RPC as the Core of Networking Protocols

Christian Tschudin and Manolis Sifalakis
University of Basel

Computer Science Technical Report CS-2014-03, May 2014

Abstract

Consider the statement that packet headers of a modern protocol should *not* have a version number field. Today, it is not immediately clear whether this is a good advice or a problematic guideline. In this technical report we argue that remote procedure calls (RPC) –usually classified as a higher level service– can operate anywhere in the protocol stack and replaces protocol version numbers. We observe an (often incomplete) RPC interaction pattern entangled in packet header formats today, which extends to issues of parameter representation with RPC. Moreover, a historical reconsideration of Internet protocols reveals an evolutionary shift towards RPC, at many places and occasions.

We experiment with an RPC-style dialect for handling two ICN transports (CCNx and NDN) at the same time –as well as itself– inside the same communication channel, without resorting to lower-level protocol demultiplexing support. Thereby we identify RPC as a vantage point in the design of new communication protocols and as a reference mechanism for embracing different information centric networking protocols.

1 Introduction

Version numbers and remote procedure calls (RPC) are not two topics that most network researchers would correlate. However, we note that this report is not the first place where such a connection is made. In the context of TCP the question was raised in 1991 whether protocol evolution should be handled by option fields [8]: The TCP header has no version field, hence TCP options are (ab-)used to introduce "extensions", each combination becoming some sort of new TCP version. O'Malley and Peterson suggested in RFC1263 to use RPC to manage the various extensions (and only if some combination would emerge as being useful, one would bake it into a monolithic spec).

A quick sampling of major Internet protocols reveals that after an initial phase of pioneering protocols without version number fields [Ethernet, TCP (RFC793), ARP (RFC 826), DNS (RFC883), FTP (RFC959)], almost all modern protocols, but also IPv4 and IPv6, feature some sort of a version information in order to manage protocol evolution [SNMP (RFC1157), HTTP (RFC2616), SDP (RFC4566), TEAP (RFC7170)]. Depending on the chosen encoding, the protocol version is put in a fixed header field for every packet or is an explicit part of an (ASCII) dialogue.

Yet, the recent NDN protocol proposal [7] goes against this line and *deliberately omits a version field*, nor does it envisage a version negotiation phase. The competing CCNx protocol [6], also released in Nov 2013, still has a version number field but a closer look reveals that conceptually the difference is minor. Nevertheless, the two dialects remain incompatible at encoding as well as semantic level; both approaches delegate protocol evolution (forking, merging) as well as version discrimination to some demultiplexing service in the lower communication layers.

The ambition of this technical report is to frame this protocol version number question in the more general context of *managing protocol dialects*. With management, the interaction patterns between peer protocol entities also come into focus, more specifically remote procedure calls. The contribution of this report is to expose this pattern in established protocols and to propose an invariant core mechanism that modern protocols should implement in some way or the other.

More specifically, we adopt a functional point of view and rely on reflection capabilities. In programming languages, reflection is the property of a system to inspect and modify its own structure and behavior. Transposed it means that a computer network provides mechanisms to examine the available network functions and to manage their use at run-time. Technically it boils down to a setup where protocol dialect management is itself a dialect that lives side by side with the "protocols under management".

2 An RPC view of the world

Historically, the concept of remote procedure call [1] only emerged once computer networks became available and core protocols were in place. The popular understanding still shared today is that RPC is implemented as a software layer on top of network transport and that RPC serves inter-process communication across machines. We argue that RPC is a much more broader concept (and pattern) that sits deep (and high) in networking stacks. In this section we identify existing uses and re-interpret hidden RPC patterns in todays protocols. Later on, in Section 4, we will argue that the low-level content centric protocols CCNx and NDN would benefit from fully adopting the RPC paradigm.

2.1 Classic RPC – Distributed Operating Systems

In distributed systems research, RPC is a framework sitting between a network and one or more program execution environments. Starting from local procedure calls, tools are provided to transfer the flow of execution from an intercepting stub procedure to a remote process that carries out the requested computation using data and code available on the remote node. Parameters for the callee are "marshalled" before transfer, in the same way as return values travel back in a special encoding (serialization) to the caller node where they are turned back into their native encoding for being returned by the stub. Often, (remote) procedures can be added at run time, wherefore mechanisms are needed to name, list and demultiplex among these computation services.

