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 same number on the other endpoint. 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 model:

- 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.

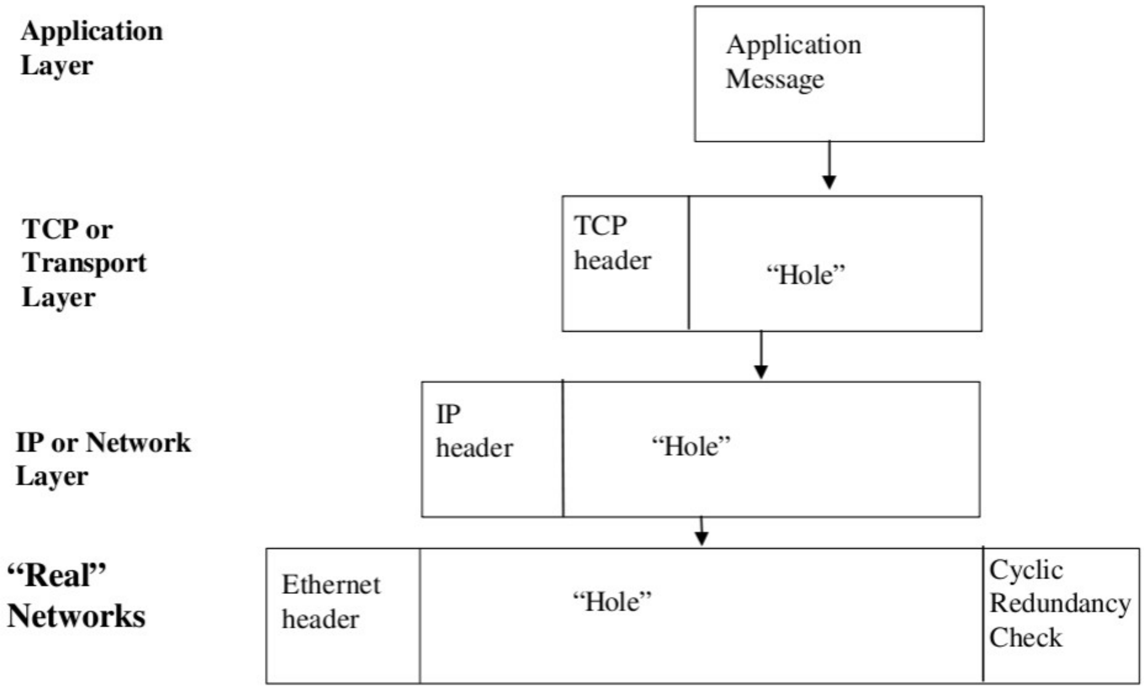


Figure 1: Sample protocol stack - TCP/IP

@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.

