ETN

Enrico Deiana, Emanuele Del Sozzo

1 Introduction

Ethos type notation (etn) is a programming-language-independent type system. It is used to type ipc and files. It serves as a bridge between the programming language and Ethos. Ethos types represent data only, they are not associated with the code for manipulating that data. Each type in etn maps to a cryptographic hash which is one-for-one with types. Thus types can be defined independently, and two types have the same hash if, and only if, they are the same type.

2 Supported Types

Here follows a list of ETN currently supported types and how they are encoded.

- Bool: represents the truth values true and false. Encoded within a byte: 1 for true, 0 for false. Represented in Go by type "bool".
- Integer: represents the set of n-bit signed/unsigned integers. Encoded in n-bit little-endian. An n-bit type is always encoded using n bits. Represented in Go by types "uint8", "uint16", "uint32", "uint64", "int8", "int16", "int32", "int64".
- Floating Point: represents the set of n-bit floating point numbers. Encoded in little-endian IEE-754. Represented in Go by types "float32", "float64".
- String: represents the set of Unicode strings. Encoded using UTF-8. The length in bytes "l" of the UTF-8 encoded string is sent first as a uint32, followed by the l bytes themselves. Represented in Go by type "string".
- Tuple: an ordered list of elements of a given type. The length in elements l of a tuple value is sent as a uint32, followed by l encoded values of the tuple's element type. Represented in Go by "array" and "slice" types.
- Dictionary: a map of keys to values. A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The number of map elements is its length l, which is encoded first as a uint32 followed by l pairs (key, value) encoded data.

- Pointer: a reference to a value of a given type. The way a pointer value is classified and encoded depends on whether or not the pointer is nil and whether it has already been encoded as part of a larger value (e.g. as part of a circular value). If a pointer value is nil, PNIL is encoded within a byte and encoding is complete. If a pointer value has been encoded before as part of the same, larger value, PIDX is encoded within a byte, followed by the index of the previously encoded value. Otherwise, PVAL is encoded within a byte, followed by the encoded value that the pointer references. Represented in Go by "pointer" types.

- Struct: a composite type containing named fields with specified types. Names are ignored when sending or receiving structs. A struct is sent by encoding each field in the order specified by the type of the structure. Represented in Go by struct types. In order to encode or decode a struct, all fields must be exported.

- Any: a reference to a value of any type. Sent by encoding the hash for the type of the concrete value references as a Tuple, followed by the encoded value that the any references. Represented in Go by "empty interface" types. In order to encode or decode a value contained in an interface, the hash for the value's type must have been registered in the typetable (se tt.go for more details).

3 Not Supported Types

List of not supported types (by ETN). Smart Pointers, Complex 64 bits and Complex 128 bits are currently not implemented. Unsigned Integers, Signed Integers and Unsigned Integer Pointers are architecture dependent, then are not supported.

- Smart Pointer: a reference to a value which may or may not be in memory.

- Complex 64 bits: represents complex values using "float32" Go type for Re and Im parts. Represented in Go by complex64 type.

- Complex 128 bits: represents complex values using "float64" Go type for Re and Im parts. Represented in Go by complex128 type.

- Unsigned Integer: positive integer numbers or zero. Represented in Go by uint. Architecture dependent.

- Signed Integer: positive or negative integer numbers, zero included. Represented in Go by int. Architecture dependent.

- Unsigned Integer Pointer: integer type that is large enough to hold the bit pattern of any pointer. Represented in Go by uintptr. Architecture dependent.

4 ETN Package

Here follows a description of the main exported functionalities of the "etn" package.

4.1 etn.go

The main structures in "ent.go" are Encoder and Decoder. The first one writes encoded values inside its []byte buffer buf and keeps track of the encoded elements by an index (integer that stores the number of encoded elements) and an addrToIndex map (which takes as key the address of the element that has to be encoded, and as value the current number of encoded elements and the type of the current element). The second one reads from its []byte buffer buf the encoded values and decodes them into ETN valid types (described above) keeping track of the decoded elements types into an indexToValue array.

Constructors:

- *func NewEncoder(io.Writer) *Encoder*
  Given a io.Writer returns the address of a new Encoder structure.
- *func NewDecoder(io.Reader) *Decoder*
  Given a io.Reader returns the address of a new Decoder structure.

