zenoh: The zero network overhead protocol

Angelo Corsaro[*], Erik Boasson[†] and Olivier Hecart[‡]
ADLINK Technology

July 23, 2018

> ""Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
>
> — Edsger W. Dijkstra

1 Introduction

Zenoh's goal is to bring data-centric abstractions and connectivity to devices that are constrained with respect to the node resources, such as computational and storage, power, and the network. Zenoh deployments such as those reported in Figure 1, highlights how devices constrained by local resources and or network can be deployed transparently using peer-to-peer or client-to-broker.

It is important to remark that constraints, on the node and on the network, are orthogonal and apply independently. For instance, there are some use cases in which non-constrained devices communicate through extremely constrained networks. In other cases, constrained devices may communicate through non-constrained networks such as WiFi.

Zenoh targets environments where devices may have very little computational power and memory, such as an Arduino[1] Uno which has 2 Kbytes of SRAM and 32 Kbytes of FLASH, may be battery powered and may communicate through constrained networks such as 40-100 Kbps LoWPANs and the newly standardised NB-IoT. Notice, that use cases for which only a subset of these requirements apply are still ideally addressed by Zenoh.

From a high-level perspective, the key requirements that a Zenoh implementation has to satisfy are (1) extremely low footprint — addressing targets such as an Arduino Uno platform, (2) extremely efficient wire protocol — inducing a protocol overhead of just a few bytes over the user data, and (3) support devices that undergo aggressive sleep cycles.

---

[*]angelo.corsaro@adlinktech.com

[†]erik.boasson@adlinktech.com

[‡]olivier.hecart@adlinktech.com

[1]The specification for existing Arduino variants is available at `https://www.arduino.cc/en/Products/Compare`
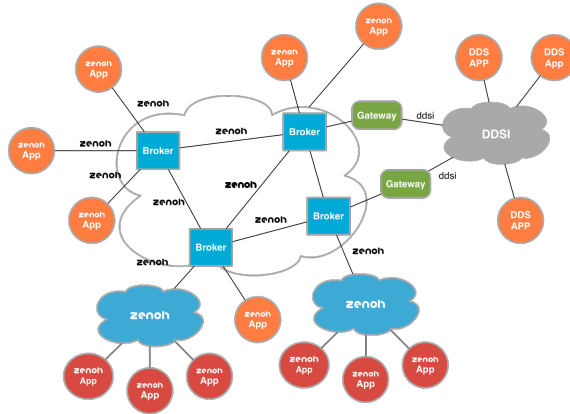
Figure 1: Typical deployment of a Zenoh system, also showing a possible bridging from/to DDS.
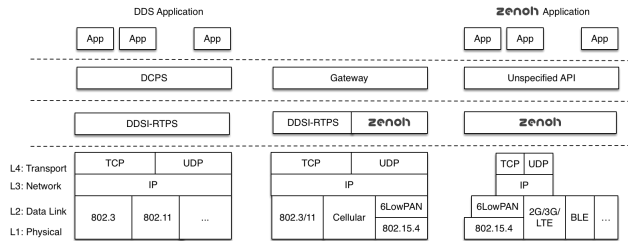

Figure 2: Typical deployment of a Zenoh system, also showing a possible bridging from/to DDS.

As shown in Figure 2, zenoh can be (1) used stand-alone, (2) bridged to DDS through an gateway, and (3) run as a replacement of the DDSI-RTPS protocol for better scalability and performance. This specification does not address the API that should be used by Zenoh applications. As far as an application exposes an "on-the-wire" behaviour in conformance with the Zenoh protocol, the API used by the application is entirely up to the vendor. The decision of keeping the API out of scope is to ensure that the right level of variability can be achieved to address the constraints imposed by different targets.

2 Zenoh

This specification defines the protocol used by Zenoh applications to communicate with a broker and/or with each other. Before delving into the details of the protocol, this specification defines the abstractions supported and how these relate to the concepts of the OMG DDS specification. Then, it precisely defines the protocol, including its semantics, state machine and framing.

2.1 Abstraction

Zenoh applications coordinate by autonomously, anonymously and asynchronously writing and reading in a data space while being decoupled in time and space. The abstraction of a time decoupled data-space is essential in supporting applications that undergo sleep cycles and specifically in decoupling the availability of data with the availability of the applications that wrote it (as the latter may be sleeping now).

In the reminder of this section we will define precisely the abstraction used by Zenoh and how they map to DDS.

Resources

Zenoh relies on *resources* to identify the information to be exchanged between readers and writers and on *resource properties* to specify the properties of exchanged data. The concepts used by Zenoh at the same time simplify and extend the concepts used by DDS. These changes were motivated by over a decade of user feedback with respect to some of the complexities of DDS.

An Zenoh *resource* is a closed description for a set. If the cardinality of the set is one then we call it a *trivial resource.* An Zenoh resource is described by means of a URI which may only include path expansions. Note, for compactness we take the liberty of representing trivial resources by the single element that constitutes the set as opposed to the single element set.

Below are some non-normative examples of Zenoh resources:

```
-- These are Resources
zenoh://myhouse/**/bedroom/*/LightStatus
zenoh://myhouse/**/musicroom/LightStatus
zenoh://myhouse/**/LightStatus
zenoh://myhouse/**

-- These are Trivial Resources
zenoh://myhouse/floor/1/musicroom/LightStatus
zenoh://myhouse/floor/2/musicroom/LightStatus
zenoh://myhouse/floor/2/bedroom/erik/LightStatus
```

The canonical representation of a resource shall not contain white space characters.

Resources have associated properties. These properties are propagated as part of resource declaration (see declaration messages) and are made available under the /property URI postfix.

The data read and written by Zenoh applications is associated with one or more resources identified by a URI. Depending on the context a resource may represent a topic or an instance. In other terms, instances may be addressed by including the instance-id in the URI, or instances may be implemented by a layer on top of Zenoh.

The formal syntax for resources is defined below:

```
resource         ::= "zenoh://" path markers
path             ::= name | name "/" path
markers          ::= "#" | "#" markerlist
markerList       ::= name | "{" nonemptynamelist "}"
nonemptynamelist ::= name | name "," nonemptynamelist
```

Where name is a valid URI name extended with "?", "*", and "**" wild-cards. Of these: "?" matches exactly one character excluding the path separator, "*" matches any number of characters excluding the path separator, and "**" matches any number of characters including the path separator.

Resource Reliability

Resources are reliable by default unless differently specified as part of the resource properties.

Resource Durability

Resources may be durable or volatile. The nature of durability is defined by naming convention in the resource and interpretation by the durability service. For instance:

```
-- Volatile Data
zenoh://myhouse/floor/1/musicroom/LightStatus
zenoh://myhouse/floor/1/musicroom/LightStatus/ID12345

zenoh://myhouse/floor/1/musicroom/LightStatus/property

-- Persistent Data
zenoh://myhouse/floor/1/musicroom/LightStatus#persistent

-- Transient
zenoh://myhouse/floor/1/musicroom/LightStatus#transient

-- All Resources that may be either transient or persistent
zenoh://**#{transient,persistent}
zenoh://**#{transient,persistent}
```

Meta Resources

These are system defined read-only resources such as:

```
zenoh://myhouse/floor/2/bedroom\$ls
```

The resource above would return the set of child resources of zenoh://myhouse/floor/2/bedroom.

meta resource	
$ls	list of child resources
$id	an ID currently associated with the resources, selected arbitrarily from the set of currently known IDs associated with it

Selections

An Zenoh *Selection* is the conjunction of a Resource, a list of hashes, and a predicate over the resource content. Zenoh provides a compact way of representing queries by concatenating the string representation of the resource, the list of hashes and the predicate. The resource and the list of hashes are separated by a "#" while the hashes and the predicate are separated by a "?".

For instance, these are legal Zenoh queries:

```
zenoh://myhouse/floor/2/bedroom/erik/LightStatus#?
zenoh://myhouse/floor/*/bedroom/*/LightStatus#?{luminosity>0}
zenoh://myhouse/floor/*/bedroom/*/LightStatus#{transient}?{luminosity>0}
zenoh://myhouse/floor/1/musicroom/LightStatus/ID12345?{inRange(property.location,
                                                      someLocation,radius)}
```

If the predicate is a tautology on the resource content, then the predicate can be omitted.

