algum mecanismo, de modo a poderem notificar a rede do seu interesse na recepção de dados enviados para um determinado endereço *multicast*. Uma vez notificada, a rede pode iniciar o envio de mensagens *multicast* para o destinatário. Esta notificação, designada de "*joining group*", é conseguida com um pedido *multicast* enviado pela interface *sockets*.

A única diferença significativa entre o receptor *multicast* e *broadcast* é que o receptor *multicast* deve aderir ao grupo *multicast*. O receptor *multicast* especifica o endereço do grupo através da estrutura *ip_mreq*.

```
struct ip_mreq{
   struct in_addr imr_multiaddr;   / * Endereço do grupo multicast * /
   struct in_addr imr_interface;    / * Endereço da interface local * /
};
```

O campo *imr_multiaddr* contém o endereço Internet para o grupo *multicast* (por exemplo, 224.1.2.3). O campo *imr_interface* especifica a interface do utente para aderir ao grupo. INADDR_ANY permite aderir a partir de qualquer interface. Uma vez especificada, a estrutura *ip_mreq* é dada como parâmetro para a opção IP_ADD_MEMBERSHIP.

4.7.5.3. *Broadcast* vs. *Multicast*

A decisão de utilizar *multicast* ou *broadcast* numa aplicação depende de várias questões, incluindo a porção de utentes da rede interessados em receber os dados. *Broadcast* funciona bem se uma grande percentagem de utentes da rede desejam receber a mensagem, no entanto, se existem mais utentes do que receptores, *broadcast* é muito ineficiente. Na Internet, o *broadcast* seria muito caro, mesmo se a comunicação tivesse 10000 receptores interessados, porque os dados teriam que ser duplicados para cada utente na Internet. Neste caso, o *multicast* limita a duplicação de dados para entrega apenas às redes que têm utentes interessados na mensagem.

A desvantagem do *multicast* é que os receptores IP *multicast* devem saber o endereço de um grupo *multicast* para se juntar. Conhecimento de um endereço que não é necessário para *broadcast*. Em alguns contextos, isto faz do *broadcast* um melhor mecanismo de descoberta do que o *multicast*.

4.8 *Domain Name Service*

A família protocolo TCP/IP usa endereços numéricos de Internet (por exemplo, 169.1.1.1) e portos numéricos (por exemplo, 5000) para descrever a comunicação entre terminais. Isto faz com que as implementações do protocolo sejam eficientes, mas *strings* de números não significam muito para o ser humano, e são portanto, difíceis de memorizar. O endereço Internet de um utente está ligado à parte da rede à qual está ligado, isto promove a

inflexibilidade na sua utilização. Se um utente se move para outra rede ou altera o *Internet Service Provider* (ISP), em geral, o seu endereço Internet tem de mudar.

Para resolver estes problemas, a maioria das implementações da *sockets* API proporcionam o acesso a um serviço de nomes, que mapeia os nomes com outras informações, incluindo endereços Internet. Se o utente mudar o seu endereço Internet, por alguma razão, basta mudar o mapeamento do nome para o novo endereço Internet, permitindo que a mudança seja invisível para programas que usam o nome.

Não é de mais lembrar, que o TCP/IP não necessita deste serviço para trabalhar. O serviço pode aceder a informações a partir de uma grande variedade de fontes. Uma das principais fontes é o *Domain Name System* (DNS). O DNS é uma base de dados distribuída que mapeia *domain names* a endereços Internet e outras informações, o protocolo DNS permite aos utentes conectados à Internet obter informações da base de dados usando o TCP ou o UDP. A configuração da base de dados local, também é outra fonte de informação, e são geralmente mecanismos específicos do sistema operativo para o mapeamento dos nomes com endereços de Internet.

4.8.1 Mapeamento entre Nomes e Endereços Internet

A função *gethostbyname()* pega no nome do utente e devolve a informação do utente disponível a partir do serviço de nomeação, incluindo o endereço Internet.

struct hostent *gethostbyname(**const char** **hostName*)

A função *gethostbyname()* devolve uma estrutura *hostent* ou NULL (erro). A estrutura *hostent* contém as informações relevantes sobre o utente.

```
struct hostent {
    char *h_name;       /* Nome oficial do utente */
    char **h_aliases;   /* Lista dos nomes alternativos */
    int h_addrtype;     /* Tipo de endereço do utente */
    int h_length;       /* Tamanho do endereço */
    char **h_addr_list; /* Lista de endereços */
};
```

O *h_name* é simplesmente uma *string* que contém o nome oficial do utente. O elemento *h_aliases* é um ponteiro para um *array* de ponteiros para *arrays* de caracteres com os

outros nomes do utente (*alias*). O *h_addrtype* indica o tipo de endereço do utente, que pode ser AF_INET para IPv4, e AF_INET6 para o IPv6. O *h_addr_list* contém uma lista dos endereços de Internet associados com o nome oficial implementado como um ponteiro para um *array* de ponteiros, onde cada elemento do *array* aponta para um endereço Internet. Por fim, o *h_length* contém o tamanho, em bytes, dos endereços em *h_addr_list*.

Podemos então obter um endereço de Internet a partir do nome de um utente, para executar o inverso (obter o nome do utente a partir de um endereço de Internet), temos a função *gethostbyaddr()*. Esta função pega na especificação de um endereço binário, na ordem de byte da rede e devolve a informação do utente disponível a partir do serviço de nomeação, que inclui o nome oficial do utente e os seus *alias*.

struct hostent *gethostbyaddr(**const char** **address* ,**int** *addressLength,* **int** *addressFamily***);**

Dado o endereço do utente, o tamanho do endereço e a família do endereço (AF_INET), a função *gethostbyaddr()* devolve uma estrutura *hostent* (descrita anteriormente) ou NULL (erro). Tal como acontece com o *gethostbyname()*, a fonte da informação é indeterminada - pode ser uma base de dados local ou o DNS.

Se o programa necessitar do nome do seu próprio utente, usamos a função *gethostname()*, que pega num *buffer* e no tamanho do *buffer* e copia o nome do utente em que o programa está a ser executado para o *buffer*.

int gethostname**(char** **nameBuffer*, **size_t** *bufferLength*)

4.8.2 Encontrar Informação do Serviço através do Nome

Por analogia com nomes de utentes, a *sockets* API fornece uma maneira de obter informações sobre uma aplicação (servidor), incluindo o número de porto que ele utiliza, pelo nome. A função *getservbyname()* pega num nome de serviço (por exemplo, *echo*) e no protocolo (TCP ou UDP) utilizado pelo servidor e devolve informações do serviço numa estrutura *servent*.

struct servent *getservbyname**(const char** **serviceName*, **const char** **protocol***);**

A função *getservbyname()* devolve uma estrutura *servent* ou NULL (erro). A estrutura *servent* contém as informações relevantes sobre o serviço.

```
struct servent {
    char *s_name;       /* Nome oficial do serviço */
    char **s_aliases;   /* Lista dos nomes alternativos */
    int s_port;         /* Número de porto */
    char *s_proto;      /* Protocolo Implementado */
};
```

A estrutura *servent* é muito semelhante à estrutura *hostent*, o *s_name* é uma *string* que contém o nome oficial do serviço, o *s_aliases* fornece a lista dos nomes alternativos como um ponteiro para um *array* de ponteiros para *arrays* de caracteres contendo esses outros nomes, o *s_port* é o porto do serviço e *s_proto* dá o nome do protocolo que está a implementar o serviço (TCP ou UDP). A *getservbyname()* devolve "0" para sucesso e "-1" para o caso contrário.

Em UNIX, as informações devolvidas por *getservbyname()* geralmente vêm de uma base de dados local. É importante notar que as informações devolvidas por *getservbyname()* podem não ser válidas para um utente que é gerido por uma administração diferente, no entanto, é indiscutivelmente melhor do que nada.

A função *getservbyport()* segue o outro caminho. Esta recebe um número de porto e o protocolo que implementa o serviço e devolve informação do serviço, incluindo o nome real do serviço.

struct servent *getservbyport**(int** **port*, **const char** **protocol**);**

Dado o porto na ordem de byte da rede e o protocolo, a *getservbyport()* devolve uma estrutura *servent* (descrita anteriormente) ou NULL (erro) [27, 28, 32, 33].

Capítulo 5

Encriptação

A comunicação secreta conseguida através da ocultação da existência da mensagem é conhecida como esteganografia. A palavra é derivada da palavra grega "*steganós"* que significa «oculto» *e "gráphein"*, que significa «escrever».

Comparado com a esteganografia que oculta a mensagem, um outro ramo da comunicação secreta, a criptografia, esconde a informação contida na mensagem. A criptografia consiste de duas operações básicas, transposição e substituição.

A transposição implica reordenar as "letras" da mensagem, a substituição envolve o mapeamento das "letras", numa mensagem de acordo com um mapeamento predeterminado. Em termos criptográficos, a nova mensagem obtida através de transformação da mensagem original usando criptografia é conhecido como o *ciphertext*, onde a mensagem original é conhecida como o *plaintext*.

A transformação do *plaintext* para o *ciphertext* é conseguida através da utilização de uma cifra (*cipher*). Cada cifra distinta pode ser descrita em termos do algoritmo e da chave. A próxima grande invenção no mundo da "escrita secreta" veio com a invenção da *cryptanalysis*, ou a ciência da destruição de cifras. A c*ryptanalysis* consiste em obter o *plaintext* a partir do *ciphertext*, sem o conhecimento da chave.

Uma rede de comunicação segura, pode ser definida como uma rede cujos utilizadores não sentem qualquer apreensão ou ansiedade enquanto utilizam a rede.

O significado de uma rede segura depende de como ela é usada, por exemplo, considerando a Internet. Desde que a Internet foi do domínio de engenheiros e cientistas, os utilizadores não se preocuparam com a segurança.

Com a comercialização da Internet, o número de utilizadores aumentou exponencialmente, com este aumento, vieram as preocupações em relação à segurança. Estas preocupações cresceram com o aparecimento da era do comércio electrónico. Hoje em dia, os utilizadores usam os seus cartões de crédito através da Internet [34].

Neste capítulo será feita uma breve introdução à criptografia, sendo então feita uma descrição pormenorizada dos principais protocolos de encriptação, existentes actualmente, o DES, RC6 e o AES Rijndael.

5.1 Criptografia

Uma rede de comunicação segura fornece as seguintes facilidades aos seus utilizadores,

- **Confidencialidade** – A não-ocorrência da divulgação não autorizada de informações. Ninguém excepto o emissor e o receptor devem ter acesso à informação a ser trocada.
- **Integridade** – A não-ocorrência da manipulação não autorizada de informação. Ninguém excepto o emissor e o receptor devem ser capazes de modificar a informação que está a ser trocada.
- **Autenticação** – O receptor tem a capacidade para verificar a origem de uma mensagem. Um intruso não deve ser capaz de passar por outro utilizador.
- ***Nonrepudiation*** – O receptor possui a capacidade de provar que o emissor enviou, de facto, uma determinada mensagem. O emissor não deve ser capaz de mais tarde, negar falsamente que enviou uma mensagem.
- **Fiabilidade do serviço** – A capacidade de proteger a sessão da comunicação contra ataques de negação de serviço.

É importante perceber que estes requisitos são distintos e independentes e a presença ou ausência de qualquer um deles não garante a presença ou a ausência de outro(s).

Após sabermos os requisitos de uma rede de comunicação segura, vamos agora ver a forma como satisfazer estes requisitos. É aqui que entra a criptografia. A criptografia é a arte e a ciência de manter mensagens seguras. Vamos agora ver como a criptografia satisfaz os requisitos de segurança das redes de comunicação [34].

5.1.1 Confidencialidade

A confidencialidade numa comunicação é alcançada através do processo de converter mensagens *plaintext* em *ciphertext*, que é designado por encriptação. A encriptação pode ser definida como o processo de mascarar uma mensagem de forma a ocultar a sua importância ou a informação contida na mensagem. Inversamente, a desencriptação é o processo da obtenção do *plaintext* a partir do *ciphertext* e pode portanto ser definida como o processo da obtenção da informação contida numa mensagem encriptada.

Existem limitações práticas no uso desta metodologia. Primeiro, este esquema requer a utilização de um algoritmo predeterminado pelos utilizadores, o que assume a existência de um canal seguro entre eles, o que nem sempre é possível. Segundo, seria necessário um algoritmo diferente para cada par de utilizadores. O que não só aumentaria a complexidade do sistema, bem como aumentava o trabalho dos criadores de sistemas, para encontrar tantos algoritmos de criptografia.

A confidencialidade do sistema é garantida através da partilha da cifra (o algoritmo de encriptação/desencriptação), e mantendo-a em segredo entre as partes envolvidas, então num grupo de pessoas que querem comunicar uns com os outros, se duas delas querem garantir a confidencialidade das mensagens entre elas, ou seja, não querem que ninguém do grupo possa aceder às mensagens que estão a trocar. Uma vez que o algoritmo é conhecido por todos os elementos do grupo, a confidencialidade numa comunicação a dois não pode ser alcançada por este sistema de segurança. É aqui que entram as chaves. Uma chave é um segredo partilhado entre as partes que comunicam que pode ser usado para garantir a comunicação entre eles.

Quando uma cifra utiliza uma chave para encriptar e desencriptar mensagens, o segredo a ser partilhado deixa de ser o algoritmo e passa a ser uma chave. Podemos então, alcançar uma comunicação segura entre qualquer número de nós num ambiente não seguro, garantindo que os utilizadores partilham uma chave secreta. Torna-se assim, muito mais

fácil de implementar, uma vez que manter uma chave secreta é muito mais fácil do que manter um algoritmo secreto.

5.1.1.1. *Symmetric Key Cryptography*

As chaves podem ser utilizadas simetricamente ou assimetricamente. A *Symmetric Key Cryptography* (SKC) refere-se ao processo em que o emissor e o receptor estão a usar a mesma chave partilhada (e a mesma cifra) para encriptar (E_K) e desencriptar (D_K) as mensagens. Este sistema pode ser representado matematicamente como $D_K(E_K(P)) = P$, onde P se refere ao *plaintext*.

5.1.1.2. *Asymmetric Key Cryptography*

A *Asymmetric Key Cryptography* (AKC) explora a matemática das funções de um sentido. Uma função de um sentido, é uma função *f* em que é fácil de calcular *f(x)* para um dado *x* mas é muito difícil de calcular *x* a partir de uma dada função *f(x)*. Mas existem funções com a peculiaridade, de tornar o calculo de *x* a partir de *f(x)* possível, se e só se soubermos uma chave secreta. Por outras palavras, *x* para *f(x)* é fácil e *f(x)* para *x* é difícil, excepto se soubermos o *y* secreto, caso em que, dado um *y* e *f(x)*, torna-se fácil de obter o *x*.

A AKC refere-se ao processo em que o emissor e o receptor usam chaves diferentes (mas a mesma cifra/algoritmo) para encriptar (E_K) e desencriptar (D_K) mensagens. Tal sistema é representado matematicamente como $E_{K1}(P) = C$ e $D_{K2}(C) = P$ ou $D_{K2}(E_{K1}(P)) = P$, onde o P se refere ao *plaintexet* e o C ao *ciphertext*. A chave de desencriptação é diferente da chave de encriptação e não deve ser produzida a partir da chave de encriptação. A AKC é também designada de *Public Key Cryptography* (PKC), uma vez que tanto a chave de encriptação ou a chave de desencriptação se podem tornar públicas, dependendo se necessitamos de confidencialidade ou *nonrepudiation*.

5.1.2 Integridade

Para o receptor garantir que a mensagem enviada pelo emissor não foi adulterada ou modificada, a criptografia utiliza funções *hash* de um sentido. Estas são funções de um sentido com a propriedade adicional de que pegam numa entrada de tamanho variável, mas produzem uma saída de tamanho fixo (e muito mais curta).

Para a criptografia, existem duas características desejáveis nas funções *hash*. Em primeiro lugar, uma pequena alteração na mensagem deve produzir uma grande alteração no *hash* da mensagem. Isto ajuda porque uma pessoa que tenta adulterar a mensagem provavelmente apenas efectuaria uma pequena alteração na informação na mensagem. Em segundo lugar, na função *hash* não devem ocorrer colisões, isto significa que a probabilidade de duas mensagens diferentes produzirem o mesmo *hash* deve ser muito baixa. Esta propriedade protege contra o ataque onde se tenta alterar completamente a mensagem mas ao mesmo tempo pretende-se manter o mesmo *hash*.

Em criptografia, o emissor da mensagem calcula o *hash* da mensagem. Este *hash* é conhecido por diversos nomes como *Message Authentication Code* (MAC), *message digest*, impressão digital, etc.

5.1.3 Autenticação

Na criptografia, a autenticação é efectuada por meio de assinaturas digitais ou esquemas de resposta ao desafio. A primeira envolve o emissor assinar digitalmente a mensagem. Se for garantido que as assinaturas digitais não podem ser copiadas ou falsificadas, o receptor pode colocar as assinaturas digitais nas mensagens de modo a autenticar a mensagem. As assinaturas digitais utilizam *Public Key Cryptography* (PKC) ao contrário.

Implementações práticas de autenticação não envolvem a encriptação da mensagem na sua totalidade com uma chave, dado que isso pode demorar muito tempo. Em vez disso, é encriptado o *hash* da função. Uma vez que o *hash* da mensagem é muito menor do que a mensagem, poupando muito tempo. O receptor pode então desencriptar o *hash* da mensagem com a chave do emissor, calcular o *hash* da mensagem, e então comparar os dois de modo a autenticar a mensagem.

Nos esquemas de resposta ao desafio, é enviado um número aleatório para o emissor. Este é o desafio. O emissor encripta este número aleatório com a chave, previamente definida entre ambos, e envia a resposta de volta para o receptor. Esta é a resposta. O receptor desencripta então a resposta e verifica se obtém o mesmo número, autenticando assim a comunicação. A autenticação pode ocorrer nos dois sentidos da comunicação.

5.1.4 *Nonrepudiation*

O problema da *nonrepudiation* é resolvido através da utilização de *timestamps*. O receptor consegue provar que o emissor enviou, de facto, uma determinada mensagem, uma vez que a mensagem contém a uma assinatura, feita com a chave privada do emissor e, por conseguinte, foi enviada por ele. No entanto, o emissor pode alegar que tinha perdido a sua chave privada. Os *timestamps* conseguem resolver o problema até certo ponto, mas o *nonrepudiation* é um dos parâmetros de segurança mais difíceis de garantir e, geralmente, exige o envolvimento de uma terceira parte de confiança.

Os *timestamps* também ajudam a resolver o problema da reutilização. A protecção contra a reutilização de uma entidade que devia ser utilizada apenas uma vez, é designada de *replay protection*. Tal protecção é conseguida através da inclusão de um *timestamp* na mensagem, fazendo o *hash* da mensagem e encriptá-lo com a chave privada. Basicamente, o mecanismo da autenticação aumenta o seu alcance para incluir um *timestamp* na assinatura digital.

5.2 *Cryptanalysis*

A antítese da criptografia é a *cryptanalysis*. Considerando que a criptografia pretende ocultar a informação contida numa mensagem, de entidades não autorizadas, a *cryptanalysis*, por outro lado, tem como objectivo recuperar a informação contida numa mensagem encriptada. Dito de outra forma, o objectivo da *cryptanalysts* é o de recuperar o *plaintext* a partir do *ciphertext*, sem ter acesso à chave. Uma tentativa de *cryptanalysis* é designada de um ataque.

A forma mais simples de *cryptanalysis* é um ataque de força bruta, onde dado o *ciphertext*, o *cryptanalyst* tenta cada uma chave possível, uma a uma, para descodificá-lo até se obter o *plaintext*. Este tipo de ataque minimiza as exigências dos dados de entrada, (tudo o que precisa é o *ciphertext*) e maximiza os recursos necessários do sistema (tempo e capacidade de computação) [34].

5.3 Tipos de Cifras

Os algoritmos de criptografia são divididos em duas categorias, dependendo do tipo de dados que têm como entrada. Cifras de bloco usam blocos de dados (geralmente de 64 bits) como entrada, enquanto que as Cifras de *Stream* operaram bit a bit (ou byte). Dada uma chave, uma cifra de bloco encripta sempre um dado bloco do *plaintext* para o mesmo bloco do *ciphertext*. Por outro lado, uma cifra de *stream* encripta um dado bit (ou byte) do *plaintext* para um outro do *ciphertext* [34]. Os protocolos de encriptação apresentados nesta dissertação, o DES, RC6 e o AES Rijndael, são do tipo de cifras de bloco.

5.3.1 Cifras de Bloco

As operações para construir cifras de bloco são a permutação e a substituição. A substituição especifica para cada um dos 2^k valores possíveis do bloco de entrada, o k-bit da saída, então, basicamente, temos uma tabela de substituição. A permutação especifica para cada um dos k bits de entrada, a posição na saída na qual é colocado. O objectivo da permutação e da substituição é de maximizar a difusão e a confusão.

A difusão é a propriedade das cifras que calcula quantos bits mudam no *ciphertext* quando um único bit for alterado no *plaintext.* As Cifras de bloco geralmente têm um elevado grau de difusão, ou seja, alterando 1 bit num bloco do *plaintext* altera quase metade dos bits no bloco do *ciphertext*. As cifras de *stream*, por outro lado, têm uma difusão nula uma vez que operam por definição, num bit de cada vez. A Confusão é propriedade das Cifras que esconde a correlação entre o *plaintext* e o *ciphertext*.

Uma das formas mais comuns para a construção de uma cifra de bloco com chave secreta é partir a entrada em pedaços manejáveis com tamanho fixo (8 bits), fazer uma substituição em cada pedaço, e após a substituição, todos os pedaços passam então por um permutador, que baralha os bits. Em seguida, o processo (um ciclo) é repetido de modo a que cada bit acabe como entrada para cada uma das substituições. Os múltiplos ciclos garantem que a cifra tem difusão suficiente. Devido a estes múltiplos ciclos, estas cifras de bloco são também conhecidos como cifras de bloco iteradas.

As redes de Feistel são cifras de bloco iteradas onde o algoritmo de encriptação é da forma: $L_i = R_{i-1}$ e $R_i = L_{i-1}$ XOR $f(R_{i-1}, K_i)$. A função *f()* pode ser tão complicado como quisermos (pode até ser uma função *hash*), mas qualquer rede Feistel pode ser reversível. O *Data*

Encryption Standard (DES) é um caso especial de redes Feistel que usa uma função "*f()*" proprietária, também conhecida como função *mangler*.

