Protocol Parsing in Scala, Part I

BY CHARLES CALKINS, PRINCIPAL SOFTWARE ENGINEER

March 2016

Introduction

With the emergence of the Internet of Things, data protocols are more ubiquitous than ever. Functional programming is also becoming mainstream, with Scala a leading language in that arena. This article discusses the scodec (http://scodec.org/) parser library for Scala to parse binary data, and part II of this article will demonstrate the Scala Standard Parser Combinator Library (https://github.com/scala/scala-parser-combinators) which is useful for parsing text. The use of libraries such as these allow the protocol definition to be translated almost exactly into executable code, making the implementation easier to understand and can be seen to be correct "by inspection," reducing the chance of programmer errors.

Source code for this article is available in the associated archive (/index.php/download_file/view/956/637/).

A Binary Protocol

There are many binary protocols in common use. In particular, Internet-related ones such as IP, TCP, and UDP are used by billions of devices. In the IoT space, MQTT (http://mqtt.org/) and CoAP (http://coap.technology/) are among the most popular. For the purposes of this article, we shall consider a fictional protocol that is reminiscent of one used in proprietary hardware for industrial control that the author has used for many years.

Consider a packet-based protocol between a host and network of devices with the following features:

1. Messages can be sent with a low or high priority.
2. Messages can be addressed to a specific device by hardware address or by network address, where the network address format also allows for broadcast messages to be expressed.
3. Multiple requests can be active at the same time — the host does not need to wait for the reply to a message to be received before another is sent.
4. Messages can be request/response, in that a message can be sent from the host and a reply is expected from a device, or a message can be unsolicited, where a device can send a message without first receiving a request from the host.

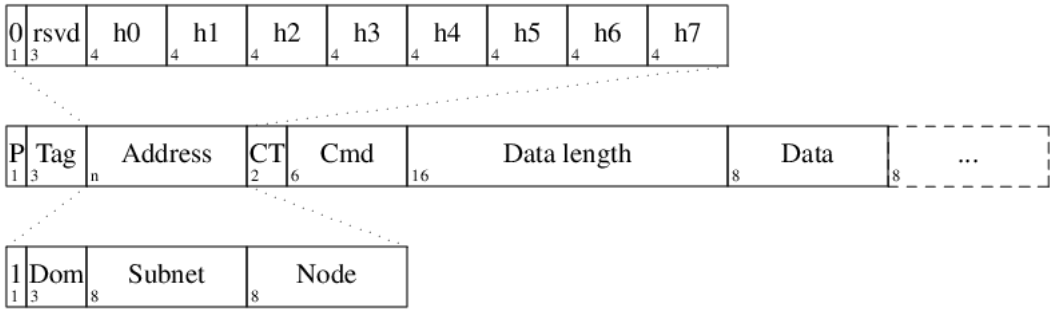
These features can be implemented as follows:

1. Message priority can be expressed by a single bit flag, interpreted as a boolean where, if true (set to 1), the message is a high priority one.
2. Two address forms can be used, also selected by a single bit flag. The hardware address form identifies a device by a hexidecimal sequence of 8 digits representing its hardware address, and the software address form is of the triple (domain, subnet, node) where (domain, 0, 0) or (domain, subnet, 0) represent broadcast messages to all devices in a domain, or all devices within a given subnet of a domain, respectively.

3. Multiple active requests can be managed by the use of a tag field, where a tag is set in a request message, and the same tag is set in the corresponding response. The host can keep track of outstanding requests by their tag, and match up responses as they arrive.

4. Message types can be defined by a command type, where commands are in request, response or unsolicited message categories.

Visually, this can be shown as:



The protocol can be expressed more formally in EBNF-style notation, where \b indicates bit values (0, 1, or x representing an unspecified digit), and {n} indicates a repetition of n times. Bit sequences are expressed in big-endian format.

```
          bool  :=  \bx
         uint8  :=  \bx{8}
        uint16  :=  \bx{16}
     hex_digit  :=  \bx{4}
           tag  :=  \bx{3}
        domain  :=  \bx{3}
        subnet  :=  uint8
          node  :=  uint8
       command  :=  uint8
   data_length  :=  uint16
    hw_address  :=  \b0xxx hex_digit{8}
   net_address  :=  \b1 domain subnet node
       address  :=  hw_address | net_address
 high_priority  :=  bool
        packet  :=  high_priority tag address command data_length uint8{data_length}
```

Defining the protocol in this manner raises a number of issues that must be taken into consideration when parsing it, such as fields with lengths of less than a byte, and fields that have their existence or length determined based on earlier values that have already been parsed.

