## Wire definition language

Wire is a network protocol definition language derived from Interface Definition Language[1] (IDL). It is used to represent an on the wire representation of a certain network protocol in an intuitive and highly abstract manner. The Wire compiler is designed to address the automatic generation of code that handles all of the defined protocol's communications, parsing and construction of packets.

Coding handlers for network protocols is time consuming and highly error prone. One must deal with sanity checks upon packet parsing/construction, integer byte ordering and sizes, various charset encodings, data alignment and padding, error reporting and debugging. Furthermore when considering protocol operations, programmers must take into account memory allocation and buffering, timing, various types of operations such as blocking/nonblocking operations and synchronous/asynchronous operations.

Programmers take various approaches for tackling coding of network protocol handlers and thus code reusability is low, modularity is weak and uniformity is non-existent (in most cases). Wire is intended to provide an intuitive way of defining a protocol that can be situated in OSI Layers 2-7. Furthermore the code generated by the compiler fits nicely with the network protocol theory and as such is easy readable. The API provided by the generated code library tends to be simple and easy to use, but of course that depends on the protocol definition. To that point Wire strives to provide an abstraction to its users from underlying networking technologies (Berkley sockets, WinSock, TLI) and host configurations (byte/bit ordering, register sizes, floating point representations).

The initial idea for Wire was to create a definition language in which one would define a protocol, run it trough a compiler and get a library that would handle that particular network protocol. The protocol at hand is already suppose to have a specification such as Internet protocol or even HTTP. Thus with a single Wire definition a compiler could generate handlers in multiple languages and/or for various systems and frameworks. For example code generation could be extended to generate Lua libraries or more specifically NMAP NSE libraries which are written in Lua but exist in a more specialized framework. Also one could generate dissection methods for Wireshark or even use ti to make fuzzers to test some network implementation.

The extension to the initial idea is to define your own network protocol for whatever purposes. This makes Wire a definition language and the underlying encoding algorithms a serialization protocol. To compare Wire to other similar projects that provide a definition language and a serialization protocol for that particular language one could mention technologies used in remote procedure call protocols such as Microsoft RPC protocol which uses IDL as a definition language and NDR as a serialization protocol. Other examples may include Google Protocol Buffers, ASN.1, JSON, Network File System (NFS) protocol and External Data Representation serialization protocol, REST, SOAP etc.

Wire conceptually differs from mentioned projects in the fact that these projects weren't designed to provide means to define an already existing protocol. The serialization protocol involved in encoding defined data types in definition language at hand is of no interest to a common user. For example a user defines an integer type to be transmitted over the communication line, all he wants is to send a number and receive that number unchanged. The format used to transfer that integer is of no interest to him. This concept is utilized in every of mentioned technologies but Wire. Wire is intended to describe the actual representation of data on the wire so the Wire definition language and the underlying serialization protocol a closely related. To elaborate this a bit further let's introduce some formalism.

---

1 Interface Definition Language is a language for specifying operations (procedures or functions), parameters to these operations, and data types.

## *Network protocol*

What is a protocol? A computer protocol can be defined as a well-defined set of messages (bit patterns or, increasingly today, octet strings) each of which carries a defined meaning (semantics), together with the rules governing when a particular message can be sent. However, a protocol rarely stands alone. Rather, it is commonly part of a "protocol stack", in which several separate specifications work together to determine the complete message emitted by a sender, with some parts of that message destined for action by intermediate (switching) nodes, and some parts intended for the remote end system.

In this "layered" protocol technique:

- One specification determines the form and meaning of the outer part of the message, with a "hole" in the middle. It provides a "carrier service" (or just "service") to convey any material that is placed in this "hole".
- A second specification defines the contents of the "hole", perhaps leaving a further hole for another layer of specification, and so on.

