Department of Computer Science and Engineering
Institute of Technology, BHU

CS-4403
Major Project Phase II

A Project Report on

Implementation of a simulated TCP/IP Stack using design Patterns

Submitted in partial fullfillment of degree Bachelor of Technology in Computer Science and Enginnering by:

Neeraj Kumar
06000044
B.Tech (4th year)

Contents

Abstract

In 1980's, Protocol Stack implementation was a difficult task. The OSI 7 layers reference model needed to be violated to ensure efficient execution.

Earlier implementations made at that time and later used a process based approach, where each layer is represented by a process or thread. Comer and Stevens [2], demonstrated such an implementation for educational purposes with their Xinu Operating system.

However, software developers at that time realized the possibility of reuse and a better design for the TCP/IP stack. Huni and others [1] in 1995 proposed a Conduit+ framework that uses Design Patterns to make the network stack easier to learn and use.

In this project, I tried to implement and test the design presented by Huni [1]. As TCP/IP stack is component of the Kernel, I simulated the packet flow in the user space by using the Raw Sockets provided by BSD Sockets API.

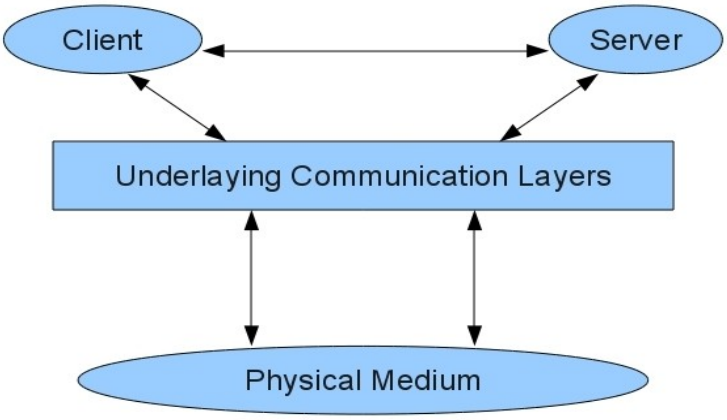
I started with implementing the Link Layer, followed by the Internet layer. The Address resolution and ICMP protocols were implemented to test reachability.
Finally, User Datagram protocol was also implemented to enable interaction between user applications (client and server) by sending and recieving data through the stack.

Keywords: protocol stack, Design Patterns, Conduit+ Framework, ARP, ICMP, UDP

Chapter:1
Introduction

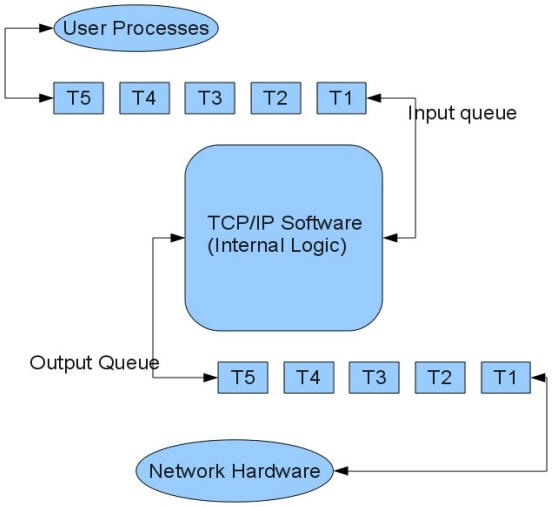
What is a TCP/IP stack ?

A TCP/IP stack or more generally a network stack is a component of an operating system that enables two computers connected by some physical medium to communicate with each other.



In most of the cases TCP/IP software is a core component of operating system kernel i.e Operating System keeps a single copy of the code and multiple programs can invoke the code.
Multiple user processes should be able to access the TCP/IP stack independently and transparent to each other so that data transferred by one program doesn't affects the data transferred by the other.

So basically TCP/IP has two major interactions :



Network Devices

Network Devices are hardware devices that place data sent to the network on the physical medium and recieve data sent over physical medium and report it to the device driver.
Examples of Network Devices are:

- Network Interface Cards (NIC)
- Wireless Adapters
- etc..

Device Drivers

Device Drivers are software components that encapsulate the various internal details that a software A needs to communicate with a hardware B independent of the kind of hardware B is.
So, for example if a user uses a network interface card provided by a vendor X Inc. The vendor also provides a driver that encapsulates the details a higher level software may need to communicate with the NIC.

User Applications

Any user application that needs to send data across a machine uses the protocol stack to appropriately place the data on the physical medium and morever make it accessible to the intended recieptent.
Examples of such applications are:

- Web Browser
- E-mail clients
- Telnet and other network facilities (ssh,rsh)
- etc..

Users interact to the application by means of some interface (Graphical or CLI).
The application software uses the system calls / Networking API provided by the underlaying OS to place a input packet to the TCP/IP input queues.

Its upto TCP/IP to handle that packet now.

Exploring the Internal Logic

Now that input and output are explained all that needs to be discussed is how TCP/IP software handles the user input data supplied by application X to a data-packet on network and send it to appropriate host, recieve the reply and send it back to the application X.

All this is accomplished by means of a protocol.

A protocol is an agreement over two communicating entities so that one can understand what the other is saying.

e.g two people knowing both english and french would communicate with one another only if there is some kind of agreement between both over the language they will communicate with.

According to OSI Model, the entire internal logic was distributed into 7 layers :

Presentation Layer
Session Layer
Application Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

However, for implementation purposes we merge the top three layers to form :

Application Layer
Transport Layer
Network Layer
Data-Link Layer
Physical Layer

In my example implementation, I have implemented the following with the necesarry protocols at each level:

Physical Layer Communication (Digital Identity Exchange)
    DIX Layer

Address Resolution (Address Resolution Protocol)
    ARP Layer

Network Layer (Internet Layer)
    IP Layer

Transport Layer
    UDP Protocol

Error Control
    ICMP Protocol

So a normal communication sequence for communication of host A with host B would involve:

Application Layer (A specifies destination host as addr(B) and source as addr(A))

Transport Layer (A UDP Packet is created):

src port : XXX
dst port : XXX
----- Data to be transferred -------

Network Layer (An IPv4 packet is created).

IPv4 header + UDP Packet from previous layer

DIX Layer (Digital Identity Exchange Layer)

Maps destination IPv4 address of next hop to hardware address.
In this process, it requires ARP layer for address resolution

DIX header + IP packet from previous layer

So, with the basic outline of the communication model described above, the implementation was started. However, a major issue was testing this protocol stack without an operating system kernel.

For this purpose a dummy wrapper interface using BSD Raw sockets was created and wrapper code was written as if the protocol stack is communicating with the native OS kernel.

These and more simulation details are outlined in next chapter.

Chapter 2:
Simulation Details

Simulating the TCP/IP stack communication as a user process involves the use of raw sockets.
Basically, a **raw socket** is a socket that allows direct sending and receiving of network packets by applications, bypassing all encapsulation in the networking software of the operating system.

Most socket application programming interfaces (APIs), especially those based on Berkeley sockets, support raw sockets.

Usually raw sockets receive packets inclusive of the header, as opposed to standard sockets which receive just the packet payload without headers. When transmitting packets, the automatic addition of a header may be a configurable option of the socket.

In BSD Sockets API, following is the syntax:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

Here, protocol is one of the constants defined as IPPROTO_xxx.
In our implementation, as everything would be an internet packet, we have used a ETH_P_ALL .

One major function used is the **ioctl** function.

ioctl() is a POSIX function to handle I/O operations.
This function affects an open file referenced by the fd argument.

```
int ioctl(int fd, int request, ... /* void *arg */ );
```

We can divide the requests related to networking into six categories:

- Socket operations
- File operations
- Interface operations
- ARP cache operations
- Routing table operations
- STREAMS system

However, in our implementation, only Interface operations would be required.

Now as basic definitions are covered, the control flow during simulation is explained below.

A class Tap is defined which takes as input an interface id to communicate with.

Tap is basically an abstraction that behaves as a network interface as well as an input output stream.