2.1.1 RPC service features

The classical and early example of an RPC use is Sun Microsystems' "Network File System" [9] which makes local system calls to the file system remotely accessible through the (enterprise) Internet. NFS also highlights some subtlety: Application must choose between one of RPC's "at-most-once" or "at-least-once" semantics. This RPC property is due to potential network failures and does not exist in a non-networked context, the problem of call-by-reference (e.g. pointers) is another difference. In general, RPC is seen as a high-level abstraction hiding as much as possible the fact that a network sits between concurrent processes, and magically mapping native data structures onto the datagrams or stream protocols available in the network.

In the following, we focus on three core features of an RPC framework, namely

1. the procedural CALL/RETURN pattern,
2. the need for selecting among many available procedures
3. the external representation problem of parameter and result values.

Today, other aspects (e.g. a service directory) also belong to RPC toolkits, but are neither key elements nor were they mentioned in the first papers on RPC. In the following we focus on the three properties above when referring to an RPC solution (and we neglect the aspect whether a full stub and marshalling framework is present or not).

2.2 RPC allover the Internet protocol stack

Traits of RPC emerge at surprisingly many and deeply embedded places of the Internet architecture, which we try to present from most obvious to more obscure.

2.2.1 Explicit RPC inside the Internet stack (FTP, HTTP and REST)

Putting FTP, HTTP and REST in a sequence highlights the evolution from a very raw form (just handling the call/return pattern and offering a small set of preinstalled procedures) to today's full blown RPC service of REST where the representation problem is handled by JSON and where arbitrary actions (procedure names) can be encoded in the URLs beyond HTTP's GET, PUT, POST and DELETE methods. On top of REST, here subsumed as being part of "the" Internet stack, storage access protocols like Dropbox's API replace today the above mentioned NFS from more than two decades ago.

2.2.2 Implicit RPC in the early Internet (ICMP, DNS, ARP)

An interesting case is DNS whose RFC was published only two years before Birrell and Nelson's RPC paper. Although not being an RPC "framework" per se, DNS adheres to the request/reply pattern and has minimal feature selection properties (e.g. iterative vs recursive resolution), and then focuses more on its specific remote database access application. In the same spirit, ICMP and ARP have to be mentioned: Note that ICMP error messages are special in the sense that ordinary Internet packet not generating

any error condition are silently processed without return code. But, as is argued by the CCNx and NDN protocols further below, this (implicit) reply is in most cases re-introduced by higher level datagrams, e.g. transport level ACKs.

2.2.3 Hidden RPC in Internet Protocols (TCP, ARQ and receiver driven transport)

From a causality point of view, an acknowledgment is the reaction to a data item that was *sent* before. However, the transfer of information can also be organized such that the *receiver* is in control instead of the sender. That is, instead of a DATA packet that is followed by an ACK, we define a DATAREQUEST packet that is followed by the DATA. From an external observer's point of view, both the sender-driven and receiver-driven philosophy lead to the same message exchange pattern and only differ by the (mental) viewpoint. This is shown in Figure 1.

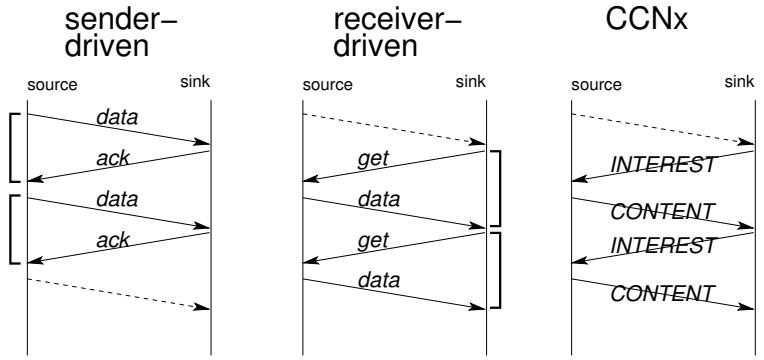

Figure 1: Sender- vs receiver-driven communication: Re-interpreting TCP's ACK messages as Interest requests.

2.3 Is "RPC in the Internet protocol stack" real?

One could argue that the comparison of RPC and protocols like ICMP, DNS and TCP is too vague and analogy-grabbing. However, we note that the original RPC paper [1] devotes several paragraphs explicitly addressing different optimization of folding the required CALL/RETURN patterns with the available networking primitives, as does the above mentioned RFC1263 [8], too. Both references considered RPC to be placed at transport layer and above, while we continue to explore the view that RPC can serve as basic interaction protocol at network and link layer. This is very obvious with the proposed new ICN protocols, as we show in the next section.