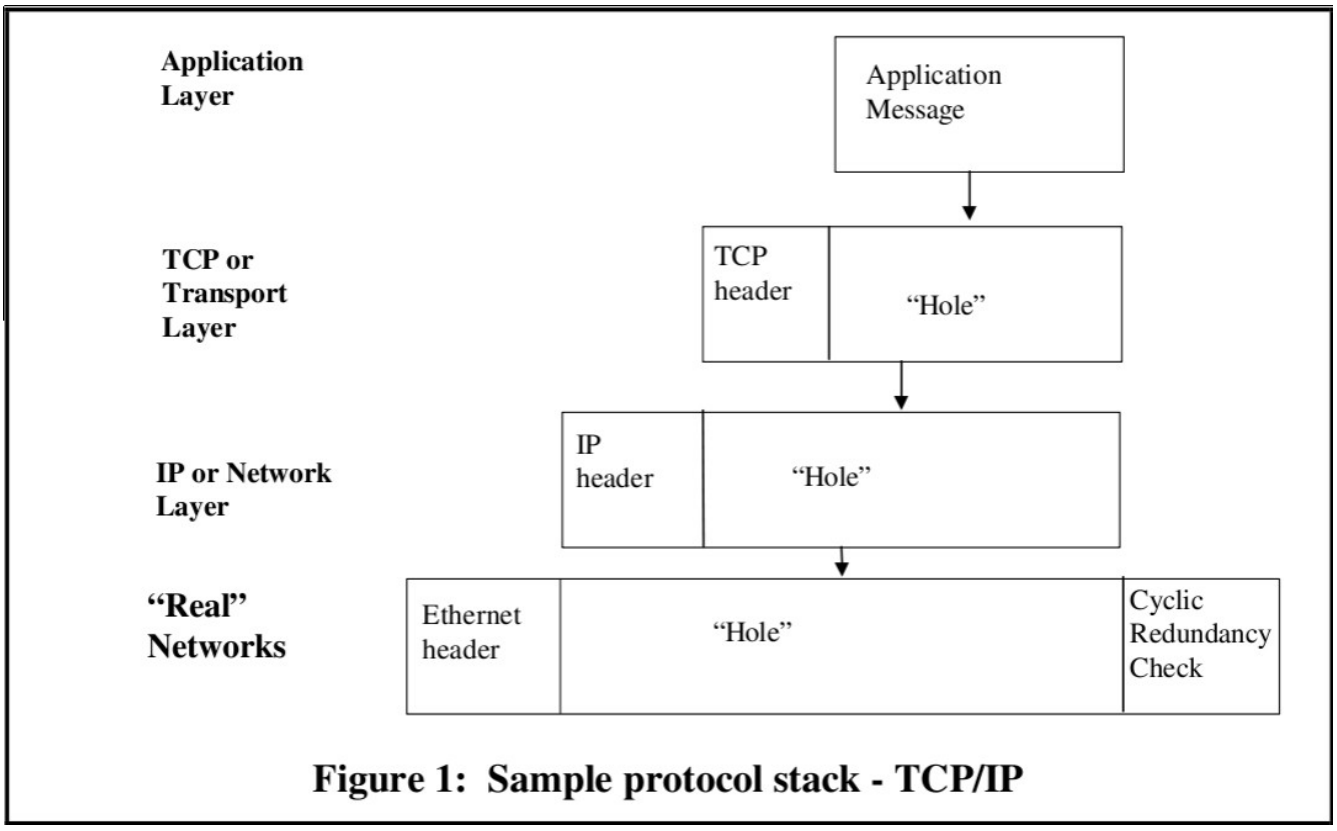



**Figure 1: Sample protocol stack - TCP/IP**

@SLIKA TCP/IP model illustrates a TCP/IP stack, where real networks provide the basic carrier mechanism, with the IP protocol carried in the "hole" they provide, and with IP acting as a carrier for TCP (or the the less well-known User Datagram Protocol - UDP), forming another protocol layer, and with a (typically for TCP/IP) monolithic application layer - a single specification completing the final "hole". The precise nature of the "service" provided by a lower layer (lossy, secure, reliable), and of any parameters controlling that service, needs to be known before the next layer up can make appropriate use of that service. We usually refer to each of these individual specification layers as "a

protocol".

