zero network over-head.

version 0.1.8

Angelo Corsaro, PhD
Chief Technology Officer
ADLINK Technologies Inc.

Erik Boasson
Senior Technologist
ADLINK Technologies Inc.

“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

Table of Contents

1 Introduction

zenoh's goals 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.

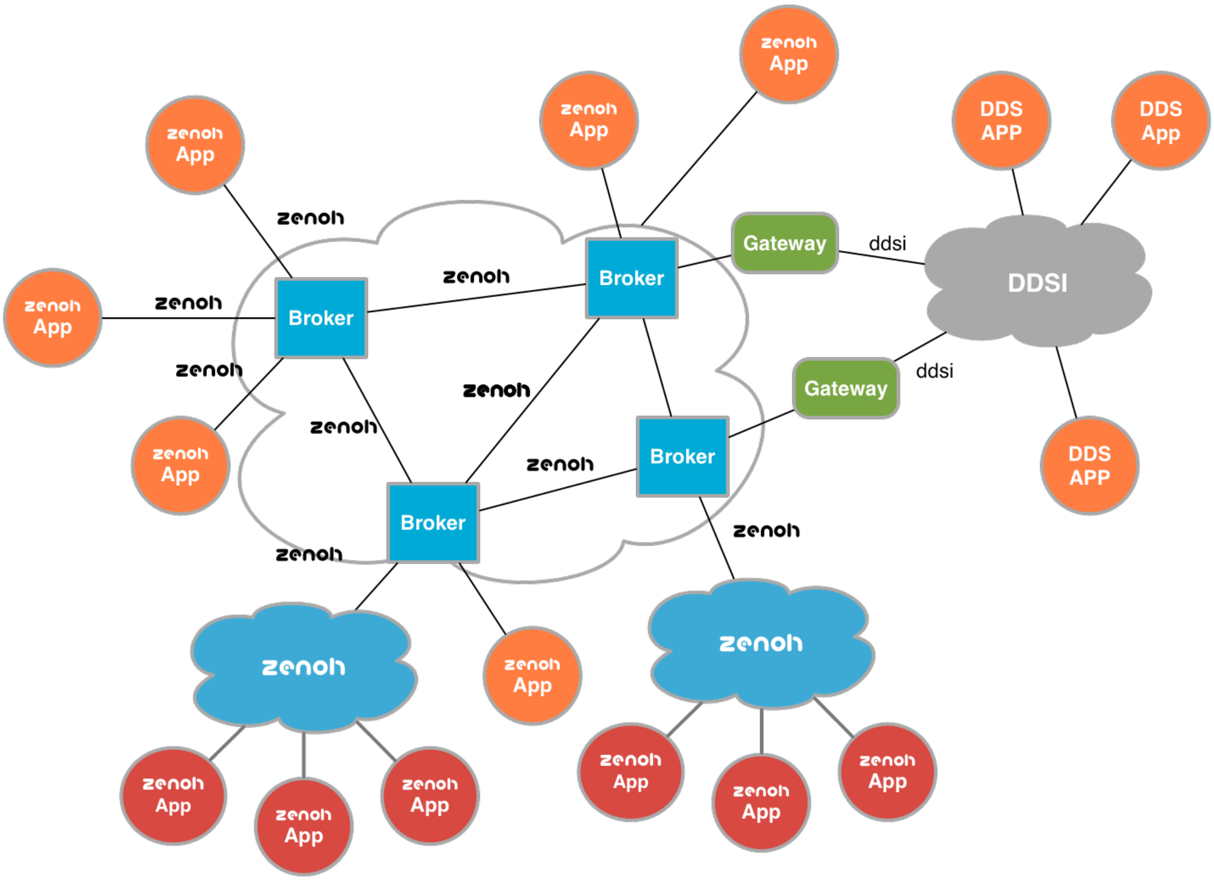


Figure 1 – Typical deployment of a zenoh system, also showing a possible bridging from/to DDS.

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.

[1] The specification for existing Arduino variants is available at https://www.arduino.cc/en/Products/Compare

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.