The canonical representation of a selection may not contain unnecessary white space characters.

For brevity, when the list of hashes is empty than the "#" can be omitted. Likewise if the predicate is a tautology the "?" can also be omitted.

Consequently, the two queries below are equivalent:

```
zenoh://myhouse/floor/2/bedroom/erik/LightStatus#?
zenoh://myhouse/floor/2/bedroom/erik/LightStatus
```

The formal syntax for queries is reported below:

```
query     ::= resource "?" predicate
predicate ::= | "{" nontrivialpredicate "}"
```

For interoperation with DDS, implementations that provide the support for queries should at least support as predicate the same subset of SQL92 supported by DDS on (X)CDR encoded data. That said, implementations are free to support many different syntaxes, such a subset of JavaScript.

Zenoh has entities that serve a role equivalent to that of DDS' Domain Participant, Data Writer and Data Reader. Zenoh terminology however, while the participant still exists and has a unique ID across the system, the entities that write and read data are called publisher and subscribers. These should not be confused with DDS Publisher/Subscriber as these entities are not exposed on the wire.

2.2 Mapping DDS Topics to Zenoh Resources and Vice-versa

The mapping of a DDS Topic into an Zenoh resource is relatively straightforward, but due to the fact that DDS matched readers and writers based on the QoS as well as the partition setting of the DDS Publisher/Subscriber entities, it is impossible to declare an Zenoh resource corresponding to a Topic. Instead one can only declare the resources for the DDS DataReaders/DataWriters — as only in this case all the relevant elements are known.

Given a DDS data-writer or data-reader declared for a topic T, we can then talk about the set of equivalent resources $\{R\}$. The cardinality of the set $\{R\}$, $|R|$ is the same as the number of partitions in which T is written by the data writer or read by the data reader.

For each partition P associated with this specific topic T, the derivation of the associated resource R for a given sequence S that acts as the equivalent of a path separator in partition names is as follows:
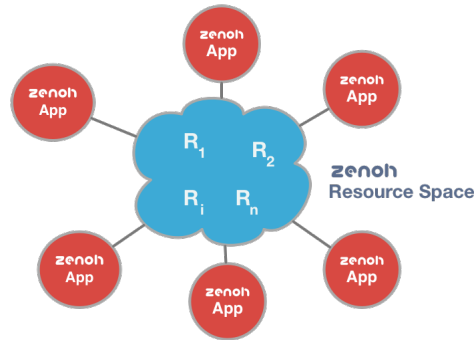
Figure 3: Zenoh interaction model

- The Topic QoS as well as the topic type information is written as a property of the Resource. Where Zenoh does not explicitly specify its own encoding, the serialisation format shall be the same as the one used by DDS when transmitting the QoS information over the wire.

- The partition P is transformed into a valid URI path by repeatedly substituting "/" for each occurrence of S, "*" for each occurrence of "**" and of "?*" for "*?" until no further substitutions can be made; then substituting "**" for each occurrence of "*".

- The resource name is then concatenation of the string "zenoh://", the URI path equivalent to the partition, a slash, the topic name, and the durability tag corresponding to the durability of the writer/reader (if necessary).

Mapping an Zenoh resource into a DDS topic, is trivial as all the information required is provided as part of the resource properties. This information, will be created by the broker to create the proper DDS entities to ensure interoperability between DDS and Zenoh.

2.3 Zenoh Programming Model

As discussed above the Zenoh standard does not define an API. That said the programming model is conceptually the same as the one for DDS. In other terms, as depicted in Figure 3, Zenoh provides a data space abstraction in which applications can read and write data autonomously and asynchronously. The data read and written by Zenoh applications is associated with one or more resources identified by a URI.

The remainder of this specification defines the protocol that makes it possible to implement the Zenoh data space abstraction. This protocol supports data space implementations that are (1) based on a network of brokers, (2) decentralised/peer-to-peer, or (3) a combination of (1) and (2).

As a result, the protocol defines how applications can discover each other, how application can declare their interests and how they can share data.

3 Message Structure, Framing and Encoding

3.1 Notation

Zenoh messages are defined using a very small domain-specific language. The primary reason for using a DSL is to efficiently and unambiguously describe the way header flags are used to elide discriminators in discriminated records. The only primitive types are natural numbers and bytes. The natural numbers are serialised using variable length encoding, whereas bytes are serialised unchanged. The language specification allows any number of definitions to be concatenated, in this document we will present individual definitions that together constitute the full set of definitions for Zenoh messages.

XXX standard tokenized language where white space is ignored
XXX tokenization follows the maximal-munch rule
XXX identifiers are a letter or underscore followed by a, possibly empty, sequence of letters, underscores or digits
XXX natural numbers are non-empty sequences of decmial digits
XXX keywords cannot be used as identifiers
XXX keywords and identifiers are case-sensitive
XXX booleans are pre-defined enumerated types with constants 'false' and 'true'

⟨*definitions*⟩ ::= ⟨*definition*⟩
 | ⟨*definition*⟩ ⟨*definitions*⟩

⟨*definition*⟩ ::= ⟨*identifiers*⟩ '=' ⟨*type*⟩

⟨*identifiers*⟩ ::= ⟨*identifier*⟩
 | ⟨*identifier*⟩ ',' ⟨*identifiers*⟩

All identifiers in the lhs of a definition are defined to be of the type specified as the rhs.

⟨*type*⟩ ::= ⟨*identifier*⟩
 | 'nat'
 | 'byte'
 | ⟨*sequence*⟩
 | ⟨*header*⟩
 | ⟨*flag*⟩
 | ⟨*enum*⟩
 | ⟨*record*⟩
 | ⟨*case*⟩
 | ⟨*option*⟩
 | ⟨*type*⟩ ⟨*constraint*⟩

⟨*sequence*⟩ ::= '[' ⟨*type*⟩ ']'

⟨*constraint*⟩ ::= 'range' ⟨*nat*⟩ '...' [⟨*nat*⟩]

A *sequence* is a (possibly empty) list of any type other than a *flag*. Serialisation of a list is the length serialised as a natural number followed by the serialisation of the elements of the list.

The only *constraint* is the range constraint, which may be applied to integral types (*nat* and *byte*) and to sequence types. For an integral type, the value is required to be within the interval; for a sequence type the constraint is on the length of the sequence. Hence

---
`[`nat `range` 1 ... 31415`]` `range` 1 ...

---

is a non-empty list of natural numbers, with each number no less than 1 and no larger than 31415.

$\langle header \rangle$ ::= 'header' $\langle nat \rangle$ $\langle type \rangle$

A *header* contrains *type* to a byte or an enumerated type. The value must be $< 2^{nat}$. A header is always serialised as a single byte.

$\langle flag \rangle$ ::= 'flag' $\langle nat \rangle$

A *flag* is a boolean that is serialised by adding 2^{nat} to the nearest preceding header iff true. The position must be available (hence a sequence of flags is impossible).

$\langle enum \rangle$ ::= 'enum' '{' $\langle symbols \rangle$ '}'

$\langle symbols \rangle$ ::= $\langle symbol \rangle$
  | $\langle symbol \rangle$ ',' $\langle symbols \rangle$

$\langle symbol \rangle$ ::= $\langle identifier \rangle$ '=' $\langle nat \rangle$

An enumerated type defines a bijective mapping of symbols to natural numbers. It is serialized as a natural number, except when used as a header.

$\langle record \rangle$ ::= 'record' '{' $\langle member \rangle$ { ',' $\langle member \rangle$ } '}'

$\langle member \rangle$ ::= $\langle identifier \rangle$ ':' $\langle type \rangle$

A record is a product type, serialised by concatenating the serialised forms of its constituing types.