5.3.2 Cifras de *Stream*

As cifras de *stream* operam no *plaintext* um bit de cada vez e consistem em dois componentes, um gerador da chave de *stream* e de uma função de mistura. A função utilizada para misturar é tipicamente a função XOR, enquanto que o gerador da chave de *stream* varia de uma cifra para outra. Apesar das cifras de bloco serem normalmente mais seguras que as cifras de *stream*, o incentivo para a sua utilização reside no facto de que são mais rápidas e mais fáceis de analisar. A primeira propriedade, a rapidez, torna-as mais adequadas para aplicações em tempo real como comunicações de voz e vídeo.

A segurança do sistema recai inteiramente sobre o gerador da chave de *stream*. Para as cifras de *stream* funcionarem correctamente, tanto o emissor como o receptor devem gerar exactamente o mesmo *stream* de números aleatórios de modo a que o emissor possa usar o XOR com o *plaintext* para obter o *ciphertext* e o receptor possa usar o XOR com o *ciphertext* para obter o *plaintext*. No entanto, há uma contradição intrínseca nestes dois objectivos de gerar números aleatórios e determinísticos.

Portanto, o gerador da chave de *stream* em ambos os lados gera números pseudoaleatórios. Estes números pseudoaleatórios são usados para encriptar os dados do lado do emissor e para desencriptar do lado do receptor. De forma a garantir que os números pseudoaleatórios gerados são iguais, é necessário primeiro negociar a chave entre o emissor e o receptor (utilizando, por exemplo o PKC).

Os números pseudoaleatórios são números que "parecem" ser aleatórios, mas que possuem um padrão determinístico. Um óptimo gerador da chave de *stream* tem um período de 2^N para uma chave de N-bits. O período de um gerador da chave de *stream* refere-se ao tempo que leva para gerar a mesma chave de *stream*.

As cifras de *stream* podem ser divididas em duas categorias. Se o estado do gerador da chave de *stream* depende apenas da chave, a cifra é designada de cifra de *stream* síncrona. Se, por outro lado, o estado do gerador da chave de *stream* depende de algum *ciphertext* produzido previamente com a chave, a cifra é designada de cifra de auto sincronização.

5.4 *Data Encryption Standard* (DES)

No final dos anos 60, a IBM iniciou o projecto de investigação Lucifer, liderada por Horst Feistel, para criptografia nos computadores. Este projecto foi concluído em 1971 e Lucifer era conhecido como uma cifra de bloco que operava em blocos de 64 bits, usando uma chave de 128 bits. Pouco depois, a IBM desenvolveu um sistema de encriptação comercial, designado de *Data Encryption Standard* (DES). Walter Tuchman liderou a equipa de investigação, e obteve, no final uma versão mais refinada do Lucifer que era mais resistente às *cryptanalysis*.

Em 1973, o *National Bureau of Standards* (NBS), agora o *National Institute of Standards and Technology* (NIST), emitiu um pedido público à apresentação de propostas para um padrão nacional de cifra. A IBM apresentou os resultados da investigação do projecto DES como um possível candidato. A NBS solicitou à *National Security Agency* (NSA) para avaliar o algoritmo de segurança e determinar a sua aptidão como uma norma federal. Em Novembro de 1976, o *Data Encryption Standard* foi aprovado como norma federal. A descrição oficial da norma, FIPS PUB 46, *Data Encryption Standard* foi publicada em 15 de Janeiro de 1977. O DES sobreviveu, notavelmente, a mais de 20 anos de intensa *cryptanalysis* e tem sido um padrão mundial à mais de 18 anos. Os recentes trabalhos de *cryptanalysis* parecem indicar que DES tem uma estrutura interna muito forte. O DES de acordo com os termos da norma foi revisto de cinco em cinco anos, até que em 2001, o *Advanced Encryption Standard* (AES), conhecido como o algoritmo Rijndael, se tornou um algoritmo avançado de cifra simétrica aprovado pelo FIPS (*Federal Information Processing Standards*), substituindo assim o DES [6, 34].

5.4.1 Descrição do Algoritmo

O DES é uma cifra de bloco simétrica, que opera em blocos de 64 bits usando uma chave de 56 bits. Durante muito tempo, o DES foi um dos algoritmos de encriptação de blocos mais usados. O DES encripta os dados em blocos de 64 bits. Na entrada do algoritmo está o *plaintext*, um bloco de 64 bits, e à saída do algoritmo temos o *ciphertext*, bloco de 64 bits do *plaintext* encriptado após 16 vezes de operações idênticas. O tamanho da chave é de 56 bits, pois são retirados os 8 bits de paridade, que são os bits de oito em oito da chave de 64 bits.

O DES foi desenhado em torno do conceito das redes Feistel. As redes Feistel, são um tipo de cifra de bloco iterada onde cada ciclo pode ser descrito como $L_i = R_{i-1}$ e $R_i = L_{i-1}$ XOR f(R_{i-1}, K_i). A função *f()* é conhecido como a função *mangler*.

Tal como qualquer sistema de encriptação de bloco, há duas entradas para a função de codificação, o *plaintext* de 64 bits a ser codificado e a chave de 56 bits. A construção do bloco do DES é uma combinação adequada de permutações e substituições no bloco *plaintext* (16 vezes). A substituição é realizada através de pesquisas nas tabelas *S-boxes*. Tanto a encriptação como a desencriptação usam o mesmo algoritmo com excepção ao processamento da *key schedule* ser pela ordem inversa.

O bloco *plaintext* X é primeiro transposto sob a permutação inicial IP, dando $X_0 = IP(X) = (L_0, R_0)$. Depois de passar por 16 ciclos de permutação, XORs e substituições, é transposto sob a permutação inversa IP^{-1} para gerar o bloco *ciphertext* Y. Se $X_i = (L_i, R_i)$ denota o resultado da encriptação do *i*-ésimo ciclo, então temos

$L_i = R_{i-1}$

$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$

O *i*-ésimo ciclo da encriptação do algoritmo DES é ilustrado na Figura 5.1. O diagrama de blocos para a computação da função f(R, K) é mostrado na Figura 5.2 O processo de desencriptação pode ser derivado dos termos de encriptação como se segue:

$R_{i-1} = L_i$

$L_i = R_{i-1} \oplus f(R_{i-1}, K_i) = R_i \oplus f(L_i, K_i)$

Se a saída do *i*-ésimo ciclo de encriptação for $L_i||R_i$, então a entrada correspondente ao (16-*i*)-ésimo ciclo de desencriptação é $R_i||L_i$. A entrada para o primeiro ciclo de desencriptação é igual à permuta de 32 bits da saída do processo do 16º ciclo da encriptação. A saída do primeiro ciclo de desencriptação é $L_{15}||R_{15}$, que é a permuta de 32 bits da entrada para o 16º ciclo de encriptação

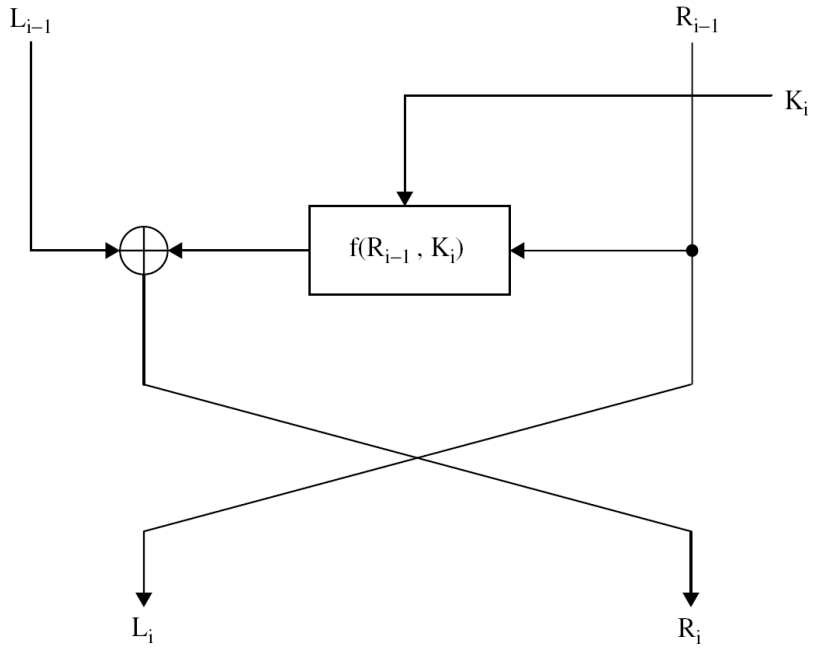

Figura 5.1 – O *i*-ésimo ciclo do algoritmo DES

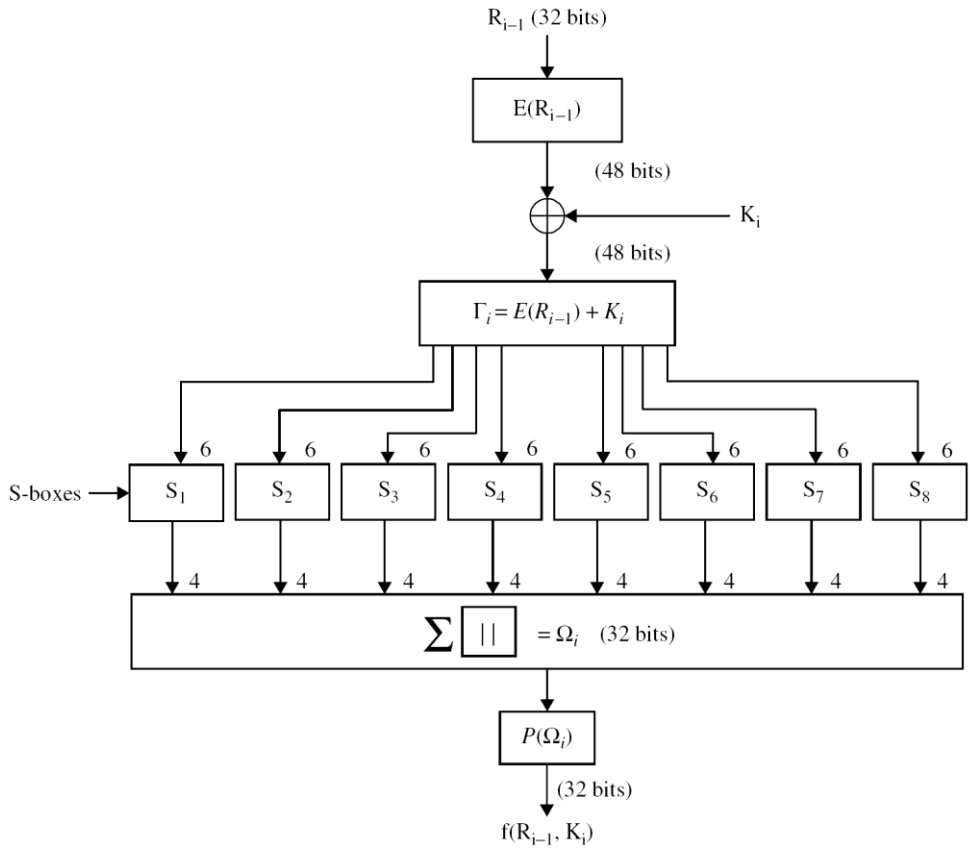

Figura 5.2 – Computação da função *f0*

5.4.2 *Key Schedule*

A chave de 64 bits é inicialmente reduzida para 56-bits, ignorando todos os oitavos bits. A Tabela 5.1 descreve a escolha permutada 1 (*Permuted Choice 1(PC1)*). Estes 8 bits ignorados, k_8, k_{16}, k_{24}, k_{32}, k_{40}, k_{48}, k_{56}, k_{64} são utilizados para verificar a paridade, a fim de garantir que não foram introduzidos erros na chave.

Depois da chave de 56 bits ter sido extraída, eles são divididos em duas metades de 28 bits e colocados em dois registos de trabalho. As metades nos registos são deslocadas para a esquerda uma ou duas posições, dependendo do ciclo. O número de bits deslocados é dado pela Tabela 5.2.

Depois de deslocadas, as metades de 56 bits (C_i, D_i), com $1 \leq i \leq 16$, são usadas como entrada da chave para a próxima iteração. Estas metades estão concatenadas, no conjunto ordenado e servem como entrada para a escolha permutada 2 (*Permuted Choice 2* (PC2)) (ver Tabela 5.3), que produz uma chave de 48 bits na saída. Assim, por cada ciclo do DES são geradas chaves de 48 bits diferentes. Estas chaves de 48 bits, K_1, K_2, ..., K_{16}, são utilizadas para encriptar em cada ciclo, na ordem de K_1 até K_{16}. O *Key Schedule* do DES é ilustrado na Figura 5.3.

Com um tamanho de chave de 56 bits, existem $2^{56} = 7{,}2 \times 10^{16}$ chaves possíveis. Assumindo que, em média, metade do espaço da chave, tem que ser pesquisado, uma única máquina a executar uma encriptação DES por μs iria demorar mais de 1000 anos para quebrar a cifra. Então, um ataque com força bruta no DES parece ser impraticável.

Tabela 5.1 – Escolha Permutada 1 (PC-1)

57	*49*	*41*	*33*	*25*	*17*	*9*	*1*	*58*	*50*	*42*	*34*	*26*	*18*
10	*2*	*59*	*51*	*43*	*35*	*27*	*19*	*11*	*3*	*60*	*52*	*44*	*36*
63	55	*47*	*39*	*31*	*23*	*15*	*7*	*62*	*54*	*46*	*38*	*30*	*22*
14	*6*	*61*	*53*	*45*	*37*	*29*	*21*	*13*	*5*	*28*	*20*	*12*	*4*

Tabela 5.2 – *Schedule* para os deslocamentos da chave

Número do Ciclo	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Número de deslocamentos para a esquerda	*1*	*1*	*2*	*2*	*2*	*2*	*2*	*2*	*1*	*2*	*2*	*2*	*2*	*2*	*2*	*1*

Tabela 5.3 – Escolha Permutada 2 (PC-2)

14	*17*	*11*	*24*	*1*	*5*	*3*	*28*	*15*	*6*	*21*	*10*
23	*19*	*12*	*4*	*26*	*8*	*16*	*7*	*27*	*20*	*13*	*2*
41	52	*31*	*37*	*47*	*55*	*30*	*40*	*51*	*45*	*33*	*48*
44	*49*	*39*	*56*	*34*	*53*	*46*	*42*	*50*	*36*	*29*	*32*

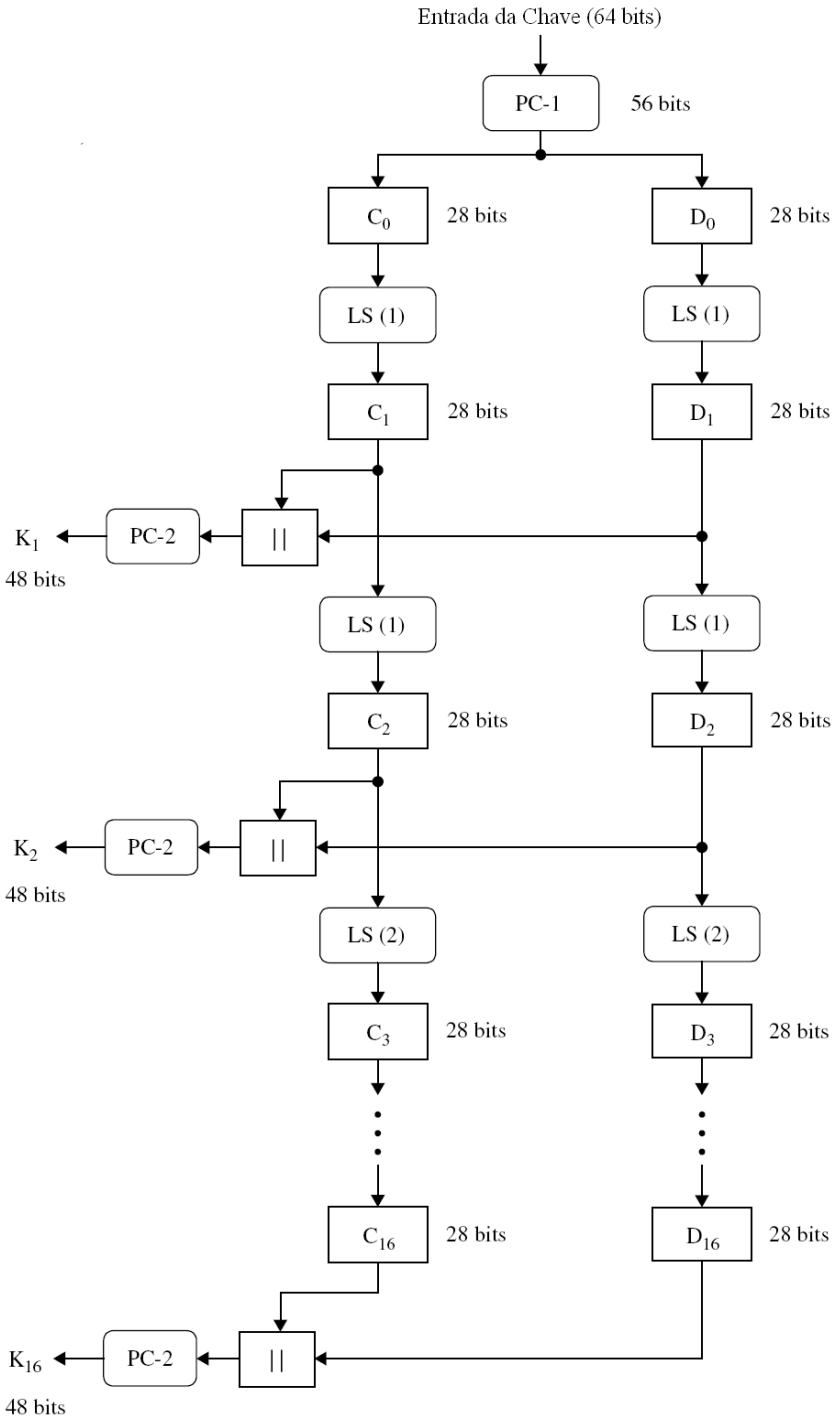

Figura 5.3 – *Key Schedule* para o DES

5.4.3 Encriptação DES

O DES opera num bloco de 64 bits de *plaintext*. Depois da permutação inicial, o bloco é dividido em dois blocos L_i (esquerda) e R_i (direita), cada com um tamanho de 32 bits. O *plaintext* permutado (ver Tabela 5.4) tem o bit 58 como o primeiro de entrada, o bit 50 como seu segundo bit, e assim por diante até que o último bit, o bit 7. A metade direita dos dados, R_i, é ampliada para 48 bits de acordo com a Tabela 5.5 de uma permutação de expansão.

Tabela 5.4 – Permutação Inicial (IP)

L_i	*58*	*50*	*42*	*34*	*26*	*18*	*10*	*2*
	60	*52*	*44*	*36*	*28*	*20*	*12*	*4*
	62	*54*	*46*	*38*	*30*	*22*	*14*	*6*
	64	*56*	*48*	*40*	*32*	*24*	*16*	*8*
R_i	*57*	*49*	*41*	*33*	*25*	*17*	*9*	*1*
	59	*51*	*43*	*35*	*27*	*19*	*11*	*3*
	61	*53*	*45*	*37*	*29*	*21*	*13*	*5*
	63	*55*	*47*	*39*	*31*	*23*	*15*	*7*

Tabela 5.5 – Tabela da selecção do bit *E*

32	*1*	*2*	*3*	*4*	*5*
4	*5*	*6*	*7*	*8*	*9*
8	*9*	*10*	*11*	*12*	*13*
12	*13*	*14*	*15*	*16*	*17*
16	*17*	*18*	*19*	*20*	*21*
20	*21*	*22*	*23*	*24*	*25*
24	*25*	*26*	*27*	*28*	*29*
28	*29*	*30*	*31*	*32*	*1*

O símbolo de expansão E do E(R_i) denota uma função que tem o R_i de 32 bits como entrada e tem como saída os 48 bits de E(R_i). O objectivo desta operação é duplo, primeiro serve para tornar a saída do mesmo tamanho que a chave para a operação XOR, e segundo é utilizado para fornecer um resultado mais longo que é comprimido durante a operação de substituição da *S-box.*

Depois da operação XOR da chave comprimida K_i, com o bloco E(R_{i-1}) tal que $\Gamma_i = E(R_{i-1}) \oplus K_i$ para $1 \leq i \leq 15$. O Γ_i vai então ser utilizado na operação de substituição, realizada por oito S_i-*boxes.* O Γ_i é então dividido em oito blocos de 6 bits. Cada bloco é operado por uma S_i-*box* diferente, como mostra a Figura 5.2. Cada S_i-*box* é uma tabela de 4 linhas e 16

colunas como ilustra a Tabela 5.6. Estes 48 bits passam por um transformação *S-box* não-linear para produzir uma saída de 32 bits.

Tabela 5.6 – *S-boxes*

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_1	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	3	4	3	2	12	9	5	15	0	11	14	1	7	6	0	8	13
S_7	0	4	11	2	14	15	0	8	3	3	12	9	7	5	10	6	1
	1	13	0	11	7	4	9	1	0	14	3	5	12	2	15	8	6
	2	1	4	11	13	12	3	7	4	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	3	2	1	14	7	4	10	8	3	15	12	9	0	3	5	6	11

Se cada S_i indica uma matriz definida na Tabela 5.6 e A indica um bloco de entrada de 6 bits, então, S_i(A) é definida da seguinte forma, o primeiro e último bits de A representam o número da linha da matriz S_i, enquanto que os 4 bits do meio representam o número de coluna no intervalo de 0 a 15.

Por exemplo, com uma entrada (101110) para S_5-*box*, os dois primeiros bits combinam formando 10, que corresponde a linha 2 (na realidade à terceira linha) da S_5. Os quatro bits do meio formam 0111, que corresponde à coluna 7 (na realidade à oitava coluna) da mesma S_5. Assim, o valor correspondente seria 8 (1000), que iria então substituir o valor de entrada 101110.

