Get stuffed: Tightly packed abstract protocols in Scheme

John P. T. Moore
Thames Valley University, UK
moorejo@tvu.ac.uk

Abstract

This paper describes a layered approach to encoding and decoding tightly packed binary protocols. The protocols developed are based on an abstract syntax described via an s-expression. This approach utilises simple built-in features of the Scheme programming language to provide a dynamic environment that facilitates the development of extensible protocols. A tool called Packedobjects has been developed which demonstrates this functionality. An example application is presented to illustrate the flexibility of both the tool and the Scheme programming language in this domain. In particular we will show how it is possible to embed this technology into another application programming language such as C to power its network communication. Using the example application we will also highlight the choices available to the developer when deciding whether or not to embed such technology.

1. Introduction

The International Standards Organisation (OSI) 7 Layer reference model provided an academic framework for the design of network protocols and standards [7]. In comparison to the OSI model the TCP/IP model adopted a more simplified approach where amongst other changes the Presentation Layer was consumed by the Application Layer. As such, a network applications programmer needs to consider how to structure their data when transferring it across an internet. A number of competing technologies have been developed and continue to develop in this area. When it comes to structuring data in a human-readable way, XML has dominated. However, approaches to structuring binary data range from serialising native data structures to transforming an abstract syntax into a more concise binary form. Binary protocols continue to play an important role in supporting network applications. Common uses include network games and mobile communication. In addition, Google released their work on Protocol Buffers which was created to address issues they faced in the area of high performance computing [2]. In this paper we discuss Packedobjects, a tool which was originally developed for the Chicken Scheme language and is now being maintained as a Guile module [6]. Before describing Packedobjects we will first provide an overview of some relevant techniques for producing binary protocols.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2009 Workshop on Scheme and Functional Programming

2. Zeros and ones

Serialising data structures for transmission across a network is a common technique. The programmer might have to handle differences in byte ordering if communication takes place across different hardware platforms. In addition, the protocol designer is restricted to describing the network protocol in terms of the native data structures available in the language used. An alternative approach might involve using an abstract syntax to describe the network protocol. This introduces some complexity. Ultimately this abstract syntax will need to be represented by the programming language. The traditional way of handling this is not dynamic. A compiler is used to transform the abstract syntax into the native language code. Typically the code generated will be combined with application specific code and linked with vendor supplied code. This is the approach which is taken by numerous Abstract Syntax Notation 1 (ASN.1) tools [1]. ASN.1 originates from the world of telecommunications. The philosophy of ASN.1 is to provide a rich abstract syntax to describe network protocols and this syntax should be transferred into binary before transmission. Different techniques, or encoding rules, can be applied to make this transition from abstract syntax to binary. The abstract syntax allows the protocol designer to think at a higher level and provides a common ground between application developers working in different programming languages. By using the Scheme programming language we can provide a more dynamic approach where s-expressions are used to describe the high level syntax. In keeping with a minimalistic tradition adopted by Scheme, we can represent a subset of the ASN.1 standard when describing our protocols. By simplifying the abstract syntax we can provide a dynamic runtime representation within an s-expression which encourages exploration in the read-eval-print loop (REPL).

3. A layered approach

Figure 1 shows how Packedobjects compares against the OSI model. At the Application Layer, Packedobjects allows the creation of buffers. A buffer contains encoded data either ready to be sent across a network or encoded data ready to be decoded. Packedobjects has been designed to allow buffers to be created within both C and Scheme. For example, the application developer can decide to use C for all network communication and therefore create the buffers in C. In either case Scheme is used to process the contents of those buffers and this takes place at the Presentation Layer. Transportation of the encoded data is shown happening at the Transport Layer. In this case we have indicated UDP is used. Packedobjects can also work over TCP, however some additional work is required to delimit application messages over this byte stream oriented transport protocol.

