Type Encoding/Decoding rules: from C to Protocol Buffer

Shen Liu

(Version 1.5, last modified 07/05/2016)

Abstract

Type conversion between C and protocol buffer is an important issue in our project. When the separation is done and the RPC tool begins working, we must automatically restore all the arguments for each RPC function in the receiver process, which means the function parameter types need to be transmitted between two processes fully and exactly. Unfortunately, the type system that protocol buffer supports is quite weak. To make our project automatically run in the end, we have to design a type conversion protocol to let protocol buffer automatically convert some advanced C types(e.g. pointer) into protocol buffer types.

1 Background

We use gRPC, which is fully based on google protocol buffer, to deal with RPC issues in our project. In gRPC, a C type must be packed to protocol buffer "`message`" type in a .proto file(IDL file) for further transmission. For example, if you have a C function `int foo(int x)` which needs to be called remotely, then in your .proto file, the argument type `int` can be packed in protocol buffer as follows:

```
message M{                          message M{
  int64 x=1;      or                  int32 x=1
}                                   }
```

Next, protocol buffer will automatically generate a group of read/write APIs for each message. Here is an API for x's value assignment:

```
void set_x(::google::protobuf::int32 value);
```

and an API for getting the value of x:

```
inline ::google::protobuf::int32 M::x() const {
  return x_;
}
```

Here is a more complex C-protobuf type conversion sample:

```
typedef struct{                   message Circle{
  int x;                            message Point{
  int y;                              int64 x=1;
}Point;                               int64 y=2;
                      --->          }
typedef struct{                     double radius=1;
  Point center;                   }
  double radius;
}Circle;
```

Our project can automatically finish this conversion for all scalar types and simple composite types as "Circle". However, when parameter types become more and more complex, especially for those structures with multi-level pointers, generating a correct .proto file automatically as before will be a real challenge. To achieve this goal, a possible way is designing a type-conversion protocol to make our project work more intelligently. Simply speaking, for any C type input, first we use such a protocol to convert it into an integer array(encoding), and then construct the "message" type in .proto. On the receiver side, we do array parsing to restore the original C types(decoding) instead of parsing the complex .proto file.

2 Type system and encoding/decoding rules

In this draft we only use a small subset of C type system to show how the encoding/decoding idea works. Here is how our toy type system looks like:

```
Type t := int | t* | struct {t1; t2; ... ;tn}
          | tname S
```

Any pair of form (`type,value`) based on this type system will be encoded as an array of bytes(`bytes[] lst`), and the first byte(see table 1) in this array denotes what type this array corresponds to.

lst[0]	type
0	int
1	pointer
2	struct $\{t_1; t_2; ...; t_n\}$
3	tname S

Table 1: type mapping rules

As we can see from Table 1, any encoding/decoding operation related to type `tname S` requires knowing the associative type `struct{...}` of `tname S`. In our framework, we use a name-type mapping table(e.g. table 2) to map each name string which represents a struct to its corresponding `struct` type.

name	struct
S1	struct {int;int;}
S2	struct {int;int*;}
S3	struct {int;int*; struct{int; int}*;}
...	...

Table 2: A name–type mapping table example

Besides, in each round for encoding, we also use an auxiliary table called pointer table to record each pointer value that ever appeared. By doing this we can identify some complex function arguments(e.g. circular linked list).