Oito blocos de quatro bits são o resultado da fase de substituição, que recombina num único bloco de 32 bits Ω_i por concatenação. Estes 32-bits Ω_i da substituição *S-box* são permutados de acordo com a Tabela 5.7. Esta permutação mapeia cada bit de entrada Ω_i para uma posição de saída de $P(\Omega_i)$.

Tabela 5.7 – Função de Permutação *P*

16	*7*	*20*	*21*
29	*12*	*28*	*17*
1	*15*	*23*	*26*
5	*18*	*31*	*10*
2	*8*	*24*	*14*
32	*27*	*3*	*9*
19	*13*	*30*	*6*
22	*11*	*4*	*25*

Tabela 5.8 – Inverso da Permutação Inicial (IP^{-1})

40	*8*	*48*	*16*	*56*	*24*	*64*	*32*
39	*7*	*47*	*15*	*55*	*23*	*63*	*31*
38	*6*	*46*	*14*	*54*	*22*	*62*	*30*
37	*5*	*45*	*13*	*53*	*21*	*61*	*29*
36	*4*	*44*	*12*	*52*	*20*	*60*	*28*
35	*3*	*43*	*11*	*51*	*19*	*59*	*27*
34	*2*	*42*	*10*	*50*	*18*	*58*	*26*
33	*1*	*41*	*9*	*49*	*17*	*57*	*25*

A saída $P(\Omega_i)$ é obtida a partir da entrada Ω_i, onde o 16º bit do Ω_i é o primeiro bit de $P(\Omega_i)$, o sétimo bit o segundo bit de $P(\Omega_i)$, e assim por diante até que o 25º bit de Ω_i é o 32º bit de $P(\Omega_i)$. Por último, é efectuada a operação XOR do resultado permutado com a metade esquerda (L_i) da permutação inicial do bloco de 64 bits. Depois, as metades são trocadas e começa outro ciclo. A permutação final é a inversa da permutação inicial, e está descrita na Tabela 5.8 IP^{-1}. É de notar que as metades esquerda e direita não são trocadas após o último ciclo do DES. Em vez disso, o bloco concatenado $R_{16}||L_{16}$ é usado como entrada para a última permutação da Tabela 5.8. Assim, a estrutura global para o algoritmo DES é apresentada na Figura 5.4.

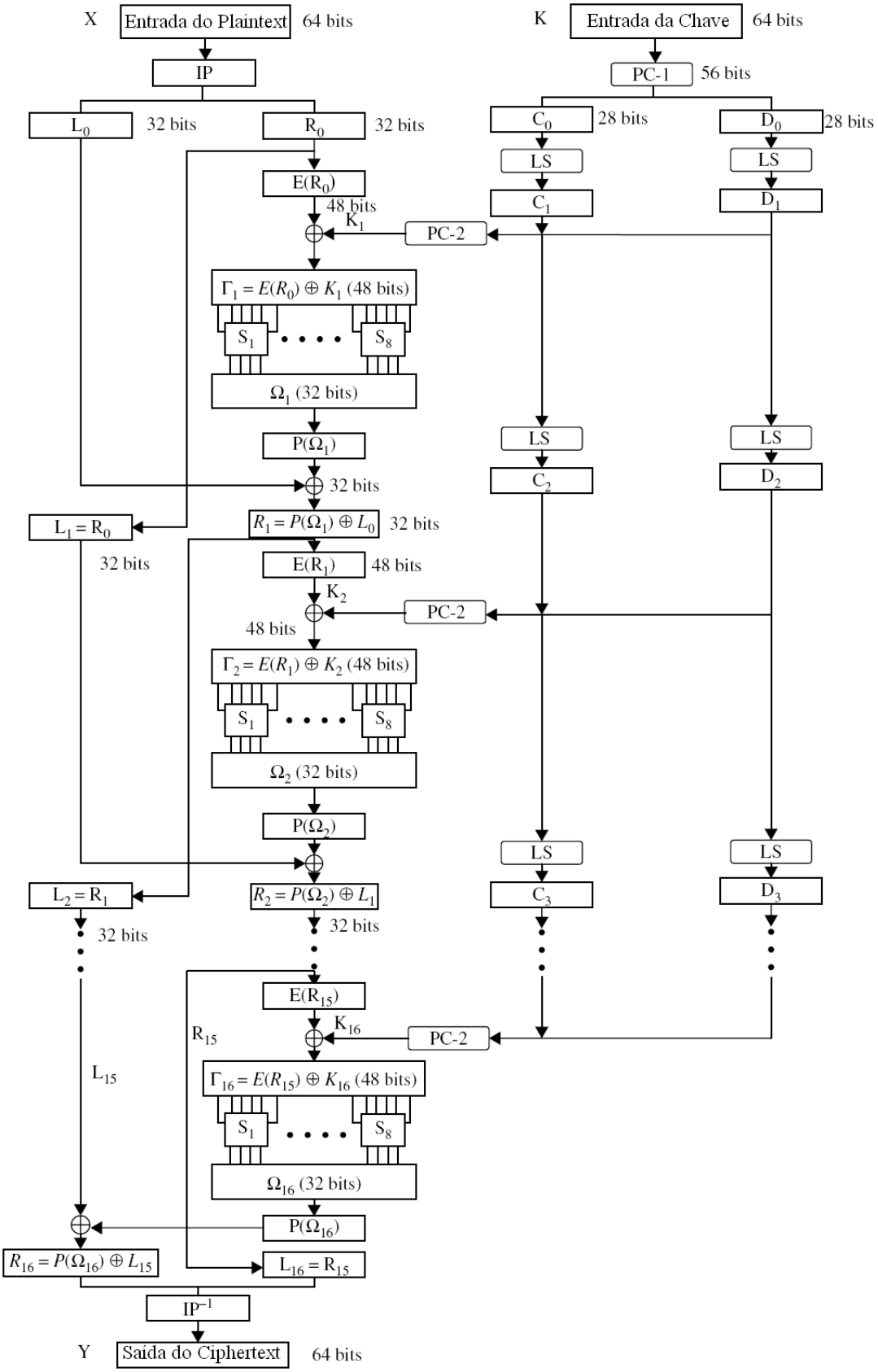

Figura 5.4 – Estrutura do algoritmo DES

5.4.4 Desencriptação DES

O algoritmo de desencriptação é exactamente idêntico ao algoritmo de encriptação, excepto que os ciclos das chaves são usados na ordem inversa. Uma vez que as chaves de encriptação para cada ciclo são K_1, K_2, ..., K_{16}, as chaves da desencriptação para cada ciclo são K_{16}, K_{15}, ..., K_1. Por isso, o mesmo algoritmo funciona tanto para encriptação como para desencriptação.

5.4.5 *Triple* DES

O *Triple* DES é muito popular em aplicações baseadas na Internet, incluindo PGP e S/MIME. A possível vulnerabilidade do DES a um ataque de força bruta leva à necessidade de encontrar um algoritmo alternativo. O *Triple* DES é uma abordagem amplamente aceite que utiliza múltiplas encriptações com DES e com múltiplas chaves, como ilustra a Figura 5.5. A alternativa preferida é o *Triple* DES de três chaves, cujo tamanho da chave é de 168 bits.

O *Triple* DES com duas chaves ($K_1 = K_3$, K_2) é uma alternativa ao DES relativamente popular. Mas o *Triple* DES com três chaves (K1, K2, K3) é preferível, uma vez que resulta num grande aumento na força criptográfica.

Fazendo referência à Figura 5.5, o *ciphertext* C é produzido da seguinte forma:

$C = E_{K3}[D_{K2}[E_{K1}(P)]]$

O emissor encripta com a primeira chave K_1 e, em seguida, desencripta com a segunda chave K_2 e, finalmente, encripta com a terceira chave K_3. A desencriptação exige que as chaves sejam aplicadas na ordem inversa:

$P = D_{K1}[E_{K2}[D_{K3}(C)]]$

O receptor desencripta com a terceira chave K_3, em seguida, encripta com a segunda chave K_2 e, finalmente, desencripta com a primeira chave K_1.

Este processo é muitas vezes conhecido como modo Encripta – Desencripta – Encripta (EDE).

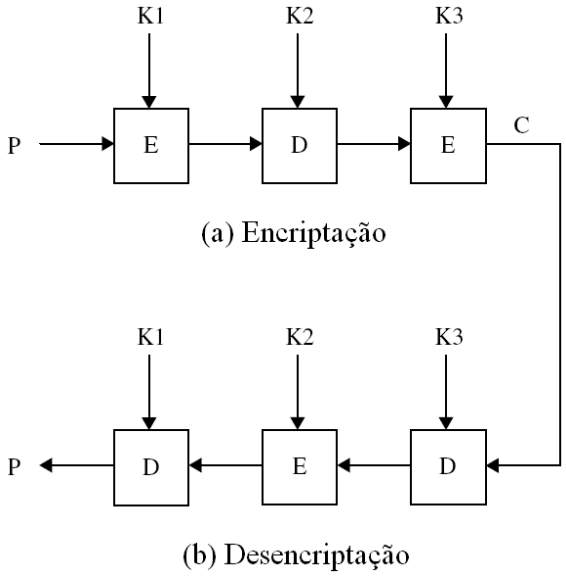

Figura 5.5 – Encriptação/Desencriptação do *Triple* DES

5.5 Algoritmo RC6

O algoritmo RC6 é uma melhoria em relação ao RC5, concebido por Ron Rivest e outros para cumprir os requisitos da competição AES. Trata-se de um algoritmo proprietário, patenteado pela RSA *Security* RC6 faz uso de rotações dependentes dos dados. Uma nova característica do RC6 é a utilização de quatro registos de trabalho em vez de dois. Enquanto o RC5 é uma cifra de bloco rápida, que actua em blocos de 128 bits usando dois registos de trabalho de 64 bits. O RC6 modificou a sua concepção ao usar quatro registos de 32 bits em vez de dois registos de 64 bits. Isto tem a vantagem de poder fazer duas rotações por ciclo em vez da uma encontrada num meio-ciclo do RC5 [6, 35].

5.5.1 Descrição do RC6

Tal como o RC5, o RC6 é uma família de algoritmos de encriptação totalmente parametrizados. A versão do RC6 também é especificada como RC6-*w/r/b* onde o tamanho da *word* é *w* bits, a encriptação consiste num número *r* de ciclos, e o *b* indica o tamanho da chave de encriptação em bytes. O RC6 foi apresentado ao NIST para ser considerado como o novo *Advanced Encryption Standard* (AES). Uma vez que o AES é orientado para *w*=32 e *r*=20, os valores especificados como parâmetro do RC6-*w/r* são usados como atalho para se referir a essas versões. Para todas as variantes, o RC6-*w/r/b* opera em quatro *words* de *w* bits, usando as seguintes seis operações básicas:

$a + b$: Adição de inteiros *modulo* 2^w.
$a - b$: Subtracção de inteiros *modulo* 2^w.
$a \oplus b$: *Bitwise exclusive-OR* de *words* de *w* bits.
$a \times b$: Multiplicação de inteiros *modulo* 2^w.
$a <<< b$: Rodar a *word* de *w* bits *a* para a direita, pela quantia dada pelo menos significativo de lg *w* bits de *b*.
$a >>> b$: Rodar a *word* de *w* bits *a* para a esquerda, pela quantia dada pelo menos significativo de lg *w* bits de *b*.

Onde lg *w* denota o logaritmo de base dois de *w* ($\log_2 w$), e a operação *modulo* (mod ou %) refere-se ao resto da divisão inteira.

RC6 tira partido das operação dependentes dos dados, dado que a multiplicação de inteiros de 32 bits é aplicada de forma eficiente na maioria dos processadores. Multiplicação de inteiros é uma difusão muito eficaz, e é utilizada no RC6 para calcular valores de rotação de modo a que estes valores sejam dependentes de todos os bits de outro registo. Como resultado, o RC6 tem uma difusão muito mais rápida do que o RC5.

5.5.2 *Key Schedule*

O *Key Schedule* do RC6-*w/r/b* é praticamente idêntico ao do RC5-*w/r/b*. Na verdade, a única diferença é que, no RC6-*w/r/b*, mais *words* são derivadas da chave fornecida pelo utilizador para o uso durante a encriptação e desencriptação.

O utilizador fornece uma chave de *b* bytes, onde $0 \leq b \leq 255$, são anexados suficientes bytes zero, de modo a termos o tamanho da chave igual ao número de *words*, os bytes da chave são então colocados num *array* de *c words* de *w* bits L[0], L[1], ..., L[*c*-1]. O número de *words* de *w* bits gerado para chaves adicionais é de $2r+4$, e estas são armazenadas no *array* S[0, 1,..., $2r + 3$].

O algoritmo do *Key Schedule* é o seguinte.

Key Schedule para RC6-w/r/b

Entrada: a chave de *b* bits fornecida pelo utilizador é colocada no *array* de *c words* L[0, 1, ..., *c*-1]

Número de ciclos: *r*

Saída: chaves de *w* bits por ciclo S[0, 1,..., $2r + 3$]

Definição das constantes mágicas

P_w = Odd ((e - 2) 2^w)

Q_w = Odd ((φ - 2) 2^w)

onde

e = 2,71828182. . . (base de logaritmos naturais)

φ = 1,618033988... (*golden ratio*)

Convertendo a palavras-chave de bytes em words

for $i = b - 1$ down to 0 do

    L[i/u] = (L [i/u] <<<8 + $K[i]$

Inicializando o array S

S[0] = P_w

for i = 1 to $2r + 3$ do

    S[i] = S[i-1] + Q_w

Misturando a chave com S

A = B = i = j = 0

$v = 3 \times \max\{c, 2r + 4\}$

for s = 1 to v do

    {

        A = S[i] = (S[i] + A + B) <<<3

        B = L[j] = (L[j] + A + B) <<<(A + B)

        $i = (i + 1) \bmod (2r + 4)$

        $j = (j + 1) \bmod c$

    }

5.5.3 Encriptação

A encriptação do RC6 funciona com quatro registos de *w* bits A, B, C e D, que contêm o *plaintext* inicial de entrada. O primeiro byte do *plaintext* é colocado no byte menos significativo de A. O último byte do *plaintext* é colocado no byte mais significativo de D. A disposição dos (A, B, C, D) = (B, C, D, A) é como a atribuição paralela de valores (bytes) sobre os registos da direita para os da esquerda, como mostra a Figura 5.6.

O algoritmo de encriptação do RC6 é exposto a seguir:

Encriptação com RC6-w/r/b

Entrada: *Plaintext* armazenado em quatro registos de *w* bits A, B, C, D

Número de ciclos: r

Chaves de *w* bits por ciclo S[0, 1, ..., 2*r* + 3]

Saída: *Ciphertext* armazenado no A, B, C, D

Procedimento: B = B + S[0]

D = D + S[1]

for *i* = 1 to *r* do

{

t = (B × (2B + 1)) <<< lg *w*

u = (D × (2D + 1)) <<< lg *w*

A = ((A ⊕ *t*) <<<*u*) + S[2*i*]

C = ((C ⊕ *u*) <<<*t*) + S[2*i* + 1]

(A, B, C, D) = (B, C, D, A)

}

A=A + S[2*r* + 2]

C=C + S[2*r* + 3]

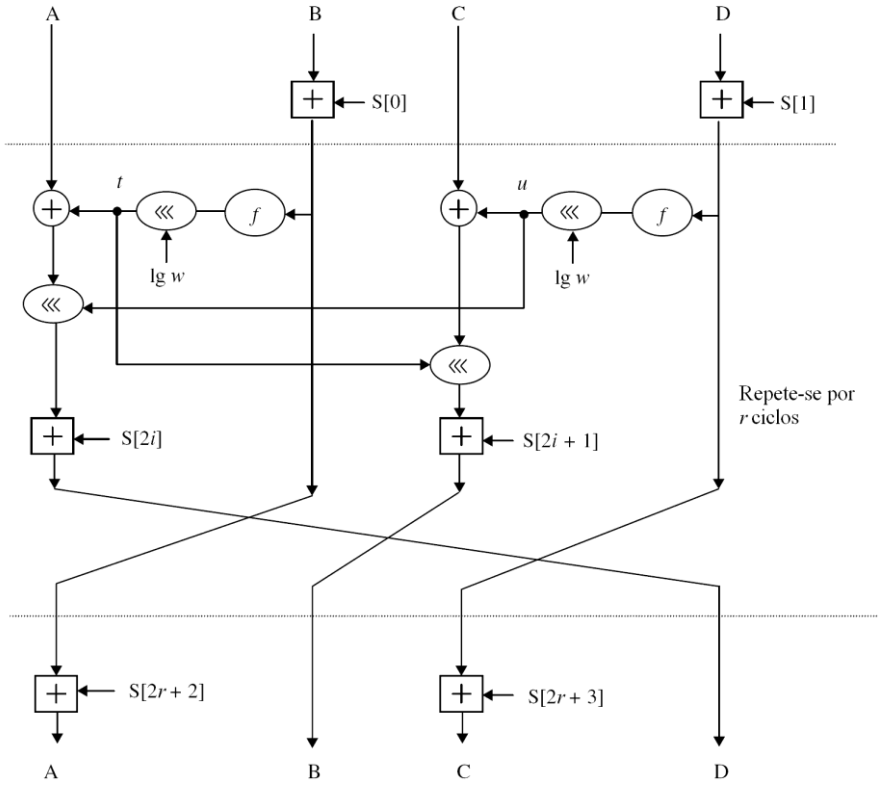

Figura 5.6 – Esquema de Encriptação RC6-*w/r/b*

5.5.4 Desencriptação

A desencriptação do RC6 funciona com quatro registos de *w* bits A, B, C e D, que contêm o *ciphertext* inicial na saída, no fim da encriptação. O primeiro byte do *ciphertext* é colocado no byte menos significativo de A. O último byte do *ciphertext* é colocado no byte mais significativo de D.

O algoritmo de desencriptação do RC6 é mostrado a seguir:

Desencriptação com RC6-w/r/b

Entrada: *Ciphertext* armazenado em quatro registos de *w* bits A, B, C, D

Número de ciclos, r

Chaves de *w* bits por ciclo S[0, 1, ..., 2*r* + 3]

Saída: *Plaintext* armazenado no A, B, C, D

Procedimento: C = C – S [2*r* + 3]

A = A – S [2*r* + 2]

for $i = r$ down to 1 do

{

$(A, B, C, D) = (D, A, B, C)$

$u = (D \times (2D + 1)) <<< \lg w$

$t = (B \times (2B + 1)) <<< \lg w$

$C = ((C - S[2i+1]) >>> t) \oplus u$

$A = ((A - S[2i]) >>> u) \oplus t$

}

$D = D - S[1]$

$B = B - S[0]$

O Esquema de desencriptação do RC6 é ilustrado na Figura 5.7.

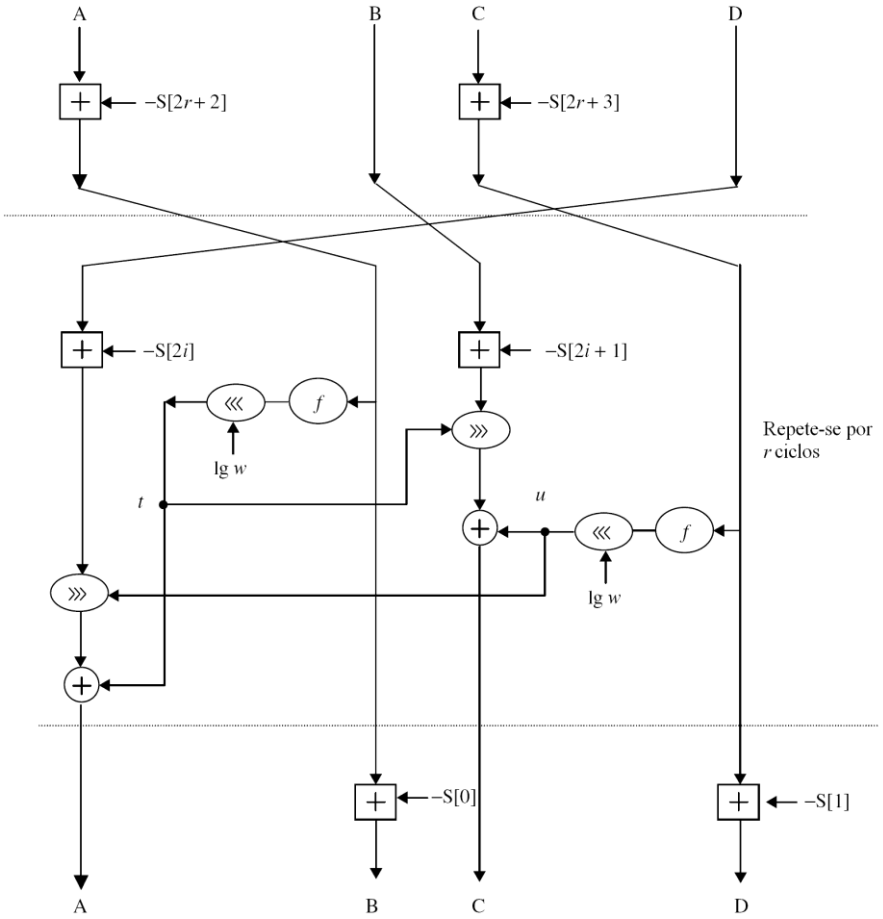

Figura 5.7 – Esquema de Desencriptação RC6-*w/r/b*

5.6 Algoritmo AES (Rijndael)

O *Advanced Encryption Standard* (AES) especificado pelo algoritmo Rijndael que é um algoritmo criptográfico aprovado pelo FIPS, desenvolvido por Daemen e Rijmen como um algoritmo AES candidato em 1999. O algoritmo Rijndael é uma cifra de bloco simétrica que pode processar blocos de dados de 128 bits usando chaves criptográficas de 128, 192 e 256 bits. Nesta secção, iremos abordar as notações e convenções, bem como a especificação do algoritmo AES Rijndael [6, 34, 36].

5.6.1 Notações e Convenções

Seguem-se algumas notações e convenções, que se podem encontrar no algoritmo do protocolo de encriptação AES Rijndael.

5.6.1.1. Entradas e Saídas

A entrada e saída para o algoritmo AES, consiste cada um em sequências de 128 bits. Estas sequências por vezes são referidas como blocos e o número de bits que contêm será referido como o seu tamanho. A chave da cifra para o algoritmo AES é uma sequência de 128, 192 ou 256 bits. Outros tamanhos para estes elementos, não são permitidos por esta norma.