Application Layer	Buffer manipulation
Presentation Layer	Encoding/Decoding
Session Layer	
Transport Layer	UDP datagrams
Network Layer	Internet
Data Link Layer	
Physical Layer	

Figure 1. The OSI and Packedobjects

4. Data is code

Using an s-expression we can make use of quasiquote, unquote and unquote-splicing to help specify and manipulate our network protocol description and its values. To illustrate some of this flexibility we will use a fictitious protocol which describes shopping for food and drink. In the process we will introduce some abstract data types used by Packedobjects.

```
(define booze
  ’(sequence-of
    (beer null)
    (nibbles null)))
```

We start by defining a *sequence-of null* types. The *sequence-of* type is a compound data type which consists of a repeating sequence of other data types, in this case a sequence of two *null* types. The *null* type is one of several atomic data types available in Packedobjects. It is an unusual data type in that it requires no value and is typically used as an acknowledgement in protocol specifications. More familiar atomic data types include integer, boolean, enumerated and various string types.

```
(define grub
  ’(sequence
    (pizza null)
    (salad null)))
```

In addition to our drinks we should have some food. In this case we use the *sequence* data type which specifies both pizza and salad.

```
(define trolley
  ‘(trolley set
    (drink ,@booze)
    (food ,@grub)))

(define basket
  ‘(basket choice
    (food ,@grub)
    (drink ,@booze)))
```

To carry our food and drink we could use a trolley or use a basket. The trolley is large enough to carry both but we must choose between the food or drink if we use a basket. We have introduced two new compound data types. The *set* data type is a flexible type which allows an unordered sequence of other types. Any item of a set is also optional. Therefore, using the trolley we could decide to only pack some food. Using the basket we must choose between either the food or drink. The *choice* data type is a compound data type which enforces this restriction. In both cases, we have used unquote-splicing to reuse our definitions of food and drink and therefore are able to produce concise protocol descriptions. Having defined a protocol we must specify values according to their description.

```
(define thirsty
  ’(basket
    (drink
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles)))))

(define hungry
  ’(basket
    (food
      (pizza)
      (salad))))

(define thirsty+hungry
  ‘(trolley
    ,(cadr hungry)
    ,(cadr thirsty)))
```

Depending on our mood, we might be thirsty, hungry or both. In the case of being both thirsty and hungry we will use a trolley. This example illustrates the use of unquote to reuse our definitions of being hungry and thirsty. Note how we apply *cadr* to represent removing the basket from the value list ready to be placed in the trolley instead. Having defined our protocol and values we are ready to encode the data ready for transmission over a network.

```
(let* ( (bufsize 10)
        (buffer (make-buffer bufsize))
        (encoder (make-encoder buffer trolley))
        (size (encoder ’pack thirsty+hungry))
        (pdu (pdu-from-buffer buffer size)) )
  pdu)
```

The output of the encoder is a tightly packed bit stream. In this case just two bytes are required to represent:

```
(trolley
  (food (pizza) (salad))
  (drink ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))))
```

Conversely, the decoder will take a tightly packed bit stream and reproduce a list of values.

```
(let* ( (bufsize 10)
        (buffer
          (make-buffer-from-string pdu bufsize))
        (decoder (make-decoder buffer trolley)))
  (decoder ’unpack))
```

The resulting output will ordinarily be equal to the original value list but in this case the *set* data type was used and this represents a special case. The ordering of decoding will match that of the protocol description. Therefore in this example we get:

```
(trolley
  (drink ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles)))
  (food (pizza) (salad)))
```

The process of encoding and decoding is completely dynamic. Figure 2 summarises the encoding process. The encoder obtains data by dynamically combining the values supplied with data from

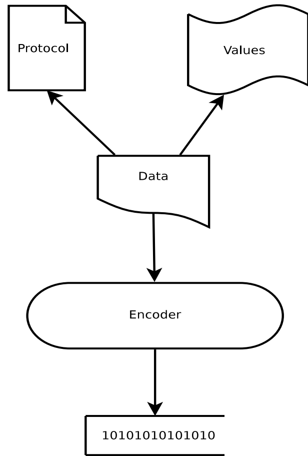

Figure 2. Dynamic encoding

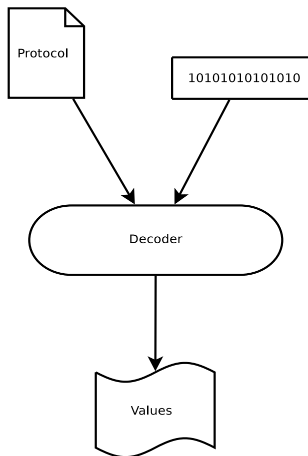

Figure 3. Dynamic decoding

the protocol specification. The encoder simply traverses the result calling the appropriate foreign functions corresponding to the underlying C based encoder. The end product is a Protocol Data Unit (PDU) which is ready to be transported across a transmission medium. The encoding produced is an unaligned bit stream based on Packed Encoding Rules (PER) [3].