Once we have such auxiliary tables, we can easily construct the encoding/decoding rules for our type system as follows:

```
Basic value conversion functions:

intToBytes(int): convert an integer to a byte string.

symbolToBytes(S): convert a symbol S to a byte string.

ptrToBytes(int): convert a pointer address(int) to a byte string.

bytesToInt(bytes[]): convert a byte string to an integer.

bytesToSymbol(bytes[]): convert a byte string to a symbol.

bytesToPtr(bytes[]) convert a byte string to a hexadecimal integer.

dereference(int): return the value that a pointer points to.

getTypeFromTable(S): look up S in the mapping table and return its
                     associative type.


Definition encode: (type,value) (t,v) -> bytes[]
  match t with
  | int    => 0::intToBytes(v)
  | t*     => 1::ptrToBytes(v)::encode(t,dereference(v))
  | struct {(t1,v1);...;(tn,vn)}
           => 2::intToBytes(n)::encode(t1,v1)::...::encode(tn,vn)
  | tname S
           => 3::symbolToBytes(S)::encode(getTypeFromTable(S),v)
  end.


Definition decode bytes[] lst =
  match lst[0] with
    | 0 => ((int, bytesToInt(lst[1...4])), lst+5)

    | 1 => let ((t1,dereference(v)), l1) = decode (lst+5) in ((t1*, v), l1)
           //v = bytesToPtr(lst[1...4])

    | 2 => let n = bytesToInt(lst[1]) in
           let ((t1,v1),l1) = decode (lst+5) in
               let ((t2,v2),l2) = decode l1 in
           ...
           let ((tn,vn),ln) = decode l_{n-1} in
           (struct {(t1,v1);(t2,v2);...;(tn,vn)}, ln)

    | 3 => let S = bytesToSymbol(lst[5...5+length(S)-1]) in
                               /*length(S) = bytesToInt(lst[1...4])*/
           decode(lst+offset)  /* offset = 1+4+length(S)*/
  end
```

Now consider a circular linked list example:

```
    typedef struct Node{
     int val;
     Node* next;
    }Node_t;

    Node_t *head = (Node_t*) malloc(sizeof(Node_t)); //head: 0x0004
    Node_t *tail = (Node_t*) malloc(sizeof(Node_t)); //tail: 0x0008

    head->val = 10;
    head->next = tail;

    tail->val = 20;
    tail->next = head; // circular linked list
```

Assume that we want to send this circular linked list from sender to receiver, then the encode/decode process is as follow:

```
encode(Node_t *head, 0x0004)

= 1::ptrToBytes(0x0004)
   ::encode(Node_t, dereference(0x0004))
   /* dereference(0x0004) = {10,0x0008} */

   (pointer table: {0x0004})
```

```
= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::encode(getTypeFromTable(Node_t),{10,0x0008})

   (pointer table: {0x0004})

= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
          ::encode(struct Node{int, struct Node*},{10,0x0008})

   (pointer table: {0x0004})

= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::2::intToBytes(2) /*two fields*/
         ::encode(int,10)
         ::encode(Node_t*,0x0008)

   (pointer table: {0x0004})


= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::2::intToBytes(2) /*two fields*/
         ::0::intToBytes(10)
         ::1::ptrToBytes(0x0008)
            ::3::symbolToBytes(Node_t)
               ::encode(getTypeFromTable(Node_t),dereference(0x0008))

   (pointer table: {0x0004, 0x0008})


= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::2::intToBytes(2) /*two fields*/
         ::0::intToBytes(10)
         ::1::ptrToBytes(0x0008)
            ::3::symbolToBytes(Node_t)
               ::encode(struct Node{int, struct Node*}, {20,0x0004})

   (pointer table: {0x0004, 0x0008})
```

```
= 1::ptrToBytes(0x0004)
    ::3::symbolToBytes(Node_t)
        ::2::intToBytes(2) /*two fields*/
            ::0::intToBytes(10)
            ::1::ptrToBytes(0x0008)
                ::3::symbolToBytes(Node_t)
                    ::2::intToBytes(2) /*two fields*/
                        ::0::intToBytes(20)
                        ::1::ptrToBytes(0x0004)
                        ...
                        (0x0004 is in pointer table already, stop here)

     (pointer table: {0x0004, 0x0008})
```

0x0004 appears again, which means there must be a circle, to remember all pointer values we need an extra data structure for pointer storage and comparison.