Os bits dentro dessas sequências serão numerados, a começar em zero e a terminar no tamanho da sequência (tamanho da chave ou tamanho do bloco) menos um. O número i anexado a um bit é conhecido como o seu índice e será entre um dos intervalos $0 \leq i < 128$, $0 \leq i < 192$ ou $0 \leq i < 256$, dependendo do tamanho do bloco e tamanho da chave.

5.6.1.2. Bytes

A unidade básica para processar no algoritmo AES é um byte, uma sequência de oito bits tratada como uma entidade única. As sequências de entrada, saída e Chave da cifra descritas na Secção anterior são processadas como *arrays* de bytes que são formados dividindo essas sequências em grupos de oito bits contíguos para formar *arrays* de bytes

Todos os valores de byte no algoritmo AES serão apresentados como a concatenação dos seus valores de bit individuais (0 ou 1), entre parênteses na seguinte ordem $\{b_7, b_6, b_5, b_4,$

b_3, b_2, b_1, b_0}. Estes bytes são interpretados como elementos de um campo finito usando uma representação polinomial:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i$$

Por exemplo, {01001011} identifica o elemento do campo finito $x^6 + x^3 + x + 1$.

Se houver um bit adicional b_8 à esquerda de um byte de oito bits, ele será exibido imediatamente à esquerda do parêntese da esquerda como 1 (00101110) = 1 (2e).

5.6.1.3. *Array* de Bytes

Arrays de bytes, a_0, a_1, a_2, ..., a_{15} são definidos a partir da sequência de entrada de 128 bits, ip_0, ip_1, ip_2,..., ip_{126}, ip_{127}, como se segue:

$a_0 = (ip_0, ip_1,..., ip_7)$

$a_1 = (ip_8, ip_9,..., ip_{15})$

⋮

$a_{15} = (ip_{120}, ip_{121},..., ip_{127})$

onde ip_k define o $input_k$ para k = 0, 1, 2,..., 127.

De um modo geral, com sequências de 192 ou 256 bits, pode ser expresso como:

$a_n = (ip_{8n}, ip_{8n+1},..., ip_{8n+7})$, $n \leq 16$.

5.6.1.4. *State*

Internamente, as operações do algoritmo AES são realizadas num *array* bidimensional de bytes designado de *State*. O *State* é composto por quatro linhas de bytes, cada uma contendo *Nb* bytes, onde *Nb* é o tamanho do bloco dividido por 32. O *array* de s*tate* $S_{r,c}$ tem um número de linha *r*, $0 \leq r < 4$, e um número de coluna *c*, $0 \leq c < Nb$.

O byte de entrada do *array* (in_0, in_1,..., in_{15}) na cifra é copiado para o *array* de *state* de acordo com o seguinte esquema:

$S_{r,c} = in(r + 4c)$ para $0 \leq r < 4$ e $0 \leq c < Nb$

e, na cifra inversa, o *state* é copiado para o *array* de saída como se segue:

out ($r + 4c$) = $S_{r, c}$ para $0 \leq r < 4$ e $0 \leq c < Nb$.

Um byte individual do *state* é referido como $S_{r, c}$ ou $S(r, c)$. As operações da cifra e a cifra inversa são realizadas sobre o *array* de *state* conforme ilustrado na Figura 5.8.

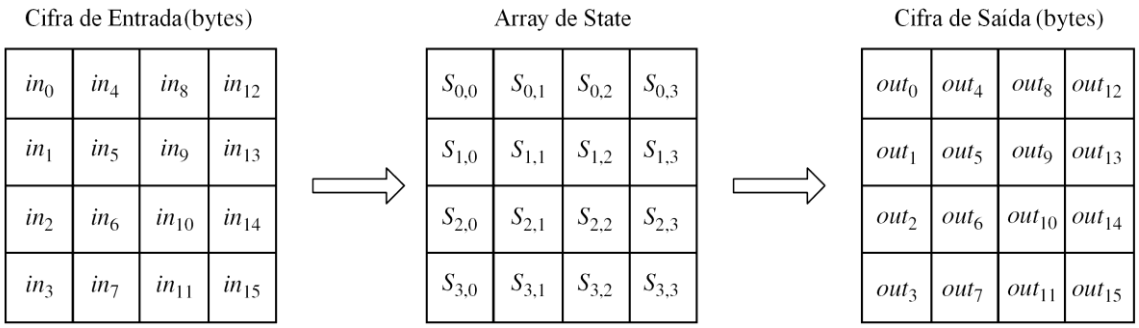

Figura 5.8 – Entrada e saída do *array* de *state*

Os quatro bytes em cada coluna do *state* formam uma *word* de 32 bits, onde o número de linha *r* fornece um índice para os quatro bytes dentro de cada *word*, e o número de coluna *c* fornece um índice que representa a coluna neste *array*.

5.6.2 Operações Matemáticas

Todos os bytes no algoritmo AES são interpretados como elementos de um campo finito usando a notação introduzida na secção anterior. Elementos de um campo finito podem ser adicionados e multiplicados, mas estas operações são diferentes das utilizadas para números. A operação *modulo* (mod ou %) refere-se ao resto da divisão inteira. As seguintes subsecções introduzem os conceitos matemáticos necessários para a secção seguinte.

5.6.2.1. Adição

A adição de dois elementos de um campo finito é conseguida "adicionando" os coeficientes das potências correspondentes dos polinómios dos dois elementos. A adição é realizada com a operação XOR (indicada por $\oplus$). Por conseguinte, a subtracção de polinómios é idêntica à adição de polinómios

Em alternativa, a adição de elementos de um campo finito pode ser descrita como a adição *modulo* 2 dos bits correspondentes no byte. Para os dois bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$e $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, a soma é $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$,onde cada $c_i = a_i \oplus b_i$.

Por exemplo, as seguintes expressões são equivalentes:

$(x^5 + x^3 + x^2 + 1) + (x^7 + x^5 + x + 1) = x^7 + x^3 + x^2 + x$ (polinomial)

{00101101} ⊕ {10100011} = {10.001.110} (binário)

(2d) ⊕ (a3) = (8e) (hexadecimal)

5.6.2.2. Multiplicação

Na representação polinomial, a multiplicação em (Galois Field) GF(2^8) (indicado por •) corresponde com a multiplicação de polinómios de grau irredutível 8. Um polinómio é irredutível se os seus únicos divisores são o um e ele próprio. Para o algoritmo AES, este polinómio irredutível é

$m(x) = x^8 + x^4 + x^3 + x + 1$,

ou {01}{1b} em notação hexadecimal.

Por Exemplo (73) • (a5) = (e3)

(01110011) • (10100101)

$(x^6 + x^5 + x^4 + x + 1) \bullet (x^7 + x^5 + x^2 + 1)$

$= x^{13} + x^{12} + x^{10} + x^9 + x^6 + x^4 + x^3 + x^2 + x + 1$

A redução *modular* por *m(x)* resulta em

$x^{13} + x^{12} + x^{10} + x^9 + x^6 + x^4 + x^3 + x^2 + x + 1 \bmod (x^8 + x^4 + x^3 + x + 1)$

$= x^7 + x^6 + x^5 + x + 1$

= (11100011) = (e3)

A multiplicação definida acima é associativa, *a(x)(b(x) + c(x)) = a(x)b(x) + a(x)c(x)*, e o elemento (01) = (00000001) é designado de identidade do multiplicativo. Para qualquer polinómio $b(x)$ de grau inferior a 8, o multiplicativo inverso de $b(x)$, designado por $b^{-1}(x)$, pode ser encontrado usando o Algoritmo de Euclides tal que

$b(x)a(x) + m(x)c(x) = 1$

do qual $b(x)a(x)$ mod $m(x) \equiv 1$. Assim, o inverso do multiplicativo de $b(x)$ torna-se

$b^{-1}(x) = a(x)$ mod $m(x)$.

Multiplicação por x

O polinómio binário, $b(x) = \sum_{i=0}^{7} b_i x^i$ multiplicado com x resulta em $xb(x) = \sum_{i=0}^{7} b_i x^{i+1}$, mas pode ser reduzido com o *modulo m(x)*.

Se $b_7 = 1$, a redução é conseguida efectuando o XOR do $m(x)$. Daqui resulta que a implicação x (i.e. $(00000010)_{(2)} = (02)_{(16)}$) pode ser implementada ao nível do byte com um deslocamento para a esquerda e o *bitwise* XOR (1b). Esta operação em bytes é designada por *xtime()*. Multiplicação de potências superiores a x pode ser implementada pela repetição da aplicação de *xtime()*.

Por Exemplo (57) • (13) = (fe)

(57) = (01010111)

(57) • (02) = *xtime*(57) = (10101110) = (ae)

(57) • (04) = *xtime*(ae) = (01011100) ⊕ (00011011)

= (01000111) = (47)

(57) • (08) = *xtime*(47) = (10001110) = (8e)

(57) • (10) = *xtime*(8e) = (00011100) ⊕ (00011011) = (07)

Assim, segue-se que

(57) • (13) = (57) • {(01) ⊕ (02) ⊕ (10)}

= (57) ⊕ (57) • (02) ⊕ (57) • (10)

= (57) ⊕ (ae) ⊕ (07)

= (01010111) ⊕ (10101110) ⊕ (00000111)

= (11111110) = (fe)

5.6.2.3. Polinómios com elementos de campo finito no GF(2^8)

Um polinómio *a(x)* com coeficiente *byte* no GF(2^8) pode ser expresso na forma de *word* como:

$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \Leftrightarrow a = (a_0, a_1, a_2, a_3)$

Para ilustrar as operações de adição e de multiplicação, temos um segundo polinómio:

$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \Leftrightarrow b = (b_0, b_1, b_2, b_3)$

A adição é realizada adicionando os coeficientes de campo finito com potências iguais de *x* de modo a que

$a(x) + b(x) = (a_3 \oplus b_3)\, x^3 + (a_2 \oplus b_2)\, x^2 + (b_1 \oplus a_1)\, x + (a_0 \oplus b_0)$

Esta adição corresponde a uma operação XOR entre os bytes correspondentes de cada uma das *words*.

A multiplicação é alcançada através de dois passos. No primeiro passo, o polinómio produto $c(x) = a(x) \bullet b(x)$ é expandido e potências iguais são recolhidas para devolver

$c(x) = a(x) \bullet b(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$

onde

$c_0 = a_0b_0$ $\qquad$ $c_4 = a_3b_1 \oplus a_2b_2 \oplus a_1b_3$

$c_1 = a_1b_0 \oplus a_0b_1$ $\qquad$ $c_5 = a_3b_2 \oplus a_2b_3$

$c_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2$ $\qquad$ $c_6 = a_3b_3$

$c_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3$

O próximo passo é reduzir *c(x)* mod ($x^4 + 1$) para o algoritmo AES, de modo a que x^i mod $(x^4 + 1) = x^{i \bmod 4}$.

O produto, $a(x) \otimes b(x)$, de dois polinómios de quatro termos *a(x)* e *b(x)*, é dado por

$d(x) = a(x) \otimes b(x) = d_3x^3 + d_2x^2 + d_1x + d_0$

onde

$d_0 = a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3$

$d_1 = a_1b_0 \oplus a_0b_1 \oplus a_3b_2 \oplus a_2b_3$

$d_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2 \oplus a_3b_3$

$d_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3$

Assim, *d(x)* escrito na forma de matriz é

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ d_3 \end{bmatrix}$$

O algoritmo AES também define o polinómio inverso como:

$a(x) = (03)\, x^3 + (01)\, x^2 + (01)\, x^1 + (02)$

$a^{-1}(x) = (0b)\, x^3 + (0d)\, x^2 + (09)\, x^1 + (0e)$

5.6.3 Especificação do Algoritmo AES

Para o algoritmo AES, *Nb* define o número de *words* de 32 bits no que diz respeito ao bloco de 128 bits da entrada, saída, ou *state* (128 = *Nb* × 32, a partir do qual *Nb* = 4).

O tamanho da Chave da cifra, *K*, é de 128, 192, ou 256 bits. O tamanho da chave é representado por *Nk* = 4, 6 ou 8, o que reflecte o número de *words* de 32 bits (número de colunas) na Chave da cifra.

Nk = 128 × 32, *Nk* = 4

Nk = 196 × 32, *Nk* = 6

Nk = 256 × 32, *Nk* = 8

Para o algoritmo AES, o número de ciclos a serem realizadas durante a execução do algoritmo depende do tamanho da chave. O número de ciclos é representado por *Nr*, onde *Nr* = 10 quando *Nk* = 4, *Nr* = 12 quando *Nk* = 6, e *Nr* = 14 quando *Nk* = 8.

Quer para a Cifra, quer para a Cifra Inversa, o algoritmo AES usa uma função por ciclo composta de quatro transformações diferentes orientadas ao byte: 1) substituição de um byte usando uma tabela de substituição (*S-box*), 2) deslocação das linhas do *array* de *state*

por diferentes *offsets*, 3) mistura dos dados de cada coluna do *array* de *state*, e 4) adição de uma chave para ao *state*. O esquema de um ciclo do AES, é ilustrado pela Figura 5.9.

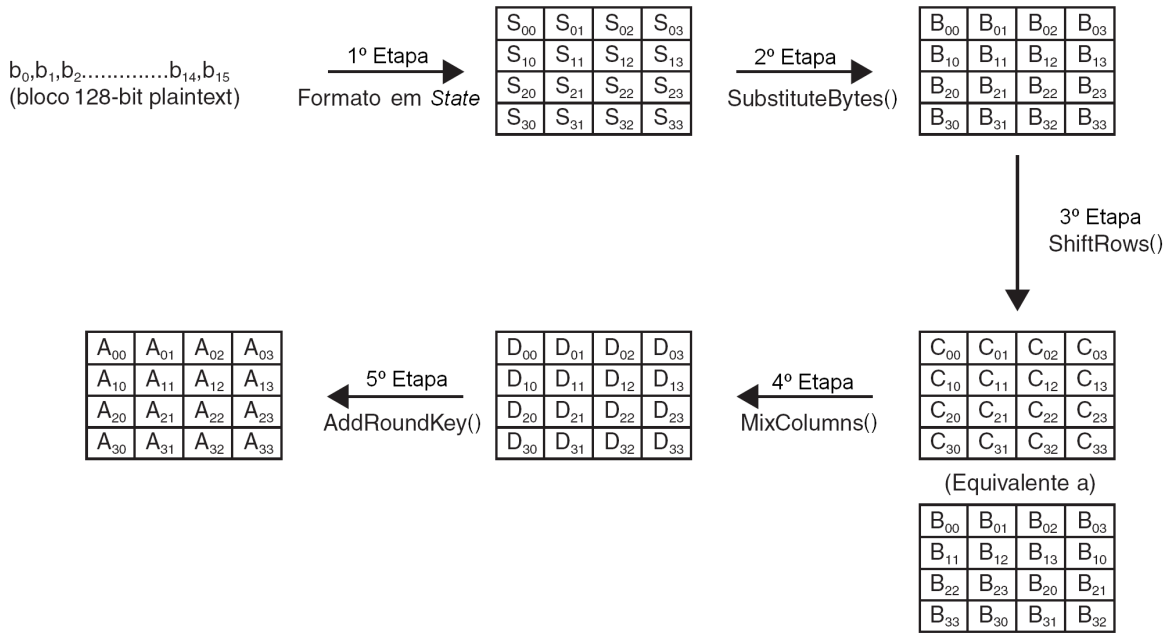

Figura 5.9 – Um ciclo do AES

5.6.3.1. *Key Expansion*

O algoritmo AES com a Chave da cifra K, executa uma rotina *key expansion* para gerar um *key schedule*. A *key expansion* gera um total de $Nb(Nr + 1)$ *words*. Assim, a *key schedule* consiste num *array* linear de *words* de quatro bytes $[w_i]$, $0 \leq i < Nb(Nr + 1)$.

A função *RotWord()* pega na *word* de quatro bytes de entrada $[a_0, a_1, a_2, a_3]$ e realiza uma permutação cíclica como $[a_1, a_2, a_3, a_0]$.

A função *SubWord()* pega na *word* de quatro bytes de entrada e aplica a *S-box* (Figura 5.10) a cada um dos quatro bytes para produzir uma *word* de saída.

O Rcon[i] representa o *array* de *words* constante do ciclo e contém os valores dados por $[x^{i-1}, \{00\}, \{00\}, \{00\}]$ com i a começar em 1.

O Rcon[i] é um elemento útil para o *array* de *words* constante do ciclo, de modo a processar a rotina *key expansion*.

O processo de *key expansion* para o *key schedule* é processado como mostra a Figura 5.11.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	ae	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	db	8b	8ª
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 5.10 – *S-box* do AES

```
Key Expansion(byte key[4 * Nk], word w[Nb * (Nr + 1)],Nk)
begin
   i= 0
   while(i< Nk)
        w[i]=word[key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]]
        i=i+1
   end while
   i= Nk
   while (i< Nb*(Nr + 1))
         word temp=w[i−1]
        if(i mod Nk =0)
             temp=SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk = 8 and i mod Nk = 4)
             temp=SubWord(temp)
        endif
   w[i]=w[i−Nk] xor temp
   i=i+1
   end while
end
```

Figura 5.11 – Pseudocódigo para a *key expansion*

5.6.3.2. Cifra

A cifra de 128 bits de entrada é copiada para o *array* de *state* como mostra a Tabela 5.9.

A Cifra é descrita no pseudocódigo na Figura 5.12. As transformações individuais – *SubBytes(), ShiftRows(), MixColumns(),* e *AddRoundKey(),* desempenham um papel na transformação do *state* e são sumariamente descritas a seguir.

Tabela 5.9 – Cifra de 16 bytes do *array* de entrada

Número da Linha	*Mapeamento do bloco de entrada num array de coluna por coluna*			
0	a_0	a_4	a_8	a_{12}
1	a_1	a_5	a_9	a_{13}
2	a_2	a_6	a_{10}	a_{14}
3	a_3	a_7	a_{11}	a_{15}

```
Cipher(byte in [4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte   state[4, Nb]
    state=in
    AddRopundKey(state, w)
    for round=1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w+round*Nb)
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w+Nr*Nb)
    out=state
end
```

Figura 5.12 – Pseudocódigo para a cifra

Transformação SubBytes()

A transformação *SubBytes()* é uma substituição de byte não linear que opera independentemente em cada byte do *state* através de uma *S-box* (Figura 5.13).

Por exemplo, se $s_{2,1}$ = {8f}, então o valor da substituição é determinado pela intersecção da linha de índice 8 com a coluna com índice f na Figura 5.10. O resultado de $s'_{2,1}$ seria o valor {73}.

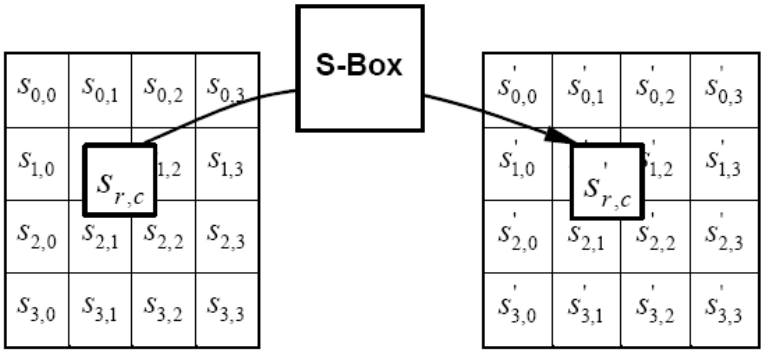

Figura 5.13 – Transformação *SubBytes()* através da *S-box*

Transformação ShiftRows()

Na transformação *ShiftRows()*, a primeira linha (linha 0) não é deslocada, as restantes linhas procedem da seguinte forma:

$s'_{r,c} = s_{r,c + shift(r, Nb)) \bmod Nb}$, para $0 < r < 4$ e $0 \leq c < Nb$

onde o valor do deslocamento $shift(r, Nb) = shift(r, 4)$ depende do número de fila r do seguinte modo:

$shift(1, 4) = 1$; $shift(2, 4) = 2$; $shift(3, 4) = 3$;

Isto tem o efeito de deslocar os bytes à esquerda para as posições à direita, um número diferente de bytes dado pelo número de linha. A Figura 5.14 ilustra a transformação *ShiftRows()*.

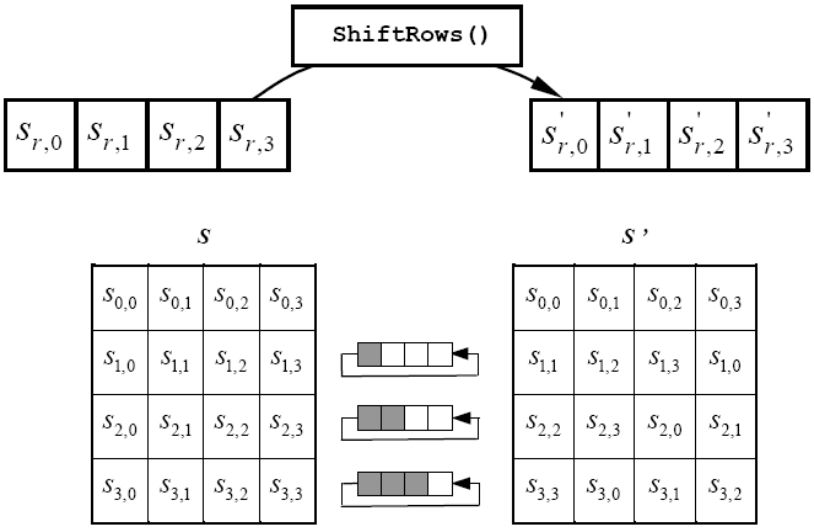

Figura 5.14 – *ShiftRows()* desloca ciclicamente a ultimas três linhas do *State*

Transformação MixColumns()

A transformação *MixColumns()* opera sobre o *state* coluna-a-coluna, tratando cada coluna como um polinómio de quatro termos sobre GF(2^8) e multiplicando o *modulo* $x^4 + 1$ com um polinómio fixo $a(x)$ como:

$s'(x) = a(x) \otimes s(x)$

onde $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, $s(x)$ é o polinómio de entrada e $s'(x)$ é o polinómio correspondente após a transformação *MixColumns()*.