Methods of Encoder (the exported ones start with capital letter):

- *func (e *Encoder) Encode(interface) os.Error*
  Wrapper for EncodeValue(), given an element of generic type (an empty interface in Go can be of any type and can implement any method) encodes that element and returns an os.Error if the element type is not supported.
- *func (*Encoder) EncodeValue(reflect.Value) os.Error*
  Given a generic element (reflect.Value is the reflection interface to a Go value) encodes it using encode() and returns an os.Error if the element type is not supported.
- *func (*Encoder) encode(reflect.Value)*
  Given a generic element encodes it following the rules described in Section 1.
- *func (*Encoder) u?int[8—16—32—64](u?int[8—16—32—64])*
  Given a u?int[8—16—32—64] encodes it writing it into the Encoder []byte buffer using little endian notation (least significant byte is stored in the smallest address).
- *func (*Encoder) string(string)*
  Given a string, writes first its length (encoded as uint32) and then the string into the Encoder []byte buffer. First encodes the length (using *func (*Encoder) length(uint32)*), then encodes the string.

- *func (*Encoder) write([]byte)*
  Given an array of bytes, writes that array using its writer io.Writer.

- *func (*Encoder) length(uint32)*
  Given a length in uint32 the method calls *(*Encoder) uint32(uint32)* to encode the length of an array, slice, map, string.

Methods of Decoder (the exported ones start with capital letter):

- *func (*Decoder) Decode(interface) os.Error*
  Wrapper for DecodeValue(), given an element of generic type (an empty interface in Go can be of any type and can implement any method) decodes that element and returns an os.Error if the element type is not supported.

- *func (*Decoder) DecodeValue(reflect.Value) os.Error*
  Given a generic element (reflect.Value is the reflection interface to a Go value) decodes it using decode() and returns an os.Error if the element type is not supported.

- *func (*Decoder) decode(reflect.Value)*
  Given a generic element decodes it following the rules described in Section 1.

- *func (*Decoder) u?int[8—16—32—64]() u?int[8—16—32—64]*
  Decodes a u?int[8—16—32—64] reading it from the Decoder []byte buffer (using little endian notation) and returns the decoded u?int[8—16—32—64] value.

- *func (*Decoder) string() string*
  Decodes a string reading it from the Decoder []byte buffer. First decodes the length (using *func (*Decoder) length() uint32*), then decodes the string.

- *func (*Encoder) read([]byte)*
  Given an array of bytes, reads that array using its reader io.Reader.

- *func (*Decoder) length() uint32*
  Calls *(*Decoder) uint32()uint32* to decode the length of an array, slice, map, string.