3 RPC traits in CCNx and NDN

The Content Centric Networking Protocol (CCNx) and the Named Data Networking (NDN) variant are recent proposals for an information centric network architec-

ture. Breaking with the node-centric model of the Internet, they emphasize the role of location-agnostic names, leading to a non-conversational model of information access and the secure binding of name and content.

A single protocol is defined and considered sufficient for all of the network's operation: The basic vertical interaction between a client and the CCNx network is the submission of INTEREST requests and reception of CONTENT reply messages (see Figure 2). The same protocol is used horizontally inside the network when peer nodes are interconnected.

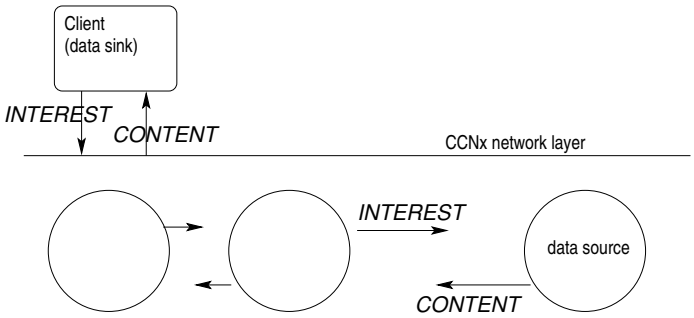

Figure 2: Content Centric Networking: A single lookup protocol based on Interest and Content messages, used both vertically (access) and horizontally (propagation).

3.1 RPC-like interaction, no parameters

In terms of RPC, one easily identifies the request/reply pattern, although conceptually only one implicit remote procedure is provided which we could call "lookup": A client, or a peer node, requests the resolution of the given name and gets back the content that is bound to it.

But then, the two other attributes of RPC, remote procedure selection and parameter marshalling, are missing. An interpretation of the CCNx' interest/content pattern as RPC is therefore not obvious and remains partial. For example, the implicit "name lookup" procedure expects a single parameter (the content's name). In case the interest message is used to access a remote multiparameter service, the parameters must be encoded in the interest's name field (like DNS can be abused to tunnel IP over DNS requests). In Section 4.3 we suggest a clean way of passing parameters explicitly.

3.2 Receiver-driven communication (and the universality of the Interest/Content primitives)

In CCNx, the explicit request for named data leads to a receiver driven setup: It is the data *sink* that has to request each data block, while in TCP it is the *source* which paces the data delivery.

As was shown in figure 1, the receiver-driven packet exchange can be arranged such that the pattern is identical as in the sender-driven case. This permits to position the interest/content protocol at the same level as IP whose forwarding semantics does not follow a request/reply pattern (in case there is no error). The argument is that when TCP is combined with IP, a request/reply pattern will nevertheless emerge and hence is equivalent to the CCNx mode of operation.

We now push this argument even further and complement CCNx such that it becomes a full fledged RPC. Once this is achieved, the RPC mechanics is able to serve as universal glue between nodes and clients, it can model TCP behavior as CCNx does, and it can address some of CCNx' shortcomings like the single-parameter setup mentioned above, but also the still debated use of NACK messages in CCNx, whose lack currently renders networking debugging virtually impossible.

4 Embedding CCNx and NDN in full RPC

In this section we describe our simple RPC framework in terms of its interaction and configuration model, the way of selecting among remote procedures, as well as an encoding scheme. Our solution was designed to "fit" both the CCNx and NDN competitors, yet allows to manage these two dialects as well as future variations. We point out that our experimental protocol does not aim at becoming *the* new information centric protocol, but rather represents a conceptual reference point and feasibility study. Ideally, these insights would be incorporated in a next versions of CCNx and NDN.

4.1 RPC interaction pattern and demux requirement

Following CCNx and NDN's basic interest/content interaction pattern, two peering entities emit RPC request datagrams for which they expect a reply. We assume that there is a means to distinguish at least four different flavors of datagrams (interest, content, RPC request, RPC reply) using dedicated "code points" in a datagram's header bytes. Different techniques can be envisaged to provide this demultiplexing feature. In our experimental setup we picked a single "magic byte" that is neutral with respect to all six different packet types and variants (interest vs content PDU, for each of the old CCNx, new CCNx and the new NDN encoding). This byte (0x80 at position 0) stands in each of these encodings either for an invalid code or has no assigned meaning yet. The neutral code point permits to declare any of the three protocol suites as being the currently active one (i.e. ship unmodified datagrams), yet have RPC packets to be continously detectable regardless of the active protocol. In modern terminology, this RPC channel is used to establish a hypervisor for the paravirtualized ICN protocols.