A matriz de multiplicação $s'(x)$ é

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad para\ 0 \le c < Nb$$

Os quatro bytes de uma coluna depois da matriz de multiplicação são

$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$

$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$

$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$

$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$

A Figura 5.15 ilustra a transformação *MixColumns()*.

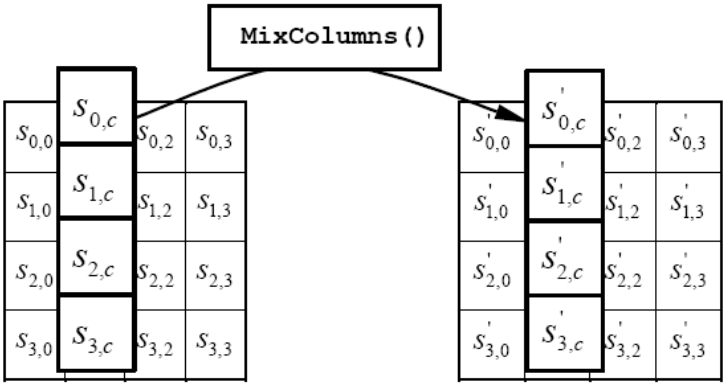

Figura 5.15 – A Transformação *MixColumns()* actua no *State* coluna a coluna

Transformação AddRoundKey()

Na transformação *AddRoundKey()*, uma chave do ciclo é adicionada ao *state* através de uma simples operação *bitwise* XOR. Cada ciclo consiste de *Nb words* de uma *key schedule*. Estas *Nb words* são adicionadas às colunas do *state* tal que

$[s'_{0,c}, s'_{1,c}, s'_{2,c} s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c} s_{3,c}] \oplus [w_{\text{round} * Nb + c}]$ para $0 \leq c < Nb$

onde $[w_i]$ são as *words* do *key shedule*, e o *round* é um valor no intervalo $0 \leq round \leq Nr$. A primeira chave de ciclo é adicionada quando *round* = 0, antes do início da primeira aplicação da função do ciclo. A aplicação da transformação *AddRoundKey()* para os *Nr* ciclos da cifra ocorre quando $1 \leq round \leq Nr$.

Esta transformação é ilustrada na Figura 5.16.

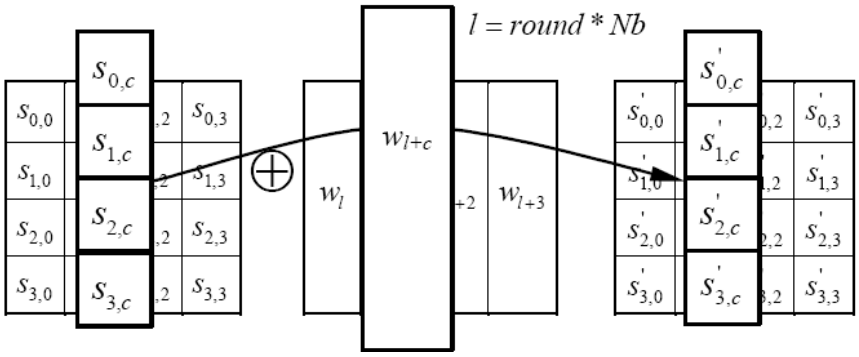

Figura 5.16 – Transformação *AddRoundKey()*

5.6.3.3. Cifra Inversa

A transformação Cifra pode ser aplicada na ordem inversa para produzir uma Cifra Inversa para o algoritmo AES. As transformações individuais utilizadas na Cifra Inversa são *InvSubBytes(), InvShiftRows(), InvMixColumn ()* e *AddRoundKey()*. Estas transformações inversas desempenham um papel na transformação do *state* e são sumariamente descritas a seguir.

Transformação InvSubBytes()

A *InvSubBytes()* é o inverso da transformação *SubBytes()*, na qual a *S-box* inversa é aplicada a cada byte do *state*. A *S-box* inversa utilizada na transformação *InvSubBytes()* é apresentada na Figura 5.17.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9b	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 5.17 – *S-box* Inversa do algoritmo AES

Transformação InvShiftRows()

A transformação *InvShiftRows()* é o inverso da transformação *ShiftRows()*. A primeira linha (linha 0) não é deslocada. Os bytes nas últimas três linhas (Linha 1, Linha 2, Linha 3) são ciclicamente deslocados um número diferente de bytes como se segue:

shift(*r, Nb*), desloca valores, onde *r* é um número de fila e o *Nb* = 4.

shift(1, 4) = 1 ; *shift*(2, 4) = 2; *shift*(3, 4) = 3; respectivamente.

Especificamente, a transformação *InvShiftRows()* procede da seguinte forma:

$s'_{r,c + \text{shift}(r, Nb)) \bmod Nb} = s_{r,c}$, para $0 < r < 4$ e $0 \leq c < Nb$

A Figura 5.18 ilustra a Transformação *InvShiftRows()*.

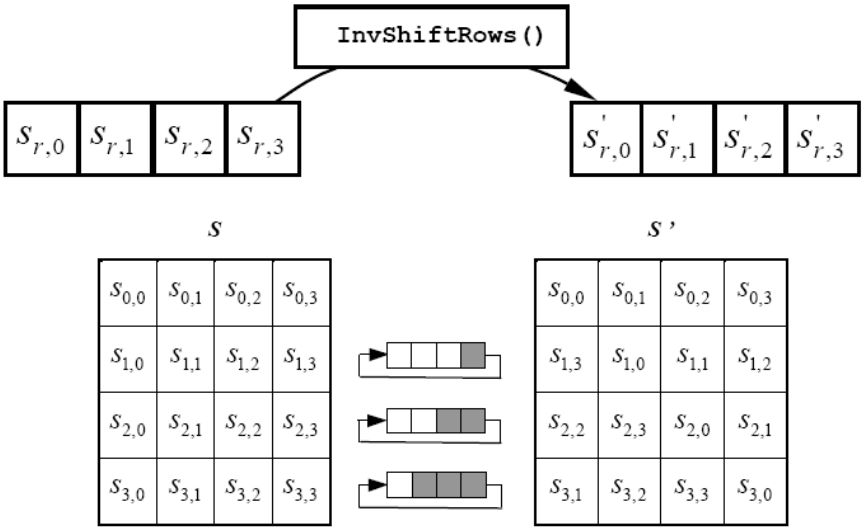

Figura 5.18 – Transformação *InvShiftRows()*

Transformação InvMixColumns()

A transformação *InvMixColumns()* é o inverso da transformação *MixColumns()*. Esta transformação opera coluna-a-coluna sobre o *state*, tratando cada coluna como um polinómio de quatro termos. As colunas são consideradas como polinómios sobre GF(2^8) e multiplicando o *modulo* $x^4 + 1$ com um polinómio fixo $a^{-1}(x)$

Se a *state* inverso $s'(x)$ é escrito como uma multiplicação de matrizes então segue-se

$s'(x) = a^{-1}(x) \otimes s(x)$

onde $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$.

A matriz de multiplicação $s'(x)$ pode ser expressa como

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad para\ 0 \leq c < Nb$$

A multiplicação irá resultar em quatro bytes numa coluna

$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$

$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$

$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$

$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$

Transformação Inversa da AddRoundKey()

A transformação *AddRoundKey()* é o seu próprio inverso, uma vez que somente envolve a aplicação da operação XOR.

Para desencriptar *ciphertext*, a Cifra Inversa é descrita no pseudocódigo mostrado na Figura 5.19.

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr + 1)])
begin
   byte  state[4,Nb]
   stae=in
   AddRoundKey(state, dw + Nr*Nb)
   for round= Nr – 1 step –1 to 1
       InvSubBytex(state)
       InvShiftRows(state)
       InvMixColumns(state)
       AddRoundKey(state, dw+round*Nb)
   end for
   InvSubBytex(state)
   InvShiftRows(state)
   AddRoundKey(state, dw)
   out=state
end
```

Figura 5.19 – Pseudocódigo para a cifra inversa

Capítulo 6

Sistema de Comunicação e Encriptação

O trabalho proposto foi a criação de um sistema de comunicação, mais especificamente, um sistema de transmissão e recepção de vídeo através da Internet. O vídeo a ser transmitido é encapsulado em pacotes RTP, este encapsulamento é efectuado pelo SUIT (*Scalable, Ultra-fast and Interoperable Interactive Television*) [37], que então transmite para a rede, em *multicast.* O objectivo do trabalho consistiu na selecção e reenvio para o cliente de um dos três serviços de vídeo (CIF (*Common Intermediate Format*), SD (*Standard Definition video*) e o HD (*High Definition video*)) disponibilizados pelo SUIT. Após a recepção dos pacotes, um dos três serviços fornecidos, é então seleccionado e reenviado para um cliente. Note-se que foi implementado um conversor de formatos de vídeo, de modo a efectuar a selecção do serviço pretendido. Esse trabalho foi desenvolvido por Tiago Caçoilo, no âmbito da sua Tese de Mestrado, "Sistema de Conversão de Vídeo". Foram também, desenvolvidos dois algoritmo de encriptação, para serem utilizados na encriptação dos pacotes, no sentido de tornar a comunicação mais segura. Todo o trabalho foi desenvolvido em Linguagem C, recorrendo às ferramentas GNU de desenvolvimento.

Este Capítulo é complementado pelo Anexo A, que expõe uma demonstração e análise dos resultados obtidos, com recurso a algumas imagens.

No Anexo B, é apresentada a plataforma, bem como as ferramentas utilizadas para a realização do trabalho.

Neste capítulo, apresenta-se o sistema de comunicação e os algoritmos de encriptação desenvolvidos, serão também apresentados os resultados de testes efectuados.

6.1 Sistema de Comunicação

Como já foi referido, o objectivo do trabalho consistiu em efectuar a recepção de pacotes RTP, transmitidos pelo SUIT, que contém três serviços de vídeo diferentes, e então reenviar o serviço pretendido para o cliente. Foram desenvolvidos dois sistemas de comunicação, um para *unicast* e outro para *multicast*. A comunicação *unicast* foi desenvolvida com o intuito de ser uma primeira versão, para a realização de testes simples que mostrassem o seu funcionamento.

Para o desenvolvimento deste sistema, foi utilizado o *Sockets* API, descrito em pormenor no Capitulo 4. Dado tratar-se de uma transmissão de vídeo, o UDP será logicamente o protocolo de transporte, será também utilizada a sinalização fornecida pela interface *socket*, de modo a se evitar o bloqueio do programa.

O sistema de comunicação, possui como um dos argumento de entrada a *Palavra Chave*, utilizada para a encriptação do pacote. Como os algoritmos de encriptação foram implementados no sistema de comunicação, serão aqui apenas referido, já que serão abordados em maior detalhe na secção seguinte.

É de seguida apresentada uma breve descrição do algoritmo desenvolvido, para os dois tipos de comunicação, bem como os resultados dos testes efectuados.

6.1.1 Comunicação *Unicast*

O desenvolvimento deste sistema consistiu em três partes: um sistema para efectuar a transmissão de pacotes (Servidor), um receptor-emissor de pacotes, que dado à integração do conversor de vídeo, designamos por Conversor e um Cliente, para a recepção final dos pacotes. Uma vez que se trata de uma transmissão UDP, não se estabelece uma conexão entre estações.

Foi então montado o esquema de rede apresentado na Figura 6.1. O Servidor é lançado da seguinte forma:

./Servidor <Endereço IP do Conversor> <Serviço a Enviar> <Número de Porto do Conversor> <Palavra Chave >

Para realizar a transmissão este necessita de saber o Endereço IP e o número de porto do Conversor. O *Serviço a Enviar* é o pacote que pretendemos transmitir neste programa.

A *Palavra Chave* é utilizada para encriptar o pacote a ser enviado para o Conversor.

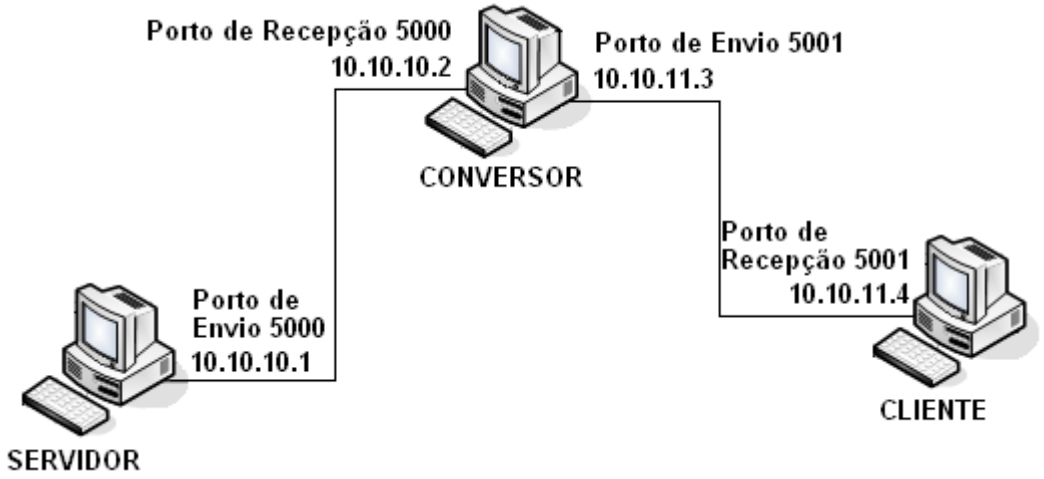

Figura 6.1 – Esquema de rede *Unicast*

O Conversor faz a recepção do pacote, que é processado pelo conversor de vídeo, e se o tipo de serviço for o pretendido, o pacote é reenviado para o Cliente. O Conversor para a comunicação com o Servidor, necessita apenas de saber qual o número de porto a escutar. Tem também, como parâmetro de entrada, o Tipo de Serviço de TV pretendido (CIF, SD, HD), este parâmetro é necessário para o conversor de vídeo. Para a transmissão do serviço pretendido é necessário o Endereço IP e o número de porto do Cliente.

./Conversor <Número de Porto do Servidor> <Tipo de Ficheiro> <Endereço IP do Cliente> <Número de Porto do Cliente>

O Conversor, após a recepção com sucesso, emite uma mensagem com o Endereço IP da máquina que enviou o pacote. São também impressas mensagens que dizem respeito ao conversor de vídeo, indicando se o pacote pertence ao nosso sistema, identifica o tipo de serviço e se é o serviço pretendido.

Já o Cliente, necessita apenas de saber o número de porto a escutar para receber os pacotes provenientes do Conversor e da palavra chave utilizada na encriptação, de modo a se poder realizar a desencriptação.

./Cliente <Número de Porto do Conversor> <Palavra Chave>

Após a recepção, o pacote é armazenado no disco, imprimindo também uma mensagem a indicar o endereço IP do Conversor. Seguem-se os resultados da transmissão efectuada.

Depois da transmissão efectuada com sucesso do Servidor para o Cliente, os pacotes são comparados, para isso utilizamos o programa *Hexedit* [38]. Verificámos que não houve qualquer alteração no conteúdo do pacote, garantindo assim a integridade do mesmo, podemos desta forma, comprovar o bom funcionamento do sistema desenvolvido. No Anexo A.1.1, é feita uma demonstração destes resultados, através da análise de várias imagens do conteúdo do pacote no Servidor e no Cliente para comparação, como as mensagens lançadas pelo Conversor quando lida com o pacote.

6.1.2 Comunicação *Multicast*

O desenvolvimento deste sistema tem algumas semelhanças com o anterior, sendo constituído agora, apenas por duas partes: o Conversor e o Cliente. Tendo em conta que o sistema de transmissão é efectuado pelo SUIT, através do PLAYOUT SYSTEM, o cliente é o mesmo que foi desenvolvido anteriormente, dado esta tratar-se de uma comunicação *unicast*. Já o Conversor, para funcionar em *multicast*, sofreu várias alterações, mas a mais significativa, é sem dúvida a inclusão da estrutura *ip_mreq*, que permite então fazer o pedido para se juntar ao grupo *multicast*, tendo assim acesso aos pacotes enviados pelo PLAYOUT.

Foi então montado o esquema de rede apresentado na Figura 6.2, que é praticamente igual ao anterior, somente o Servidor foi substituído pelo PLAYOUT SYSTEM do SUIT, que é lançado da seguinte forma

mcsend

Esta aplicação pertence ao SUIT, e foi utilizada apenas para efectuar testes. Esta aplicação, dado tratar-se de uma transmissão *multicast*, envia vários pacotes RTP para o grupo do endereço *multicast*. O conversor, depois de fazer o pedido para se juntar ao grupo *multicast*, e este ser aceite, começa então a receber vários pacotes pertencentes a vários serviços de TV (CIF, SD e HD).

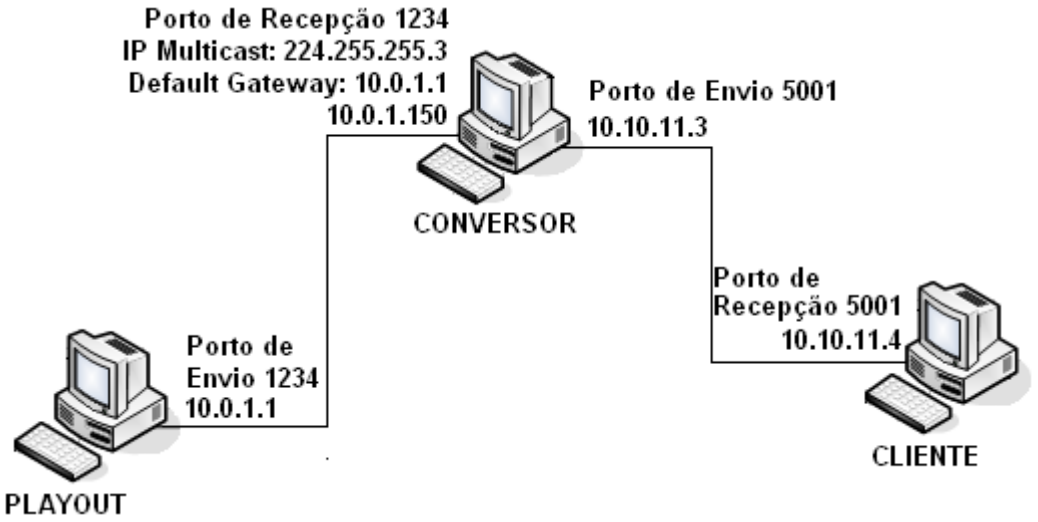

Figura 6.2 – Esquema de rede *Multicast*

O Conversor efectua a recepção dos pacotes, um a um, sendo estes então processados pelo conversor de vídeo. Se o tipo de serviço coincidir com o pretendido, os pacotes são reenviados para o Cliente, senão são descartados. O Conversor para a comunicação com o PLAYOT SYSTEM, necessita de saber qual o endereço IP *multicast* do grupo e o número de porto a escutar. Tem também como parâmetro de entrada, o Tipo de Serviço pretendido (CIF, SD, HD), este parâmetro é necessário para o conversor de vídeo. Para a transmissão do serviço pretendido é necessário o Endereço IP e o número de porto do Cliente.

./Conversor <Endereço IP Multicast> <Número de Porto Multicast> <Tipo de Ficheiro> <Endereço IP do Cliente> <Número de Porto do Cliente>

O Conversor, depois de realizar uma recepção com sucesso, emite uma mensagem, a informar o Endereço IP da máquina que enviou o pacote. São também impressas mensagens que dizem respeito ao conversor de vídeo, indicando se o pacote pertence ao nosso sistema, e identifica se o tipo de serviço é o serviço pretendido.

Já o Cliente, tal como o anterior, necessita apenas de saber o número de porto a escutar, para receber os pacotes provenientes do Conversor e a palavra chave utilizada na encriptação, para ser possível realizar a desencriptação.

./Cliente <Número de Porto do Conversor> <Palavra Chave>

Após a recepção, os pacotes são armazenados no disco, sendo também impressa uma mensagem a indicar o endereço IP do Conversor. Para este sistema foram realizados testes para o acesso do Cliente aos três serviços: CIF, SD, HD. Seguem-se os resultados, da transmissão efectuada

Para a análise deste sistema, foi utilizada a ferramenta de rede *Wireshark* [39] de modo a se realizar a captura dos pacotes, que chegavam ao Cliente para cada um dos serviços disponíveis, no sentido de se poder comparar o número de pacotes enviados, com o número de pacotes recebidos.

Após a análise da transmissão efectuada, para cada um dos serviços, verificou-se que não existiu qualquer perda de pacote durante a transmissão, ou seja todos os pacotes enviados pelo PLAYOUT SYSTEM, chegaram ao seu destino, o Cliente. É relevante referir que este resultado foi obtido sob condições favoráveis, ou seja, os pacotes transmitidos eram os únicos a circular na rede. É também importante referir que o vídeo utilizado, tinha apenas uma duração de cerca de 1 minuto.

No Anexo A.1.2, é possível visualizar imagens destes resultados, através das capturas realizadas pelo *Wireshark*, para os três tipos de serviço, CIF, SD e HD.

6.2 Encriptação

Após os resultados positivos, verificados no sistema de comunicação, faltava então o segundo objectivo do trabalho, garantir segurança da transmissão através da encriptação dos dados. Foram desenvolvidos dois sistemas de encriptação. O primeiro é uma simples, mas eficaz criptografia, bit a bit, utilizando a operação lógica XOR; O segundo é uma implementação do algoritmo AES.

6.2.1 Encriptação bit a bit

Este algoritmo consiste na encriptação dos dados através da operação lógica XOR com uma palavra-chave, escolhida pelo utilizador.

O algoritmo foi então implementado no sistema de comunicação *unicast*. Sendo incluído, apenas, no Servidor e no Cliente, isto porque, não há necessidade de realizar a operação de

desencriptação no Conversor, uma vez, que de seguida ter-se-ia de voltar a encriptar para realizar a transmissão para o Cliente.

A função de encriptação tem a seguinte forma:

```
void Encrypt(unsigned char *Buffer, int Filesize, char *key,
                    int EncryptKeyLength);
```

O argumento *Buffer* indica o ficheiro a encriptar, o *Filesize* é o tamanho do ficheiro a ser encriptado. Os parâmetros *key* e *EncryptKeyLength* dizem respeito à chave usada para encriptar e ao tamanho da chave respectivamente.

Dado tratar-se da operação lógica XOR, a desencriptação efectuada pelo Cliente é exactamente a mesma, ou seja necessita apenas, de saber qual a palavra-chave.

