# Generic frame parsing model

Most of the currently widely used communication protocols such as those in OSI/ISO layer exhibit the same pattern of data representation formats. As shown in Table 1, their data frames generally have a header, a message body or a payload, and a footer.

Most of the time the metadata of a message and synchronization information are included in the header section while error detection information like checksums or CRCs are included in the footer section. Apart from this resemblance, the message body or the payload can have some form of structural similarity which can be used to express information carried by different frame formats. To represent this structural similarity the following entity models are introduced:

Data Field: A data field is a conceptual representation of a single information that is carried by a frame. This can be a method name in HTTP or a destination address inside IP protocol. It has a name, data type and other properties that are relevant in encoding and decoding values to and from raw data. Filter Field: A filter field is basically a data field with one minor difference. Filter fields are used for identification mechanism to distinguish one frame from another or to gather a specific intelligence out of a data stream. These fields make use of expressions to evaluate whether part of data stream fulfills a given criteria and help identify frames of interest. Most of the time filter fields reside inside frame headers but users can specify filter fields that evaluate the message body. Unlike lower level filtering mechanisms like in BPF et al [7] our model is implemented at higher application levels which brings its own performance issue. Therefore more

efficient filtering algorithms will be included to gain better performance in the future.

This is an information source that define part of a protocol and contains related data fields which are transmitted together as a chunk over a communication medium. For example, telemetry frames which inform status of a remote vehicle or orbital commands to be sent to a satellite can be defined as a single data frame.

Functions: To supplement data processing task at a field or frame level basic algorithms such as CRC calculation and simple mathematical functions are included in the model. Apart from generic functions provided, custom functions can be composed from existing functions in the model because a simple compiler enables accessing information through simple syntaxes.

Logical Conditions: This feature is one of the planned features to be included in the future. The general representation model entities described above can be used to describe protocol frames in general. A given frame can have a text encoded message, like in HTTP, or be composed of a series of binary encoded data structures, like in IP. Therefore, our model, shown Fig. 2, shall shall contain two models that handle both data stream representations namely, binary streams and text streams.

For text encoded data frames a generic representational model that has text fields with some separation character or string of characters can be formulated. To identify such frames one from the other a separator character or a string of characters can be added at the beginning (as preamble) or the

end (as post amble) of the model. This model is illustrated in the following table.

For binary encoded data frames, on the other hand, we can formulate a similar model with some modification. Most of binary encoded frames don’t need separators since every binary encoded data structure has specific length. The length and position of each field is determined by its data type and its predecessor’s position. For example, a field with single point precision is 4 bytes long and as such the next field’s position is calculated by adding 4 to the current field’s position in the frame.

Formulating a model makes data stream parsing job easier for users or developers. This is because it can substantially abstract the details of handling data structures and does the heavy lifting in the process. All fields have their parsed values stored in text encoded format which helps in reconstructing their original data type representation because it is easier to parse values from text to common data structures. The text encoding standard can be ASCII, Unicode, UTF7, UTF8 or any other standard and it will be set by the user.

Text Stream Parsing

Some data frame formats like those in the application layer of OSI/ISO standard represent information as a text encoded using standard encoding mechanisms usually ASCII and Unicode. Since the length of bytes included in these fields of such frames is variable depending on their content, separator characters like spaces, commas and others are used to distinguish frames and fields. A text stream parser will use these characters to split and process

information included in each field. Since string manipulation can be a resource intensive job for large data streams Enumerables in the . Net framework are used whenever possible to avoid memory usage for intermediate states.

Binary Stream Parsing

Frames that have their information serialized as binary stream can be parsed using standard data structures. Based on data structure algorithms, inputs of a position and location of a field in a frame are enough to get its value out of a stream. In Table 4, some standard data types that are supported by the parsing model are listed with their binary representation lengths.

As stated before our data frame representation model in communication protocols gives us the ability to manage information at a higher level than what would otherwise be a cumbersome job to handle whenever data processing is needed. One of such data processing cases is in translating information from one format to another in communicating two or more nodes. In this section, a simple yet powerful routing mechanism developed on top of the parsing model above is described. The process of routing employs a routing table that is filled with a list of route entries. A data route entry specifies from which source node information be taken and to which sink node it be put. An additional data processing needed during the routing can be specified in the route definition. These additional processing specifics, which will be studied in the future works, can be trimming, filtering, multiplying or others manipulations necessary for a specific applications.

A route entry has a source node name, source frame name, source field name, sink node name, sink frame name and sink field name properties. Based on what information is included in these properties a routing process can be of field-by-field, frame-by-frame or node-by-node type.

Field-by-Field Routing: This is the basic form of routing where a specific source field value is mapped directly to a sink field. Since this routing needs to identify fields by name the user should know and specify the name of each field explicitly.

Frame-by-Frame Routing: Mapping each and every data fields in all frame definitions can be unnecessary task if connected nodes have similar frame definitions. In such cases all raw data can be directly forwarded to the sink node without any additional processing. This is especially useful when the information nodes are interfaced by incompatible hardware and special bridge is needed that trans code information. For instance, packets received through UDP protocol can be relayed to a microcontroller connected through a serial port and vice versa.

Node-by-Node: This is basically passing on all data frames of any kind from a source node to sink node and it is the extreme case of frame-by-frame routing.

Auto-Mapping

To avoid the cumbersome job of listing each field name in routing tables a simple algorithm is used to automatically map fields across connected nodes

based on the metadata included in frames and the similarity of field names and data types.

Empty Fields

During the process of frame parsing and data routing some fields may end up being empty. This will create a problem when decoding data at the receiver end if the sender and the receiver have no common understanding how empty fields are represented. To avoid this issue this model uses special characters that are to be specified by the user such as zero, empty text or any other character of choice.

Endianness

Binary data representation can have different value depending on the endianness supported by the system it is encoded on. Therefore, the binary parser should check the endianness of the system before returning field values to the user which can be used by a simple XOR logic. Convert with Big Endian — (Is System Big Endian) XOR (User Need is Big Endian) Value = Convert with Little Endian — Otherwise

IMPLEMENTATION

This generic data parsing model is implemented in C# . NET with its own editor UI. The UI includes features for creating parsers, editing frames and fields, and testing against data sources. The newly created parser models can be saved into a disk for later usage. The implementation was used to communicate with Flightgear simulator, Matlab models over UDP and serial port interfaced microcontrollers with more tests planned in the future.

There is no special requirement in using C# . NET apart from familiarity and faster prototyping need. Therefore, anyone can implement this model with other programming languages for additional requirements like getting better performance or cross-platform deployment need.

The generic parsing model described here has proved to be an important tool for creating a parser model for many protocols tested. Besides, it was used as a data routing tool to interpret and transfer remote telemetry data from different devices that are incompatible in their interfacing mechanism and in their data frame formats. Information received from these nodes was filtered and saved to and replayed back from storage disks with ease. With more works in improving performance and adding features, it can be one of a must have tool inside the toolset of protocol analyzers and developers.

Future works and improvements

Performance: programming languages such as C or C++ can bring more performance in parsing algorithm implementations. Additionally fast and efficient parsing algorithms like the ones discussed in [1] and [7]can be included for binary parsing. Concurrency: most data streams have high throughput that can easily overwhelm our model. So our implementation should consider concurrency both in design and programming language used. Usability: Firing a UI every time an analyst tries to do a job might not be a feasible form of doing things. Other forms of user interaction like CLI or command terminal integration can bring better usability. Logic Conditions: Having logical conditions like if then, for loops and others can make the model more robust.