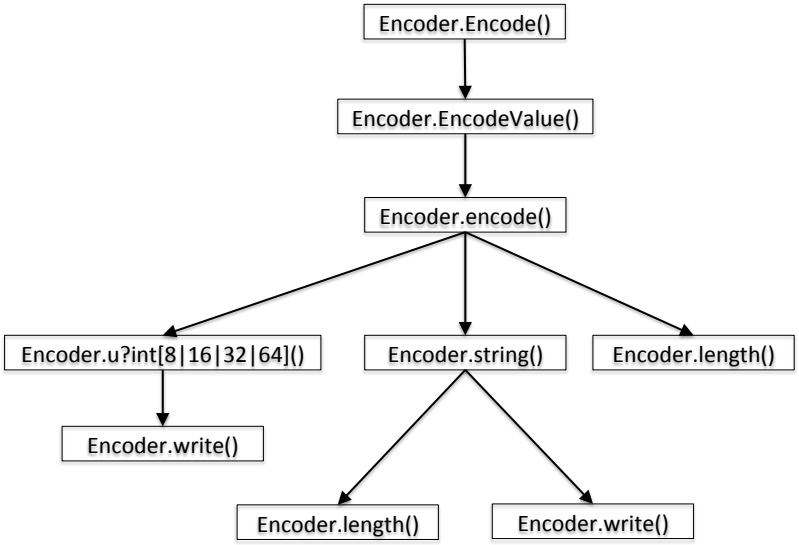

Figure 1: etn.go package: Encoder

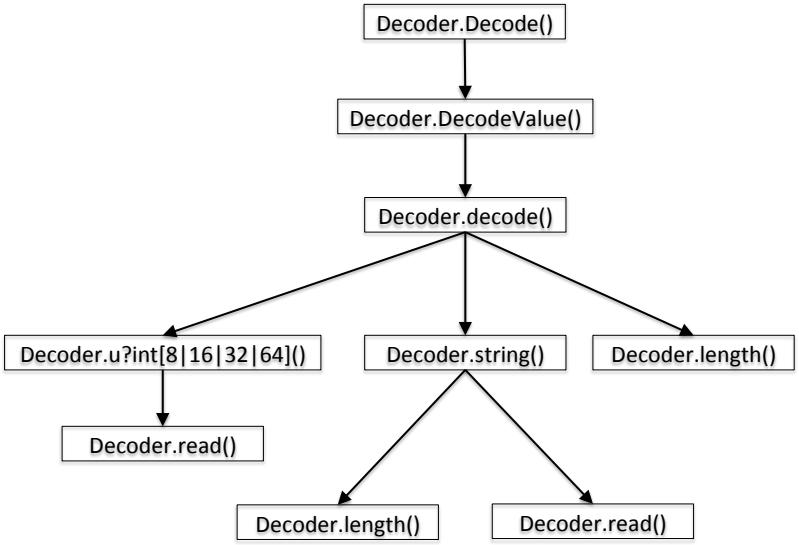

Figure 2: etn.go package: Decoder

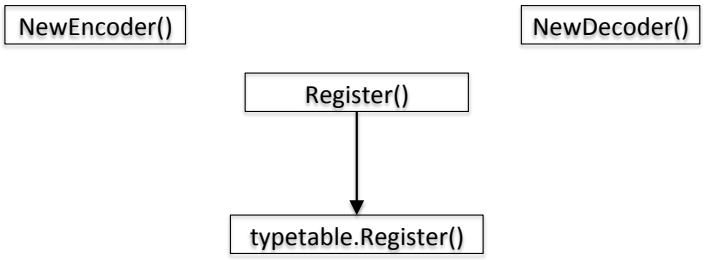

Figure 3: etn.go package: Constructors

4.2 rpc.go

The data structures defined here are: *id*, *header*, *Call*, *callId*, *Server* and *Client*. The most important are *Server* and *Client*.
Type Server contains a *reflect.Value* used during the creation of the Server, an array of *int* which contains the indexes of the exported methods the Server is supposed to respond to, an *Encoder* and a *Decoder*, used to transmit and receive values.
Type Client contains a map where the element type is **Call* and the key type is *int* to keep track of the calls, an *Encoder* and a *Decoder*. Type Call contains two interfaces (one for the arguments and the other for the reply), an *os.Error* and a boolean *Channel*.
Here the constructors of *Server* and *Client*:

- *func NewServer(v interface, e *Encoder, d *Decoder) (s *Server)*
  It creates a server using the value stored in the empty interface that transmits and receives values using the supplied Encoder and Decoder. The newly-created server will respond to the exported methods of this value.
- *func NewClient(e *Encoder, d *Decoder) *Client*
  It creates a client that transmits and receives values using the supplied Encoder and Decoder.

Here the function of *callId* type:

- *func (c *callId) Increment() (i id)*
  It locks variable c, increments its id, unlocks c and returns the *c* previous id.

Here the functions of *Client* type:

- *func (c *Client) Dispatch() bool*
  It collects, decodes and dispatches replies.
- *func (c *Client) Go(id uint64, args, reply interface) (call *Call)*
  It creates a new call using the parameters. If either *call.reply* or *c.d* (Client's decoder) are]nil, the call is done, otherwise, the call is added to the Client's calls map.
- *func (c *Client) Call(id uint64, args, reply interface) (err os.Error)*
  Synchronous function. It uses method *Go* to create a new call and does not return until call is finished.

Here the function of *Server* type:

- *func (s *Server) Handle() os.Error*
  It handles *Server*'s requests.