The reverse process of decoding the PDU is more straight forward as summarised in figure 3. Here the protocol specification is used to drive foreign function calls to the C decoder which returns back values to Scheme to be combined into a list. Both the encoding and decoding processes illustrate how a clear divide exists between the low level C routines and the high level s-expression used to represent data. In the following section we will further describe the lower layer routines.

5. Bit fiddling

Bit manipulation is sometimes viewed as the practice of hackers [8]. In this section we will attempt to describe the techniques used by Packedobjects to pack and unpack bits in an accessible way to the reader. Both the encoder and decoder operate on words. Thus, working with strings involves multiple calls to encode or decode individual characters. As a result, protocols that are dominated with

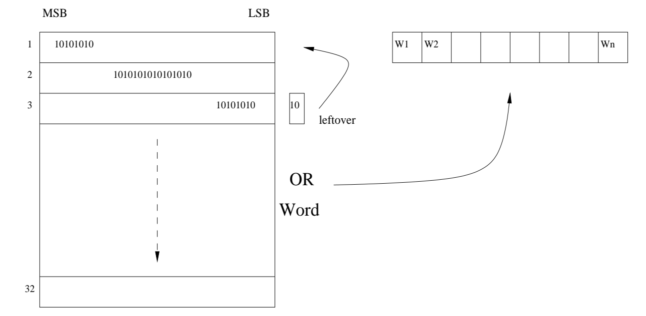

Figure 4. Encode buffers

string data are not well suited to Packedobjects. In a worst case scenario you may try to encode or decode a large amount of 8 bit strings. Packedobjects has no notion of byte boundaries, therefore the strings could start at any bit position within a contiguous block of memory. Reading the contents of such a string would require bit manipulation. The approach taken by Packedobjects is that ”every bit counts”. This is reinforced by the fact the default string type encodes in 7 bits. Although it may appear an extreme approach to take it does allow for a simplified view of how all types are encoded and decoded into words. The following subsections will illustrate this.

5.1 The encoder

The encoder works using two buffers [4]. One is fixed in size corresponding to the number of bits within a word, the other can be dynamically allocated. The word size is determined by the hardware platform and equals 32 bits in the example given. The fixed buffer can be visualised as an array of 32 words as depicted in figure 4. The dynamic buffer is typically created to accommodate the largest PDU so effectively operates as a statically allocated piece of memory. The fixed buffer is used to construct the bit sequences before they are copied across to the dynamic buffer. A bit sequence is copied to the appropriate word of the fixed buffer and then shifted into position. The bits are aligned so that after an OR operation on the array of words a single word is produced which can then be copied across to the dynamic buffer. This sequence is illustrated in figure 4. To begin with the bit pattern ”10101010” is copied to the first word in the fixed buffer. The eight bit pattern must be shifted so that it follows the network byte order and therefore has its most significant bit (MSB) at bit position 32. The next bit pattern ”1010101010101010” is added to the second word and shifted so that its MSB starts at bit 24. This leaves room for only eight more bits to be added within the third word. If, for example, the ten bit pattern ”1010101010” is to be added, then the eight most significant bits would be copied to the remaining room in the fixed buffer. The entire fixed buffer then has its contents OR’ed and the resulting word is copied to the dynamic buffer. The two bits left over are put back into the fixed buffer starting from the MSB of the first word. The pseudo code for the encode algorithm is provided in figure 5. The algorithm makes just two tests to see whether a word boundary is crossed in the fixed buffer and whether a full word exists already. The algorithm is recursive. It calls itself whenever there is a value left over to encode after a full word has been copied to the dynamic buffer.

5.2 The decoder

The decoder algorithm (figure 6) is slightly more straight forward than the encoder algorithm. A PDU is decoded by masking off the desired bits to form a value. The size of a word determines the size of the window which is placed over the data to decode. The window

```
BEGIN /* encode */

   accept a number and a bit length

   IF the bit pattern is unable to fit into the space
   available in the current word of the fixed buffer THEN
      fit as many bits in as possible
      OR the fixed buffer
      copy the result to the dynamic buffer
      reset the fixed buffer
      GOTO BEGIN to encode the leftover bit pattern
      RETURN
   ENDIF

   copy the number to the correct position in the
   next word of the fixed buffer

   IF we have a full word already THEN
      OR the fixed buffer
      copy the result to the dynamic buffer
      reset the fixed buffer
   ENDIF

END */ encode */
```