The basic functions provided by Tap being a network interface are:

- getType()

returns the type of network device, ethernet in our case

- start() /stop()
- getMacAddress()
- getMulticastAddress()
- addMulticastAddress()
- getLinkState()

get the status of Link.
Implemented using SIOCGIFLAGS and ifreq structure.

- getMTU()

get the Maximum transmission unit.

- Statistical functions

- getTransmittedCount()
- getDroppedCount() etc.

Functions provided by Tap Class being a I/O stream:

- read()

read from the socket

- write()

write on to the socket.

- getPosition()

get current head position

- setPosition()
  set current head position on the stream
- flush()

flush the stream

All these clearly suggest Tap being a class derived from network interface and the stream.

It then overrides the methods as required using the Raw socket API provided by the BSD sockets API.

Chapter 3:
Design Patterns used

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

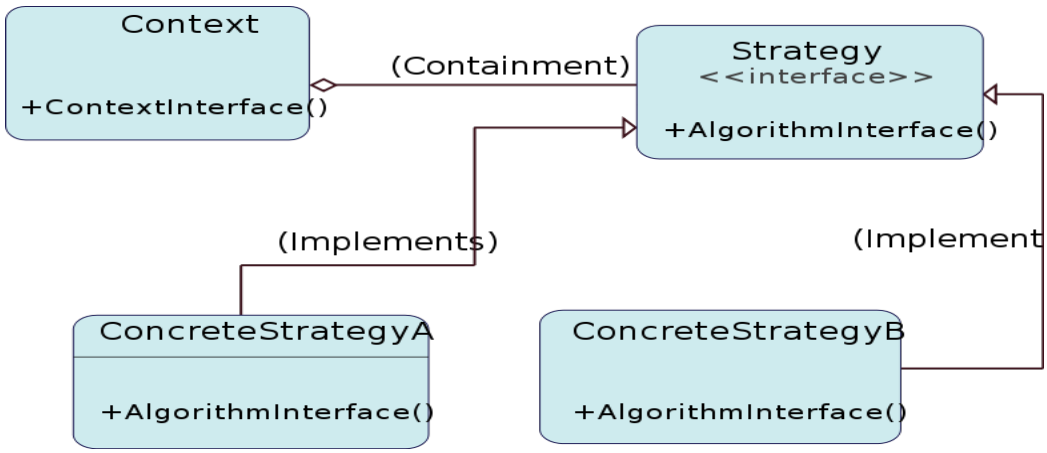
Design patterns reside in the domain of modules and interconnections. At a higher level there are Architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.

Not all software patterns are design patterns. For instance, algorithms solve computational problems rather than software design problems.

The design Patterns were employed in designing the TCP/IP protocol stack..Out of many exixting patterns, the following were applied.

1. **Strategy Pattern**
   The strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime.

   The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.



2. State Singleton and Command Patterns

The state pattern is a behavioral software design pattern, also known as the objects for states pattern. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime.

the singleton pattern is a design pattern that is used to restrict instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

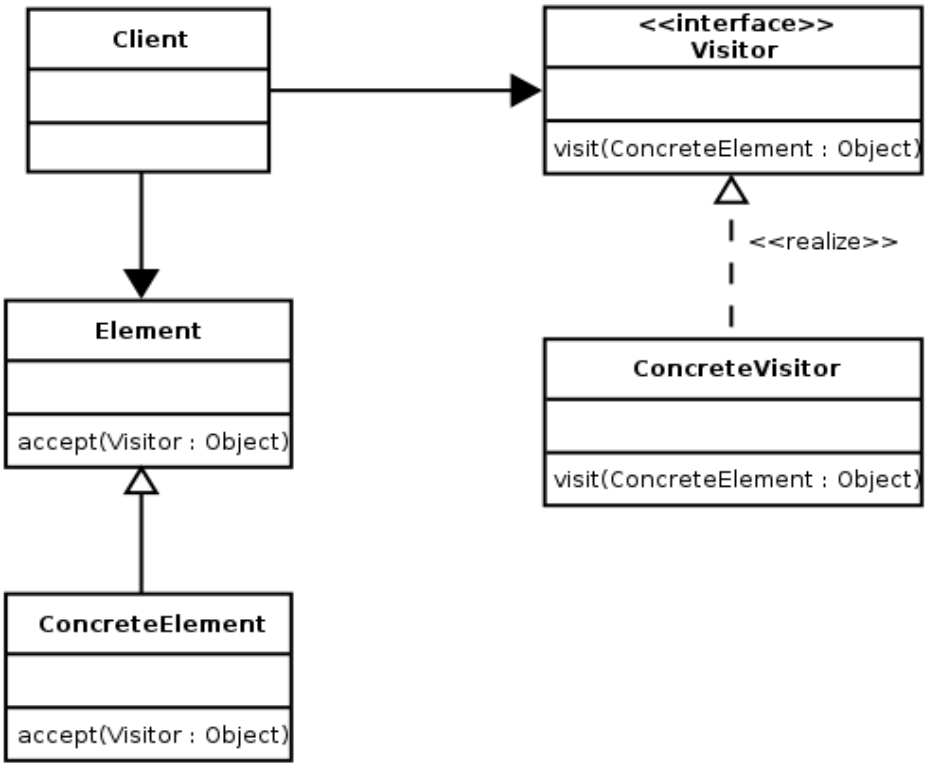
the command pattern is a design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Three terms always associated with the command pattern are client, invoker and receiver. The client instantiates the command object and provides the information required to call the method at a later time. The invoker decides when the method should be called. The receiver is an instance of the class that contains the method's code.

3. Visitor Pattern

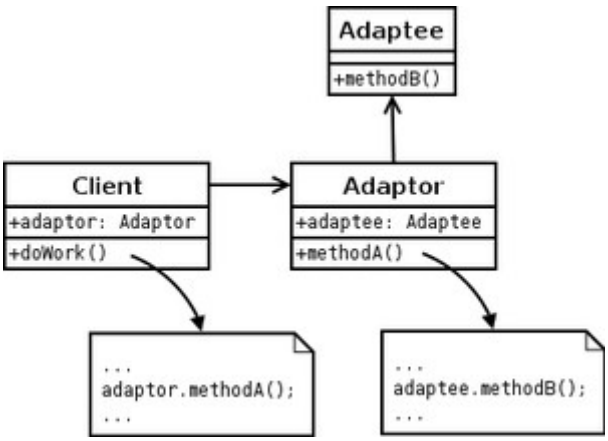
Visitor pattern represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which operates.

It is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. Thus, using the visitor pattern helps conformance with the open/closed principle.



4. Adapter Pattern

In computer programming, the adapter design pattern (often referred to as the wrapper pattern or simply a wrapper) translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. For instance, if multiple boolean values are stored as a single integer but your consumer requires a 'true'/'false', the adapter would be responsible for extracting the appropriate values from the integer value.

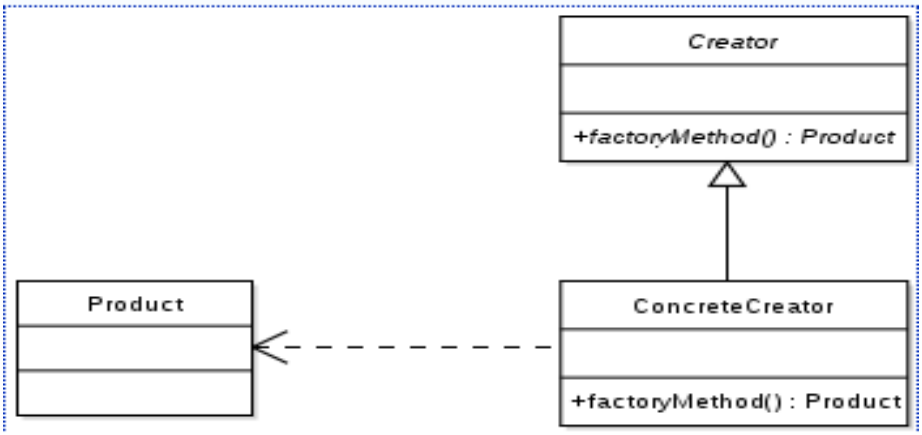


5. Factory Pattern

The factory method pattern is an object-oriented design pattern to implement the concept of factories.

Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

Outside the scope of design patterns, the term factory method can also refer to a method of a factory whose main purpose is creation of objects.



Chapter 4:
The Conduit+ framework

The entire stack is based on a paper titled " A Conduit based framework" for network protocols. If interested reader may refer to [1].

Nonetheness, a summary of the paper and its implications are presented here.

Contrary to earlier methods, Conduit+ framework extended reusability of components across the different layers by application of various design patterns discussed in section 2.5.

The framework is made up of basically two objects: **Conduits** and **information chunks.**

A Conduit is a software component with two distinct sides, sideA and sideB which maye be connected to other conduits called neighbor conduits.

An information chunk is a piece of message that is to be passed from one conduit to the other conduit.

Based upon number of neighbors connected on either side of conduits, conduits are classified into four groups:

1. **Mux**

A mux is a conduit which can have more than one neighbors on either sideB.

Properties of mux:

1. Multiple Input single output
2. Single Input multiple output can be achieved by an inverted mux.
3. Each Layer of protocol can contain a mux, for example in protocol stack, the ethernet layer contains mux to connect to ARP and IP layer and so on.
4. A Mux implements startegy pattern. Based upon the accessor class contained with it, it redirects the information chunk to appropriate conduit.

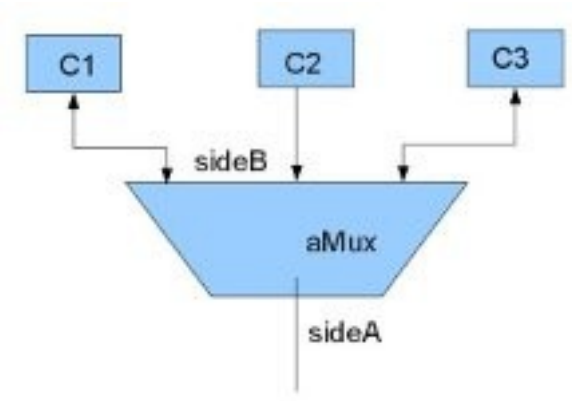


fig: A multiplexer

2. **The Protocol**

A communication protocol can be described as a finit state machine which is implemented by a protocol conduit which is where information chunks are produced, consumed and tested.
The protocol remembers the current state of the communication.

Properties of protocol are:
1. It has only one neighbor conduit on its either side.
2. A protocol implements the reciever pattern and the strategy pattern.

3. It consists of a Receiver class pointer contained in it (receiver pattern).
Next, the reciever class handles all the logic of handling a particular message chunk.
Which is receiver pattern.

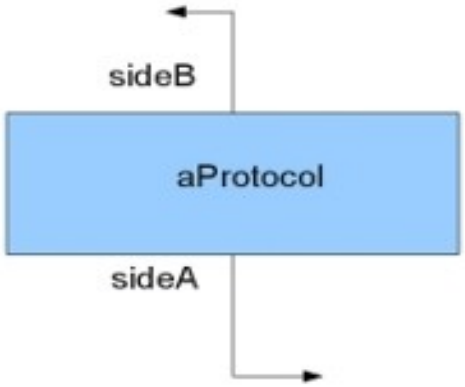


Fig: a protocol conduit

3. The Conduit Factory

One important problem is how to add new conduits to sideB of a mux.
Another problem is what should a mux do with an information chunk from sideA that addresses a non-existing conduit on sideB.
Both these problems are addressed by using a default conduit **Conduit Factory.** The default conduit handles packets with unmatched sideB conduits.
Next it can also be used to create new conduits with its own information as prototype.

A few important features are:

1. Only two neigbors, one on each side.
2. Implements prototype pattern.
3. Can be used for error logging and access details.
4. Default conduit if there is no matching conduit found corresponding to that key.

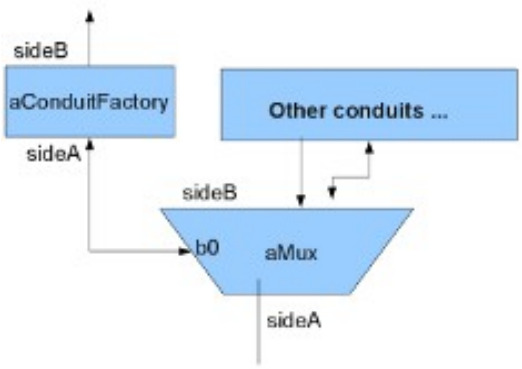
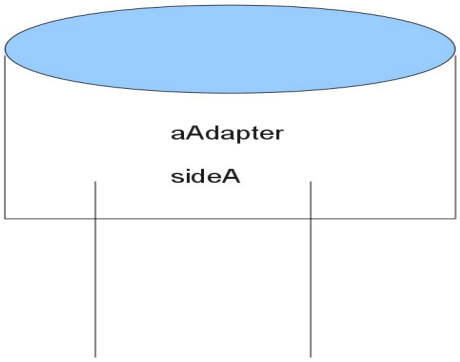


fig: a conduit factory

4. The Adapter

Adpater is a kind of conduit that has no neigbor conduit on its sideB. Thus only sideA is connected to another conduit. It used to interface the framework to some other hardware or software.

A few features are:
1. Single Neighbour, only on sideA.
2. sideB implementation is hardware or software dependent.
3. It is the entry point to both the ends of the conduit framework.
4. The adapter often converts information chunks to stream oriented format.



Now that different types of conduits have been explained, the next thing is how to apply the various design patterns listed in previous chapter on these conduits.

As in mux and adapter themselves are examples of composite and adapter patterns respectively. The other patterns are used as follows:

Application of strategy pattern

We want to be able to reuse the same mux class at different places in the conduit lattice. Therefore we call these components as accessors since they access the relevent information key within the information chunk.

Each mux has an accessor which has two responsiblities:

1. To compute the index of sideB conduit to which the current information chunk be dispatched.
2. To compute an index for the new sideB conduit.

Thus, a mux will handle an information chunk that arrives on sideA by using its accessor to get an index and then sending the information chunk out with the sideB conduit of that index.

This leads to a design like this:

```
class Mux {
     Accessor*     myAccessor;
     .......
     public:
     void* getDispatchKey(){ return myAccessor->dispatchKey();}
```

So the type of adpater decides all the dispatching logic and thus it is seperated from the Mux.
Therefore, this is an example of strategy pattern. Separating algorithm from implementation is the basic aim of strategy pattern which is achieved here.

Application of State, Singleton and Command Patterns

A communication protocol is usually realized as some extended finite state machine. It changes state when it receives the information chunk or when the timer times out.

Protocols are implemented using state pattern, i.e each state of a protocol is represented by a seperate object. Protocols delegate their behavior to the state objects, thus letting the protocol change the behavior once the state changes.

The state pattern in [2] gives a broad interface to context (protocol) and state object (state). This interface has an operation for each possible event. However a traditional state pattern would limit the reusability of protocol conduit.

So, in the given implementation, the protocols define a simple interface:
accept() : A method to accept information chunks.
Now when a protocol is given an information chunk it performs the apply(state, protocol) on that information chunk.
Thus, appropriate action is invoked based on the state of the object.

Thus **state pattern** is applied here.

Now, as all the state classes are similar and can be used again and again, it is better to have them as only one instance throughout and use them again and again.

Hence they could be made singleton instances by making sure only one instance of them can be created and that be resused. This is **singleton pattern**.

Raw information chunks are often an arrays of bytes, so it is not appropriate to define

operations like apply() on them. Therefore we introduce a new class messenger, that has a variable command associated with it. Thus a messenger knows what to do when it passes the graph of conduits.
Hence **command pattern** is applied.

Application of Visitor Pattern.