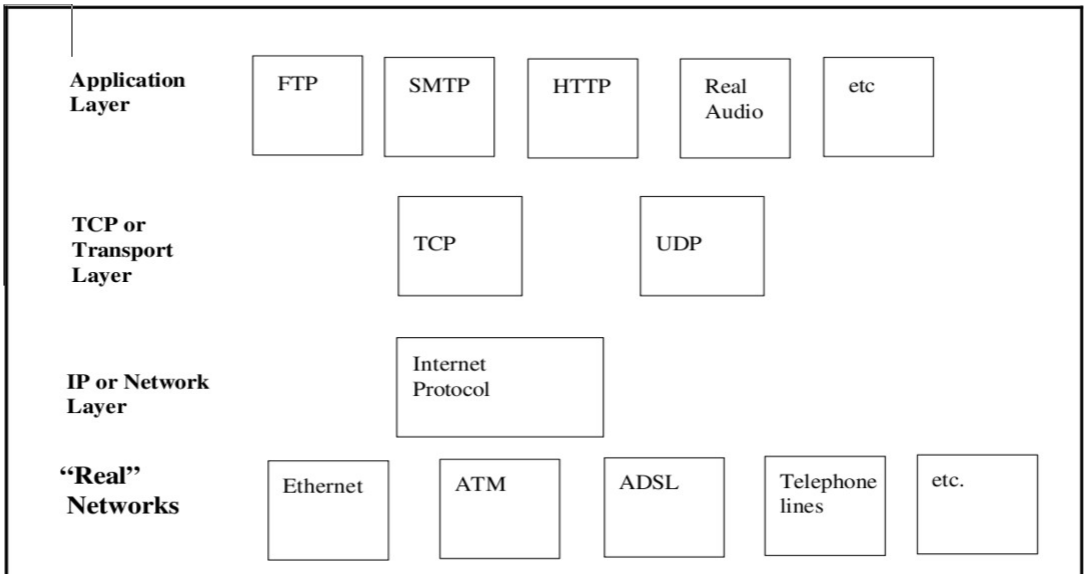


Figure 2: Layered protocols - TCP/IP

Protocol operations

Considering the layered model, lower layer network protocol provides services to a higher layer protocol. This services are exposed trough protocols interface. A protocol then performs certain operations to the point of servicing a higher layer protocol which requested a service. For example TCP provides connection-oriented, ordered and reliable transfer of data from one TCP endpoint to another, for higher level protocol such as SMTP or FTP. This is achieved using operations such as handshaking, acknowledging and signaling.

A simple protocol operation would be to start the operation and then wait for it to complete. But such an approach (called **synchronous** or **blocking operation**) would block the progress of a program while the communication is in progress, leaving system resources idle. The thread of control is blocked within the function performing the protocol operation, and it can use the result immediately after the function returns. This means that the processor can spend almost all of its time idle waiting for a certain protocol operation to complete.

Alternatively, it is possible, to start the operation and then perform processing that does not require that the operation has completed. This type of operation is called **asynchronous** or **non-blocking** operation. Any task that actually depends on the operation having completed (this includes both using the return values and critical operations that claim to assure that a protocol operation at hand has been completed) still needs to wait for the protocol operation to complete, and thus is still blocked, but other processing which does not have a dependency on the protocol operation can continue. Situations in which a protocol operation should operate in asynchronous mode are those that can get extremely slow, for reasons such as writing or reading from a hard drive (in the context of network file systems).

Additional issue which needs to be addressed concerning protocol operations is the time period in which they must perform. This are called operation timeouts and they vary a lot and usually depend on the semantics and the context of the operation itself.

Further we can separate operations to ***passive*** and ***active***, considering if the operation is initiating communication with the other endpoint (eg. active), or is simply waiting for the communication to be initiated by the other endpoint (eg. passive). Also we can utilize terminology such as server operations and client operations, for passive and active operations, respectively.

Protocol data units

What it boils down to, a protocol operation is actually the exchange of messages. These messages are transmitted across a virtual communication line between endpoints or peers that reside on the same layer and thus speak the same protocol. The abstraction level that the layering approach provides us allows us to think of these peers being directly connected. The message that carries the required semantics among the protocol peers at hand is called the ***Protocol Data Unit*** *(PDU).*

A PDU in general consists of a header which contains some kind of protocol-control information and possibly user data of that layer. The other part is considered to be the payload data, formally referred to as ***Service Data Unit*** (*SDU*). The semantics and syntax of the SDU is know to the higher layer protocol which is being serviced by the lower layer protocol. The lower layer protocol has no such knowledge and thus SDU is considered as a “hole” to the protocol at hand.

For example in relation to the OSI model layers, the Physical layer PDU is a bit, the Data Link layer PDU is referred to as a ***frame****,* while the Network layer and the Transport layer use the terms ***packet*** and ***segment***, respectively.

PDUs are commonly ***binary-based*** or ***text-based.*** Generally with binary-based PDUs protocol gains in speed and bandwidth usage, but in turn has to deal with different integer sizes and sign, floating point representations, bit and byte ordering. On the other hand text-based PDUs are relatively simple to handle as they are most commonly ASCII encoded and thus human-readable and easy debugged. Of course it's clear that such PDUs are heavy on bandwidth usage.