4.2 Lambda expressions for named functions

An important element of an RPC framework is the breadth of procedure signatures that it supports. Typically, an RPC request names the remote procedure to be invoked, so we need a way to express "application of a function" to some parameter values:

```
REQ( fct(param1, .. paramN) )
```

We expect these parameters to be names (as is the case in an information centric network) and apply this not only to the parameters but also to the function that has to be invoked:

```
REQ( /name/of/fct(/name/of/param1 ..) )
```

This scheme is made more general by permitting parameters as well as function names to be recursively expressed by function applications. That is, instead of providing the name of a remote function, it is also possible to provide a recipe how the peer should compute the remote function's name, for example first doing a lookup:

```
REQ( /name/of/lookup(/name/of/fct) (/name/of/param ..) )
```

In a last generalization step we add a syntactic lambda construct for defining name shuffling operations. Overall this means that the RPC call, if implemented in its fullest form, requests the remote execution of an arbitrary lambda expression, where these expressions are recursively defined by the three forms "variable", "application" and "abstraction". The following table summarizes the syntactic forms that our RPC supports:

`REQ(name)`	**"Variable"**: returns the value to which 'name' is bound. Note that this copies the behavior of the CCNx and NDN: an INTEREST(**name**) packet requests a CONTENT(**value**) packet.
`REQ(fct(expr))`	**"Application"**: returns the result of applying the named function to the parameters defined by expression 'expr'.
`REQ(λx.expr)`	**"Abstraction"**: creates an anonymous function based on 'expr' with 'x' as its formal parameter (and in this case might return an opaque and transient name for a so called thunk which can be used in subsequent requests).

4.3 TLV-Encoding

The original CCNx [5] protocol introduces a binary encoding format for XML called "ccnb". Both interest and content packets are then defined as an XML data structure and do not need any fixed packet header anymore. This approach is interesting because

it permits to transfer complex data structures in a standard way (like e.g., JSON) yet remain extensible, what in a fixed header world would require option fields.

Although designed as a new packet formatting strategy for CCNx, we can make perfect use of this approach for RPC, both for parameter data structures *and* the syntactic structure of lambda expression. In Appendix A we give the EBFN grammar which governs our encoding of lambda expressions, as well as basic data types and structures.

The ccnb format was criticized for its parsing overhead, leading to a next generation encoding using a type-length-value style. Unfortunately, two different TLV encodings were chosen for the new CCNx and the new NDN format such that some start byte sequences could be valid for both protocol suites. In our implementation we picked NDN's TLV encoding, but ultimately this choice does not matter as long as recursive data structures as those defined in Appendix A can be represented. This is also the reason why we omit here the details of the TLV formats which can be found in [6] and [7].

An encoding example (for tunneling packets with alien encodings)

Note that the incompatible encodings chosen by new CCNx and new NDN foil an easy demultiplexing approach. As a consequence, their implementations need different TCP and UDP ports, or different Ethertypes if to be run on the same LAN. We avoid this problem through a "tunnel" function that forwards a packet passed as a parameter. The remote procedure call

```
REQ(/rpc/builtin/forward(/rpc/const/ccn2013enc, data))
```

asks for forwarding of the given a packet with given encoding and would be represented as (the length of a TLV block 'len' is shown as a symbol):

```
0x80 len          # =APPL (RPCREQUEST)
  0x83 len        # =NAME (EXPR, VAR)
    "/rpc/builtin/forward"
  0x82 len        # =SEQ (ARG)
    0x83 len      # =NAME (BASIC)
      "/rpc/const/ccn2013enc"
    0x85 len      # =BIN (BASIC)
      data
```

The following paragraphs introduce more such examples.

4.4 Examples of link-layer usage of RPC

We illustrate the use of the RPC framework with four use cases.

4.4.1 Selecting the default protocol suite

One design goal of our RPC was that it would not introduce any changes to existing ICN protocols, yet permit the handling of several of them potentially running over the

same channel at the same time and without recourse to outer demultiplexing support. To this end, one of these suites is declared the default protocol and is permitted to run natively. Thanks to the neutral code point, we can invoke our RPC mechanism to switch protocols at any time:

```
REQ(/rpc/builtin/setDefaultSuite(/rpc/const/ndn2013))
```

and the reply could for example be

```
REP(200, "ok")
```

In a similar way, the currently active suite can be inquired, or a list of supported encodings be requested.