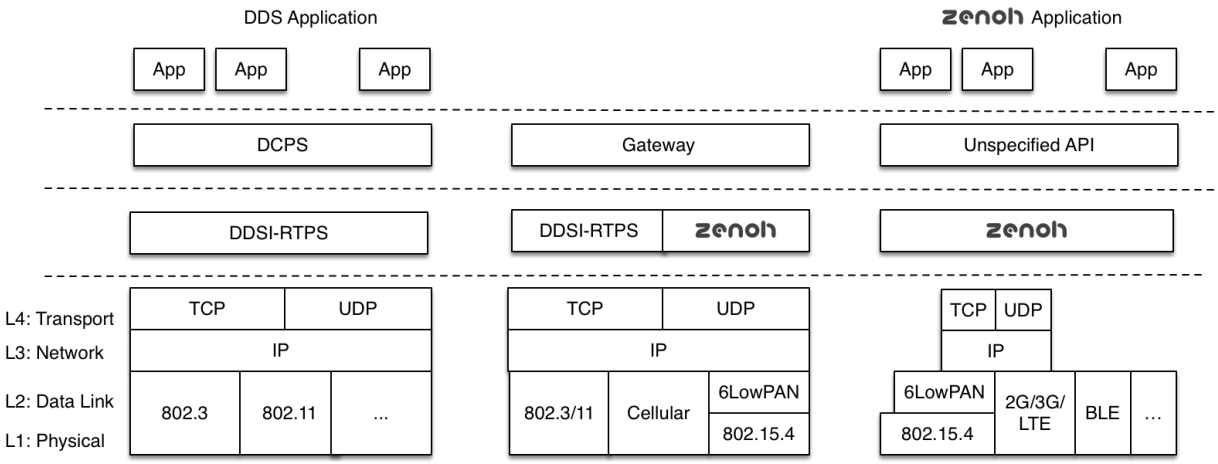


Figure 2 -- Comparison of the traditional DDS and the zenoh stacks.

2 zenoh

This specification defines the **zenoh** protocol which applications can use to communicate with a broker as well as between them, *i.e.*, peer-to-peer. This protocol has also been designed to address the broker-to-broker communication.

2.1 zenoh Abstractions

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 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).

2.1.1 Resources

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

A zenoh **Resource** is a closed description for a set. If the cardinality of the set is one then we call it a **Trivial Resource**. A 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 constitute the set as opposed to the single element set.

Below are some non-normative examples of zenoh resources:

```
-- This are Resources

xrce://myhouse/**/bedroom/*/LightStatus

xrce://myhouse/**/musicroom/LightStatus

xrce://myhouse/**/LightStatus

xrce://myhouse/**


-- This is a Trivial Resource

xrce://myhouse/floor/1/musicroom/LightStatus

xrce://myhouse/floor/2/musicroom/LightStatus

xrce://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 are addressed by including the instance-id in the URI.

The formal syntax for resources is defined below:

$$\boldsymbol{resource} = xrce\text{:}//\boldsymbol{path\ markers}$$

$$\boldsymbol{path} = (\boldsymbol{name} \mid \boldsymbol{wildcard}) \mid (\boldsymbol{name} \mid \boldsymbol{wildcard})/\boldsymbol{path}$$

$$\boldsymbol{wildcard} = * \mid **$$

$$\boldsymbol{markers} = \#|\#\boldsymbol{markerlist}$$

$$\boldsymbol{markerList} = \boldsymbol{name} \mid \{\boldsymbol{nonemptynamelist}\}$$

$$\boldsymbol{nonemptynamelist} = \boldsymbol{name} \mid \boldsymbol{name},\boldsymbol{nonemptynamelist}$$

Where **name** is a valid URI name.

2.1.1.1 Resource Reliability

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

2.1.1.2 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
xrce://myhouse/floor/1/musicroom/LightStatus
xrce://myhouse/floor/1/musicroom/LightStatus/ID12345

--
xrce://myhouse/floor/1/musicroom/LightStatus/property


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


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

-- All Resources that may be either transient or persistent

xrce://**#{transient, persistent}
xrce://**#{transient, persistent}
```

2.1.2 Selections

A **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:

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

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

The canonical representation of a selection shall 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:

xrce://myhouse/floor/2/bedroom/erik/LightStatus#?

xrce://myhouse/floor/2/bedroom/erik/LightStatus

The formal syntax for queries is reported below:

$$\boldsymbol{query} = \boldsymbol{resource}?\boldsymbol{predicate}$$

$$\boldsymbol{predicate} = \in \mid \{\boldsymbol{nontrivialpredicate}\}$$

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 other syntax too, such a subset of JavaScript.

zenoh has entities that serve a role equivalent to that of DDS' Domain Participant, Data Writer and Data Reader. In 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.