$\langle case \rangle$ ::= 'case' $\langle identifier \rangle$ 'of' '{' $\langle cases \rangle$ '}'

$\langle cases \rangle$ ::= $\langle labels \rangle$ $\rightarrow$ $\langle type \rangle$

$\langle labels \rangle$ ::= $\langle label \rangle$
  | $\langle label \rangle$ ',' $\langle labels \rangle$

$\langle label \rangle$ ::= $\langle identifier \rangle$
  | $\langle nat \rangle$ ['...' $\langle nat \rangle$]

The members of a record may be sum types, where the discriminator is specified explicitly as a preceding member of the containing record. It is serialised by serialising the case selected by the discriminator; if no case matches the discriminator, nothing is serialised. E.g.:

```
record {
  a: header 5 enum { x = 1, y = 2, z = 3 },
  b: flag 5,
  c: case b of { true -> record { d: flag 6, e: nat } }
}
```

serialises to a one-byte header of which the least significant 5 bits are used for encoding the enumerated type, and where bit 5 is used to encode b. If b is false, that is all; if b is true, however, bit 6 of this header byte is used to encode the value of d, and the header byte is followed by the value of e in variable-length encoding.

$\langle option \rangle ::= \text{`if'} \langle type \rangle \text{`then'} \langle type \rangle$

This is syntactic sugar for:

```
record { present: A, value: case A of { true -> B } }
```

where A is the first *type* and B the second.

Variable-length encoding

Natural numbers are serialised using variable-length encoding, the number n is serialised as a sequence of bytes $128 + s_0, 128 + s_1, \ldots, 128 + s_{k-1}, s_k$

$$n = \sum_{i=0}^{k} 2^{7i} s_i \wedge 0 \leq s_i < 128 \wedge k = \begin{cases} 0, & \text{for } n = 0 \\ \lceil (\log_2 n)/7 \rceil, & \text{for } n > 0 \end{cases}$$

Strings

Strings are simply sequences of bytes, i.e.:

```
string = [byte]
```

3.2   Framing

The zenoh protocol assumes that communication occurs as arbitrary sized messages containing a concatenation of zenoh messages. For a particular transport, implementations may limit the size of the messages, e.g., on Ethernet it makes sense to adhere to the Ethernet MTU of 1500 bytes.

For message-preserving transports such as UDP/IP, each of these arbitrary sized messaged corresponds to a single transport message (UDP datagram). For reliable streaming transports such as TCP/IP, the transport layer shall insert information to reconstruct the message boundaries. Each arbitrary sized message shall be preceded with its size in bytes using VLE encoding. An implementation may signal an unbounded message size by specifying a size of 0 bytes, after which the stream consists of nothing but zenoh messages.

XXX picture

3.3 Message Structure

An Zenoh message is composed of a single byte header and a body:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
|X|X|X|MessageId|
+---------------+
~      body      ~
+---------------+
```

Any flag field that is not defined shall always be set to zero in outgoing messages, and shall be ignored in received messages.

```
Message = record {
  kind: header 5 MessageId,
  body: case kind of {
    ScoutId       -> Scout,
    HelloId       -> Hello,
    OpenId        -> Open,
    AcceptId      -> Accept,
    CloseId       -> Close,
    DeclareId     -> Declare,
    StreamDataId  -> StreamData,
    BatchedDataId -> BatchedData,
    WriteDataId   -> WriteData,
    PullId        -> Pull,
    PingPongId    -> PingPong,
    SynchId       -> Synch,
    AckNackId     -> AckNack,
    KeepAliveId   -> KeepAlive,
    FragId        -> Frag,
    ConduitId     -> Conduit,
    MigrateId     -> Migrate,
    RSpaceId      -> RSpace
  }
}

MessageId = enum {
  ScoutId           = 1,
  HelloId           = 2,
  OpenId            = 3,
  AcceptId          = 4,
  CloseId           = 5,
  DeclareId         = 6,
  StreamDataId      = 7,
  BatchedDataId     = 8,
  WriteDataId       = 9,
  QueryId           = 10,
  PullId            = 11,
  PingPongId        = 12,
  SynchId           = 14,
  AckNackId         = 15,
  KeepAliveId       = 16,
```

```
  ConduitCloseId   = 17,
  FragId           = 18,
  ConduitId        = 19,
  MigrateId        = 20,
  StreamDeltaId    = 21,
  BatchedDeltaId   = 22,
  WriteDeltaId     = 23,
  RSpaceId         = 24
}
```

Flags

The header flags have single-letter names, but to let these letters be mnemonic, each flag has several names depending on context:

```
S, I, M, P, O, L = flag 5
C, R, H, N       = flag 6
F, U, G, A, Z    = flag 7
```

XXX I think we should rename the O flag to the E flag (for "Echo"): then it reads "SIMPLE" :)

Their (primary) use is as outlined in the below table, for easy reference. For the definitions, see the message definitions that use them. The Z, L and H flags are odd ones and used only for Conduit message. XXX what about I, F XXX wasn't there another one besides Conduit now? XXX

flag	name	interpretation
S	sync	used in reliable messages to request a prompt acknowledgement
M	mask	presence of a mask in an acknowledgement
P	properties	presence of properties in the message
C	committed	a set of declarations that has already been committed
R	reliable	indicates that the message is reliable
N	number	the presence of a count
A	actual	an "actual" resource id is present in the contents of a data set
U	unacked	count of messages available for retransmission
G	global	indicates the selection id is globally unique

Properties

A property is a pair of a numerical id and a octet sequence as value. Properties are always optional, their presence indicated by the P flag:

```
Property   = record {
  id: nat,
  value: [byte]
}
Properties = if P then [Property]
```

Even property ids are reserved for standardised properties, odd ones are reserved for vendor-specific properties. The standardised properties and their

values are as follows, where the specified value shall be a prefix of the "value" sequence of octets:

id	interpretation
0	vendor id a DDSI vendor code in DDSI encoding (default is an unspecified vendor)
2	maximum number of conduits (that is, highest supported conduit id + 1), VLE encoded (default is 1)
4	sequence number length in bits, VLE encoded (default is 14)
6	reliability flag, either 1 for reliable, or 0 for unreliable (default is 1, reliable)
8	durability flag, either 0 for volatile, 1 for transient, or 2 for persistent (default is 0, volatile)
10	commit mode, the number of pending declarations in VLE encoding that the sender can handle while processing a COMMIT message. When 0, no declarations may be sent to it following a COMMIT until after the corresponding RESULT has been received; when 1, one may continue, but must stop at the 2nd COMMIT until after a RESULT for the first COMMIT has been received; &c.
12	authentication data ...
14	client hash ...

Locators

Locators are ... udp/A.B.C.D:PORT ...

```
Locator = string
```

Peer IDs

...

```
PeerID = [byte] range 1 ...
```

3.4   Message Decoration

Messages can be "decorated" by prefixing them with certain markers. A message together with its decoration shall be treated as an atomic unit on the network. In this version of the protocol, the only markers are Conduit and Frag, both described the following sections. When a message is decorated with multiple markers, these markers shall occur in the order:

- Conduit

- Frag

- RSpace

3.5 Scouting

Scouting is used to discover anything that may engage in an Zenoh interaction, such as brokers, services, peers, etc. For those familiar with the DDS discovery protocol, Zenoh scouting can be seen as some kind of Participant Discovery Protocol.

Scout Message

A node may send a Scout message to probe responses from specific kind of nodes it is interested in. The Scout message has the following structure:

```
Scout = record {
  mask: nat,
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|    1    | (ScoutId)
+-+-+-+---------+
~      mask      ~
+---------------+
~  properties   ~ if P
+---------------+
```

Its attributes have the following meaning:

mask The bitmask identifying the kind of Zenoh nodes being scouted, and is unbounded. Implementations are required to handle at least masks < 16384. the meaning of the bits of mask is:

bit	interpretation
0	broker
1	durability
2	peer
3	client
4 ... 13	reserved

The default address for scouting on a UDP/IPv4-based transport is udp/239.255.0.1:7447.

Hello Message

Upon receiving a Scout message, a node matching the scout mask shall promptly respond with a Hello message:

```
Hello = record {
  mask: nat,
  locators: [Locator],
  properties: Properties
}
```
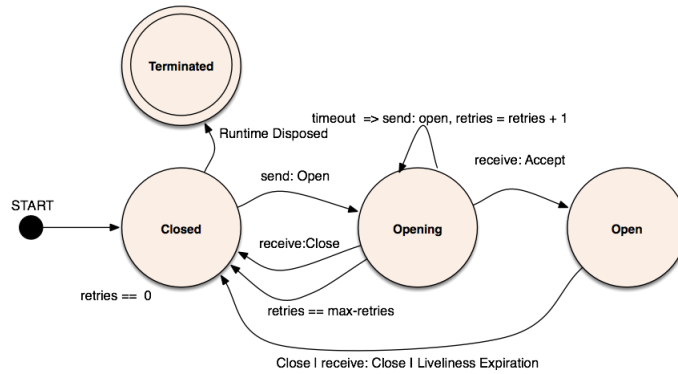
Figure 4: Session management state machine for the session initiator

```
+-+-+-+---------+
|X|X|P|    2    | (HelloId)
+-+-+-+---------+
~      mask     ~
+---------------+
~   locators    ~
+---------------+
~  properties   ~ if P
+---------------+
```

Its attributes have the following meaning:

mask The bitmask describing the Zenoh node that is sending the Hello; see
the Scout message for additional information on the interpretation of the
mask.

locators The list of locators at which this node can be reached, in addition
to the source address of the Hello message.

3.6 Session Management

Once scouting has resulted in the discovery of other nodes, or alternatively if
scouting information is configured statically, an Zenoh node can start estab-
lishing sessions with the entities with which it wishes to interact. An Zenoh
session is characterized by the state machine depicted in Figure 4. A newly
started application does not have any session, thus starts in the closed session
state. The session management protocol is then used to open as well close
sessions.

Session Establishment

The protocol exchange for opening a session between an Zenoh entities is de-
scribed in the sequence diagram described in Figure 6. As represented in
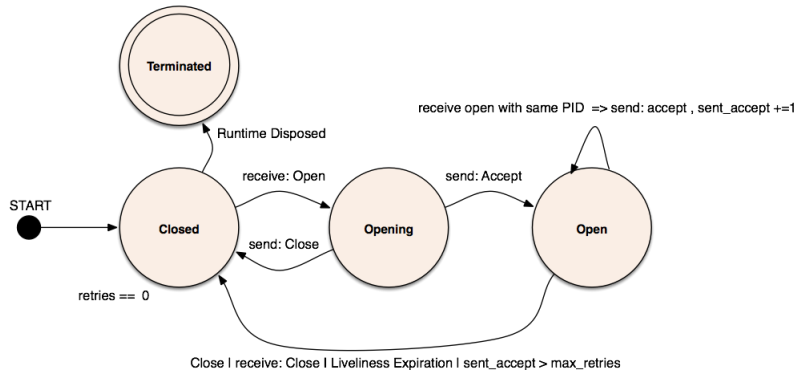Figure 6a, to establish a session an Zenoh application sends an Open message

Figure 5: Session management state machine for the broker

(a) Successful session establishment.

(b) Unsuccessful session establishment.
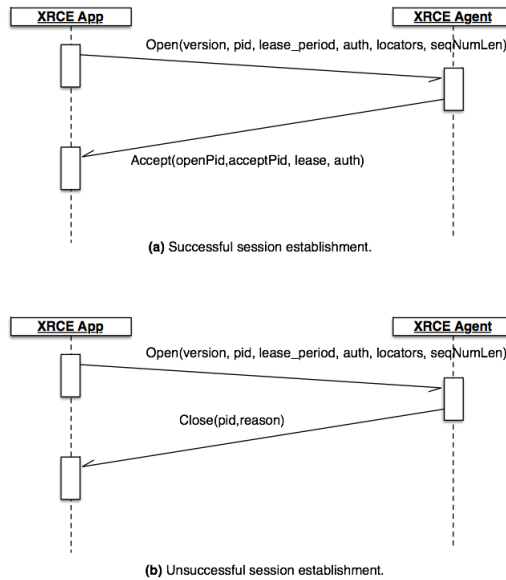
Figure 6: Successful session establishment

to the broker which in turn responds with an Accept if the section can be established or with a Close otherwise (see Figure 6b).

The Open message is sent with best effort reliability. As a consequence, an Zenoh application should resend an Open for which no Accept or Close has been received after a configurable timeout. Likewise, the Accept and the Close messages are also sent with best effort reliability. Thus, an Accept should be resent — up to a configurable maximum number of times — for session that have been already accepted each time a request is received by the broker.

The sequence number size is negotiated, that is, the Open message contains a requested sequence number size, and the recipient of the open message shall reject the session if it can't handle the requested sequence number size. The negotiated sequence number size is denoted by snsize.

Open Message

The Open message has the following structure:

```
Open = record {
  version: byte,
  pid: PeerID,
  lease: nat,
  locators: [Locator],
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|    3    | (OpenId)
+-+-+-+---------+
|    version    |
+---------------+
~      pid      ~
+---------------+
~     lease     ~
+---------------+
~   locators    ~
+---------------+
~  properties   ~ if P
+---------------+
```

Its attributes have the following meaning:

version The protocol version.

pid The participant id, a sequence of bytes that is unique within the scope of the application. Implementations are required to accept at a minimum ids of at least 1 byte and at most 16 bytes in length.

lease This is the period — expressed in hundreds of milliseconds — for which the accepting party should assume the application is alive, even if no traffic has been received. A lease period of 0 (zero), represents an infinite lease. There is no specified upper bound to a valid lease duration, but implementations are allowed to impose an arbitrarily chosen bound for a finite lease. If the specified lease duration is larger than the upper bound for the finite lease, the session shall be rejected with the appropriate error code (cf. the Close message)

locators The list of locators at which this node can be reached, in addition to the source address of the Open message.

Accept Message

The Accept message has the following structure:

```
Accept = record {
  openPid: PeerID,
```
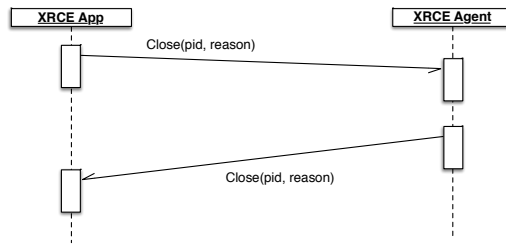
Figure 7: Session termination

```
  acceptPid: PeerID,
  lease: nat,
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|   4     | (AcceptId)
+-+-+-+---------+
~     openPid     ~
+---------------+
~    acceptPid    ~
+---------------+
~      lease      ~
+---------------+
~   properties   ~ if P
+---------------+
```

Its attributes have the following meaning:

openPid The id of the participant that sent the open message.

acceptPid The id of the participant that is accepting the session.

lease See the lease field of the Open message.

Session Termination

The protocol exchange for gracefully closing a session between an Zenoh application and a broker is described in the sequence diagram described in Figure 7. As represented in Figure 7, to close a session an Zenoh application (broker) sends a Close message to the broker (application) which in turn, when ready to close the session, responds with a Close. Notice that implementations that decide not to wait for the matching close may end up losing data. A session is implicitly closed after the expiration of the lease period.

Close Message

The Close message has the following structure:

```
Close = record {
  pid: PeerID,
  reason: byte
}
```

```
+-+-+-+---------+
|X|X|X|    5    | (CloseId)
+-+-+-+---------+
~       pid     ~
+---------------+
|     reason    |
+---------------+
```

Its attributes have the following meaning:

pid The pid of the peer closing the session (or rejecting the opening of a session).

reason The reason why the session is being closed/rejected. Predefined codes for Reason are:

reason	meaning
0	success
1	invalid authentication data
2	unsupported protocol (version)
3	out of resources
4	unsupported sequence number length
5	unsupported parameter (e.g., lease, locators, peer id, subscription mode)
6	incompatible pre-committed declaration
255	generic error

Error codes in the range $[64, 254]$ are reserved for vendor-specific error codes.[2]

Session Liveliness

The session is kept alive as far as there is at least a protocol message sent per lease period. If the lease expires without the reception of any message, the session loses its liveliness. The timed state machine of Figure 8 describes the operations of the liveliness protocol.

As indicated by the liveliness state machine any protocol message renews the lease, that said it is important to remark that when two entities communicate there are two sides of the session that need to be maintained.

KeepAlive Message

A session lease can be renewed explicitly by sending a KeepAlive message defined as follows:

---

[2]Note that on a two's complement machine, this means any codes less than -1 when the octet is interpreted as a signed 8-bit number are vendor-specific — as are any negative ones on a ones' complement machine.
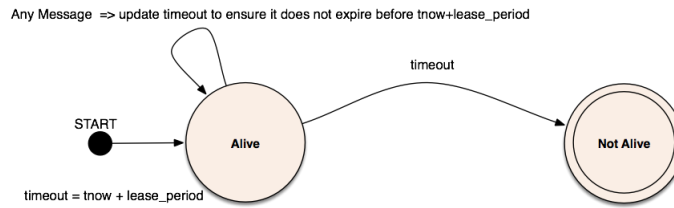
Figure 8: Liveliness state machine

```
KeepAlive = record {
  pid: PeerID
}
```

```
+-+-+-+---------+
|X|X|X|   16    | (KeepAliveId)
+-+-+-+---------+
~       pid       ~
+---------------+
```

Its attributes have the following meaning:

pid The pid of the peer wants to renew the lease.

Session Endpoints

Now that we have specified how a session can be established and torn-down, we need define how the endpoints of a session are identified and what is associated with a session. From a broker perspective an application session is represented by the tuple (pid, locators), where the pid is the unique identifier of the participant while the locators are the set of locators received from participant pid in the hello and/or open messages, augmented with the source address of the latest received message from that participant that contained its pid. In essence, a client application can open a session across multiple transports and networks (such as TCP/IP, UDP/IP) but use all of those transport session to carry on in the most optimal manner communication between the client and the broker.

Notice that the locators uniquely identify a participant/client session. That means that multiple participant/client sessions cannot be multiplexed on the same connection. This is not a limitation since a client session is associated with a Domain Participant, but all the reader/writer traffic associated with a domain participant will be multiplexed across the transport connection belonging to the session. Finally, it is worth pointing out that typically DDS applications have a single domain participant per domain, thus constraining a session to a participant is not introducing any limitation.

Entity Declaration

Once a session has been established, entities such as resources, publisher and subscribers can be declared. To maintain entity declaration atomic while ensuring that messages can be kept small, Zenoh has a concept of commit for

19

declarations. In other terms, declarations are taken into account only after a commit is received. The result of the declaration is provided back to inform the sender of the declarations whether the declarations have been accepted or not.

Resource and Selection Identifiers

Throughout the specification, numerical resource and selection identifiers are used. Resource identifiers are positive even numbers and selection identifiers are positive odd numbers. There is no specified upper bound for resource or selection identifiers, but it is recommended that general, interoperable implementations provide support for identifiers in $1 \dots 2^{63} - 1$.

Whenever an identifier is required in a declaration but the identifier does not have to correct form or is out of bounds for the implementation, the declaration shall be rejected.

Declare Message

The message used to declare entities has the following format:

```
Declare = record {
  sync: S,
  committed: C,
  sn: nat,
  declarations: [Declaration]
}
```

```
+-+-+-+---------+
|X|C|S|   6     | (DeclareId)
+-+-+-+---------+
~       sn      ~
+---------------+
~ declarations  ~
+---------------+
```

Its attributes have the following meaning:

sync iff set, asks for receiving an ack immediately.

committed iff set indicates that the contained declarations have already been committed to by the sender and are not part of a transaction, and that the recipient shall close the session if it would otherwise reject some of the contained declarations.

sn The sequence number of this message

declarations A sequence of declarations (see following subsections).

The Declare message is always sent over the reliable channel.

Declarations

As we saw above, a Declare message contains a sequence of declarations. Similar to messages, the wire format of each declaration begins with a header of one octet encoding the kind as well as a number of flags. Again, like in messages, undefined flags shall be set to 0 in outgoing declarations, and shall be ignored in received declarations.

```
Declaration = record {
  kind: header 5 DeclarationId,
  declaration: case kind of {
    ResourceDeclId          -> ResourceDecl,
    PublisherDeclId         -> PublisherDecl,
    SubscriberDeclId        -> SubscriberDecl,
    SelectionDeclId         -> SelectionDecl,
    BindingDeclId           -> BindingDecl,
    CommitDeclId            -> CommitDecl,
    ResultDeclId            -> ResultDecl,
    ForgetResourceDeclId    -> ForgetResourceDecl,
    ForgetPublisherDeclId   -> ForgetPublisherDecl,
    ForgetSubscriberDeclId  -> ForgetSubscriberDecl,
    ForgetSelectionDeclId   -> ForgetSelectionDecl
  }
}

DeclarationId = enum {
  ResourceDeclId        = 1,
  PublisherDeclId       = 2,
  SubscriberDeclId      = 3,
  SelectionDeclId       = 4,
  BindingDeclId         = 5,
  CommitDeclId          = 6,
  ResultDeclId          = 7,
  ForgetResourceDeclId  = 8,
  ForgetPublisherDeclId = 9,
  ForgetSubscriberDeclId = 10,
  ForgetSelectionDeclId = 11
}
```

Resource Declaration. The resource declaration corresponds to the ResourceDeclId clause in the Declaration type defined above.

```
ResourceDecl = record {
  rid: nat,
  resource: string,
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|    1    | (ResourceDeclId)
+-+-+-+---------+
```

```
~      rid      ~
+--------------+
~    resource   ~
+--------------+
~  properties  ~ if P
+--------------+
```

The attributes of this declaration have the following meaning:

rid the numerical identifier of the resource.

resource the URI defining the resource, notice that this is one of the few messages in which the name of the resource appears, almost everywhere else the resource id is used to save space and allow to bound the message size.

properties optional properties associated with the resource; the P flag shall be set iff properties are present in the message.

Notice that resources are immutable, that means once declared it cannot be mutated. Finally resource declaration is idempotent.

Publisher Declaration. The publisher declaration corresponds to the PublisherDeclId clause in the Declaration type defined above.

```
PublisherDecl = record {
  rid: nat,
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|    2    | (PublisherDeclId)
+-+-+-+---------+
~      rid      ~
+--------------+
~  properties  ~ if P
+--------------+
```

The attributes of this declaration have the following meaning:

id the numerical identifier of the resource or of the selection. properties: optional properties associated with the publisher; the P flag shall be set iff properties are present in the message.

Subscriber Declaration. The subscriber declaration corresponds to the SubscriberDeclId clause in the Declaration type defined above.

```
SubscriptionModeId = enum {
  PushModeId        = 1,
  PullModeId        = 2,
  PeriodicPushModeId = 3,
```

```
    PeriodicPullModeId = 4
}

SubscriberDecl = record {
  id: nat,
  mode: SubscriptionModeId,
  params: case mode of {
    PeriodicPushModeId, PeriodicPullModeId -> record {
      temporalOrigin: nat, period: nat, duration: nat
    }
  },
  properties: Properties
}
```

```
+-+-+-+---------+
|X|X|P|   3     | (SubscriberDeclId)
+-+-+-+---------+
~       id      ~
+---------------+
~      mode     ~
+---------------+
~temporalOrigin ~ if mode in {PeriodicPushModeId,PeriodicPullModeId}
+---------------+
~     period    ~ if mode in {PeriodicPushModeId,PeriodicPullModeId}
+---------------+
~    duration   ~ if mode in {PeriodicPushModeId,PeriodicPullModeId}
+---------------+
~   properties  ~ if P
+---------------+
```

The attributes of this declaration have the following meaning:

id the numerical identifier of the resource or of the selection.

mode indicates the mode in which this subscriber wishes to receive data. Supported modes are:

> **PushModeId** updates shall be pushed to this subscriber;
>
> **PullModeId** data shall be stored until the subscriber pulls it (using PULL messages);
>
> **PeriodicPushModeId** updates shall be stored and pushed periodically, with the push occurring in the intervals [temporalOrigin + n * period, temporalOrigin + n * period + duration), n = 0, 1, 2, ...
>
> **PeriodicPullModeId** updates shall be stored until the subscriber pulls it, where the pulling occurs in the intervals [temporalOrigin + n * period, temporalOrigin + n * period + duration), n = 0, 1, 2, ... The temporalOrigin, period and duration fields are in milliseconds. Implementations shall support at least a period and duration of 214ms, and duration shall be less than period.

properties optional properties associated with the publisher; the P flag shall be set iff properties are present in the message.

Selection Declaration. The Selection declaration corresponds to the SelectionDeclId clause in the Declaration type defined above.