4.4.2 Tunneling non-default packet formats

In case packets with incompatible encodings interleave, one can invoke the remote forwarding routine for the packets with an encoding that deviates from the default one, as was already shown in the previous section:

```
REQ(/rpc/builtin/forward(/rpc/const/ccn2013enc, data))
```

while the default encoding would run as normal and in parallel:

```
INTEREST_NDNenc(someNDNname)
CONTENT$_NDNenc(anotherNDNname, data)
```

4.4.3 Header compression, link fragments

Using RPC and remote state change, a peer can define mappings between a long byte sequence and a user assignable code point[1]. Once defined, such code points can be used to request expansion at the remote side, as the following lines illustrate:

```
REQ(/rpc/builtin/define(\%08, /rpc/builtin/forward))
REQ(/rpc/builtin/define(\%09, /some/very/long/name))
REQ(/rpc/builtin/define(\%0a, /rpc/builtin/insertname))
REQ(/rpc/builtin/define(\%0b, /rpc/builtin/concat))
...
REQ(\%08(\%0a(pktdata1,\%0b(\%09,"/001"))
REQ(\%08(\%0a(pktdata2,\%0b(\%09,"/002"))
...
```

Similarly, different fragmentation schemes can operate in parallel and retransmission schemes can be configured on a on-demand basis. The following hypothetical example requests the retransmission for a range of sequence numbers:

```
REQ(/rpc/builtin/frag2014/arq(6625,6639))
```

[1] We envisage the use of an "encoding vector" similar to PostScript where character code points are mapped to the glyph rendering procedures.

4.4.4 Conditional network actions, network management

Coming back to the theme of version numbers and protocol options, a peer can discover run-time facts and act differently, all expressed as a lambda expression:

```
REQ(if(gt(/rpc/builtin/firmwarevers, 2.98)/rpc/builtin/optimze0n))
```

As a final example consider the task of specifying more than one remote action on a packet. Here, we use lambda expression abstraction in order to define a function that first copies each packet and then forwards it. Note that the packet's data is only shipped once but acted on twice:

```
REQ((λ x.(/rpc/builtin/syslog(x);
          /rpc/builtin/forward(x))) pktdata)
```

4.5 Summary of extensions over CCNx and NDN

Like a keyhole, we use a single code point to insert our RPC packets into the "packet header name space" of CCNx and NDNx. This would be a structural requirement for our approach when it should be applied to a non-ICN environment.

Both ICN protocols studied here, although incompatible at several levels, can run unchanged either natively or tunneled. With the RPC mechanism in place, one can control the channel in which this happens. The name space rooted in the neutral code point is used for a TLV-encoded expression format that captures full lambda terms as well as binary data necessary for shipping parameter values. This goes beyond traditional RPCs where only a single procedure name and its parameters can be specified. An important property is that this setup permits to change, on a per-direction basis, the default protocol suite *including* the low level bit encoding. We consider this to become an important feature and much better tool than version numbers for bridging the vast range from highspeed networks to challenged networks for the Internet of things, where non-conventional encodings will be needed for sure.

4.6 Implementation in CCN-lite and integration with NFN

The RPC and TLV scheme described in this technical report has been implemented in a simple form inside CCN-lite [2], an independend implementation of the CCNx and NDN protocols. The next and natural step will be to integrate it with Named Function Networking [10], which lifts the face-local RPC view to global level.

5 Related Work and Discussion

The 4D proposal [3] suggests to bootstrap link-level communication through management functionality, relying on a remote procedure call interface to discover and configure a peer interface's characteristics. In general, network management is related to our "dialect management" view, as well as the encoding question. A huge and complex body of work was done on ASN.1 (used inside SNMP) and the basic encoding rules for representing arbitrary data structures (another approach would be SUN's XDR, the

external data representation toolkit). CCNx and NDN encompass several key infastructure elements in that context, for example the hierarchical name space concept which lends itself well for representing management information bases (MIBs).