```
void Decrypt(unsigned char *Buffer, int Filesize, char *key,
                    int EncryptKeyLength);
```

O algoritmo de encriptação, inclui duas operações antes de efectivamente encriptar, a primeira é verificar se o pacote, pertence ao nosso sistema, ou seja os dois primeiros bytes são 0x80 e 0x60, e segundo, de modo a não alterar o cabeçalho RTP, para que seja possível o conversor efectuar a selecção do serviço pretendido, ou seja o início do ficheiro, para a encriptação encontra-se após o cabeçalho RTP, ou seja, 12 bytes depois.

Após a recepção do pacote, este foi comparado com o pacote enviado é verificou-se que não houve alteração nos dados. Foram também criados dois ficheiros extra, somente para testes, que representavam o pacote encriptado a enviar pelo Servidor e o pacote encriptado recebido pelo Cliente, também estes eram exactamente iguais. Comprovando assim que os pacotes que circularam na rede eram os encriptados, tornando assim impossível a utilização dos dados por utilizadores não autorizados.

Estes resultados, são visíveis através da análise das figuras do conteúdo do pacote antes e depois da encriptação e desencriptação, figuras estas que se encontram no Anexo A.2.1.

6.2.2 AES Rijndael

Este algoritmo consiste na implementação do algoritmo AES Rijndael, descrito em pormenor no Capítulo 5.

O algoritmo AES Rijndael é uma versão *open source*, disponibilizada pelos autores do algoritmo, a qual foi utilizada na implementação do algoritmo.

O algoritmo foi então implementado no sistema de comunicação *unicast.* Sendo incluído, no Servidor e no Cliente, isto porque, não existe necessidade de realizar a operação de desencriptação no Conversor, para de seguida voltar a encriptar para realizar a transmissão para o Cliente, dado que iria apenas aumentar a carga computacional.

A função de encriptação tem a seguinte forma.

void EncryptAES(**unsigned char** **Buffer*, **int** *Filesize*, **char** **key*);

O argumento *Buffer* indica o ficheiro a encriptar, o *Filesize* é o tamanho do ficheiro a ser encriptado. O parâmetro *key* diz respeito à chave usada para encriptar o ficheiro.

Dado se tratar de uma cifra de bloco, é necessário garantir que os dados a encriptar têm um tamanho múltiplo de 16, pois a encriptação ocorre em blocos de 16 bytes. Caso não seja múltiplo de 16, é então feito o *padding*, mas para sabermos onde começa o *padding*, foi necessário introduzir um código para identificar o início do *padding*, para então do lado do Cliente os bytes de *padding* poderem ser ignorados.

A desencriptação efectuada pelo Cliente é o inverso da encriptação, como foi visto no Capitulo 5.

void DecryptAES(**unsigned char** **Buffer*, **int** *Filesize*, **char** **key*);

O algoritmo de encriptação, inclui duas operações antes de efectivamente encriptar, a primeira é verificar se o pacote, pertence ao nosso sistema, ou seja os dois primeiros bytes são 0x80 e 0x60, e segundo, a encriptação é efectuada apenas aos dados, de modo a não alterar o cabeçalho RTP, para que seja possível o conversor efectuar a selecção do serviço pretendido, a encriptação inicia-se então após o cabeçalho RTP, ou seja, 12 bytes depois.

Após a recepção do pacote, este foi comparado com o pacote enviado é verificou-se que não houve alteração nos dados. Foram também criados dois ficheiros extra, somente para testes, que representavam o pacote encriptado a enviar pelo Servidor e o pacote encriptado recebido pelo Cliente, também estes eram exactamente iguais. Comprovando assim que os

pacotes que circularam na rede eram os encriptados, tornando assim impossível a utilização dos dados por utilizadores não autorizados.

Estes resultados, são visíveis através da análise das figuras do conteúdo do pacote antes e depois da encriptação e desencriptação, figuras estas que se encontram no Anexo A.2.2.

6.3 Conclusões

Neste capítulo, foi apresentado o sistema de comunicação implementado, quer para comunicação *Unicast*, quer para *Multicast*. Possibilitando assim a comunicação, para a transmissão dos pacotes. Foram também apresentados dois algoritmos de encriptação que foram incluídos no sistema de comunicação. No primeiro algoritmo, os dados a transmitir são encriptados através da operação lógica XOR, bit a bit dos dados com uma chave, escolhida pelo utilizador. O segundo diz respeito à implementação do algoritmo AES Rijndael.

Os resultados obtidos, demonstram que os dois algoritmos de encriptação efectivamente encriptaram os dados, deixando o cabeçalho RTP intacto, permitindo assim uma transmissão mais segura dos pacotes que circulam na rede. O primeiro algoritmo, não é infalível, a sua grande vantagem é basicamente a velocidade de processamento e a sua eficiência contra ataques de força bruta, tornando a sua desencriptação por este método impraticável.

O segundo algoritmo, tratando-se de uma implementação do AES Rijndael, oferece mais garantias de segurança, pois é praticamente impossível a sua desencriptação, independentemente do método utilizado, sem se saber a chave utilizada para encriptar. Mas, para conseguir este nível de segurança, é necessário uma maior capacidade de computação, bem como mais tempo de processamento e apesar de nos testes realizados não se notar esse aumento, nem existir perda de pacotes, é possível que com um maior tráfego se venha efectivamente a observar a uma pequena perda de pacotes, não sendo talvez o algoritmo ideal para a transmissão em tempo real de vídeo.

Capítulo 7

Considerações Finais

A procura de serviços multimédia e transferências exaustivas de dados irá estimular o desenvolvimento de novos sistemas de comunicações, e por isso o desenvolvimento de mecanismos que nos permitam aumentar a eficiência e segurança da sua utilização, serão determinantes para concretizar os objectivos das futuras gerações dos sistemas de comunicações.

Esta dissertação concentrou-se numa abordagem à interface *sockets,* bem como a protocolos de encriptação existentes, que permitam uma transmissão mais segura através da Internet.

No trabalho realizado no âmbito desta dissertação estudaram-se e exploraram-se os protocolos de encriptação e a estrutura da interface *socket,* direccionados para transportar tráfego baseado em IP.

Fundamentalmente, neste trabalho procurou-se estudar e melhorar o desempenho do sistema de comunicações, pela definição e caracterização de algoritmos, quer dos *sockets*, quer de encriptação, sendo de seguida apontadas as principais conclusões resultantes do trabalho desenvolvido, bem como algumas perspectivas de trabalho futuro.

7.1 Principais Conclusões

No sexto capítulo, propôs-se um algoritmo para o sistema de comunicação e encriptação, para a transmissão de vídeo, de forma segura. Os módulos desenvolvidos para o sistema de

comunicação mostraram ser eficientes para os serviços de vídeo, utilizados pelo SUIT, e com baixo consumo de recursos computacionais quanto ao seu processamento.

De uma forma geral, demonstrou-se, ao longo desta dissertação, que o sistema desenvolvido, com a inclusão do Sistema de Conversão de Vídeo é uma boa solução a considerar pelos clientes, que possuem poucos recursos computacionais e que pretendam aderir ao nosso serviço de transmissão. Dado a maior carga de processamento ocorrer no Conversor e não no Cliente.

Para além do sistema de comunicação, este trabalho apresenta também, dois algoritmos de encriptação para a protecção dos conteúdos que se encontram a ser transmitidos na rede.

O primeiro algoritmo tem como grande vantagem a velocidade de processamento e a sua eficiência contra ataques de força bruta, permitindo assim uma transmissão em tempo real com alguma segurança. O segundo algoritmo consegue uma maior segurança, pois trata-se de uma implementação do AES Rijndael, tornando a sua desencriptação praticamente impossível, se a chave utilizada para encriptar não for conhecida. Apesar de não ser o algoritmo ideal para a transmissão em tempo real de vídeo, devido à sua carga computacional, nos testes realizados, não existiu qualquer perda de informação.

Finalmente, pensamos que com o trabalho realizado no âmbito desta dissertação se deram alguns passos promissores na definição da estrutura para a transmissão de vídeo através da Internet, com recurso à interface *sockets* e aos protocolos de encriptação.

7.2 Perspectivas de Trabalho Futuro