Note that in @SLIKA TCP/IP model, the "hole" provided by the IP carrier can contain either a TCP message or a UDP message - two very different protocols with different properties (and themselves providing a further carrier service). Thus one of the advantages of "layering" is in reusability of the carrier service to support a wide range of higher level protocols, many perhaps that were never thought of when the lower layer protocols were developed.

When multiple different protocols can occupy a hole in the layer below (or provide carrier services for the layer above), this is frequently illustrated by the layering diagram shown in @SLIKA LAYERING.

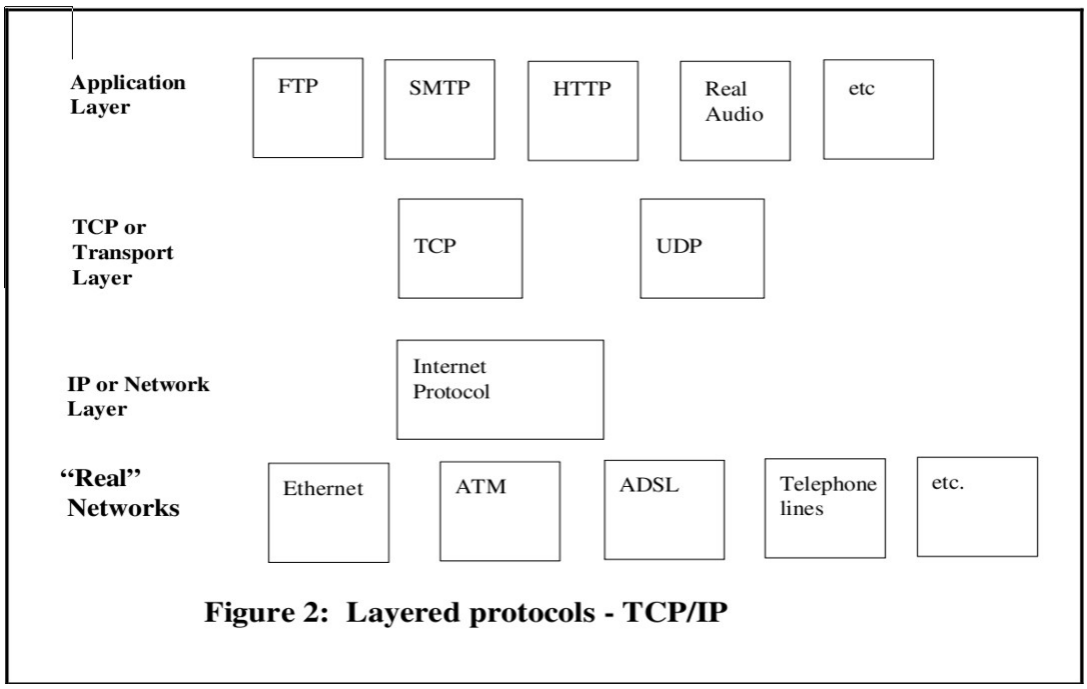



**Figure 2: Layered protocols - TCP/IP**

## *Network protocol specification*

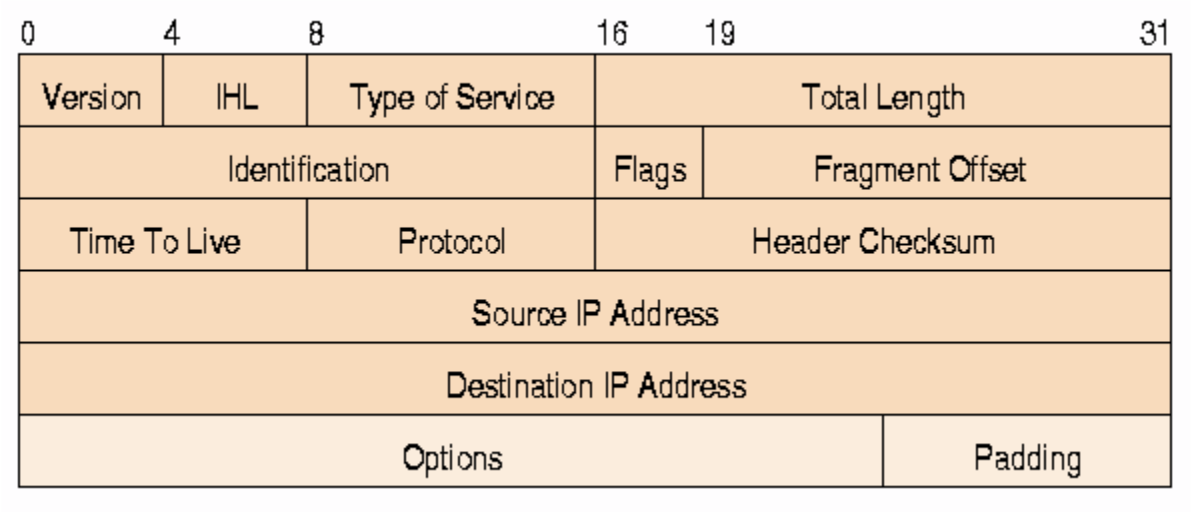
Protocols can be (and historically have been) specified in many ways. One fundamental distinction is between character-based specification versus binary-based specification:

1. *Character-based* The "protocol" is defined as a series of lines of ASCII encoded text. Also referred to as *text-based* protocols.
2. *Binary-based* The "protocol" is defined as a string of octets or of bits.

Character-based specifications are often designed as a command line or statement-based protocols. The communication of such protocols consist of series of lines of text each of which can be thought of as a command or a statement, with textual parameters (frequently comma separated) within each command or statement. The examples of such text based protocols are HTTP, FTP, POP3, IMAP, SMTP, SIP, XMPP, IRC etc. The common way of defining a text based protocol is with use of *Backus Naur Form* or simply BNF. It is very powerful for defining arbitrary syntactic structures, but it does not in itself determine how variable length items are to be delimited or iteration counts determined.