The decoding process for the generated `bytes[] lst` can be illustrated as follows:

```
decode(bytes[] lst)
= decode(1::ptrToBytes(0x0004)
             ::3::symbolToBytes(Node_t)
                 ::2::intToBytes(2)
                     ::0::intToBytes(10)
                     ::1::ptrToBytes(0x0008)
                         ::3::symbolToBytes(Node_t)
                             ::2::intToBytes(2)
                                 ::0::intToBytes(20)
                                 ::1::ptrToBytes(0x0004))

= decode(3::symbolToBytes(Node_t)
             ::2::intToBytes(2)
                 ::0::intToBytes(10)
                 ::1::ptrToBytes(0x0008)
                     ::3::symbolToBytes(Node_t)
                         ::2::intToBytes(2)
                             ::0::intToBytes(20)
                             ::1::ptrToBytes(0x0004)) in ((t1*, 0x0004), null)

= decode(2::intToBytes(2)
             ::0::intToBytes(10)
             ::1::ptrToBytes(0x0008)
                 ::3::symbolToBytes(Node_t)
                     ::2::intToBytes(2) /*two fields*/
                         ::0::intToBytes(20)
                         ::1::ptrToBytes(0x0004)) in ((Node_t*, 0x0004), null)
```

Once we have the name Node_t, we can directly look up and retrieve its associative struct type in the mapping table, and then restore a new list on the receiver side like:

```
Node_t* head = (Node_t*) malloc(sizeof(Node_t));
```

During the left decoding process, we set up a table(see table 3) which records the values on both sender and receiver sides for each pointer, to help us conveniently restore the sender side point-to relationships in the receiver side.

pointer	value in sender	value in receiver
head	0x0004	0x0012
tail	0x0008	0x0016
head→next	0x0008	0x0016
tail→next	0x0004	0x0012
...	...	...

Table 3: Pointer values in both sender and receiver

For example, assume in the receiver side pointer head equals to 0x0012, and its sender counterpart equals to 0x0004, then we have an entry like head–0x0004–0x0012 in our table. After we finished the restoration, we can also use this table to check whether the old pointer-to relationships are maintained correctly.

Let's continue decoding the bytes above, we have:

```
= decode(2::intToBytes(2)
          ::0::intToBytes(10)
          ::1::ptrToBytes(0x0008)
              ::3::symbolToBytes(Node_t)
                  ::2::intToBytes(2)
                      ::0::intToBytes(20)
                      ::1::ptrToBytes(0x0004))

= decode(3::symbolToBytes(Node_t)
          ::2::intToBytes(2)
              ::0::intToBytes(20)
              ::1::ptrToBytes(0x0004))
  head->val = 10;
  head->next = (Node_t*)malloc(sizeof(Node_t)); //assume head->next = 0x0016
  ("Node_t" can be directly retrieved from the name-type mapping table)

  Pointer table:
  Pointer       sender    receiver
  head          0x0004     0x0012
  head->next    0x0008     0x0016
```

```
= decode(empty)

  head->next->val = 20;
  head->next->next = (Node_t*)malloc(sizeof(Node_t));//assume 0x0020

  Pointer table:
  Pointer           sender   receiver
  head              0x0004    0x0012
  head->next        0x0008    0x0016
  head->next->next  0x0004    0x0020(wrong value!)
```

By checking the pointer table we know pointer `"head→next→next"` should be an alias of pointer `"head"`. So the new allocated value for `"head→next→next"` should be updated immediately from 0x0020 to 0x0012 as follows:

```
= decode(empty)

  head->next->val = 20;
  head->next->next = 0x0012;
  or
  head->next->next = head;

  Pointer table:
  Pointer           sender   receiver
  head              0x0004    0x0012
  head->next        0x0008    0x0016
  head->next->next  0x0004    0x0012
```

3 Program dependence graph

Building the program dependence graph(PDG) is a key step in our paper.

Our separation is fully based on the partitioning of the program dependence graph(PDG) generated for each input program.