No âmbito deste trabalho existem ainda vários aspectos que podem ser aprofundados. Não se pretende enumerar todos os assuntos que merecem uma investigação mais cuidada, mas sim, apresentar algumas preferências que de momento parecem ser merecedoras de dedicação. Destacam-se, por isso, os seguintes tópicos:

- Desenvolvimento de um sistema de comunicação, onde o cliente efectua o pedido ao conversor do serviço que pretende, bem como acerta a chave a utilizar na encriptação, permitindo deste modo a utilização do sistema por mais clientes.

- Desenvolvimento de algoritmos mais eficientes e seguros do sistema de comunicação, recorrendo, não só mas principalmente, aos protocolos *Secure Sockets Layer* (SSL) e *Transport Layer Security* (TLS).
- Desenvolvimento de algoritmos mais eficientes de encriptação, especificamente desenvolvidos para a transmissão de vídeo em tempo real.

Apêndice A

Demonstração dos Resultados

Este anexo é um complemento ao Capitulo 6, dado que é feita uma demonstração dos resultados obtidos do trabalho desenvolvido recorrendo à apresentação de algumas imagens, quer para o sistema de comunicação, como para os algoritmos de encriptação.

A.1 Sistema de Comunicação

Nesta secção é apresentada uma análise dos resultados, através da apresentação de algumas imagens para a comunicação *Unicast* e *Multicast.*

A.1.1 Comunicação *Unicast*

O Conversor lida com a recepção do pacote, como mostra a Figura A.1, onde se podem observar as mensagens do tratamento, bem como mensagens a identificarem que o pacote pertence ao nosso sistema e a identificarem o tipo de serviço autorizado.

Figura A.1 – Tratamento do pacote CIF no Conversor

Através da análise da Figura A.2 e da Figura A.3, podemos verificar que o pacote não se perdeu, nem o seu conteúdo foi alterado, garantindo assim a integridade do pacote.

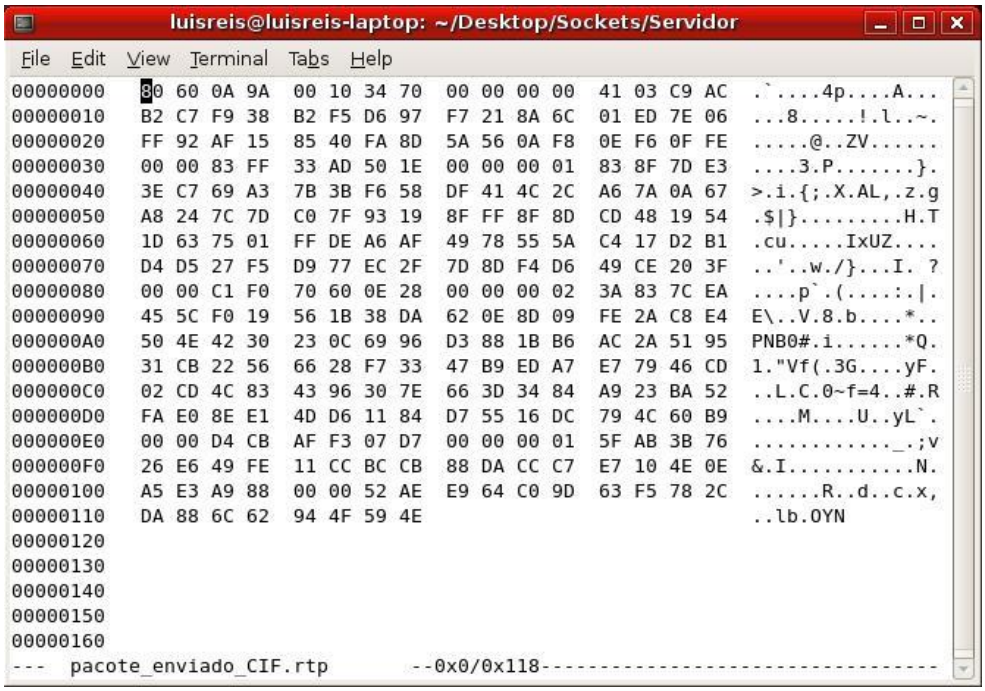

Figura A.2 – Pacote enviado do Servidor para o Conversor

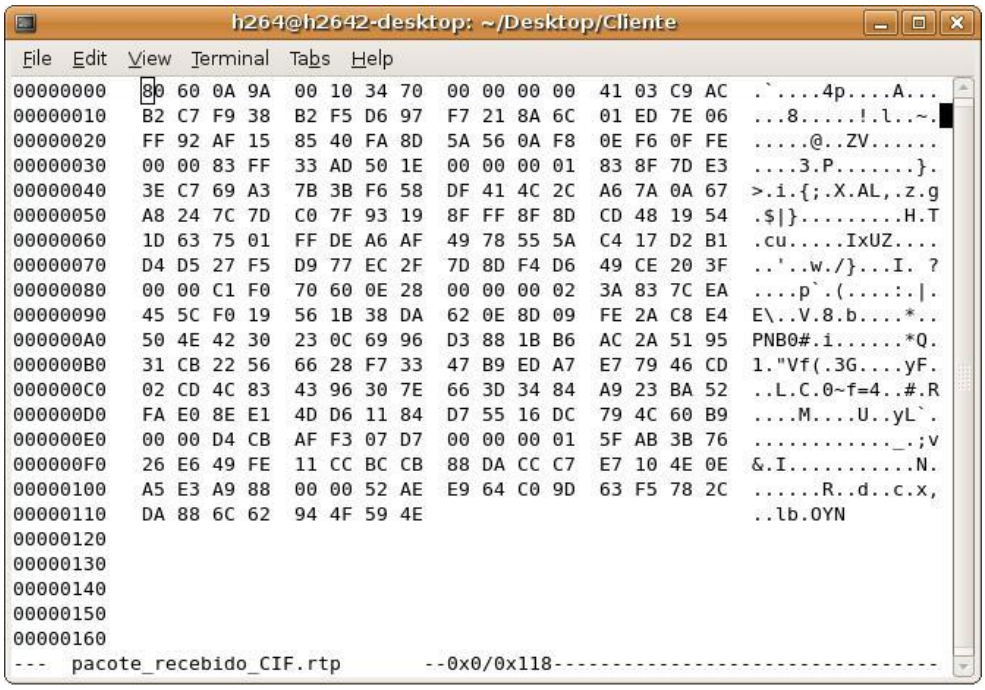

Figura A.3 – Pacote recebido pelo Cliente do Conversor

A.1.2 Comunicação *Multicast*

A Figura A.4 mostra o lançamento dos serviços, a partir do PLAYOUT, foram enviados 10177 pacotes RTP correspondentes ao serviço CIF, 11691 correspondentes ao serviço SD e 22621 pacotes correspondentes ao serviço HD. As capturas que se apresentam de seguida foram, efectuadas no Cliente. Para a realização destes testes foi utilizada uma sequência de vídeo com cerca de 60 segundos de duração.

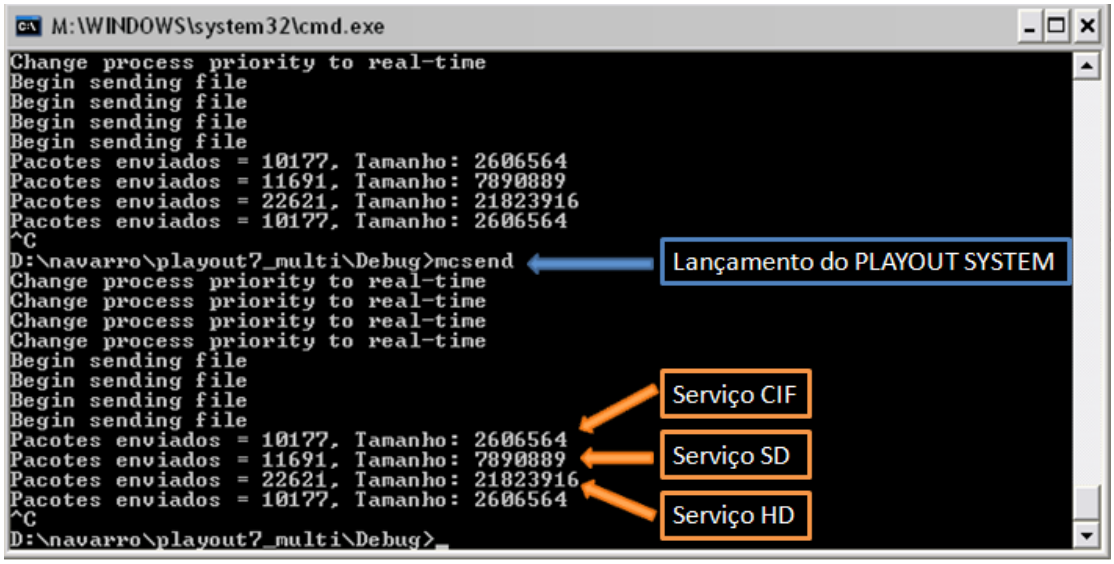

Figura A.4 – Envio de pacotes RTP no PLAYOUT

A.1.2.1. *Serviço CIF*

A Figura A.5 mostra a captura realizada pelo *Wireshark* dos pacotes enviados pelo PLAYOUT SYSTEM.

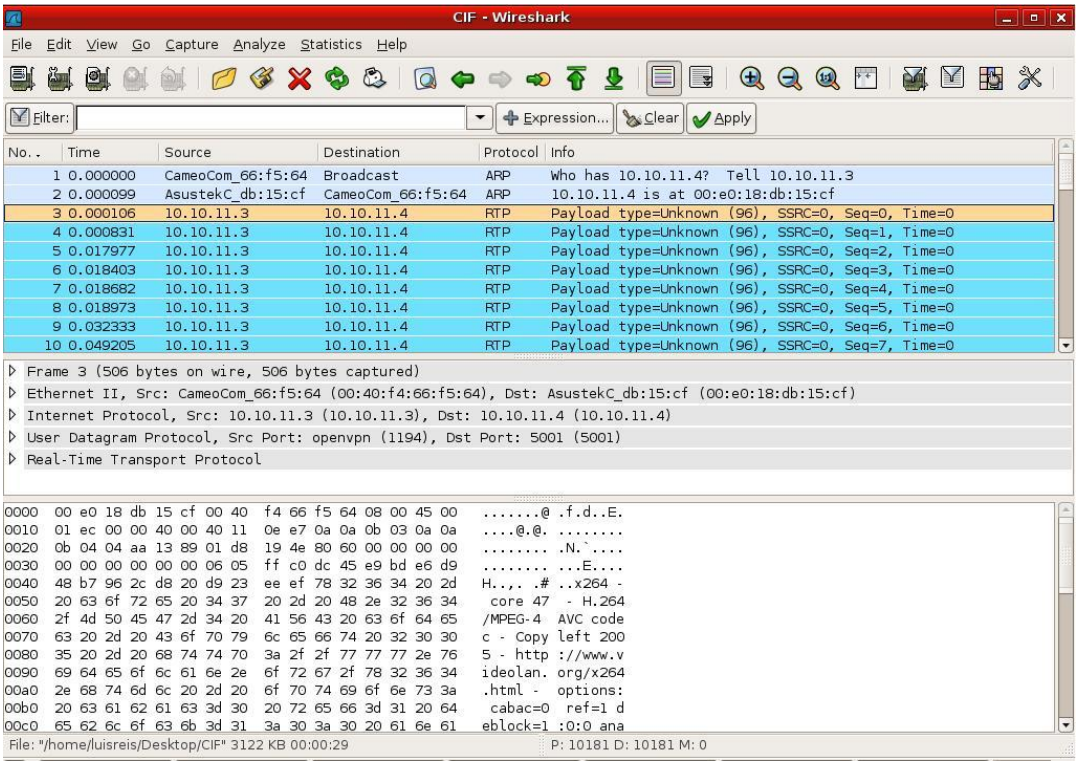

Figura A.5 – Captura dos pacotes relativos ao serviço CIF

Após a captura foi realizada uma análise aos pacotes, também com o *Wireshark*. Os resultados são apresentados na Figura A.6.

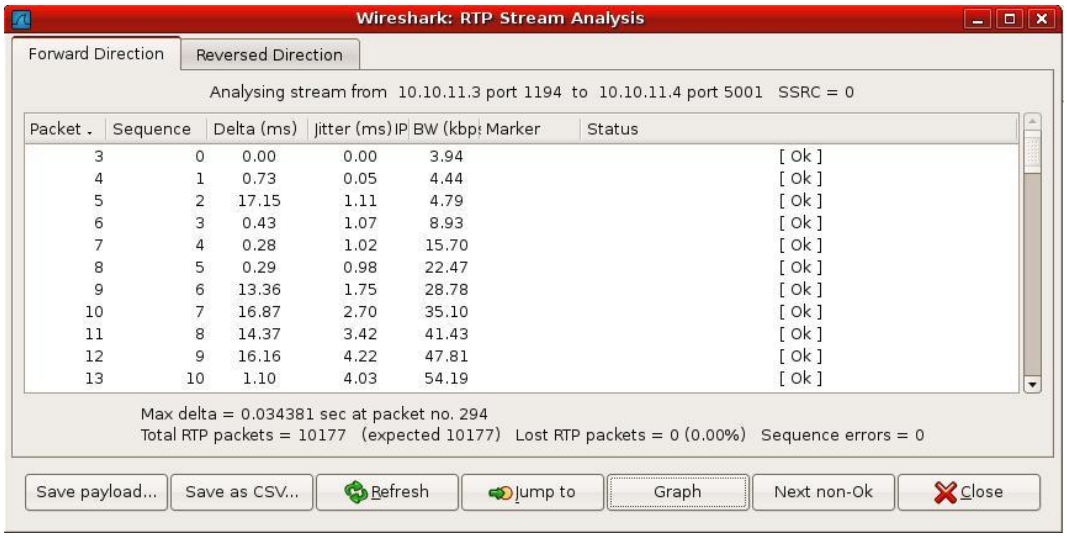

Figura A.6 – Análise do serviço CIF

O número de pacotes recebidos pelo Cliente é exactamente o mesmo que foi enviado pelo PLAYOUT, não existindo qualquer perda de pacotes.

A.1.2.2. *Serviço SD*

A captura seguinte, Figura A.7, foi realizada para o serviço SD, fazendo assim a captura dos pacotes enviados pelo PLAYOUT para o cliente.

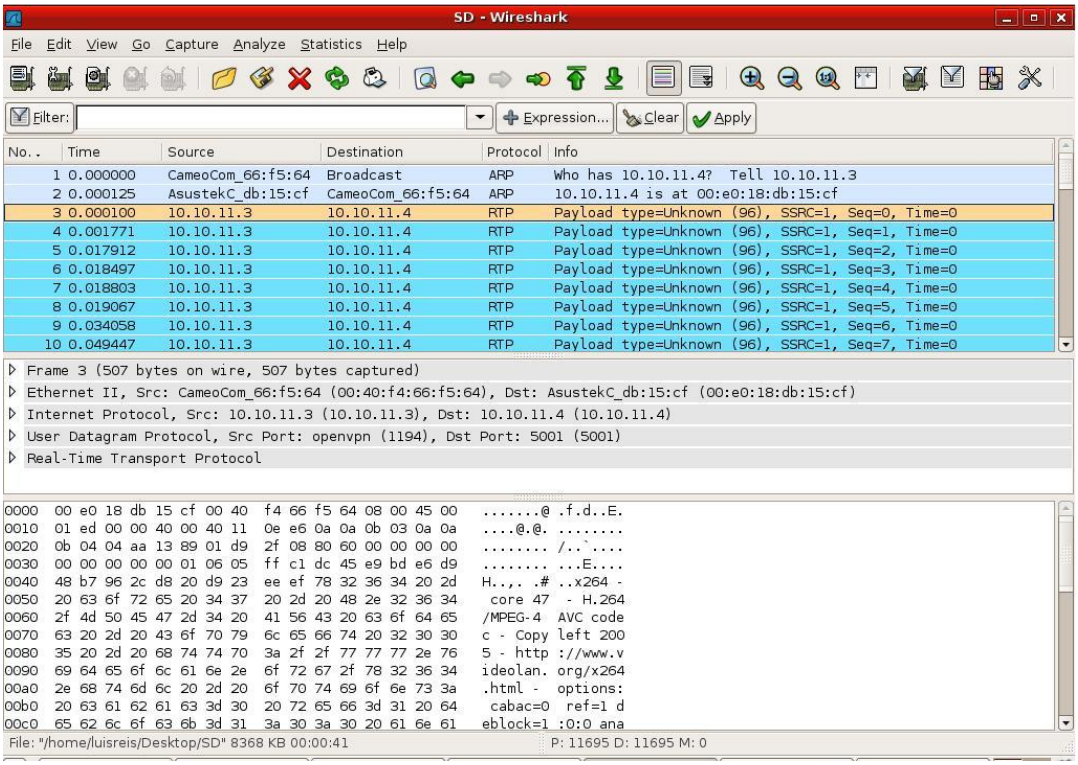

Figura A.7 – Captura dos pacotes relativos ao serviço SD

Do mesmo modo, foi realizada a análise aos pacotes recebidos, mostrada pela Figura A.8.

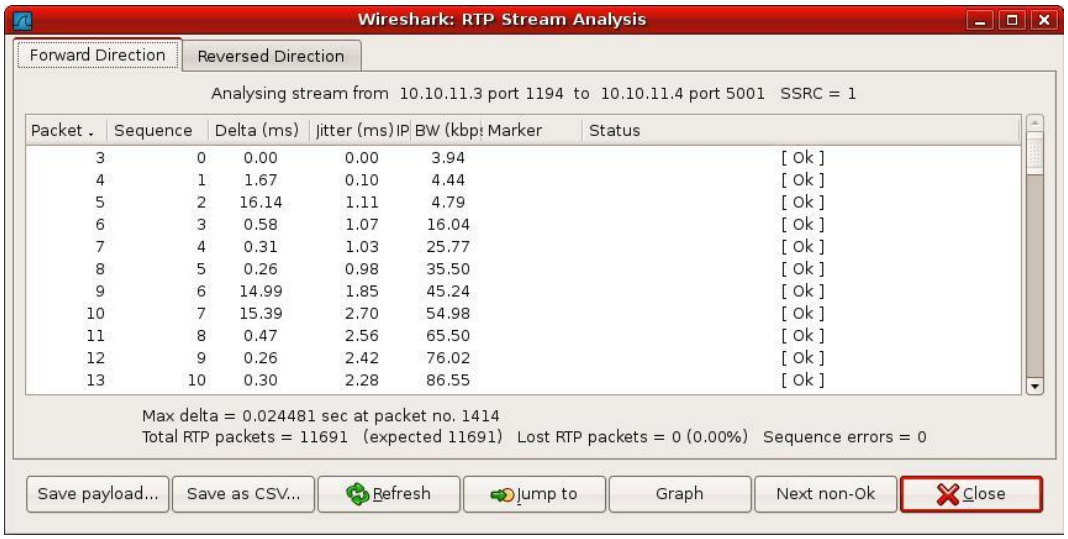

Figura A.8 – Análise do serviço SD

Novamente todos os pacotes enviados pelo PLAYOUT foram recebidos pelo Cliente.

A.1.2.3. *Serviço HD*

Foi realizada nova captura, desta vez para o serviço HD, obtendo assim a captura que a Figura A.9 mostra.

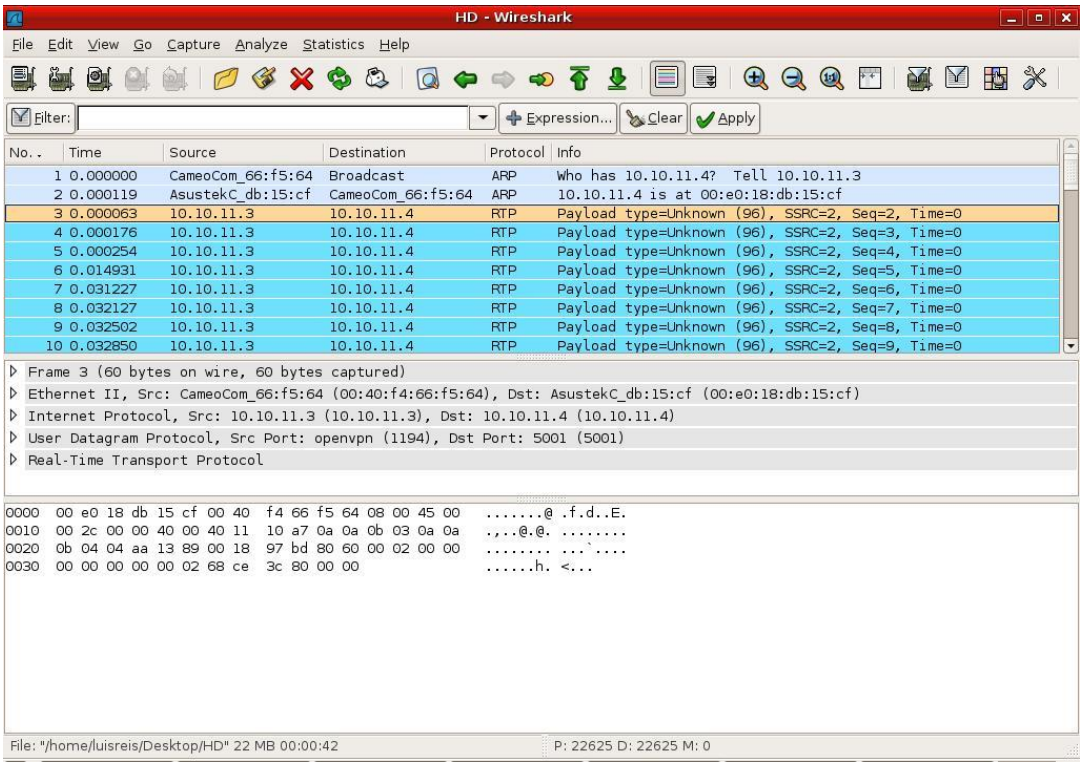

Figura A.9 – Captura dos pacotes relativos ao serviço HD

A Figura A.10 apresenta a análise realizada aos pacotes recebidos pelo Cliente, e tal como nos outros serviços o número de pacotes recebidos coincide com o número de pacotes enviados a partir do PLAYOUT.

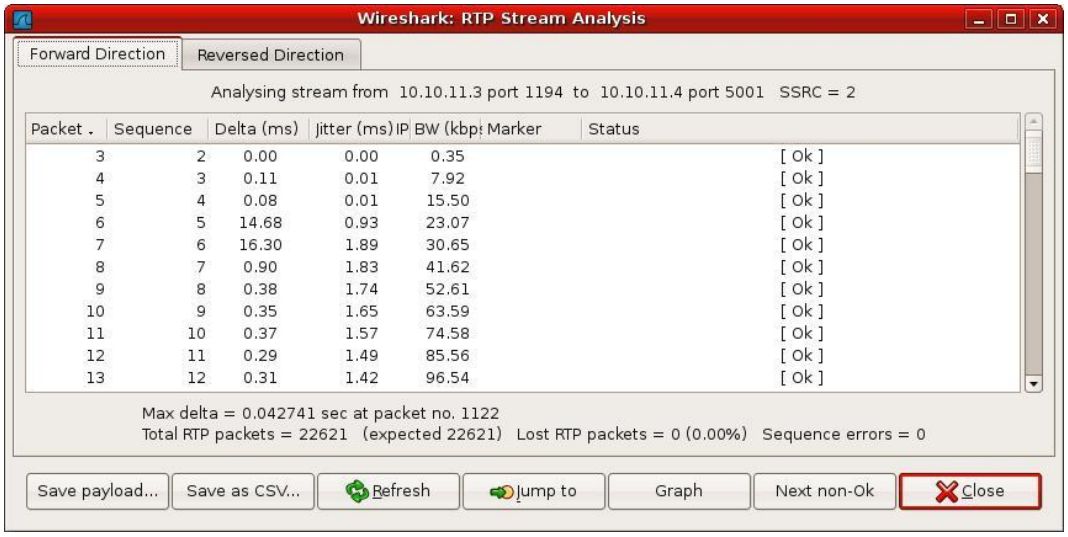

Figura A.10 – Análise do serviço HD

Podemos, então comprovar que o Sistema de Comunicação apresenta um comportamento linear, isto é, o seu comportamento não é afectado com o aumento do número de serviços que se encontram a ser enviados para a rede, garantindo que todos os pacotes são recebidos pelo Cliente.

A.2 Encriptação

Nesta secção, são apresentadas imagens do conteúdo do pacote antes e depois da encriptação, bem como antes e depois da desencriptação para os dois algoritmos de encriptação implementados.

A.2.1 Encriptação bit a bit

Como se pode ver, o conteúdo da Figura A.12, está completamente diferente do conteúdo original do pacote apresentado pela Figura A.11, à excepção dos 12 primeiros bytes, correspondentes ao cabeçalho RTP.

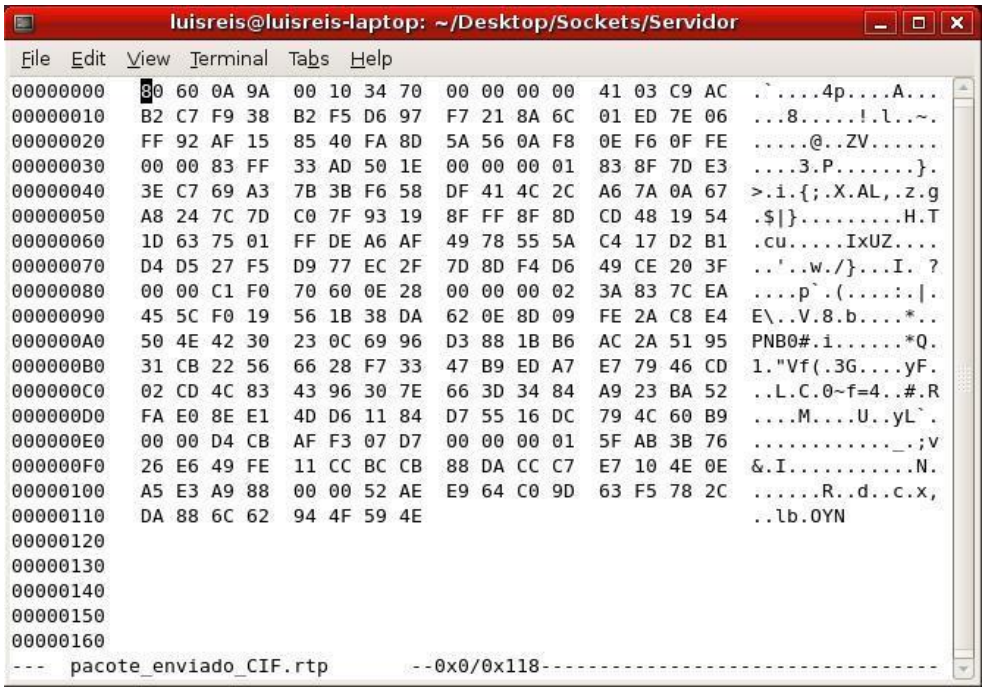

Figura A.11 – Pacote antes da encriptação no Servidor

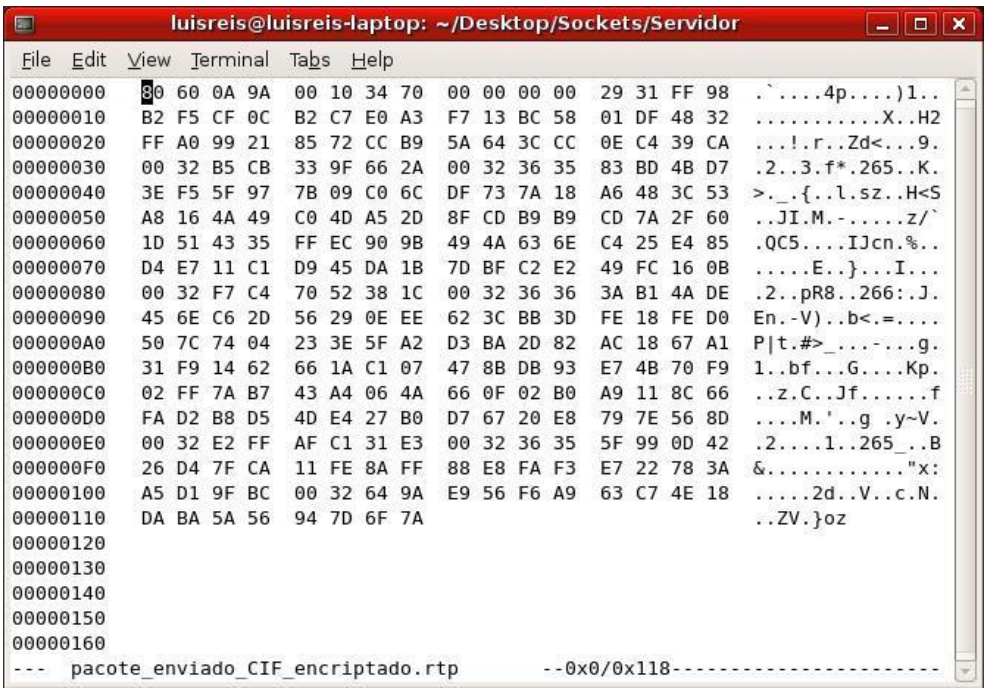

Figura A.12 – Pacote depois da encriptação no Servidor

Da mesma forma, o conteúdo do pacote, como mostra a Figura A.12 é igual ao da Figura A.13, continuando o pacote encriptado.

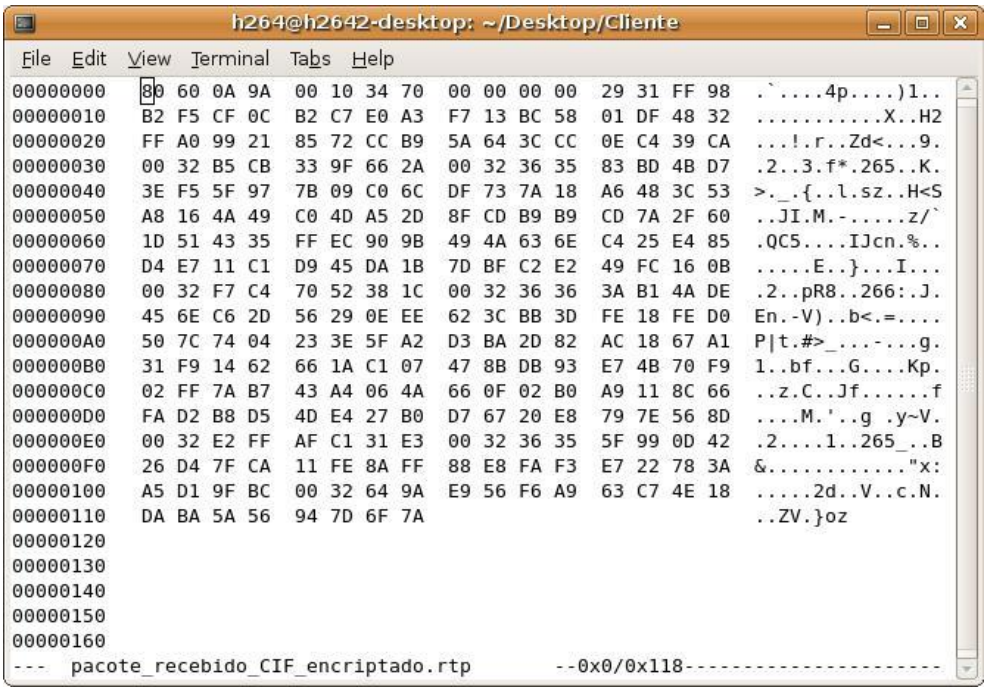

Figura A.13 – Pacote antes da desencriptação no Cliente

A Figura A.14 apresenta o conteúdo do pacote após a desencriptação, sendo este igual ao pacote original, apresentado pela Figura A.11.

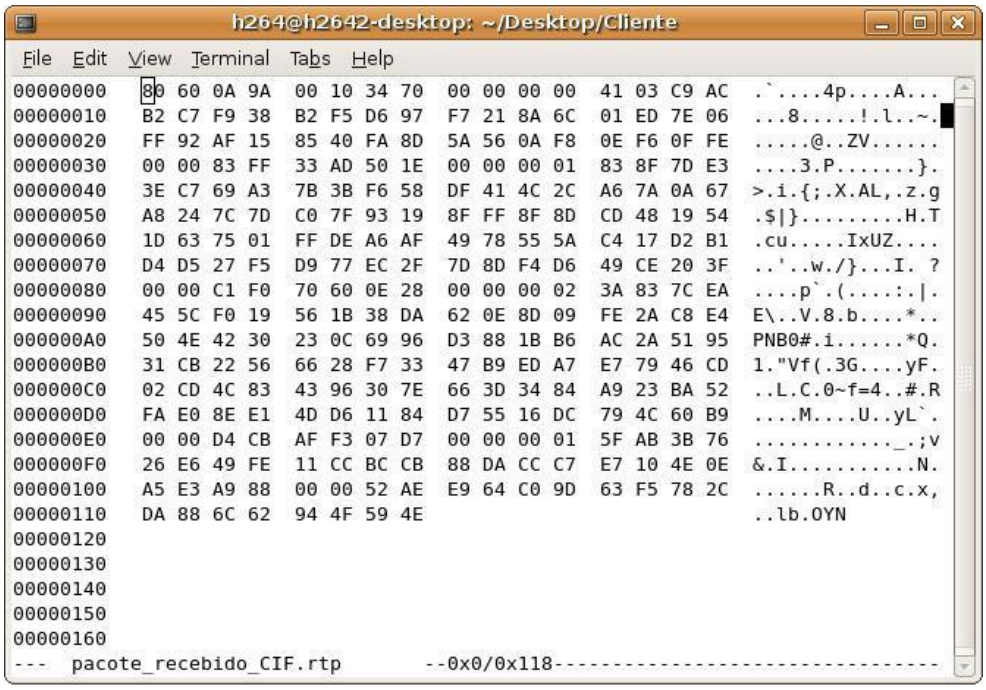

Figura A.14 – Pacote após a desencriptação no Cliente

Como se pode comprovar, através da análise das figuras, o pacote transmitido na rede está encriptado, garantindo a segurança do pacote, sendo então desencriptado no Cliente, podendo este então aceder às informações.

A.2.2 AES Rijndael

Também com este algoritmo, foram obtidos os mesmos resultados do algoritmo anterior. O pacote utilizado foi o mesmo, apresentado pela Figura A.11. Como se pode ver o conteúdo da Figura A.15, é completamente diferente do conteúdo original do pacote, à excepção dos 12 primeiros bytes, correspondentes ao cabeçalho RTP. Também se pode observar, que foi necessário utilizar o *padding*, foram acrescentados 4 bytes ao final do pacote, de modo a que este fosse múltiplo de 16. Os três primeiros bytes inseridos representam o código "A28369", sendo então acrescentados zeros até que o tamanho seja múltiplo de 16, neste caso foi necessário introduzir apenas 1 byte, além do código.

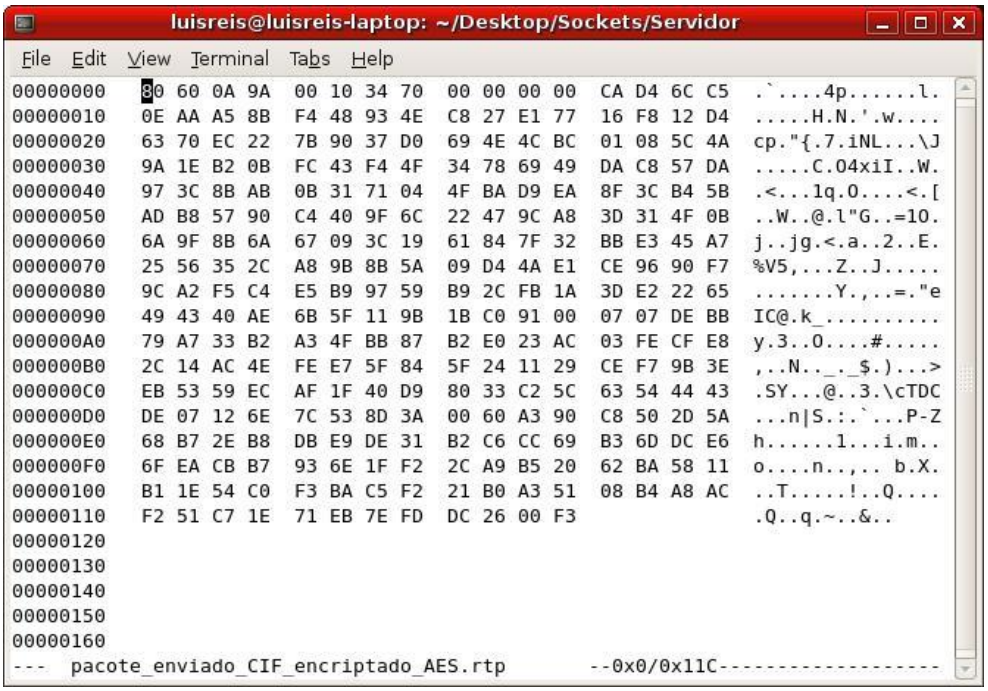

Figura A.15 – Pacote depois da encriptação AES Rijndael no Servidor

Da mesma forma, o conteúdo do pacote, como mostra a Figura A.15 é igual ao da Figura A.16, continuando o pacote encriptado, após a recepção no Cliente.

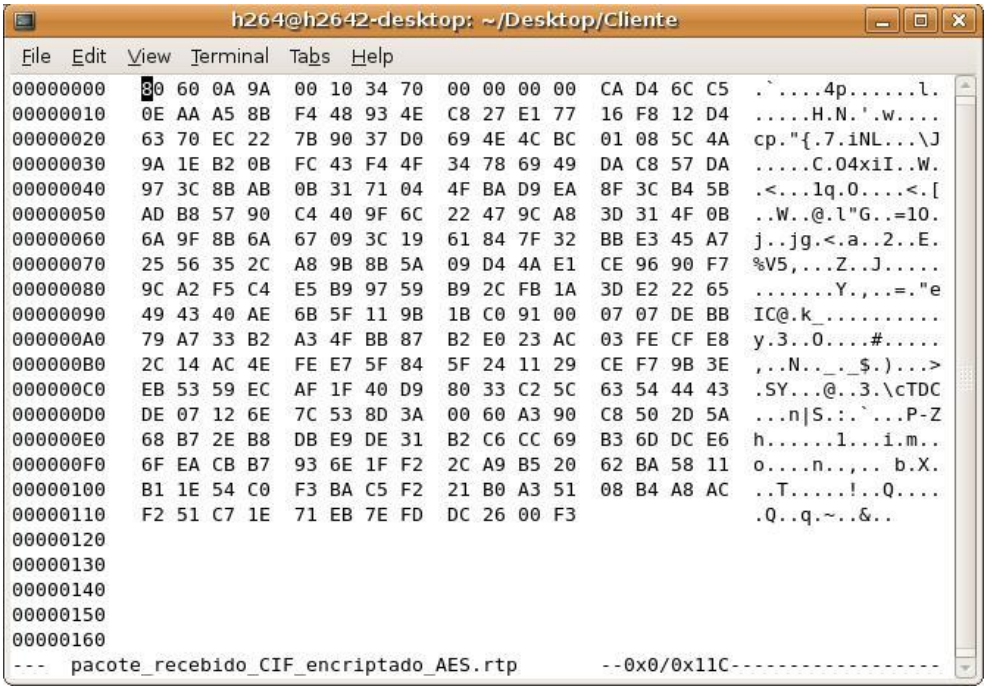

Figura A.16 – Pacote antes da desencriptação AES Rijndael no Cliente

A Figura A.17, apresenta o conteúdo do pacote após a desencriptação, que contém o pacote original com o código de identificação e os bytes de *padding* no final.

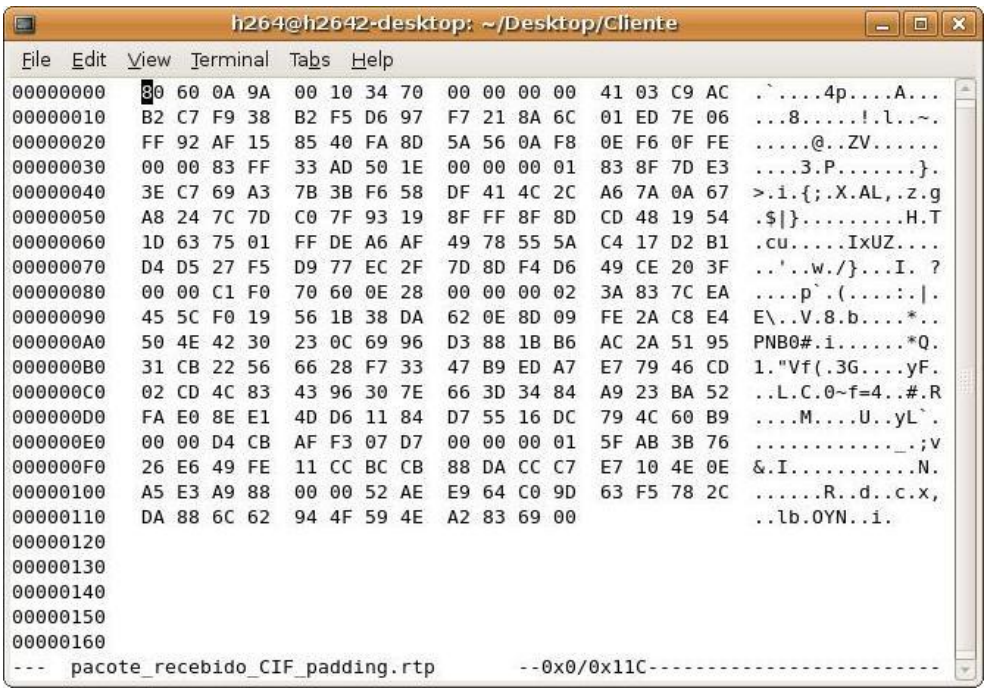

Figura A.17 – Pacote após a desencriptação AES Rijndael no Cliente

A Figura A.18, apresenta o conteúdo do pacote após a eliminação do código de identificação e dos bytes de *padding*, sendo este igual ao pacote original, apresentado pela Figura A.11. Como se pode ver, em relação ao pacote desencriptado, foram eliminados os bytes de *padding*.

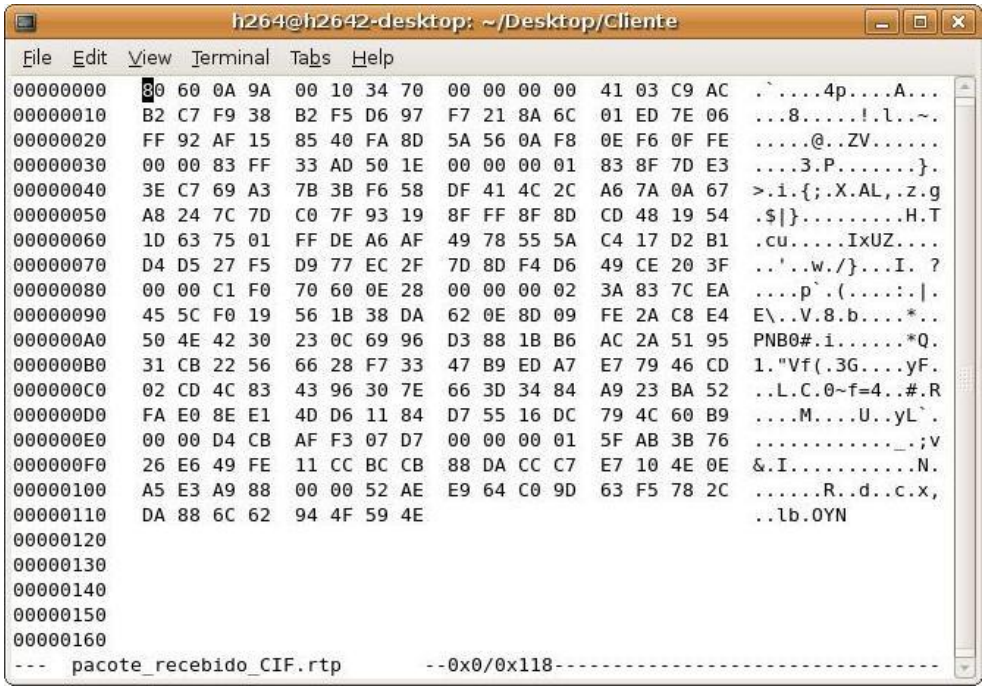

Figura A.18 – Pacote após a eliminação dos bytes de *padding* no Cliente

Tal como o algoritmo anterior, também com este se pode comprovar, através da análise das figuras, que o pacote transmitido na rede está encriptado, garantindo a segurança do pacote, sendo então desencriptado no Cliente, procedendo à eliminação dos bytes de *padding*, caso seja necessário, para então ser possível o acesso aos dados.

Apêndice B

Plataforma e Ferramentas de Desenvolvimento

Este anexo tem como principal objectivo, apresentar a plataforma utilizada para implementação do trabalho, bem como as ferramentas utilizadas para o desenvolvimento e análise do trabalho.

A plataforma de hardware utilizada foi um mini PC, que tinha instalado o sistema operativo *open source*, *Ubuntu 7.04*. Todo o projecto foi desenvolvido em linguagem C, compilando a mesma com o *GNU Compiler Collection* (GCC). Foi utilizado um analisador de protocolos de rede, o *Wireshark*, de modo a podermos realizar uma análise mais detalhada, do sistema implementado. Foi também utilizado o *Hexedit*, que permitia a leitura do conteúdo do pacote, permitindo deste modo a sua análise.