A typical messenger is routed through a mux until it gets a protocol where it interacts with the state. However some operations must involve traversing conduit graph differently especially when conduits are added and removed from the mux.
So, that leads to either add new kind of accept but that defeats the purpose of command pattern or make messenger responsible for traversing the conduit graph.

Either way is harmful as it leads to more coupling.

One better solution is to capture the non-specific and reusable elements to a seperate class and have a separate visitor for each operation.

Based upon requirements following visitor classes are defined:

class Installer : public Visitor
Installs a conduit on sideB of the mux.

class Uninstaller : public Visitor
Removes a conduit from sideB of the mux.

class BroadcastVisitor : public Visitor
demonstrate the packet flow in the graph.

class Transporter : public Visitor
Traverses the conduit graph with default keys when no matches are found.

class SocketInstaller : public Visitor
Installs a new adapter corresponding to the socket.

class SocketUninstaller : public Visitor
Removes the adapter corresponding to the socket.

class SocketDisconnector : public Visitor
remove a new sideB protocol conduit corresponding to a socket.

class SocketConnector : public Visitor
add a new sideB protocol conduit corresponding to a socket.

Chapter:5

Protocol Details

The ARP Protocol

ARP stands for Address Resolution Protocol.
Transmission at physical layer is done using mac addresses. So for a device A to transfer data to device B, A needs to know the logical address of B.
The IP layer then decides the destination host/next hop based upon various entries in the routing table.Now, the protocol stack needs to know the hardware address of next hop.

This is accomplihed by ARP.

32 Bit Internet Address <----------> 48 bit hardware address
ARP

ARP Header:

Hardware type		Protocol type
HW addr lth	P addr lth	Opcode
Source hardware address		
Source protocol address		
Destination hardware address		
Destination protocol address		

ARP message

ARP Requests:

ARP Requests are normally of two types: ARP request and ARP reply.
ARP requests are broadcasts whereas ARP replies are unicast.

So for host A to resolve logical address 10.8.20.134 of B it will:

1. Send a broadcast ARP request asking for **"whois 10.8.20.134"**
2. The host with Internet address 10.8.20.134 replies his mac address via a unicast.
3. If a reply is not recieved, within a given time, the sender host retries.

Caching:

As it would not be very reasonable to have a ARP request for every communication, it simply makes sense to have a cache of ARP addresses, so that mapping is stored in the local machine itself.

In this implementation, a least recently used caching scheme is used.
The IP address which has not been lately used is discarded if a new binding is to be stored

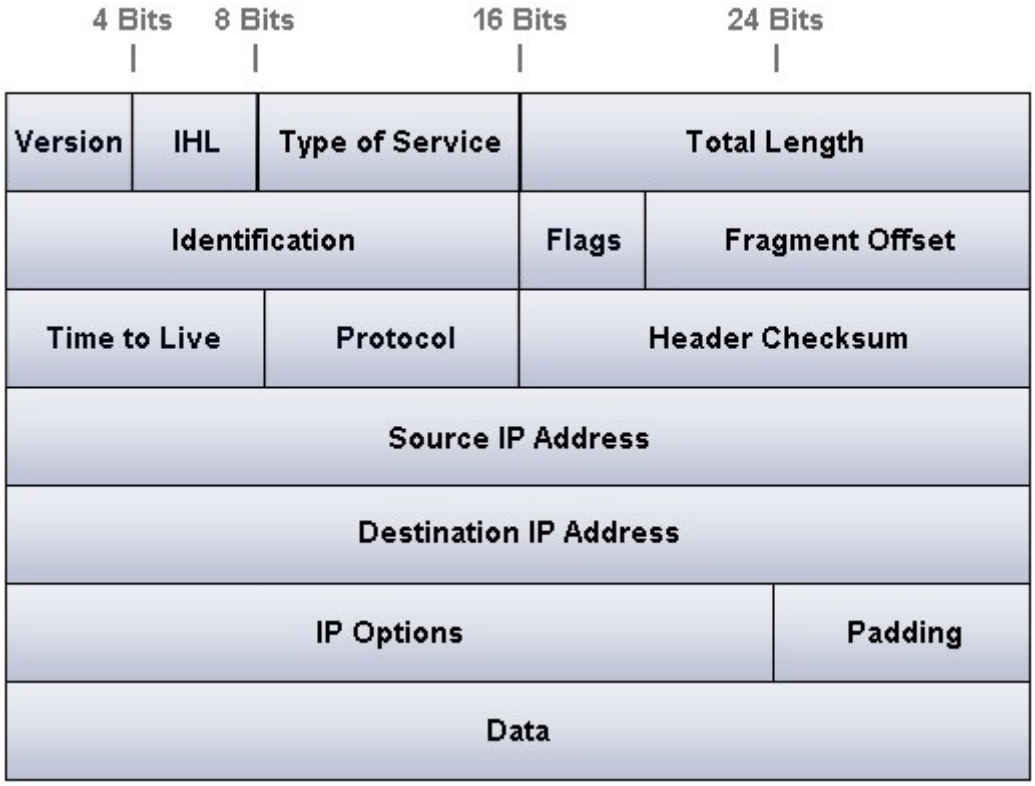
The Internet Protocol (IP)
The internet protocol is the hub of all the activities, the link layer transfers packets meant for Internet layer and the IP acts on these packets.

Based upon the type of packet, it extracts header information and then demultiplexes the packet to appropriate transport layer protocols.

The task of IP is to ensure source to destination delivery.

It maintains a routing table to map routes to IP address of the gateways. If a packet is not meant for the input host the packet is discarded or forwarded based upon whether or not the host is a router.

IP header:

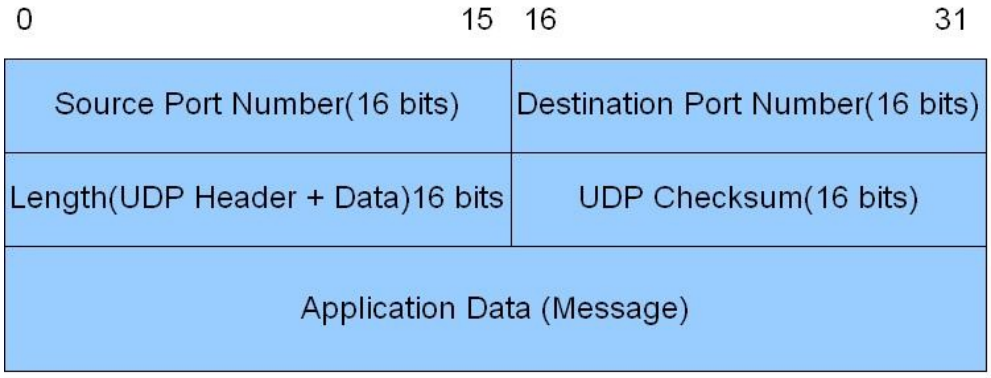


The UDP Protocol

UDP is a transport layer protocol. It enables applications to talk.
However, there is no reliablity in deliver of UDP packets. Each packet goes as a seperate datagram.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [4]

UDP header:



Port numbers are 16 bit numbers defined by the Internet standards authority.
Port numbers from 1-1024 are well known ports, rest are ephermal ports.

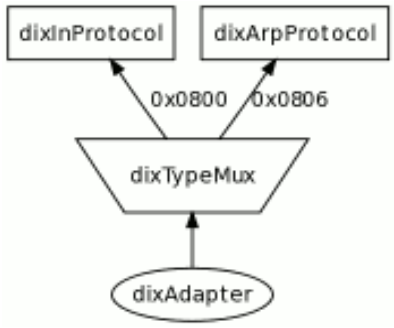
In case, an application doesnt uses the well known ports, it may then randomly select one of the ephermal ports.

Chapter 6:
Architecture and Implementation Details

In this section the implementation architecture of the various implemented protocols along with minor class details.