While bit manipulation to parse this protocol can be done "by hand" by many bit shift and logical operations, the scodec library allows the grammar that defines the protocol to be transcribed almost directly into executable code. As described on scodec's Github (https://github.com/scodec/scodec) project page, scodec is a "Scala combinator library for working with binary data." Several design constraints include that the "binary structure should mirror protocol definitions and be self-evident under casual reading" as referred to previously, "mapping binary structures to types should be statically verified" in that the protocol is implemented as compiled code, "encoding and decoding should be purely functional" so protocol parsing fits within the functional programming paradigm, and "failures in encoding and decoding should provide descriptive errors" so debugging is aided during runtime operation.

Since case classes hold immutable data and automatically provide support for comparison and pattern matching, the software pattern used in this article is to define a high-level Scala case class that is used to store an element of the protocol, plus a corresponding codec object that matches the grammar which converts a bit sequence to and from the case class.

The scodec library is divided into several components. The two that are used in this article are `scodec-core` and `scodec-bits`. When building with `sbt`, the `build.sbt` build description file must have these dependencies added to the `libraryDependencies` list:

```
libraryDependencies ++= Seq(
    "org.scodec" %% "scodec-core" % "1.8.3",
    "org.scodec" %% "scodec-bits" % "1.0.12"
)
```

To use these libraries, several imports are recommended instead of fully-qualifying type names:

```
import scodec._
import scodec.bits._
import scodec.codecs._
import scodec.codecs.implicits._
```

Importing `scala.language.implicitConversions` also eliminates additional warnings when using Scala 2.11.

THE ADDRESS

We shall start with the address element. The address is in two forms: a hardware address and a network address. The case class for the hardware address can be expressed as:

```
// src\main\scala\Parse.scala
case class HardwareAddress(address: Vector[Int]) {
  require (address.length == 8)
  address.foreach(digit => require ( (0x00 <= digit) && (digit <= 0x0F) ))
}
```

The class constructor accepts an array of integers representing the hexidecimal digits, and two `require` statements ensure that the address provided is in the correct form — that it consists of 8 digits, each between 0x0 and 0xF. An scodec `Codec` object is defined in a companion object to the case class:

```
object HardwareAddress {
  implicit val codec: Codec[HardwareAddress] = {
    (constant(bin"0")) ::
    (ignore(3)) ::
    ("address" | vectorOfN(provide(8), uint(4)))
  }.as[HardwareAddress]
}
```


A `Codec` is implemented as a Shapeless (https://github.com/milessabin/shapeless) `HList`, allowing heterogeneous types to be elements of a single list. Each element of the list is an scodec `Codec` itself, allowing codecs to be nested. A number of codecs are provided by scodec, four of which are used here.

The first codec is the `constant` codec. This codec accepts an scodec `BitVector` object as an argument, and will absorb or emit this bit vector during the decode or encode process, respectively. The `bin` helper function converts from a string representation of a bit pattern to a `BitVector` for convenience of expression. So, in this instance, a bit with the value 0 is expected as the first bit in the bit sequence. During the decoding process, this bit will be removed from the bit sequence, and the remaining bits passed on to the next codec that is to process the bit sequence. During the encoding process, a 0 bit will be emitted.

The second codec is the `ignore` codec. This codec, when decoding, will absorb the requested number of bits from the bit stream and discard them. When encoding, bits with the value 0 will be emitted the requested number of times.

The third codec is the `vectorOfN` codec. This codec accepts two arguments and will yield a Scala `Vector` of elements. The first argument to `vectorOfN` is the number of elements that the associated `Vector` will contain, and the second is a `Codec` object that will be used to encode or decode each element of the `Vector`. The codec used for each element is the fourth codec type used in the `HardwareAddress` codec, the `uint` codec, which encodes or decodes an unsigned integer value. As with the `ignore` codec, the value in parentheses indicates the number of bits to use — here, four bits are needed for each hexidecimal digit, and 8 digits are

present. The `"address"` label is used by scodec to display more informative error messages when they arise to show at what point a codec has failed, but isn't a necessary part of the codec description — the `constant` and `ignore` codecs also could be labeled in a similar way, but they aren't in this instance.