B.1 Plataforma de Hardware – Mini PC

A plataforma utilizada como base deste projecto foi o mini PC apresentado na Figura B.1

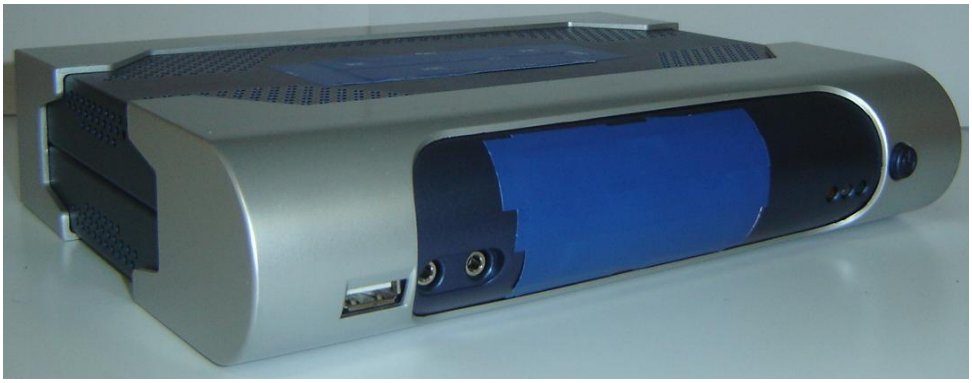

Figura B.1 – Mini-PC

Esta plataforma apresenta dimensões reduzidas (23x18x5 cm) e possui três placas de rede Ethernet (Lan1, Lan2, Lan3), como se pode ver na Figura B.2, principal razão pela qual foi escolhida. Torna-se importante a presença deste tipo de característica dado que o sistema a ser implementado, irá funcionar como um *gateway*. Para além das três placas de rede este computador possui ainda três entradas USB, que podem ser utilizas para a expansão da plataforma.

Figura B.2 – Face Posterior da plataforma

Para além destas características, a plataforma possui um processador Via Erza a 800 MHz, uma memória de 256 Gbytes e um disco de 60 Gbytes. Possui também uma placa gráfica e uma placa de som, ambas integradas na *motherboard* da plataforma.

B.2 GCC – *GNU Compiler Collection*

O GCC é um projecto iniciado por Richard Stallman em 1984. O objectivo do GNU (*GNU's Not Unix*) era dar aos utilizadores liberdade. Por isso, era necessário utilizar termos de distribuição que impediria o software GNU, de transformar-se em software proprietário. O método utilizado foi designado de "*Copyleft*", que ao contrário do *Copyright* tem a finalidade de manter o software livre, em vez de ser um meio para privatizar o software. A ideia principal do *copyleft* é permitir qualquer pessoa executar, copiar, modificar e distribuir versões modificadas do programa, mas não permite adicionar restrições próprias. Assim, os fundamentos que definem software livre são garantidos a qualquer pessoa que possua uma cópia, pois eles se tornam direitos inalienáveis.

A partir de 1984, *Stallman* e outros programadores desenvolveram os módulos principais do sistema operativo, como é o caso do compilador C e dos respectivos editores de texto. Em 1990, o sistema GNU estava quase completo, mas faltava uma parte muito importante, o *kernel*. Começaram por implementar o próprio *kernel* como uma colecção de processos a

serem executados em cima do *Mach*. O *Mach* é um *microkernel* desenvolvido na *Carnegie Mellon University*, e depois na *University of Utah*. O desenvolvimento do GNU *Hurd*, como foi designado, foi atrasado pois o lançamento do *Mach* como software livre, estava constantemente a ser adiado.

Felizmente, outro *kernel* ficou disponível. Em 1991, Linus Torvalds desenvolveu um *kernel* compatível com Unix que designou de Linux. Por volta de 1992, a combinação do Linux com o sistema GNU resultou num sistema operativo livre. Esta versão foi designada de sistema GNU/Linux, para expressar a sua composição como uma combinação do sistema GNU com o *kernel* do Linux [40].

Actualmente o sistema *GNU* vem integrado na maioria das distribuições de Linux como é o caso do *Ubuntu* utilizado no desenvolvimento deste trabalho.

B.3 *Wireshark*

O *Wireshark*, anteriormente conhecido por *Ethereal*, é um analisador de protocolos de rede, criado e desenvolvido por Gerald Combs. O *Wireshark* funciona em vários sistemas operativos, devido à utilização da plataforma *GTK+ widget toolkit* e foi lançado sob os termos da *GNU General Public License*, sendo um software livre.

Este programa permite a resolução de problemas na rede, bem como a análise, o desenvolvimento de software e de protocolos de comunicações. O desenvolvimento deste programa deve-se também às contribuições de especialistas de redes de todo o mundo. É a continuação de um projecto que teve início em 1998.

O *Wireshark* é um dos analisadores de protocolos de rede mais utilizado no mundo e é de facto um dos mais importantes ao nível da indústria e do ensino [39].

Esta aplicação permite a análise dos conteúdos transmitidos, utilizando o protocolo RTP (Figura B.3).

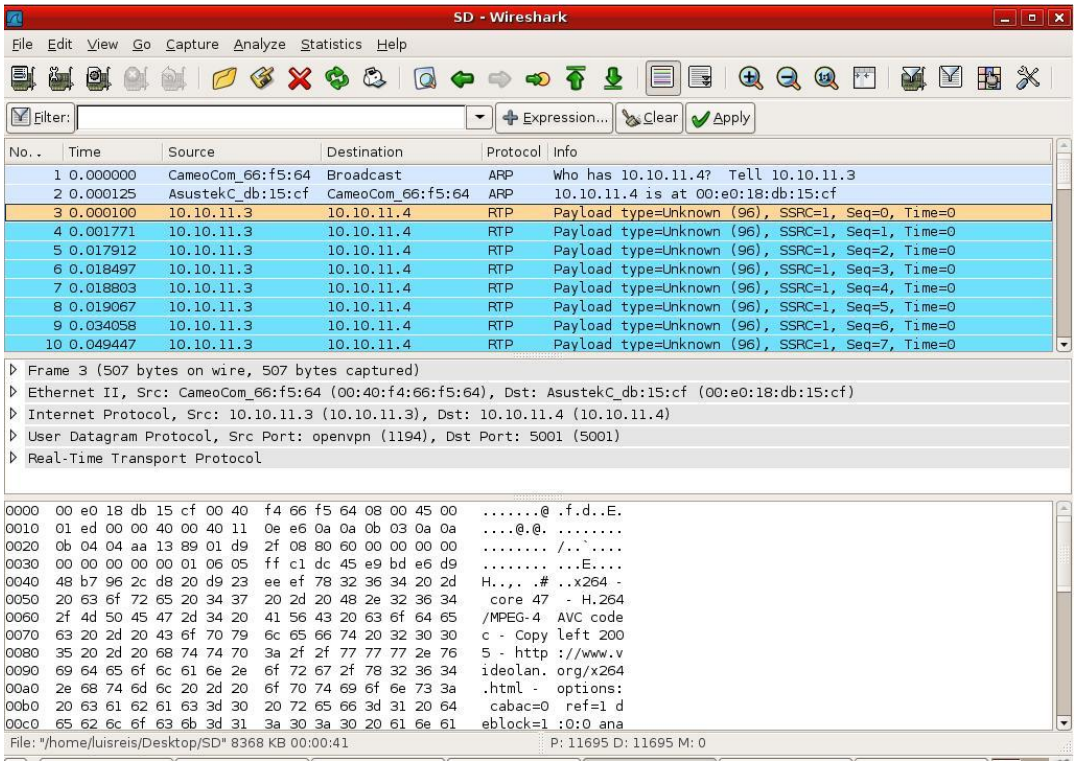

Figura B.3 – Aplicação *Wireshark*

B.4 *HexEdit*

O *HexEdit* é uma aplicação de leitura e edição de ficheiros em formato hexadecimal ou ASCII. Esta aplicação é um tipo de programa que permite ao utilizador visualizar e editar o conteúdo exacto dos ficheiros, ou seja, ao nível do *byte*, em contraste com as interpretações de nível mais elevado do mesmo conteúdo que são fornecidas por outros programas. O *Hexedit* permite ainda cortar, copiar, colar, inserir e eliminar qualquer informação [38].

Esta aplicação foi utilizada para a verificar se o conteúdo do pacote enviado para o cliente sofre qualquer alteração durante a sua transmissão.

Referências Bibliográficas

[1] *Intenet World Stats*, http://www.internetworldstats.com/stats.htm.

[2] Comer, D.E., *Internetworking with TCP/IP - Principles, Protocols, and Architectures*, 1, Fourth, Upper Saddle River, N.J., Prentice Hall, 2000.

[3] Cisco Systems Inc., *Internetworking Technologies Handbook*, Fourth Edition, Cisco Press, 2004.

[4] Blank, A.G., *TCP/IP Foundations*, San Francisco, SYBEX, 2004.

[5] Blank, A.G., *TCP/IP Jumpstart: Internet Protocol Basics*, Second Edition, San Francisco, Sybex, 2002.

[6] Rhee, M.Y., *Internet Security: Cryptographic Principles, Algorithms and Protocols*, Chichester, West Sussex, England, J. Wiley, 2003.

[7] Clark, M.P., *Data networks, IP and the Internet : Protocols, Design and Operation*, Chichester, England, Wiley, 2003.

[8] Groth, D. and T. Skandier, *Network+ Study Guide*, Fourth Edition, Sybex, 2005.

[9] Peterson, L.L. and B.S. Davie, *Computer Networks: A Systems Approach*, Third, Boston, Morgan Kaufmann Publishers, 2003.

[10] Simpson, W., *The Point-to-Point Protocol (PPP)*, July 1994.

[11] Simpson, W., *PPP in HDLC-like Framing*, July 1994.

[12] Cisco Systems Inc., *Guide to ATM Technology*, Cisco Press, 2000.

[13] Stevens, W.R., B. Fenner, and A.M. Rudoff, *UNIX Network Programming - The Sockets Networking API*, 1, Third Edition, Boston, MA Addison-Wesley, 2004.

[14] Reis, J., "Técnicas de Cross Layer para Optimização do Desempenho em Comunicações Móveis," Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, Aveiro, 2005.

[15] Stevens, W.R. and G.R. Wright, *TCP/IP Illustrated - The Protocols*, Volume 1, Reading, Mass, Addison-Wesley Pub. Co., 1994.

[16] Postel, J., *Internet Protocol*, September 1981.

[17] Almquist, P., *Type of Service in the Internet Protocol Suite*, July 1992.

[18] Fuller, V., et al., *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*, September 1993.

[19] Plummer, D.C., *An Ethernet Address Resolution Protocol*, November 1982.

[20] Finlayson, R., et al., *A Reverse Address Resolution Protocol*, June 1984.

[21] Deering, S. and R. Hinden, *Internet Protocol, Version 6 (IPv6)*, December 1998.

[22] Hinden, R. and S. Deering, *IP Version 6 Addressing Architecture*, July 1998.

[23] Postel, J., *Internet Control Message Protocol*, September 1981.

[24] Postel, J., *Transmission Control Protocol*, September 1981.

[25] Postel, J., *User Datagram Protocol*, August 1980.

[26] Perkins, C., *RTP : Audio and Video for the Internet*, Boston, Addison-Wesley, 2003.

[27] Donahoo, M.J. and K.L. Calvert, *TCP/IP Sockets in C: Practical Guide for Programmers*, San Francisco, Morgan Kaufmann Publishers, 2001.

[28] Stevens, W.R. and S.A. Rago, *Advanced Programming in the UNIX Environment*, Second Edition, Addison Wesley Professional, 2005.

[29] Comer, D. and D.L. Stevens, *Internetworking with TCP/IP - Client-server Programming and Applications*, Volume 3, BSD Sockets Version Second, Upper Saddle River, N.J., Prentice Hall, 1996.

[30] Gay, W.W., *Linux Socket Programming by Example*, 2000.

[31] Stevens, W.R. and G.R. Wright, *TCP/IP Illustrated - The Implementation*, Volume 2, Reading, Mass., Addison-Wesley Pub. Co., 1995.

[32] Mockapetris, P., *Domain Names: Concepts and Facilities*, November 1987.

[33] Mockapetris, P., *Domain Names: Implementation and Specification*, November 1987.

[34] Chandra, P., *Bulletproof Wireless Security*, Elsevier Inc., 2005.

[35] Rivest, R.L., et al., *The RC6™ Block Cipher*. 1998.

[36] FIPS Publication 197, *Announcing the Advanced Encryption Standard (AES)*, US DoC/NIST, 2001.

[37] *SUIT - Scalable, Ultra-fast and Interoperable Interactive Television*, http://suit.av.it.pt.

[38] *Hexedit*, http://linux.die.net/man/1/hexedit.

[39] *Wireshark*, http://www.wireshark.org.

[40] *GNU Operating System*, http://www.gnu.org/.