Network protocol specification

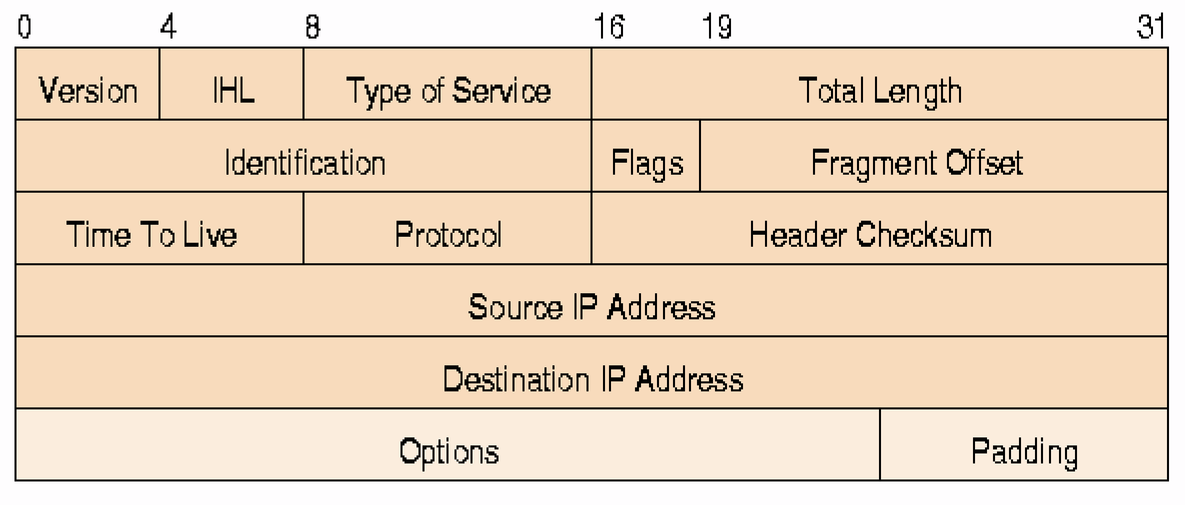
Protocols can be (and historically have been) specified in many ways. One fundamental distinction is between protocols that utilize character-based PDUs versus binary-based PDUs. Such specifications are commonly referred to as character-based and binary-based specification, respectively:

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

Character-based protocols 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.

```
SPACE := “ “
CRLF := “\r\n”

HTTP-REQUEST := HTTP-REQUEST-LINE HTTP-REQUEST-HEADERS HTTP-MESSAGE-BODY

HTTP-REQUEST-LINE := HTTP-METHOD SPACE HTTP-URI SPACE HTTP-VERSION CRLF

HTTP-METHOD :=“OPTIONS” | “GET” | “HEAD” | “POST” | “PUT” | “DELETE” | “TRACE” |
“CONNECT”

HTTP-VERSION := “HTTP/1.0” | “HTTP/1.1”

HTTP-REQUEST-HEADERS := HTTP-REQUEST-HEADERS HTTP-REQUEST-HEADER | HTTP-REQUEST-
HEADER

HTTP-URI := ...
HTTP-REQUEST-HEADER := ...
HTTP-MESSAGE-BODY := ...
```

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).