The `as` method transforms the codec of the HList to a codec of the type of the case class. For convenience, a variable `codec` is created of the codec type, and the variable is marked as `implicit` so any time a `HardwareAddress` codec is needed, this one will be found and used automatically.

Next, we do a similar thing for the network address format. The case class is defined in likewise manner, using `require` statements as safety checks to ensure all values are within range:

```
1. case class NetworkAddress(domain: Int, subnet: Int, node: Int) {
2.   require ((0 <= domain) && (domain <= 7))
3.   require ((0 <= subnet) && (subnet <= 255))
4.   require ((0 <= node) && (node <= 255))
5. }
```

It, too, has a codec defined in an associated object:

```
1. object NetworkAddress {
2.   implicit val codec: Codec[NetworkAddress] = {
3.     (constant(bin"1")) ::
4.     ("domain" | uint(3)) ::
5.     ("subnet" | uint8) ::
6.     ("node" | uint8)
7.   }.as[NetworkAddress]
8. }
```

Here, the `constant` codec is used again, but this time for a single 1 bit. The `uint` codec is used for the remaining fields. Certain bit sizes are predefined in scodec, such as for byte-sized unsigned integers, so codecs such as `uint8` can be used directly, instead of being expressed as `uint(8)` . A 3-bit unsigned int codec doesn't exist, so the parameterized `uint` codec form is needed.

At a high-level, a developer would not want to distinguish between the address forms by a binary digit, as it is hard to remember and prone to error. Instead, we can define an enumeration:

```
1. object AddressType extends Enumeration {
2.   type AddressType = Value
3.   val HARDWARE_ADDRESS = Value(0)
4.   val NETWORK_ADDRESS = Value(1)
5. }
6. import AddressType._
```

In addition to the codecs provided by scodec, custom codecs can be written by implementing the codec interface. In our case, we can implement a codec that, when decoding a bit stream, returns either the `HARDWARE_ADDRESS` or `NETWORK_ADDRESS` enumeration value based on the bit, but not consume the bit so it is still available for a successive codec to use.

The first method of the codec interface that must be implemented is `sizeBound` . This method returns the number of bits needed by the codec. Here, as a single bit is being tested, only one bit is required to be in the input for decoding to be allowed (an error will be generated if a sufficient number of bits is not present in the bit sequence to decode), and only one bit at most would be emitted during encoding.

```
1. class AddressTypeCodec() extends Codec[AddressType] {
2.   override def sizeBound = SizeBound.exact(1)
```

The second method of the codec interface is the `encode` method. As this codec only looks at the bit stream without consuming or emitting anything, encoding is always successful, and as no bits are emitted, an empty `BitVector` is supplied.

```
override def encode(a: AddressType) = Attempt.successful(BitVector.empty)
```

Decoding is more complex, and the implementation of the third method of the codec interface, `decode`, first copies one bit from the bit stream via a call to `acquire`. The `acquire` method returns an `Either[String, BitVector]`, which is either an instance of `Left(`*`error message`*`)` or `Right(`*`the extracted bits`*`)`. In the former case, the decoding fails by returning an `Attempt.Failure` object containing an appropriate error message. In the latter case, the decoding is successful, so an `Attempt.Successful` object containing a `DecodeResult` is returned. The first parameter of the `DecodeResult` is the result of the decoding process itself, which in this case is an instance of the `AddressType` enumeration depending on whether the extracted bit is a 0 or a 1. The second parameter of the `DecodeResult` is the bit stream that is to be passed to the codec that follows this one. Here, we pass the entire incoming buffer, as the decoding process only looks at the bit to determine which enumeration value to return without consuming it. If the bit were consumed, the returned bit sequence would be `buffer.drop(1)` to remove the bit from the sequence, instead of just `buffer`.

```
override def decode(buffer: BitVector) =
  buffer.acquire(1) match {
    case Left(e) => Attempt.failure(Err.insufficientBits(1, buffer.size))
    case Right(b) =>
      Attempt.successful(
        DecodeResult(
          if (b(0)) NETWORK_ADDRESS else HARDWARE_ADDRESS,
          buffer
        )
      )
  }
```