Figure 5. Encode algorithm

```
BEGIN /* decode */

   accept a bit length

   IF current bit position has reached a word boundary THEN
      fetch a word from the buffer
      store a copy of the word
   ENDIF

   mask out (AND) the bit pattern from current word

   IF bit pattern crosses word boundary THEN
      fetch the next word from the buffer
      store a copy of the word
      obtain the missing part of the bit pattern
      merge (OR) the two bit patterns together
   ENDIF

   return the result

END */decode */
```

Figure 6. Decode algorithm

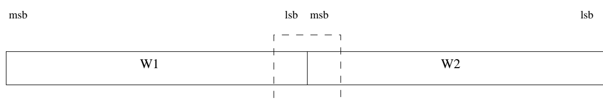

Figure 7. Decode window

moves in word sized increments over the PDU. Provisions must be made to handle bit patterns that cross over word boundaries. Values obtained from different words must be merged together to form a single bit pattern. Figure 7 shows that the area between two words can contain the desired bit pattern. As with the encode algorithm, just two test conditions exist: one to examine whether a new word should be fetched from the PDU buffer and one to examine if the value to extract lies between two word boundaries. By storing a copy of the last word obtained, it may be possible to avoid fetching a word each time the routine is called. Having described the encoder and decoder we will have completed our discussion on the layered approach of the tool and can now focus on a practical example of its use.

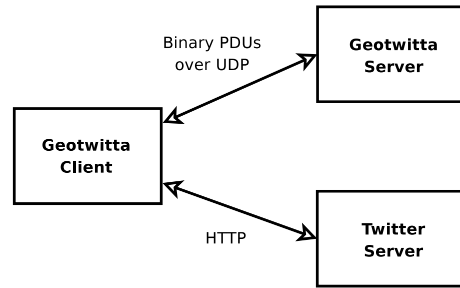

Figure 8. Application architecture

6. Example application

The previous sections provided an insight into the flexibility of using an s-expression to represent an abstract syntax and described its transformation into bits. In this section we will provide a more practical example that also highlights the flexibility of the Scheme language itself. We will describe an application that interfaces to the social networking and micro-blogging service Twitter. The application, known as geotwitta, is able to calculate the distance of other users and then post the result to the user's account [5]. Figure 8 summarises the architecture of the application. In order to calculate the distance of other users a server is required to manage the location of each user. Each client simply ”pings” in its coordinates to the server and in response retrieves a list of the distances of other users. Any new responses returned are then posted to Twitter using the HTTP protocol. An example post might appear as follows: #geotwitta @jptmoore appears to be about 18791.955 kilometers away from me.

6.1 The design

Although a simple protocol and simple application, it provides enough scope to show how the Scheme language and in particular Guile can be embedded inside a C application to help power the network protocol. However, we should first state some influencing design criteria for our example application other than the fact it must post to Twitter. Firstly, we want the application to be light-weight in terms of network usage. We also want to be able to easily build a packaged version which could get distributed and installed on well known Linux distributions such as Ubuntu. The first design condition is not really relevant to the client but rather to the server. We would like our low-cost server to be-able to handle multiple client requests without issues of bandwidth or load. Therefore, we shall use UDP to transport the data and Packedobjects to tightly pack the application messages (PDUs). In terms of the second design decision we would like users to be able to easily install the application without compiling from source. Scheme implementations such as Guile provide excellent support for using open source tools such as autoconf. This in turn allows us to easily apply automated routines to transfer the builds into Debian packages. Having provided some background to the design we can now discuss implementation specifics.