Clearly, our approach has limitations and critical aspects out of which we highlight two. First, RPC only works if bidirectional communication is available, and is more difficult to implement and exploit in a broadcast or multicast setup. For Ethernet, for example, no request/reply pattern is available at the link layer control level, representing a lower boundary for the applicability of the RPC pattern. An obvious critical aspect is security where both access control for limiting execution of arbitrary builtin functions is needed, but also restrictions on the language level if full lambda calculus expression should be envisaged (long-running or non-terminating loops, for example). While access control and the use of security credentials can be captured by appropriate builtin functionality (hence within our RPC framework), pragmatic RPC implementations will put strict resource limits for each RPC request in terms of resolution steps or memory consumption, which is more an orthogonal intervention.

6 Conclusions

The CCNx protocol designers have made it clear that CCNx aims at a clean slate approach to network design and that its role is to ultimately replace IP as the universal glue of a future Internet. This position was backed for example by showing how packet pair interaction patterns at transport level (e.g., data/ack) naturally map to network level, for which a universal interest/content exchange was proposed and deemed sufficient once and for all.

But with the original CCNx protocol currently being redefined, and a competing variant (NDN) being explored, these researchers are faced with incompatible protocols and no interworking solution that could handle these or any future variants. In this technical report, we provide such an interworking solution. We immediately point out that this solution cannot make the competing protocols interoperable nor bridgable, except that protocol evolution itself becomes managable. Surprisingly enough, this is achieved with almost the same interest/content pattern, and extending the named data approach to named function, which in this report we lift to full RPC capabilities. With such rich expressiveness there is no need for version numbers in protocol headers anymore.

References

[1] Andrew D. Birrell and Bruce Jay Nelson. *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems 2(1), Feb 1984

[2] `http://www.ccn-lite.net/`

[3] A. Greenberg et al. *A clean slate 4D approach to network control and management*. ACM SIGCOMM CCR, Volume 35 Issue 5, Oct 2005

[4] Van Jacobson, Diana K. Smetters, Nicholas H. Briggs, James D. Thornton, Michael F. Plass and Rebecca L. Braynard. *Networking Named Content*. CoNext, Dec 2009

[5] M. Mosco. *CCN Binary Encoding (CCNB)*. `http://www.ccnx.org/wp-content/uploads/2011/06/mosko-ccnb-01.txt`, 2007 – Jul 2013

[6] M. Mosco. *CCNx 1.0 Protocol Specifications Roadmap*. `http://www.ccnx.org/wp-content/uploads/2011/06/CCNx_v1_Roadmap.pdf`, Nov 2013

[7] *NDN Packet Format Specification 0.1 documentation*. `http://named-data.net/doc/ndn-tlv/`, Oct 2013

[8] S. O'Malley and L. Peterson. *TCP Extensions Considered Harmful*. RFC1263, Oct 1991.

[9] Sun Microsystems. *NFS: Network File System Protocol Specification*. RFC1094, Mar 1989

[10] C. Tschudin and M. Sifalakis. *Named Functions and Cached Computations*. In Proc 11th Annual IEEE Consumer Communications and Networking Conference, Jan 2014.

Appendix A) EBNF for RPC

EBNF grammar for our RPC datagrams and their TLV encoding, where the "type of the TLV block" is some code point (see table below) and where 'len' is the number of bytes needed to store the octets of the rest of the grammar line (i.e., the "value of the TLV block"):

```
RPCREQUEST ::= APPL
RPCRESULT  ::= appl_cp len INT STR ARG*

APPL   ::= appl_cp len EXPR ARG*
VAR    ::= NAME | user_defined_cp 0
LAMBDA ::= lambda_cp len VAR EXPR*

EXPR   ::= APPL | VAR | LAMBDA

ARG    ::= EXPR | SEQ | BASIC
SEQ    ::= seq_cp len ARG*
BASIC  ::= NAME | INT | BIN | STR

NAME   ::= nam_cp len octet*
INT    ::= int_cp len octet*
BIN    ::= bin_cp len octet*
STR    ::= str_cp len octet*
```

Appendix B) List of assigned code points

The following table lists the reserved code points. Note that only the first three type values (interest, data, appl_cp) appear at the front of a datagram, acting as a magic number, and that the other code points are internal to our RPC packets.

0x05	*interest* (defined by NDN)
0x06	*data* (defined by NDN)
0x80	`appl_cp` (reserved across all protocol suites)
0x81	`lambda_cp`
0x82	`seq_cp`
0x83	`nam_cp`
0x84	`int_cp`
0x85	`bin_cp`
0x86	`str_cp`
0x00..0x7F	The values 0x00…0x7F are names with length 0 and an empty value field. They only appear inside a TLV structure. These names have no (globally) assigned meaning and are available for users to bind them to arbitrary objects (functions, constants).