Lastly, it is good to override the `toString` method to provide a user-friendly name for the codec to allow error messages to be clearer.

```
  override def toString = s"AddressTypeCodec"
}
```

For convenience, we can create an implicit instance of this codec so it can be automatically selected.

```
object PacketImplicits {
  implicit val addressTypeCodec = new AddressTypeCodec()
}
import PacketImplicits._
```

Now that the codec for `AddressType` has been created, we can create a case class for an address that contains either a hardware address or a network address, distinguished by the address type, by using the codec for `AddressType`.

```
case class Address(addressType: AddressType, hwAddress: Option[HardwareAddress],
  netAddress: Option[NetworkAddress])

object Address {
  implicit val codec = {
    (("address_type" | Codec[AddressType]) >>:~ { addressType =>
      ("hardware_address" | conditional(
        addressType == HARDWARE_ADDRESS, Codec[HardwareAddress])) ::
      ("network_address" | conditional(
        addressType == NETWORK_ADDRESS, Codec[NetworkAddress]))
    }).as[Address]
  }
}
```

This codec uses the `>>:~` operator, which allows a value that has been parsed, the address type in this case, to be used in successive parts of the codec. The `conditional` codec, if a predicate is true, executes the supplied codec. Here, if the address type is a hardware address, the hardware address codec is invoked, otherwise the network address codec is used. The `conditional` codec, as it might not execute if the predicate is false, is associated with an `Option` type in the case class. For example, the `hwAddress` variable will be set to `Some(HardwareAddress(...))` if `addressType` is `HARDWARE_ADDRESS`, otherwise it will be set to `None`.

THE COMMAND

Since a requirement of the protocol is to be able to classifly messages as request, response or unsolicited, so an enumeration can be established for these categories:

```
object CommandType extends Enumeration {
  type CommandType = Value
  val REQUEST = Value(0)
  val RESPONSE = Value(1)
  val UNSOLICITED = Value(2)
}
import CommandType._
```

The scodec-supplied `mappedEnum` codec creates a codec based on an enumeration. By adding the following to `object PacketImplicits`, an object of the codec can be instantiated and ready for use. A `mappedEnum` can be constructed by explicitly listing the enumeration values, or, as done here, by setting them automatically by mapping over the enumeration itself.


```
implicit val commandTypeCodec: Codec[CommandType] =
  mappedEnum(uint(2), CommandType.values.map(v => (v, v.id)).toMap)
```

Taking advantage of the `mappedEnum` is possible in this case, but wasn't in the `AddressType` case above, because `mappedEnum` will emit bits when encoding and consume bits when decoding, but the `AddressType` codec needed to leave the bit sequence unchanged.

THE PACKET

With this, we can now complete the codec for the packet as a whole:

```
case class Packet(highPriority: Boolean, tag: Int, address: Address,
  command:Int, dataLength: Int, data: Vector[Int]) {
  require (dataLength == data.length)
}

object Packet {
  implicit val codec: Codec[Packet] = {
    ("high_priority" | bool) ::
    ("tag" | uint(3)) ::
    ("address" | Codec[Address]) ::
    ("command_type" | Codec[CommandType]) ::
    ("command" | uint(6)) ::
    (("data_length" | uint16) >>:~ { length =>
      ("data" | vectorOfN(provide(length), uint8)).hlist
    })
  }.as[Packet]
```

The `bool` codec parses a single bit as a boolean value, and the `>>:~` operator is again used to reference a parsed value in a later codec — in this case, as the number of bytes to consume for the data portion of the packet. The call to `hlist` converts the single `vectorOfN` codec into an `HList`, as the `>>:~` operator requires an `HList` to operate on. The `::` operator concatenates codecs into `HLists`, but as there is only one codec after the `=>`, it must be converted to an `HList` explicitly.