```
SelectionDecl = record {
  global: G,
  sid: nat,
  query: string,
  properties: Properties
}
```

```
+-+-+-+---------+
|G|X|P|   4     | (SelectionDeclId)
+-+-+-+---------+
~       sid     ~
+--------------+
~      query    ~
+--------------+
~   properties  ~ if P
+--------------+
```

The attributes of this declaration have the following meaning:

sid the numerical identifier of the selection. This identifier can be locally generated and specific to a runtime or global. As discussed next, a binding mechanism is provided to agree a common selection id between different nodes.

query the string representing the predicate associated with this selection.

properties optional properties associated with the publisher.

global when true indicates that sid is a global selection id.

Binding Declaration. The Binding declaration is used to change the id associated with a selection, this is typically done with a selection has associated multiple local id and the broker want to be able to multicast data belonging to the selection to the interested parties.

```
  BindingDecl = record {
  global: G,
  oldId: nat,
  newId: nat
}
```

```
+-+-+-+---------+
|G|X|X|   5     | (BindingDeclId)
+-+-+-+---------+
~      oldId    ~
+--------------+
~      newId    ~
+--------------+
```

The attributes of this declaration have the following meaning:

oldId the old id of the selection.

newId id for the selection to be bound/aliased with the old id.

global when true indicates that newId is a global selection identifier.

Commit Declaration. The Commit declaration is used to indicate that the receiving party should process all the received declaration as an atomic unit. In other terms, received declarations are not evaluated until a commit is received and then they either all succeed, or none of them succeed.

```
CommitDecl = record {
  commitId: byte
}
```

```
+-+-+-+---------+
|X|X|X|    6    | (CommitDeclId)
+-+-+-+---------+
|    commitId   |
+---------------+
```

The attributes of this declaration have the following meaning:

id the identifier associated with this transaction.

Whenever a commit declaration is included in a sequence of declarations, it shall be the last.

Result Declaration. The Result declaration is used to communicate the result of applying a lot of declarations.

```
ResultDecl = record {
  commitId: byte,
  status: byte range 0 ... 254,
  id: case status of { 1 ... 254 -> nat }
}
```

```
+-+-+-+---------+
|X|X|X|    7    | (ResultDeclId)
+-+-+-+---------+
|    commitId   |
+---------------+
|     status    |
+---------------+
~       id       ~ if 1 <= status <= 254
+---------------+
```

The attributes of this declaration have the following meaning:

commitId the identifier of the commit for which this a result.

status the result of the declaration, where zero represent a success while a non-zero value represents an error.

id the optional id of a resource or selection included in the current transaction whose declaration failed.

ForgetResource Declaration. The forget resource declaration informs the receiver that, as far as the sending node is concerned, the specified resource with the specified id may be removed from the system.

```
ForgetResourceDecl = record {
  rid: nat
}
```

```
+-+-+-+---------+
|X|X|X|    8    | (ForgetResourceDeclId)
+-+-+-+---------+
~       rid     ~
+---------------+
```

The attributes of this declaration have the following meaning:

rid the numerical identifier of the resource.

ForgetPublisher Declaration. The forget publisher declaration informs the receiver that the sender will no longer be publishing for the specified resource or selection id.

```
ForgetPublisherDecl = record {
  id: nat
}
```

```
+-+-+-+---------+
|X|X|X|    9    | (ForgetPublisherDeclId)
+-+-+-+---------+
~       id      ~
+---------------+
```

The attributes of this declaration have the following meaning:

id the numerical identifier of the resource or selection.

ForgetSubscriber Declaration. The forget subscriber declaration informs the receiver that the sender is no longer subscribed to the specified resource or selection.

```
ForgetSubscriberDecl = record {
  id: nat
}
```
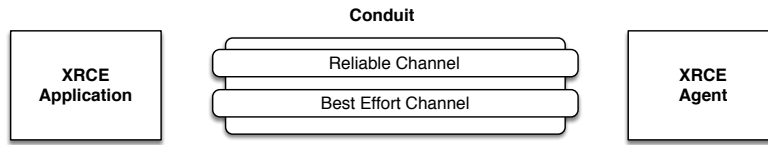
Figure 9: Logical communication channels between the Zenoh application and the broker

```
+-+-+-+---------+
|X|X|X|   10    | (ForgetSubscriberDeclId)
+-+-+-+---------+
~       id      ~
+--------------+
```

The attributes of this declaration have the following meaning:

id the numerical identifier of the resource or selection.

ForgetSelection Declaration. The forget selection declaration informs the receiver that the sender will no longer be using the specified selection.

```
ForgetSelectionDecl = record {
  sid: nat
}
```

```
+-+-+-+---------+
|X|X|X|   11    | (ForgetSelectionDeclId)
+-+-+-+---------+
~       sid     ~
+--------------+
```

The attributes of this declaration have the following meaning:

sid the numerical identifier of the selection.

3.7 Data Exchange

As a result of a session establishment between two Zenoh runtimes, a conduit composed by two logical channels, is established, as depicted in Figure 9. The two conduits' channels may be implemented over one or more transports, for instance the best effort channel could be implemented over UDP/IP while the reliable channel over TCP/IP. However, the mechanism provided to implement an Zenoh reliable channel doesn't require a reliable transport. Each conduit has a numerical identifier, starting from zero for the default conduit. Each of the channels belonging to the conduit has its own sequence number.

Message exchanges that take place before successfully establishing a session are best effort. Once the session is established, messages that can be either best-effort or reliable use a header flag R to discern between best-effort R=0 and reliable R=1.

Conduits

As explained above, as soon as a session is established the default conduit is available along with its two channels, one reliable the other best effort. The conduit concept allows creating another pair of channels with their own sequence numbers that can be used to either unicast or multicast data. The conduit associated with a message is identified by a message marker. This message marker has the following structure:

```
Conduit = record {
  compact: Z,
  id: case compact of {
    false -> nat,
    true  -> record { id1: H, id0: L }
  }
}
```

```
+-+-+-+---------+
|Z|H|L|   19    | (ConduitId)
+-+-+-+---------+
~       id        ~ if not Z
+---------------+
```

The id attribute represents the conduit identifier.

If "compact" is true, that the compact format is used and in this case (as specified by the Conduit message format above), there is no id following the header, but rather the H and L flags are used to represent a conduit id: $1+l+2h$ where $l = 0$ if "id0" is false and 1 if it is true (analogously for h and "id1"). The compact representation is an optimisation and implementations are free to always set "compact" to false and include the conduit id as a separate value. This takes just an additional byte — but when you have an MTU of 20 bytes even a single byte matters!

The highest supported conduit id is implementation-defined, and messages received on a conduit with an unsupported conduit id shall result in closing the session.

As a final remark, recall that the conduit marker is prepended to messages that are sent over a conduit whenever the target conduit is different from the default, conduit id 0. Message markers, as the name suggest, are sent as if they were a decoration of the message, thus a message and its markers are always sent as a unit.

The migration of a resource across conduits is supported through the Migrate message defined as follows:

```
Migrate = record {
  ocid: nat,
  id: if I then nat,
  rchLastSN: nat,
  bechLastSN: nat
}
```