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
      rfc1123-date = wkday "," SP date1 SP time SP "GMT"
      rfc850-date  = weekday "," SP date2 SP time SP "GMT"
      asctime-date = wkday SP date3 SP time SP 4DIGIT
      date1        = 2DIGIT SP month SP 4DIGIT
                     ; day month year (e.g., 02 Jun 1982)
      date2        = 2DIGIT "-" month "-" 2DIGIT
                     ; day-month-year (e.g., 02-Jun-82)
      date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
                     ; month day (e.g., Jun  2)
      time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                     ; 00:00:00 - 23:59:59
      wkday        = "Mon" | "Tue" | "Wed"
                   | "Thu" | "Fri" | "Sat" | "Sun"
      weekday      = "Monday" | "Tuesday" | "Wednesday"
                   | "Thursday" | "Friday" | "Saturday" | "Sunday"
      month        = "Jan" | "Feb" | "Mar" | "Apr"
                   | "May" | "Jun" | "Jul" | "Aug"
                   | "Sep" | "Oct" | "Nov" | "Dec"
```

Binary protocols are more difficult to implement and their wire representation is not human-readable, but generally they are more efficient in both bandwidth usage and speed. For binary-based specification, approaches vary from various picture-based methods to use of separately defined notation (syntax) with associated application-independent encoding rules (serialization protocols).

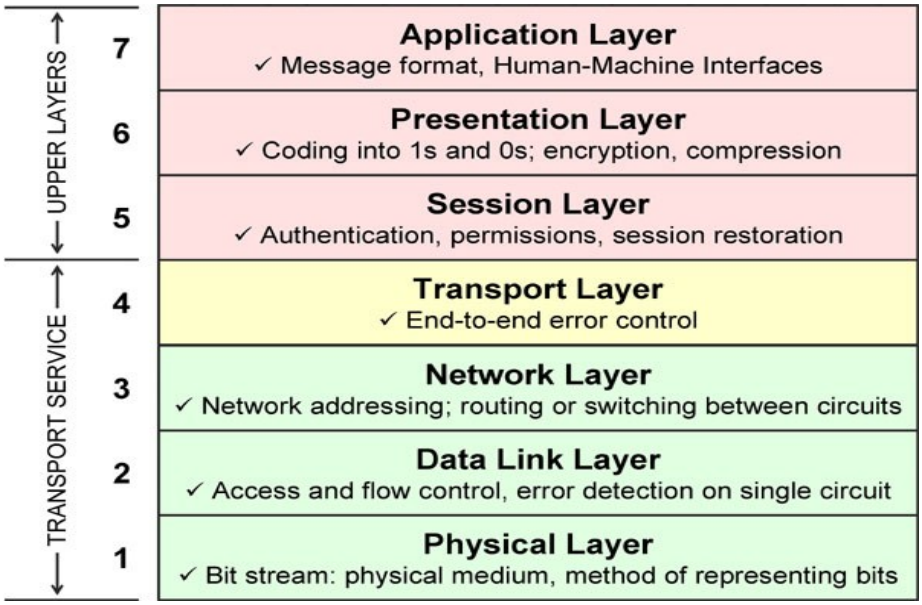



The later is called the “abstract syntax” approach. This is the approach taken with technologies such as ASN.1, Protocol Buffers, SUN-RPC, ONC-RPC, SOAP, REST etc. It has the advantage that it enables designers to produce specifications without undue concern with the encoding issues, and also permits application-independent tools to be provided to support the easy implementation of protocols specified in this way. Moreover, because application-specific implementation code is independent of encoding code, it makes it easy to migrate to improved encodings as they are developed.

```
FooProtocol DEFINITIONS ::= BEGIN

    FooQuestion ::= SEQUENCE {
        trackingNumber INTEGER,
        question       IA5String
    }

    FooAnswer ::= SEQUENCE {
        questionNumber INTEGER,
        answer         BOOLEAN
    }

END
```

## *Layering*

The layering concept is perhaps most commonly associated with the International Standards Organization (ISO) and International Telecommunications Union (ITU) “architecture” or “7-layer model” for Open Systems Interconnection (OSI) shown in @SLIKA OSI LAYERS.




While many of the protocols developed within this framework are not greatly used today, it remains an interesting academic study for approaches to protocol specification. In the original OSI concept in the late 1970s, there would be just 6 layers providing (progressively richer) carrier services, with a final "application layer" where each specification supported a single end-application, with no "holes". It became apparent, however, over the next decade, that even in the "application layer" people wanted to leave "holes" in their specification for later extensions, or to provide a means of tailoring their protocol to specific needs.

It is thus important that any mechanism or notation for specifying a protocol should be able to cater well for the inclusion of "holes".

## *Abstract and transfer syntax*

The terms abstract and transfer syntax were primarily developed within the OSI work, and are variously used in other related computer disciplines.

The following steps are necessary when specifying the messages forming a protocol (see Figure 8):

- The determination of the information that needs to be transferred in each message. We here refer to this as the semantics associated with the message.
- The design of some form of data-structure (at about the level of generality of a high-level programming language, and using a defined notation) which is capable of carrying the required semantics. The set of values of this data-structure are called the abstract syntax of the messages or application. We call the notation we use to define this data structure or set of values we the abstract syntax notation for our messages.
- The crafting of a set of rules for encoding messages such that, given any message defined using the abstract syntax notation, the actual bits on the line to carry the semantics of that message are determined by an algorithm specified once and once only (independent of the application). We call such rules encoding rules, and we say that the result of applying them to the set of (abstract syntax) messages for a given application defines a transfer syntax for that application. A transfer syntax is the set of bit-patterns to be used to represent the abstract values in the abstract syntax, with each bit-pattern representing just one abstract value.