TESTING

For ease of testing, we can add a convenience method to `object Packet` which sets the length based on the length of the data vector, and allows the vector to be optionally provided.

```
  def apply(highPriority: Boolean, tag: Int, address: Address,
    commandType: CommandType, command:Int, data: Vector[Int] = Vector()) =
    new Packet(highPriority, tag, address, commandType, command, data.length, data)
}
```

Additionally, we can also add two more implicits to `object PacketImplicits` to automatically create `Address` objects when an `Address` object is required, but a `HardwareAddress` or `NetworkAddress` object is given instead.

```
// in object PacketImplicits
  implicit def hardwareAddress2Address(address: HardwareAddress) =
    new Address(HARDWARE_ADDRESS, Some(address), None)

  implicit def networkAddress2Address(address: NetworkAddress) =
    new Address(NETWORK_ADDRESS, None, Some(address))
```

The codecs are used by invoking the `encode` and `decode` methods on `Codec`, as appropriate. The `encode` method returns either `Attempt.Successful(`*`encoded bits`*`)` or `Attempt.Failure(`*`Error object containing an error message`*`)`. For example, a `Packet p` can be created in a compact way by taking advantage of the implicit conversion from `NetworkAddress` to `Address`, and invoking the constructor that does not require a packet data argument, as follows:

```
val p = Packet(false, 0, NetworkAddress(1,2,3), REQUEST, 4)
```

Executing `println(p)` yields:

```
Packet(false,0,Address(NETWORK_ADDRESS,None,Some(NetworkAddress(1,2,3))),REQUEST,4,0,Vector())
```

which uses the Scala default `toString()` methods for the elements that compose the packet. Overriding these methods would provide more friendly output, but isn't done in this case.

The result of encoding this `Packet` by executing `println(Codec.encode(p))` displays:

```
Successful(BitVector(48 bits, 0x090203040000))
```

The `require` method extracts the successfully encoded bit vector, and the `toBin` method converts the `BitVector` to a string of binary digits, so executing `println(Codec.encode(p1).require.toBin)` outputs:

```
000010010000001000000011000001000000000000000000
```

The `decode` method is the process in reverse. Given a `BitVector`, the `decode` method returns either `Attempt.Successful(DecodeResult(`*`decoded object, remaining buffer`*`))` or `Attempt.Failure(`*`Error object containing an error message`*`)` as before. For instance, both of these invocations of the method `decode`:

```
println(Codec[Packet].decode(hex"0x090203040000".bits))
println(Codec[Packet].decode(bin"000010010000001000000011000001000000000000000000"))
```

result in the following being displayed:

```
Successful(DecodeResult(Packet(false,0,
  Address(NETWORK_ADDRESS,None,Some(NetworkAddress(1,2,3))),
  REQUEST,4,0,Vector()),BitVector(empty)))
```

The first parameter of `DecodeResult` is the same as `p` shown previously, demonstrating that the decoding is successful. The remaining buffer equal to `BitVector(empty)` means that all input was consumed, which would be the normal case when the bit sequence does not contain extraneous data (or the start of the next packet, say). To double-check the successful parse, executing this:

```
println(Codec[Packet].decode(Codec.encode(p).require).require.value == p)
```

displays `true`. When encoding and decoding are implemented consistently, a complete encode/decode round-trip should always be possible.

Summary

Using the scodec library makes parsing of binary data in Scala much simpler than crafting a comparable parser by hand. Having the ability to represent the protocol in code in a way that is very close to the original grammar helps to ensure that, just by inspection, the parser is correct. The ability to easily round-trip the parse also ensures the correctness of the implementation.

References

- [1] scodec
  http://scodec.org/ (http://scodec.org/)
- [2] Scala Standard Parser Combinator Library
  https://github.com/scala/scala-parser-combinators (https://github.com/scala/scala-parser-combinators)
- [3] MQTT
  http://mqtt.org/ (http://mqtt.org/)
- [4] CoAP
  http://coap.technology/ (http://coap.technology/)

- [5] Shapeless
https://github.com/milessabin/shapeless (https://github.com/milessabin/shapeless)

The **Software Engineering Tech Trends** is a monthly newsletter featuring emerging trends in software engineering.

Subscribe (https://objectcomputing.com/resources/publications/)

Let's connect.

(https://www.twitter.com/objectcomputing)(https://www.linkedin.com/company/oci)(https://www.facebook.com/objectcomputing)(https://

JOIN OUR MAILING LIST ˅

NEWS ˅

EVENTS ˅

CONTACT ˅

Email