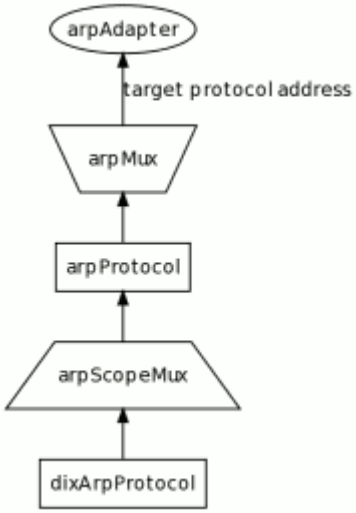
DIX Conduit
DIX stands for digital Identity exchange. This conduit is responsible for :

1. Recieving the packet from IP/ARP protocols and putting them on the physical network.
2. Recieving the packets from physical network and putting it on the appropriate protocol.



A dix header flag of 0x0800 indicates an IP packet whereas 0x0806 indicates ARP packet.

ARP Conduits

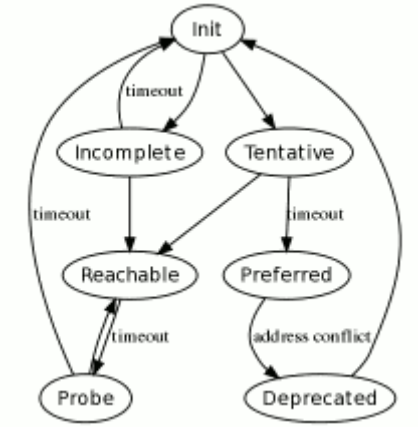


At the Ethernet level, packets are transmitted to the MAC address specified in their Ethernet packet headers. The role of the ARP protocol is to resolve the mappings between IP addresses and Ethernet MAC addresses. The very basic mechanism of ARP is to broadcast an inquiry to see whether there is a node that has been assigned the IP address trying to communicate. If there is the recipient of the inquiry who has been assigned the requested IP address then it reports its own MAC address to the original sender. Since is is not very efficient to request a MAC address with an ARP request broadcast every time sending an IP packet, ARP caches mappings between MAC addresses and IP addresses that it has inquired about.

Another feature of ARP is to obtain link-local IP addresses. Link-local IP addresses are addresses that can be assigned dynamically to the appliances on a LAN without using DHCP; so information appliances on a home LAN can communicate each other using TCP/IP without any manual network configuration. In obtaining link-local IP addresses, an appliance broadcasts an inquiry to see whether the link-local IP address it wants to use is already in use or not, and if no reply is received (i.e. not used), the IP address is assigned to that appliance.

States of onlink Inet4Address object

ArpProtocol conduit looks for the source protocol address of the received ARP packet and, if the Inet4Address object for that address is already created, changes its state to Reachable. ArpMux moves received ARP packets to ArpAdapter inspecting their target protocol addresses. ArpAdapter receivers are the Inet4Address objects, and they process received ARP packets based on their state.



The following table expresses the states for ARP processing:

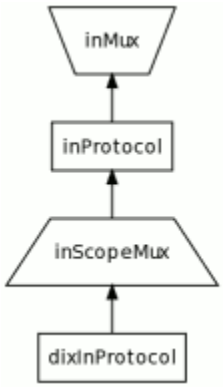
State	Description	RFC
Init	The MAC address corresponding to the IP address is not yet known.	826
Incomplete	The MAC address corresponding to the IP address is being requested.	826
Reachable	The MAC address corresponding to the IP address is known.	826
Probe	The MAC address corresponding to the IP address is being re-checked.	826
Tentative	Checking whether a local address can be used.	3927
Preferred	A local address can be used.	3927
Deprecated	A local address that is due to be discontinued.	3927

IP Conduits

InProtocol conduit looks for the protocol number of received IP packets and sends the packets to InMux. If the received IP packets are fragmented packets, it sets IPPROTO_FRAGMENT as a pseudo-protocol number to the packets so that they can be processed by the higher fragment reassembly conduits. (IPPROTO_FRAGMENT is originally a protocol number for IPv6.) In the case that the destination address of an IP packet being sent is not the loopback address or the on-link IP address, InProtocol conduit configures the messenger to send the packet to a router. It also inspects the packet length and if it is larger than the path MTU, fragments it before sending it on to the lower InScopeMux.

InMux inspects the protocol number of the received IP packet and sends it to the higher conduits like UDP, TCP, etc.

InScopeMux inspects the scope ID of the packet to be sent and moves the packet to the loopback interface or Ethernet interface.



UDP Conduits

The UDP conduits process UDP datagrams. UDP-related processes peculiar to IPv4 are handled in the lower UdpProtocol. Meanwhile, UDP-related processes common to IPv4 and IPv6 are handled in the upper DatagramProtocol.

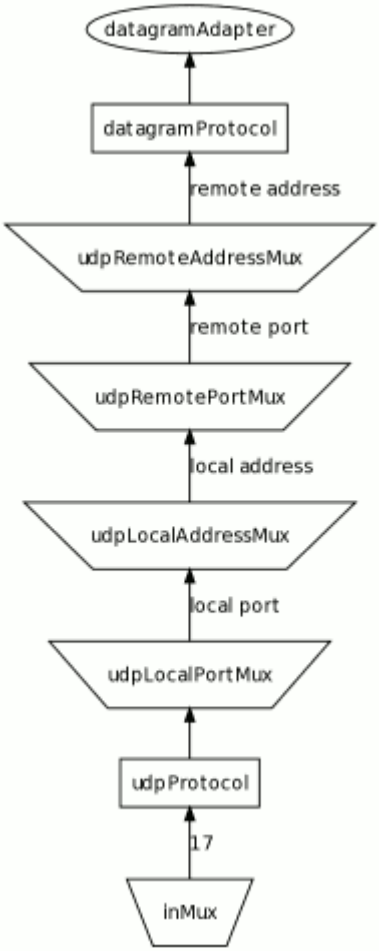


fig:UDP Conduit

Chapter 7
Implementation and Class Diagrams

Putting everything together, the following diagram for flow of packets can be derived.