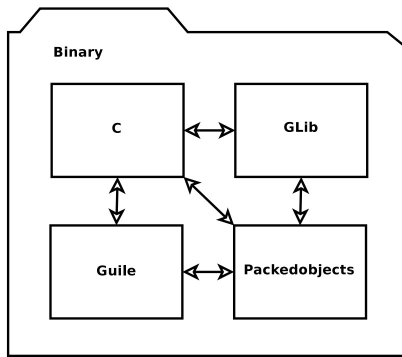

Figure 9. Client technology

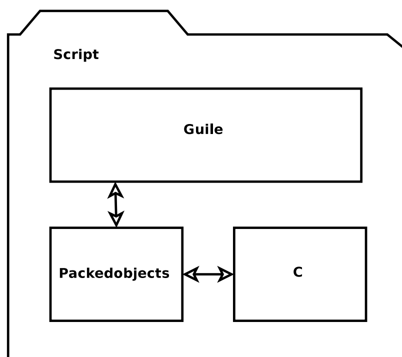

Figure 10. Server technology

6.2 Implementation choices

The client consists of a C application which uses features of GLib [1] to simplify tasks such as calling the Twitter web API. The other main feature of the client is it embeds Guile (including Packedobjects) to facilitate the encoding and decoding of network packets. Figure 9 summarises the technologies built into the client. The end product is an application binary compiled from C source code. Figure 9 shows a relationship between Packedobjects and C. Although Packedobjects is a Guile module it also heavily relies on calls to C for its low level functionality. This highlights the true flexibility with working with an embeddable language where callbacks to the host language may also occur. The server, however, takes a different approach and is completely written in Guile. In this case the end product is a script. Figure 10 summarises the technologies used. The server is simply a Guile script which uses the Packedobjects module. This illustrates one design choice available to the developer when using embeddable Scheme implementations. Do you write the application in Scheme and perhaps interface to C or do you write the application in C and embed Scheme? If the developer decides to embed Scheme into their C application, another choice exists. How much should be done in C and how much should be done in Scheme? In some cases there may be an obvious technical divide. However, often less technical factors influence the decision, such as the ability to re-use code. For example, client software that talks to well known Web 2.0 services is not difficult to find amongst various open source C based projects. Therefore, although it would not be difficult to write this functionality completely in Scheme it was more straightforward to simply use some existing C code. The end product is a binary that is not only easily distributable but also dynamically configurable.

[1] GLib is a utility library developed as part of the GNOME project.

7. Future work

Challenges exist from taking such a dynamic approach to network protocol design. Improvements to the Packedobjects tool can be made in areas such as performance and safety.

In section 4 we saw how expressive a protocol could be but how does this compare to tools like Protocol Buffers? Although subjective it provides a useful additional metric of comparison.

8. Conclusion

The designer of a network protocol must make a number of choices. The choices taken will have an impact on the size and structure of the data communicated. In some cases it is necessary to try and encode the data as efficiently as possible, in which case a binary format may be used. Similar to the way we might migrate from a low-level language and think about a problem in a high-level language, the protocol designer should not think in terms of a low-level binary format. Instead the designer should use a more expressive alternative, one that will still produce equivalent concise binary output. In this paper we presented Packedobjects, a tool which provides such an alternative. By utilising s-expressions from the Scheme programming language, Packedobjects is able to describe network protocols using an abstract syntax. This abstract syntax is dynamically transformed into a tightly packed bit stream for communication across a network. The Scheme programming language provides a number of advantages for the design of such a tool. Firstly the concept of "data is code" eliminates the need for using a compiler to transfer the abstract syntax into a concrete syntax which is usable in the native programming language. Instead we gain the benefits of using a Scheme interpreter to design and test our protocols. In addition, we obtain expressive features such as quasi-quote to help create concise and re-usable protocol definitions. The other main benefit of using Scheme for a tool like Packedobjects is that it provides some implementation specific choices. We have the choice of building solutions completely in Scheme itself but also have the ability to embed the language into a host language such as C. In this paper we have illustrated the benefits of this approach such as code reuse and the ability to easily package and distribute the application. Even though we use C as the host language we are still able to dynamically control the network protocol using the embedded Scheme.

References

[1] DUBUISSON, O. *ASN. 1 Communication between Heterogeneous Systems*. Morgan Kaufmann, 2001.

[2] GOOGLE. Protocol Buffers. http://code.google.com/p/protobuf/, July 2007.

[3] INTERNATIONAL TELECOMMUNICATION UNION. Information Technology — ASN.1 Encoding Rules — Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, July 2002.

[4] MOORE, J. *On the Performance of Unaligned Packed Encoding Rules when Applied to a Non-optimised Protocol Specification.* PhD thesis, University of Sheffield, 2001.

[5] MOORE, J. Geotwitta. http://zedstar.org/blog/2009/05/02/geotwitta/, May 2009.

[6] MOORE, J. Packedobjects. http://packedobjects.sourceforge.net/, 2009.

[7] TANENBAUM, A. *Computer Networks*. Prentice hall PTR, 2002.

[8] WARREN, H. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.