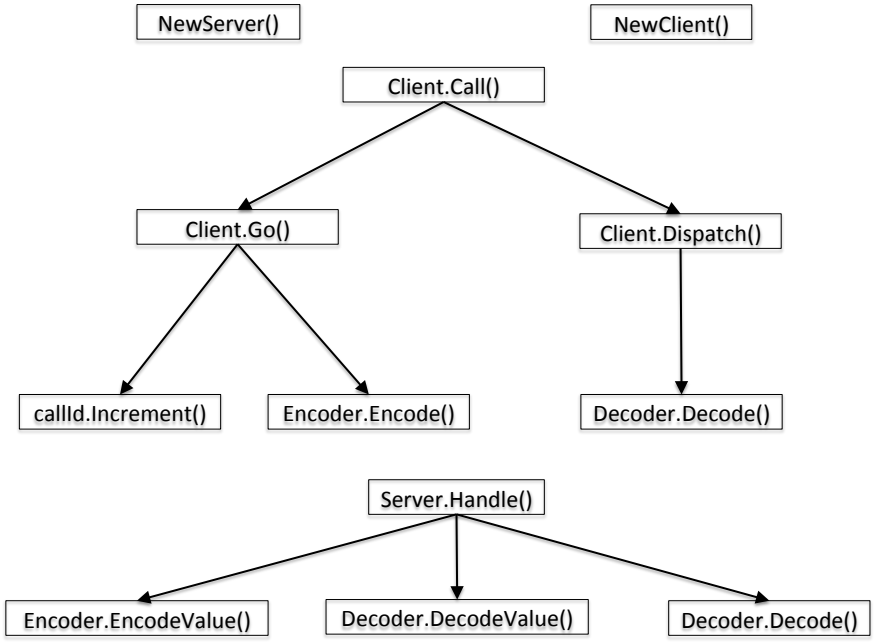

Figure 4: rpc.go package

4.3 trie.go

The data structure defined here is *trie*, that is used to build a tree. It contains a byte (used as index), interface and two trie type pointers (one to the child and the other to the sibling).
Here the functions of *trie* type:

- *func (t *trie) prefix(b []byte) (s []byte, r *trie)*
  It finds the longest present prefix of b. It returns the prefix and pointer to a trie variable that is the last node whose index matches the prefix or *t* if there is no matching index in *t's* children.
- *func (t *trie) Lookup(key []byte) (value interface, ok bool)*
  It looks up for a key in *t* variable. If the key doesn't exist or the interface returned by prefix function is nil, it returns *nil* and *false*, otherwise, it returns the interface and *true*.
- *func (t *trie) Insert(key []byte, value interface)*
  It inserts a new interface using the key received as parameter. If the key already exists, it updates the interface value.
- *func (t *trie) Delete(key []byte)*
  It deletes an interface by setting it to nil according to the key received as parameter. It searches and records the deepest node with multiple children and siblings.

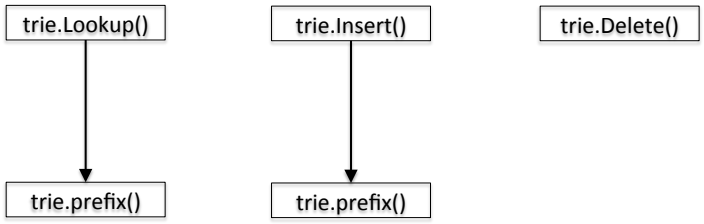

Figure 5: trie.go package

4.4 tt.go

The main structure in "tt.go" is typetable. It contains information for mapping a concrete type to its hash using a "*trie" data structure for hash to type and a "map[reflect.Type]types.Hash" for type to hash information. Detailed information of "*trie" and "types.Hash" are in "trie.go" and "types.go".

The constructor for typetable is:

- *func newTypeTable() *typetable*
  Returns a new typetable structure with an empty *trie and map[reflect.Type]types.Hash.

Exported methods of typetable:

- *func (*typetable) Register(reflect.Type, types.Hash) bool*
  Maps the concrete type of the value contained in the empty interface to its hash and stores it into the *trie and map[reflect.Type]types.Hash data structures.
- *func (*typetable) TypeForHash(types.Hash) reflect.Type*
  Given an hash returns the corresponding type.
- *(*typetable) HashForType(reflect.Type) types.Hash*
  Given a type returns the corresponding hash value.

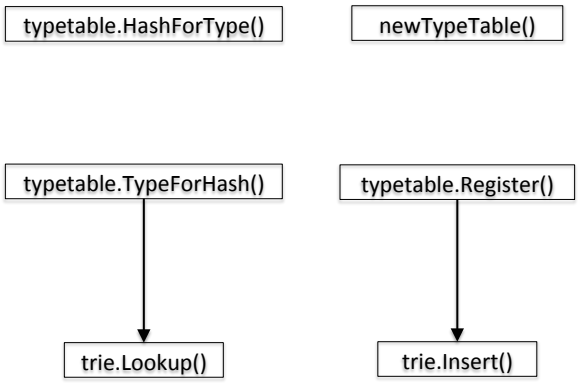

Figure 6: tt.go package