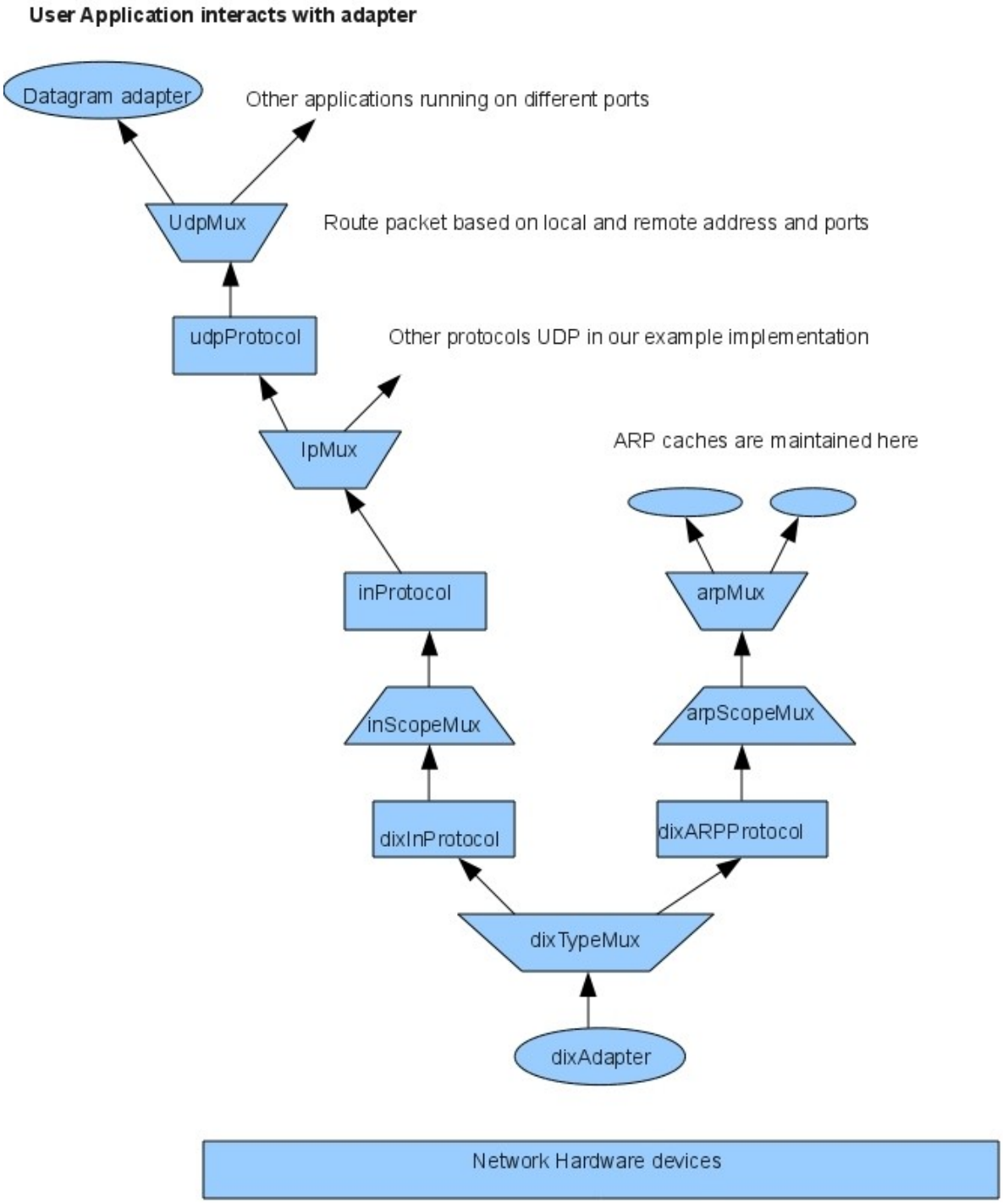


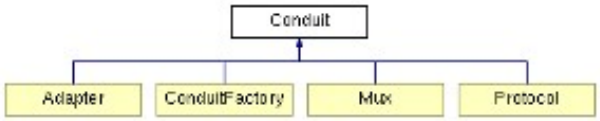
Fig : Conduit graph for our implementation

A successful implementation must route packets in accordance with the conduit graph.

Class Diagram

Conduit Class Reference

Inheritance diagram for Conduit:



List of all members.

Public Member Functions

virtual unsigned int	**release** ()
virtual bool	**accept** (**Messenger** *m)=0
virtual bool	**accept** (**Visitor** *v, **Conduit** *sender=0)=0
virtual **Conduit** *	**clone** (void *key)=0
Receiver *	**getReceiver** () const
void	**setReceiver** (**Receiver** *receiver)
bool	**toA** (**Visitor** *v)
bool	**toB** (**Visitor** *v)
virtual void	**setA** (**Conduit** *c)
virtual void	**setB** (**Conduit** *c)
virtual **Conduit** *	**getA** () const
virtual **Conduit** *	**getB** () const
virtual bool	**isEmpty** () const
virtual const char *	**getName** () const

Static Public Member Functions

static void	**connectAA** (**Conduit** *x, **Conduit** *y)
static void	**connectAB** (**Conduit** *x, **Conduit** *y)
static void	**connectBA** (**Conduit** *x, **Conduit** *y)
static void	**connectBB** (**Conduit** *x, **Conduit** *y)
static void	**connectAB** (**Conduit** *x, **Mux** *y, void *keyX)
static void	**connectBA** (**Mux** *x, **Conduit** *y, void *keyY)
static void	**connectBB** (**Mux** *x, **Mux** *y, void *keyX, void *keyY)

Protected Attributes

Receiver *	**receiver**
Conduit *	**sideA**
Conduit *	**sideB**

The documentation for this class was generated from the following file:

- conduit.h

Chapter 8
Conclusion and Future Work

The entire project was an effort to explore and test the possiblity of implementing a TCP/IP protocol stack using design patterns. The stack was tested using the raw socket interface provided by the BSD socket API.

The application was found to run as desired for the various input cases. Morever, the implementation is much better than the traditional methods as in there is too much flexiblity in adding a new feature to the stack.

However, the capabilities of this implementation are very limited. So far, it only supports Datagram Sockets. Implementing TCP support is the next big task ahead. Similarly error control and flow control mechanisms need to be improved.

However, the task serves its purpose of demonstrating the possiblity of TCP/IP stack implementation using conduit+ framework. The work can later be extended to integrate the stuff to a real time operating system.

Chapter 9
References

[1] A framework for network protocol software, Huni,Johnson, Angel, ACM 1995
[2] TCP/IP illustrated volume2 , Comer and Stevens
[3] Design Patterns by GoF, Addison-Wesely
[4] http://faqs.org/ rfcs
[5] The C++ object model, Stanley B. Lippman
[6] http://code.google.com/p/es-operating system
[7] Essential COM, Don Box Addison Wesely