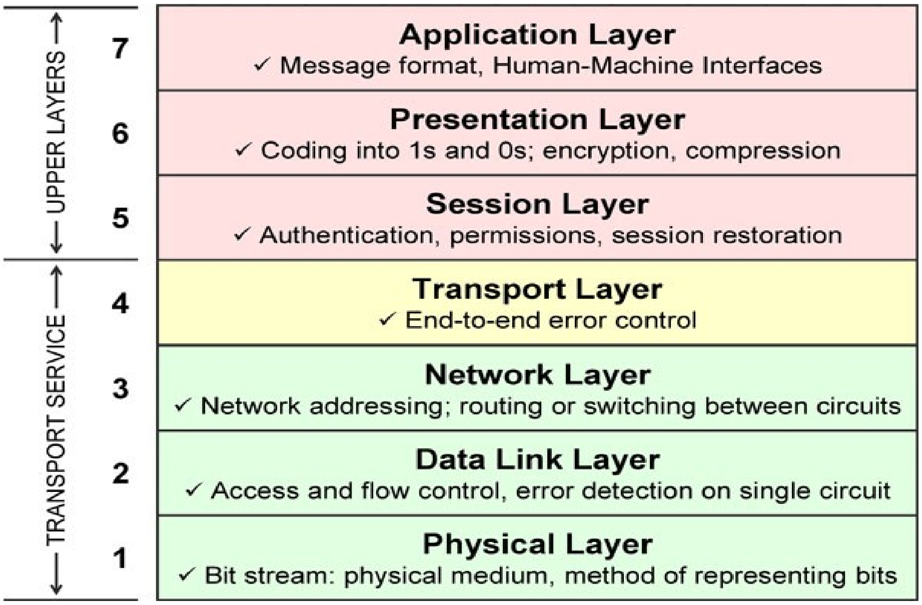


The later is called the “abstract syntax” approach. This is the approach taken with technologies such as ASN.1, Protocol Buffers, SUN-RPC, ONC-RPC, SOAP, REST etc. It has the advantage that it enables designers to produce specifications without undue concern with the encoding issues, and also permits 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. To reduce their design complexity, most networks are organized as a stack of layers or levels, each one built upon the one below it. The number of layers, the name of each layer, the contents of each layer, and the function of each layer differ from network to network. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. This is know as ***encapsulation***. In a sense, each layer is a kind of virtual machine, offering certain services to the layer above it.


UPPER LAYERS
TRANSPORT SERVICE
7
Application Layer
✓ Message format, Human-Machine Interfaces
6
Presentation Layer
✓ Coding into 1s and 0s; encryption, compression
5
Session Layer
✓ Authentication, permissions, session restoration
4
Transport Layer
✓ End-to-end error control
3
Network Layer
✓ Network addressing; routing or switching between circuits
2
Data Link Layer
✓ Access and flow control, error detection on single circuit
1
Physical Layer
✓ Bit stream: physical medium, method of representing bits

Between each pair of adjacent layers is an interface. The interface defines which primitive operations and services the lower layer makes available to the upper one. When network designers decide how many layers to include in a network and what each one should do, one of the most important considerations is defining clean interfaces between the layers. Doing so, in turn, requires that each layer perform a specific collection of well-understood functions. In addition to minimizing the amount of information that must be passed between layers, clear-cut interfaces also make it simpler to replace the implementation of one layer with a completely different implementation (eg., all the telephone lines are replaced by satellite channels) because all that is required of the new implementation is that it offer exactly the same set of services to its upstairs neighbor as the old implementation did. In fact, it is common that different hosts use different implementations.

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".

PDU specification

A PDU is specified using various data types. Let's divide data types into primitive types or basic types and constructed types or composite types. Primitive types include integers, floating point numbers, characters, booleans etc. Constructed data types are constructed using primitive data types and other constructed types, they provide enclosure for some kind of set consisting of primitive types. Structures, arrays, strings, unions, enumerators etc., fall into constructed data type category.

Different kinds of computers use different conventions for the ordering of bytes within data types that are multiple of a byte. Some computers put the most significant byte (eg. MSB) within such data type first, this is called “***big endian***” order, and others put it last, thus called “***little endian***” order (eg. LSB). The same goes for bit ordering, all though it's rare to find little endian bit ordering into the wild, both on processor architecture and network protocol specifications. Integer sizes also differ amongst architectures, not to mention floating point representations. Strings have different character encodings (ie. character set or simply ***charset***).

@PROC_ARCH_ATTR

So that machines with different conventions and specifications can communicate, the network protocol specification must clearly define these attributes to every data type transmitted over the network.

Formally we can define integers as a data type which represents some finite subset of mathematical integers (integral data type). When specifying an integer data type one must consider the following attributes:

1. **Size** – Most commonly integer sizes are byte multiples, and as such are usually named with the following size specifiers: *char, short, long, long long* or *hyper* respectively pertaining to sizes of 1,2,4,8 bytes. It's not uncommon to define a bit multiple integer size, for instance 13 bits offset field in the IP PDU specification. The smallest integer size is of course 1 bit, and the largest is undefined.
2. **Byte order** – Byte order concept is only valid with byte multiple sized data types so only byte sized integers must have byte ordering defined. Thus called byte-sized integers.
3. **Bit order** – Data types that have size defined as bit multiple are called bit-sized data types, therefor such integers are called bit-sized integers. Bit order can be specified for both byte-sized and bit-sized integers. When specified for bit-sized integer the bits are arranged accordingly for the integer as a whole. For byte-sized integers the bit order is considered bytewise and thus is set for each byte.
4. **Sign –** An integer can be signed or unsigned. Signed integers are stored in a computer using 2's complement. Distinction must be made as integer operations are different for unsigned versus signed integers.