```
+-+-+-+---------+
|X|X|I|   20    | (MigrateId)
+-+-+-+---------+
~      ocid     ~
+---------------+
~       id      ~ if I
+---------------+
~    rchLastSN  ~
+---------------+
~   bechLastSN  ~
+---------------+
```

where the *ocid* attribute is the id of the conduit from which traffic is being migrated to the conduit on which the Migrate message is received. The Migrate message shall always be sent sent on the reliable channel.

If the I flag is clear, all traffic from the conduit *ocid* is migrated and the conduit is considered closed. If the I flag is set, only the resource or selection identified by *id* is being migrated and the conduit remains open.

Best-Effort Channel

Channel Specification

The Best-Effort Channel supports the following primitives:

send Channel × Msg → Channel

receive Channel → Msg The semantic of the channel operations are as follows:
send: given a sequence of messages $m_0, m_1, \ldots, m_k$, sent over the channel, using the channel primitive send, the channel shall deliver an ordered subsequence: $m_{s_0}, m_{s_1}, \ldots, m_{s_h}$ where:

$$\begin{cases} s_0 < s_i < \ldots < s_h \\ s_i \in \{x \in \mathbb{N}_0 : 0 \le x \le k\} \end{cases}$$

In other terms, the Zenoh best-effort channel may drop messages but always deliver messages in order. To maintain ordering the channel relies on a sequence number. The sequence number sn is initialized to zero at the creation of the channel and is incremented by one for every message sent on the channel.

receive returns a message, previously sent on the other end of this channel, if available. Otherwise it returns nothing.

Channel Implementation

The best-effort channel implementation is relatively straight forward. The Zenoh runtime has to make a reasonable effort to send the message over the associated transport and ensure that messages, whilst may be dropped, shall never be delivered to receive out of order.

Reliable Channel

Channel Specification

The Reliable Channel supports the following primitives:

rsend Channel × Msg → Channel

rreceive Channel → Msg

The semantic of the channel operations are as follows:

open creates a new Reliable Channel;

close closes the channel, in other terms no messages can be sent anymore over it;

send given a sequence of messages $m_0, m_1, \ldots, m_k$ sent over the channel using the rsend primitive, the channel will deliver (on the other end) exactly the sequence:

$$m_0, m_1, \ldots, m_k$$

In other terms, the Zenoh reliable channel shall not drop messages nor deliver them out of order. This behaviour has to be guaranteed only under the assumption that the communicating parties are correct, i.e., don't fail. To maintain ordering the channel relies on a sequence number. The sequence number sn is initialized to zero at the creation of the channel and is incremented by one for every message sent on the channel.

receive returns a message, previously sent on the other end of this channel, if available. Otherwise it returns nothing.

Channel Implementation

The Petri Net in Figure 10 describes the behaviour of a Reliable Channel with a buffer of n places. It is worth to mention that the specification in Figure 10 assumes that a NACK can only be received for the head of the queue. This may seem a restriction, but in reality it only means that when receiving a NACK for message with sequence number k the previous messages have been acknowledged.

Synch Message

Synch messages are sent on a newly created reliable channel to set the initial sequence number. If a synch message is not received, then zero should be considered as the first sequence number for the reliable channel. Synch message are sent when needed to inform the other end of the channel of the sequence number of the latest message sent. This allows the other end to detect message loss without necessarily waiting for another regular message. The structure of the synch message is:

```
Synch = record {
  reliable: R,
  sync: S,
```
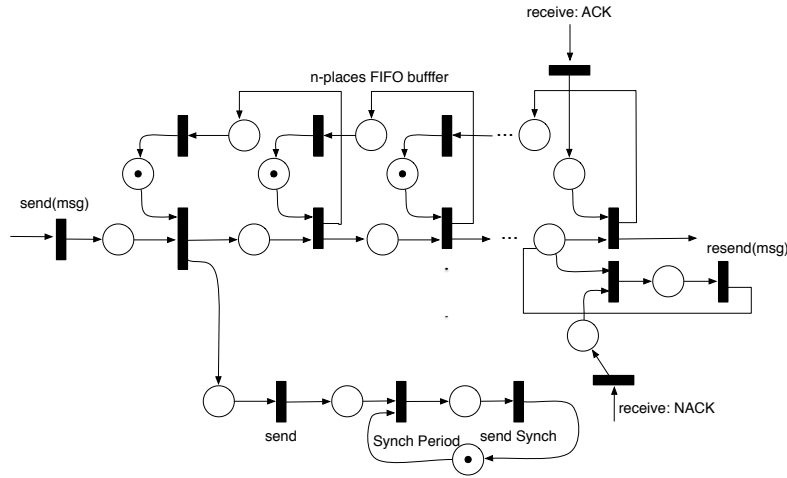
Figure 10: Petri Net describing the Reliable Channel behaviour

```
  sn: nat,
  count: if U then nat
}
```

```
+-+-+-+---------+
|U|R|S|   14    | (SynchId)
+-+-+-+---------+
~       sn       ~
+---------------+
~      count     ~ if U
+---------------+
```

Where the message attributes have the following meaning:

reliable set if this concerns the reliable channel, clear if it concerns the unreliable channel.

sync if set, the message shall be acknowledged promptly.

sn the sequence number of the next message to be transmitted on this channel

count optional number of unacknowledged messages, that is, messages with sequence numbers $s_{n-\text{count}}, s_{n-\text{count}+1}, \ldots, s_{n-1}$ are available for retransmission. If "count" is absent, there are none. Uflag $\Leftrightarrow 0 < \text{count} < 2^{\text{snsize}-1} - 1 \wedge \text{reliable}$.

Positive and Negative Acknowledgements

A single message is used to indicate both positive as well as potentially negative acknowledgements. A node may ignore a negative acknowledgement for a message a finite number of times, but eventually, receipt of a negative acknowledgement shall result in the retransmitting of at least the message with sequence number sn, the oldest message for which retransmit is requested.

The structure of the message is the following:

```
AckNack = record {
  sn: nat,
  mask: if M then nat
}
```

```
+-+-+-+---------+
|X|X|M|   15    | (AckNackId)
+-+-+-+---------+
~       sn      ~
+---------------+
~      mask     ~ if M
+---------------+
```

Where the attributes have the following meaning:

sn the first sequence number not received; all preceding messages are acknowledged.

mask if present, retransmission is requested for the messages with sequence numbers $\text{sn} + i$ s.t. $\lfloor 2^{-i}m \rfloor - 2\lfloor \lfloor 2^{-i}m \rfloor /2 \rfloor \neq 0$, where $m = 2\text{mask} + 1$.

The synch, positive and negative acknowledgements described in Figure 10 shall be implemented by the messages specified before.

Writing Data

After a publisher has been successfully created and the declaration of a matching data reader has been received, an Zenoh implementation shall start to send data to the matching entity(s). To avoid any ambiguity, an Zenoh runtime shall avoid to sends data on a write for writers that are not matched. The Zenoh protocol provides different ways of ways of writing data, one which maps to that provided by DDS.

Write Data

Differently from DDS, Zenoh allows to write samples for resources for which a publisher was not declared and for which only resource name is known (not the resource id). This makes the write a bit less wire efficient because the resource name is part of the data message, but is quite handy for dealing with one-off writes of a resource. These kind of writes are very cumbersome on DDS.

The WriteData message has the following structure:

```
WriteData = record {
  reliable: R,
  sync    : S,
  sn      : nat,
  resource: string,
  payload : [byte]
}
```

```
+-+-+-+---------+
|X|R|S|    9    | (WriteDataId)
+-+-+-+---------+
~       sn      ~
+---------------+
~    resource   ~
+---------------+
~     payload   ~
+---------------+
```

Where the attributes have the following meaning:

reliable set if this concerns the reliable channel, clear if it concerns the unreliable channel.

sync if set, the message shall be acknowledged promptly.

sn the sequence number for this message.

resource the resource for which we are writing data.

payload the serialised data.

To the Zenoh protocol the format of the serialised data is transparent. Thus, when interoperating with DDS it shall be the same as that used by DDS. In stand-alone situations can be whatever the application deems to be the most appropriate.

Stream Data

Zenoh provides a very wire efficient to produce data belonging to a stream, namely the StreamData message. The structure of this message is as follows:

```
StreamData = record {
  reliable: R,
  sync    : S,
  sn      : nat,
  id      : nat,
  prid    : if A then nat,
  payload : [byte]
}
```

```
+-+-+-+---------+
|A|R|S|    7    | (StreamDataId)
+-+-+-+---------+
~       sn      ~
+---------------+
~       id      ~
+---------------+
~      prid     ~ if A
+---------------+
~     payload   ~
+---------------+
```

Where the attributes have the following meaning:

reliable set if this concerns the reliable channel, clear if it concerns the unreliable channel.

sync if set, the message shall be acknowledged promptly.

sn the sequence number for this message.

id the matching resource/selection id for which we are writing data.

prid optionally the id of the resource to which the data belongs — this may be different to the matching resource/selection id for non-trivial resources and selections, or 0 to indicate the end of a response to a PULL request

payload the serialised data.

Notice that the message overhead is at least 4 bytes, three bytes of protocol data and one byte — at minimum — to encode the payload length.

Batch Data

Zenoh supports message batching for the same resource/selection. The Batched-Data message further improves the wire efficiency of the protocol. The structure of this message is as follows:

```
BatchedData = record {
  reliable: R,
  sync    : S,
  sn      : nat,
  id      : nat,
  hasPrid : A,
  payload : case hasPrid of {
    false -> [[byte]],
    true  -> [record { prid: nat, payload: [byte] }]
  }
}
```

```
+-+-+-+---------+
|0|R|S|    8    |  (BatchedDataId)
+-+-+-+---------+
~        sn     ~
+---------------+
~        id     ~
+---------------+
~     payload   ~
+---------------+
+-+-+-+---------+
|1|R|S|    8    |  (BatchedDataId)
+-+-+-+---------+
~        sn     ~
+---------------+
~        id     ~
```

```
+---------------+
~ prid,payload  ~
+---------------+
```

Where the attributes have the following meaning:

reliable set if this concerns the reliable channel, clear if it concerns the unreliable channel.

sync if set, the message shall be acknowledged promptly.

sn the sequence number for this message.

id the matching resource /selection id for which we are writing data

payload the batched data

Notice that the minimal overhead in this case is: $(3 + n)/n$ which gets very quickly close to 1 as the number n of batched message grows beyond a few.

Reading Data

While in regular DDS reads are always local, in Zenoh that is not always possible or desirable due to memory or network constraints. As such Zenoh needs a message to pull data remotely.

Pull Message

Pull is a reliable message with the following structure:

```
Pull = record {
  final: F,
  sync: S,
  sn: nat,
  id: nat,
  maxSamples: if N then nat
}
```

```
+-+-+-+---------+
|F|N|S|   11    | (PullId)
+-+-+-+---------+
~        sn       ~
+---------------+
~        id       ~
+---------------+
~   maxSamples   ~ if N
+---------------+
```

Its attributes have the following meaning:

final if set, the indicates that this pull-session shall be considered concluded after serving this request

sync if set, the message shall be acknowledged promptly.

sn the message sequence number.

id the resource / selection id for which data has to be pulled.

maxSamples optionally the maximum number of samples that this node is willing to receive before having to issue another pull. Implementations shall at least support $0 < \text{maxSamples} < 2^{16}$.

Notice that the PULL message does not provide explicitly a way of expression the full resource for which data is being pulled. It is assumed that even for a "one-shot" pull, a temporary selection will be created using the Declaration messages.

The puller can send a PULL message with F=1 N=0 to indicate that no data will be requested for the current pull-id. XXX

The Forget message may be used to indicate that the associated SID may be reused for other selections.

The end of the sequence of data samples associated the pull session associated with the SID can be indicated by sending a DATA message with PRID=0 and A=1. In this case the payload shall be ignored, which suggest that implementations should send a zero-length payload. Notice that as a consequence the cursor associated with this selection session will be reset.

Data Fragmentation

Zenoh supports fragmentation to accommodate sending user data of any size. This is done by a message marker called Frag. As in the case of the Conduit message marker, these messages have to be sent in the same packet as the message they mark. Fragmentation shall not be used for messages that fit in one MTU, and whenever data is fragmented into $n > 1$ fragments, the first $n-1$ fragments shall have the same size and the last fragment shall be no larger than the preceding fragments. The structure of the Frag message is defined below:

```
Frag = record {
  snBase: nat,
  n: if N then nat
}
```

```
+-+-+-+---------+
|X|N|X|   18    | (FragId)
+-+-+-+---------+
~     snBase    ~
+---------------+
~       n       ~ if N
+---------------+
```
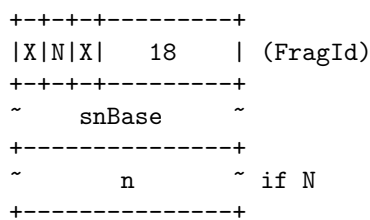
Where the attributes have the following meaning:

snBase the sequence number of the first message in this series of fragment.

n if present tells the total number of fragments. Otherwise it can be deduced by looking at the first in order message that does not have the fragment marker.

Resource Spaces

For multi-tenancy as well as to allow brokers to freely chose RID without
conflicting with user defined resources we can define different resource spaces.

The message is defined as:

```
RSpace = record {
  compact: Z,
  id: case compact of {
    false -> nat,
    true  -> record { id1: H, id0: L }
  }
}
```

```
+-+-+-+---------+
|Z|H|L|   24    | (RSpaceId)
+-+-+-+---------+
~       id       ~ if not Z
+--------------+
```

XXX encoding of resource space is the same is in Conduit — should perhaps
define a "compact id" type for that

Resource space 0 is the default one. The Resource space 1 is reserved for
Brokers.

Roundtrip-Time Estimation

Latency estimation is essential in order to do proper reliability window man-
agement. As such the Zenoh protocol provides a couple of messages to estimate
the roundtrip-time between to nodes. These messages are called Ping and Pong
respectively and their structure is shown below.

```
PingPong = record {
  ispong: O,
  hash: nat
}
```

```
+-+-+-+---------+
|X|X|O|   12    | (PingPongId)
+-+-+-+---------+
~      hash      ~
+--------------+
```

On receipt of a PingPong with *ispong* false, a PongPong with *ispong* set to
true and the same hash shall be sent in response. An implementation shall at
least support hashes $< 2^{16}$.

4 Resource Usage

One of the key aspects defining the quality and the fitness of a zenoh for
constrained environments is its wire-overhead. zenoh has a wire overhead for

user data of only 4 bytes. This wire-overhead can additionally be reduced by leveraging batching to $(3+n)/n$ bytes – where n is the number of batched samples.

Additionally, our current implementation of the protocol has proven that:

- The protocol can be implemented for very small targets, such as having a few Kbytes of memory. Notably we have implemented and demonstrated at the Huawei European Connect an implementation of this protocol for an Arduino Uno platform communicating through BLE with MTU of 20 bytes.

- The protocol can be used for both client to broker/agent as well as peer-to-peer communication. The submitters have built implementations of both and are ready to demonstrate to interested parties.

- The implementation of the core protocol fits in about 2500 lines of C code.

- We have worked with a FPGA team to ensure that the protocol can be easily implemented in as an FPGA IP.

5   Conformance

This specification defines the following modules:

Discovery The discovery module allows application to dynamically discover each other and exchange entities declarations.

- SCOUT, HELLO, KEEPALIVE, PING, PONG
- DECLARE: PUB, SUB, COMMIT, RESULT

Query The query profile allows applications to execute queries over resources.

- DECLARE: RESOURCE, SELECTION, BIND

Core The core profile defines the session establishment and the data exchange.

- OPEN, ACCEPT, CLOSE,
- WDATA, SDATA, BDATA
- PULL

Minimally, an implementation to comply with zenoh has to implement the functionalities specified as part of the Core Module. The Query and Discovery module represent additional point of conformance.