Appendix A
Source Listing

Source File : conduit.h

```
#ifndef CONDUIT_H_INCLUDED
#define CONDUIT_H_INCLUDED

// cf. H. Hüni, R. Johnson, R. Engel,
//     A Framework for Network Protocol Software, ACM, 1995.

#include <string.h>
#include <es.h>
#include <es/ref.h>
#include <es/types.h>
#include <es/tree.h>

class Accessor;
class Adapter;
class Conduit;
class Messenger;
class Mux;
class Protocol;
class ConduitFactory;
class Receiver;
class Visitor;

class Messenger
{
    Ref     ref;

    char*   chunk;
    long    len;
    long    position;
    int     type;       // type of data pointed by position

    long    saved;      // saved position
    bool    internal;   // true if chunk is allocated internally

public:
    Messenger(long len = 0, long pos = 0, void* chunk = 0) :
        chunk(static_cast<char*>(chunk)),
        len(len),
        position(pos),
        type(0),
        saved(0),
        internal(false)
    {
        ASSERT(0 <= len);
        if (0 < len && this->chunk == 0)
        {
            this->chunk = new char[len];
            internal = true;
```

```
    }
 }
 virtual ~Messenger()
 {
     ASSERT(0 <= len);
     if (internal)
     {
         delete[] chunk;
     }
 }

 virtual bool apply(Conduit* c)
 {
     return true;
 }

void dump(const char* header, ...)
{
     va_list list;
     va_start(list,header);

     esReport("---\n");
     esReportv(header,list);
     for (int i = 0; i < len && len > 0; i++)
     {
         if (i % 16 == 0)
         {
             esReport("\n");
         }
         esReport("%02x ", (u8)chunk[i]);
     }
     esReport("\n---\n");
     return;
 }

 long getSize() const
 {
     return len;
 }
 void setSize(long len)
 {
     this->len = len;
 }

 long getPosition() const
 {
     return position;
 }
 void setPosition(long pos)
 {
     if (pos < 0 || len < pos)
     {
         return;
     }
     position = pos;
```

```
    }
    long movePosition(long delta)
    {
        if (delta < 0)
        {
            if (delta < -position)
            {
                return position;
            }
        }
        else if (position + delta < position)
        {
            return position;
        }
        position += delta;
        return position;
    }

    void savePosition()
    {
        saved = position;
    }
    void restorePosition()
    {
        position = saved;
    }

    long getLength() const
    {
        ASSERT(position <= len);
        return len - position;
    }
    void setLength(long len)
    {
        ASSERT(0 <= len && position + len <= this->len);
        this->len = position + len;
    }

    int getType() const
    {
        return type;
    }
    void setType(int type)
    {
        this->type = type;
    }

    int read(void* dst, int count, long offset)
    {
        if (offset < 0 || len <= offset || count <= 0 || offset +
count < offset)
        {
            return 0;
        }
        if (len < offset + count)
```

```
        {
            count = len - offset;
        }
        memmove(dst, chunk + offset, count);
        return count;
    }
    int write(const void* src, int count, long offset)
    {
        if (offset < 0 || len <= offset || count <= 0 || offset +
count < offset)
        {
            return 0;
        }
        if (len < offset + count)
        {
            count = len - offset;
        }
        memmove(chunk + offset, src, count);
        return count;
    }

    void* fix(long count, long offset) const
    {
        if (offset < 0 || len <= offset || count <= 0 || offset +
count < offset ||
            len < offset + count)
        {
            return 0;
        }
        return chunk + offset;
    }
    void* fix(long count) const
    {
        return fix(count, getPosition());
    }

    s32 sumUp(long count) const
    {
        register u16* ptr = static_cast<u16*>(fix(count));
        register s32 sum = 0;

        while (1 < count)
        {
            // This is the inner loop
            sum += *ptr++;
            count -= 2;
        }

        // Add left-over byte, if any
        if (0 < count)
        {
            sum += *reinterpret_cast<u8*>(ptr);
        }

        return sum;
```

```
    }

    unsigned int addRef()
    {
        return ref.addRef();
    }

    unsigned int release()
    {
        unsigned int count = ref.release();
        if (count == 0)
        {
            delete this;
        }
        return count;
    }
};

class Visitor
{
    Messenger*  messenger;

public:
    Visitor(Messenger* messenger) :
        messenger(messenger)
    {
    }
    virtual ~Visitor()
    {
    }

    Messenger* getMessenger() const
    {
        return messenger;
    }

    virtual bool at(Adapter* a, Conduit* c);
    virtual bool at(Mux* m, Conduit* c);
    virtual bool at(Protocol* p, Conduit* c);
    virtual bool at(ConduitFactory* f, Conduit* c);

    virtual bool toB(Mux* m);
};

class Accessor
{
public:
    virtual void* getKey(Messenger* m) = 0;
};

class Receiver
{
public:
    Receiver() {}
    virtual ~Receiver() {}
```

```
    virtual Receiver* clone(Conduit* conduit, void* key)
    {
        return this;
    };
    virtual unsigned int release()
    {
        return 1;
    }
};

class Conduit
{
protected:
    Receiver*       receiver;
    Conduit*        sideA;
    Conduit*        sideB;

public:
    Conduit() :
        receiver(0),
        sideA(0),
        sideB(0)
    {
    }
    virtual ~Conduit()
    {
    }

    virtual unsigned int release()
    {
        if (receiver)
        {
            receiver->release();
            receiver = 0;
        }
        delete this;
        return 0;
    }

    virtual bool accept(Messenger* m) = 0;
    virtual bool accept(Visitor* v, Conduit* sender = 0) = 0;
    virtual Conduit* clone(void* key) = 0;

    Receiver* getReceiver() const
    {
        return receiver;
    }
    void setReceiver(Receiver* receiver)
    {
        ASSERT(receiver);
        this->receiver = receiver;
    }

    bool toA(Visitor* v)
    {
```

```
    if (sideA)
    {
        return sideA->accept(v, this);
    }
    return true;
}
bool toB(Visitor* v)
{
    if (sideB)
    {
        return sideB->accept(v, this);
    }
    return true;
}

virtual void setA(Conduit* c)
{
    sideA = c;
}
virtual void setB(Conduit* c)
{
    sideB = c;
}

virtual Conduit* getA() const
{
    return sideA;
}
virtual Conduit* getB() const
{
    return sideB;
}

virtual bool isEmpty() const
{
    return sideB ? false : true;
}

virtual const char* getName() const
{
    return "Conduit";
}

static void connectAA(Conduit* x, Conduit* y)
{
    x->setA(y);
    y->setA(x);
}

static void connectAB(Conduit* x, Conduit* y)
{
    x->setA(y);
    y->setB(x);
}
```

```
    static void connectBA(Conduit* x, Conduit* y)
    {
        x->setB(y);
        y->setA(x);
    }

    static void connectBB(Conduit* x, Conduit* y)
    {
        x->setB(y);
        y->setB(x);
    }

    static void connectAB(Conduit* x, Mux* y, void* keyX);
    static void connectBA(Mux* x, Conduit* y, void* keyY);
    static void connectBB(Mux* x, Mux* y, void* keyX, void* keyY);
};

class Protocol : public Conduit
{
public:
    bool accept(Messenger* m)
    {
        return m->apply(this);
    }
    bool accept(Visitor* v, Conduit* sender = 0)
    {
        // Determine the direction to forward the messenger before
calling the
        // at() method as sideB can be reset in the at() method.
Note the
        // default direction is to B.
        bool (Protocol::*to)(Visitor*);
        to = (sender == sideB) ? &Protocol::toA : &Protocol::toB;
        if (!v->at(this, sender))
        {
            return false;
        }
        (this->*to)(v);
    }

    Protocol* clone(void* key)
    {
        Protocol* p = new Protocol();
        if (receiver)
        {
            p->setReceiver(receiver->clone(p, key));
        }
        return p;
    }

    const char* getName() const
    {
        return "Protocol";
    }
};
```

```
class ConduitFactory : public Conduit
{
    Conduit*    prototype;
    void*       defaultKey;
    bool        hasDefault;

public:
    ConduitFactory(Conduit* prototype = 0) :
        prototype(prototype),
        defaultKey(0),
        hasDefault(false)
    {
    }

    unsigned int release()
    {
        if (prototype)
        {
            prototype->release();
            prototype = 0;
        }
        return Conduit::release();
    }

    bool accept(Messenger* m)
    {
        return m->apply(this);
    }
    bool accept(Visitor* v, Conduit* sender = 0)
    {
        bool (ConduitFactory::*to)(Visitor*);
        to = (sender == sideB) ? &ConduitFactory::toA :
&ConduitFactory::toB;
        if (!v->at(this, sender))
        {
            return false;
        }
        (this->*to)(v);
    }
    Conduit* create(void* key)
    {
        if (!prototype)
        {
            return 0;
        }
        return prototype->clone(key);
    }

    ConduitFactory* clone(void* key)
    {
        ConduitFactory* f = new ConduitFactory(prototype ?
prototype->clone(key) : 0);
        if (receiver)
        {
```

```
            f->setReceiver(receiver->clone(f, key));
        }
        if (hasDefaultKey())
        {
            f->addDefaultKey(getDefaultKey());
        }
        return f;
    }

    bool hasDefaultKey() const
    {
        return hasDefault;
    }

    void* getDefaultKey() const
    {
        ASSERT(hasDefault);
        return defaultKey;
    }

    void addDefaultKey(void* key)
    {
        hasDefault = true;
        defaultKey = key;
    }

    void removeDefaultKey(void* key)
    {
        ASSERT(key == defaultKey);
        hasDefault = false;
        defaultKey = 0;
    }

    const char* getName() const
    {
        return "ConduitFactory";
    }
};

class Adapter : public Conduit
{
    Conduit* getB() const
    {
        return 0;
    }
    void setB(Conduit* c)
    {
    }

public:
    bool accept(Messenger* m)
    {
        return m->apply(this);
    }
    bool accept(Visitor* v, Conduit* sender = 0)
```

```
    {
        Conduit* conduit = sideA;
        if (!v->at(this, sender))
        {
            return false;
        }
        if (sender != conduit)
        {
            return toA(v);
        }
        else
        {
            return true;
        }
    }

    Adapter* clone(void* key)
    {
        Adapter* a = new Adapter();
        if (receiver)
        {
            a->setReceiver(receiver->clone(a, key));
        }
        return a;
    }

    const char* getName() const
    {
        return "Adapter";
    }
};

class Mux : public Conduit
{
    Accessor*               accessor;
    ConduitFactory*         factory;
    Tree<void*, Conduit*>   sideB;

    Conduit* getB() const
    {
        return 0;
    }
    void setB(Conduit* c)
    {
    }

public:
    Mux(Accessor* a, ConduitFactory* f) :
        accessor(a),
        factory(f)
    {
        factory->setA(this);
    }
    ~Mux()
    {
```

```
}

unsigned int release()
{
    if (factory)
    {
        factory->release();
        factory = 0;
    }
    return Conduit::release();
}

ConduitFactory* getFactory() const
{
    return factory;
}

void* getKey(Messenger* m)
{
    if (m)
    {
        return accessor->getKey(m);
    }
    else
    {
        return 0;
    }
}
Accessor* getAccessor() const
{
    return accessor;
}

bool accept(Messenger* m)
{
    return m->apply(this);
}
bool accept(Visitor* v, Conduit* sender = 0)
{
    Conduit* conduit = sideA;
    if (!v->at(this, sender))
    {
        return false;
    }
    if (sender != conduit)
    {
        ASSERT(sideA);
        ASSERT(sender);
        return toA(v);
    }
    else
    {
        return v->toB(this);
    }
}
```

```
void addB(void* key, Conduit* c)
{
    sideB.add(key, c);
}
void removeB(void* key)
{
    sideB.remove(key);
}

bool isEmpty() const
{
    return sideB.isEmpty();
}

Conduit* getB(void* key) const
{
    return sideB.get(key);
}

bool contains(void* key) const
{
    return sideB.contains(key);
}

Tree<void*, Conduit*>::Iterator list()
{
    return sideB.begin();
}

Mux* clone(void* key)
{
    Mux* m = new Mux(accessor, factory->clone(key));
    if (receiver)
    {
        m->setReceiver(receiver->clone(m, key));
    }
    return m;
}

const char* getName() const
{
    return "Mux";
}

void* getKey(Conduit* b)
{
    Tree<void*, Conduit*>::Node* node;
    Tree<void*, Conduit*>::Iterator iter = list();
    while ((node = iter.next()))
    {
        if (node->getValue() == b)
        {
            return node->getKey();
        }
    }
```

```
        return 0;
    }
};

class Installer : public Visitor
{
public:
    Installer(Messenger* messenger) :
        Visitor(messenger)
    {
    }

    bool at(Adapter* a, Conduit* c)
    {
        return true;
    }

    bool at(Mux* m, Conduit* c)
    {
        return true;
    }

    bool at(Protocol* p, Conduit* c)
    {
        return true;
    }

    bool at(ConduitFactory* f, Conduit* c)
    {
        Mux* mux = dynamic_cast<Mux*>(c);
        ASSERT(mux);
        ASSERT(mux->getFactory() == f);
        void* key = mux->getKey(getMessenger());
        if (Conduit* conduit = f->create(key))
        {
            Conduit::connectAB(conduit, mux, key);
            return false;
        }
        return true;
    }
};

class Uninstaller : public Visitor
{
public:
    Uninstaller(Messenger* messenger) :
        Visitor(messenger)
    {
    }

    bool at(Adapter* a, Conduit* c)
    {
        return true;
    }
```

```
    bool at(Mux* mux, Conduit* c)
    {
        if (c->isEmpty())
        {
            c->setA(0);
            mux->removeB(mux->getKey(getMessenger()));
            c->release();
            return true;
        }
        return false;       // To stop this visitor
    }

    bool at(Protocol* p, Conduit* c)
    {
        return false;       // To stop this visitor
    }

    bool at(ConduitFactory* f, Conduit* c)
    {
        return true;
    }
};

inline bool Visitor::at(Adapter* a, Conduit* c)
{
    return messenger->apply(a);
}

inline bool Visitor::at(Mux* m, Conduit* c)
{
    return messenger->apply(m);
}

inline bool Visitor::at(Protocol* p, Conduit* c)
{
    return messenger->apply(p);
}

inline bool Visitor::at(ConduitFactory* f, Conduit* c)
{
    return messenger->apply(f);
}

inline bool Visitor::toB(Mux* m)
{
    void* key = m->getKey(getMessenger());
    do {
        try
        {
            Conduit* b = m->getB(key);
            if (b->accept(this, m))
            {
                return true;
            }
        }
```

```
        catch (SystemException<ENOENT>)
        {
        }
    } while (!m->getFactory()->accept(this, m));    // Have factory
added a new conduit to sideB?
    return false;
}

class BroadcastVisitor : public Visitor
{
public:
    BroadcastVisitor(Messenger* messenger) :
        Visitor(messenger)
    {
    }

    bool toB(Mux* m)
    {
        Tree<void*, Conduit*>::Node* node;
        Tree<void*, Conduit*>::Iterator iter = m->list();
        while ((node = iter.next()))
        {
            Conduit* b = node->getValue();
            b->accept(this, m);
        }
        return true;
    }
};

// Transporter extends Visitor to visit conduits using the default
factory keys
// if no matches are found.
class Transporter : public Visitor
{
public:
    Transporter(Messenger* messenger) :
        Visitor(messenger)
    {
    }

    bool toB(Mux* m)
    {
        void* key = m->getKey(getMessenger());
        ConduitFactory* factory = m->getFactory();

        esReport("transport: %p (%ld)\n", key, (long) key);

        do {
            try
            {
                Conduit* b = m->getB(key);
                if (b->accept(this, m))
                {
                    return true;
                }
```

```
            }
            catch (SystemException<ENOENT>)
            {
                if (factory->hasDefaultKey())
                {
                    try // with default key
                    {
                        Conduit* b = m->getB(factory-
>getDefaultKey());
                        if (b->accept(this, m))
                        {
                            return true;
                        }
                    }
                    catch (SystemException<ENOENT>)
                    {
                    }
                }
            }
        } while (!factory->accept(this, m));    // Have factory
added a new conduit to sideB?
        return false;
    }
};

class TypeAccessor : public Accessor
{
public:
    void* getKey(Messenger* m)
    {
        return reinterpret_cast<void*>(m->getType());
    }
};

inline void Conduit::
connectAB(Conduit* x, Mux* y, void* keyX)
{
    x->setA(y);
    y->addB(keyX, x);
}

inline void Conduit::
connectBA(Mux* x, Conduit* y, void* keyY)
{
    x->addB(keyY, y);
    y->setA(x);
}

inline void Conduit::
connectBB(Mux* x, Mux* y, void* keyX, void* keyY)
{
    x->addB(keyY, y);
    y->addB(keyX, x);
}
```

```
#endif // CONDUIT_H_INCLUDED
```