In computer science, floating point describes a system for representing real numbers which support a wide range of values. The following are the attributes applicable to floating point data type:

1. **Representation –** There are several floating point representations used today in computing. Different processor architectures utilize different representations. These are: IEEE754, VAX, Cray, IBM...
2. **Size** - The size of a floating point type is usually determined by it's representation, and most commonly are byte sized.
3. **Byte order** – As a byte-sized data type it must have a defined byte order.
4. **Bit order** – Similar to byte-sized integers bit order is defined bytewise (not as a whole).

One more primitive data type is a character. A character data type is used to store symbols such as alphanumeric text, whitespace, punctuation and others. These symbols exist at a higher level of abstraction then integers and floating point numbers. But similarly to these primitive types, characters also must have a mapping from character abstraction to a certain binary representation that can be stored in computer memory or transmitted across a network. Essentially a character is mapped into an integer data type, so we can use object oriented paradigm terminology to describe a character type as being a specialized form of an integer type. As such a character inherits all of the mentioned integer data type attributes to additionally introducing some of its own:

1. **Character set** – Also referred to as a charset, character encoding, character map or a code page. It represents a mapping of symbols into an integer for the purpose of storing these symbols in the computer memory or transmission over the network. These mapping can be either specified using a predefined set of symbol to number conversion (ASCII) or using an encoding algorithm (Unicode).

Boolean types are simple types for representing logical values. Their mapping to some form of binary representation must be defined. This data type is also derived from integer data type and additionally must define two attributes:

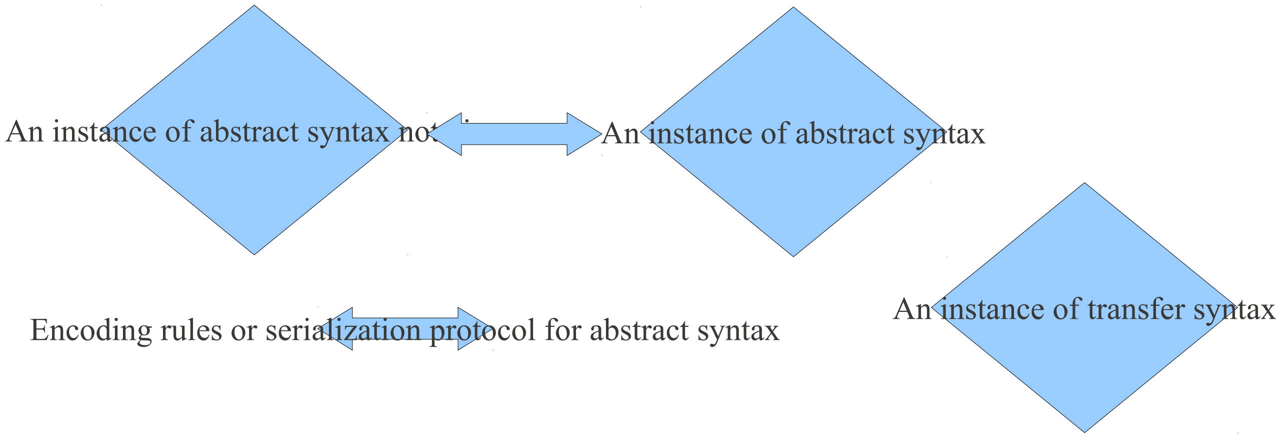
1. **True value –** The integer value that represents a true boolean value.
2. **False value** – The integer value that represents a false boolean value.

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. These terms will provide us with terminology for formally defining Wire language and it's purpose.

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. 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 messages for a given application defines a ***transfer syntax*** for that particular abstract syntax. Therefor, 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.


An instance of abstract syntax not
An instance of abstract syntax
An instance of transfer syntax
Encoding rules or serialization protocol for abstract syntax