Source file: arp.h

```
#ifndef ARP_H_INCLUDED
#define ARP_H_INCLUDED

#include <es.h>
#include <es/endian.h>
#include <es/timeSpan.h>
#include <es/net/arp.h>
#include "socket.h"
#include "inet4address.h"

class ARPReceiver : public InetReceiver
{
    InFamily*   inFamily;

public:
    ARPReceiver(InFamily* inFamily) :
        inFamily(inFamily)
    {
    }

    bool input(InetMessenger* m, Conduit* c);
};

class ARPFamily : public AddressFamily
{
    InFamily*                   inFamily;

    // Scope demultiplexer
    InetScopeAccessor           scopeAccessor;
    ConduitFactory              scopeFactory;
    Mux                         scopeMux;

    // ARP protocol
    ARPReceiver                 arpReceiver;
    Protocol                    arpProtocol;
    InetLocalAddressAccessor    arpAccessor;
    Adapter                     arpAdapter;     // prototype
    ConduitFactory              arpFactory;
    Mux                         arpMux;

public:
    ARPFamily(InFamily* inFamily);

    int getAddressFamily()
    {
        return AF_ARP;
    }

    Conduit* getProtocol(Socket* socket)
    {
        return &arpProtocol;
```

```
    }

    void addInterface(NetworkInterface* interface)
    {
        Conduit* c = interface->addAddressFamily(this, &scopeMux);
        if (c)
        {
            scopeMux.addB(reinterpret_cast<void*>(interface-
>getScopeID()), c);
        }
    }

    void addAddress(Inet4Address* address)
    {
        ASSERT(address);
        if (address)
        {
            InetMessenger m;
            m.setLocal(address);
            Installer installer(&m);
            arpMux.accept(&installer, &arpProtocol);
        }
    }

    void removeAddress(Inet4Address* address)
    {
        ASSERT(address);
        if (address)
        {
            Adapter* adapter = dynamic_cast<Adapter*>(address-
>getAdapter());
            if (adapter)
            {
                InetMessenger m;
                m.setLocal(address);
                Uninstaller uninstaller(&m);
                adapter->accept(&uninstaller);
            }
        }
    }
};

#endif  // ARP_H_INCLUDED
```