2.1.3 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.

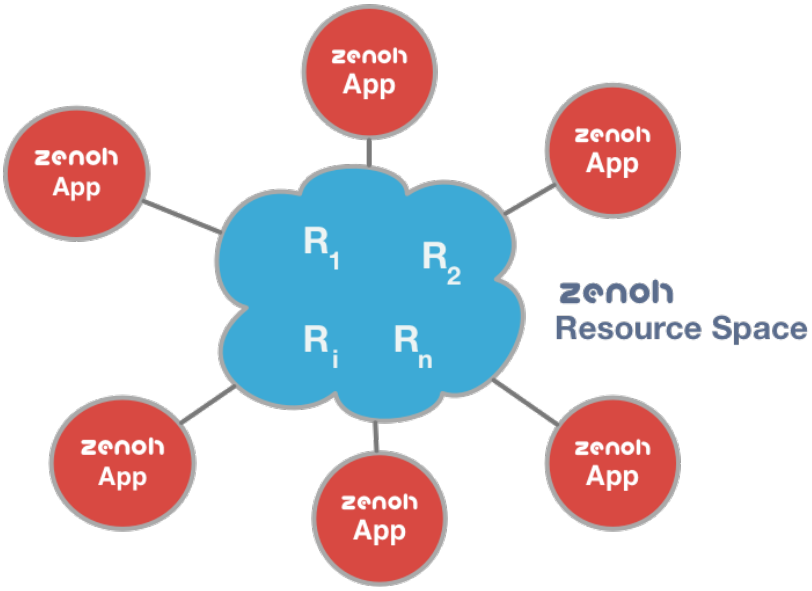


*Figure 3 – **zenoh interaction model.***

The reminder of this specification defines the protocol that makes it possible to implement the zenoh

data space abstraction. This protocol, supports data space implementation that are (1) based on a network of brokers, (2) decentralised / peer-to-peer, (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.

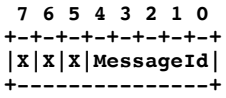
2.2 zenoh Protocol

The zenoh protocol description is decomposed in scouting, session management, entities declaration, and data exchange. Before describing the details of each of these protocol elements, let's define how the protocol data is encoded.

2.2.1 Message Structure, Framing and Encoding

2.2.1.1 Message Structure

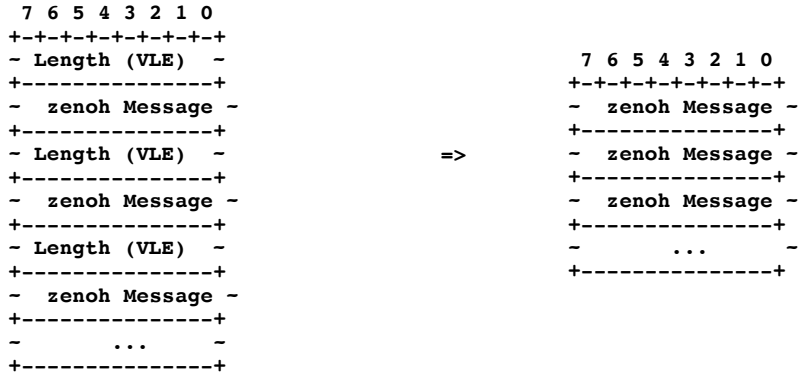
zenoh messages are all composed by a single byte header and a body. The header has always the following structure:



where the least significant 5 bits represent the message Id and consequently define the format of the remainder of the message. The three most significant bits are used as header flags. As different classes of messages have different header flags, we'll describe these in the context of each message. That said, any flag field that is not defined shall always be set to zero.

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.



2.2.2 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`
- `Migrate`
- `RSpace`

2.2.3 Encoding

The zenoh protocol uses variable length encoding. The encoding rules are defined below.

2.2.3.1 Variable Length Encoding (VLE)

The zenoh protocol makes use of Variable Length Encoded (VLE) integral types. The VLE encoding of an integral number **n**, obtained first by decomposing the number into the sequence $\mathbf{a_0}$, $\mathbf{a_1}$, …, $\mathbf{a_k}$ such that:

$$n = \sum_{i=0}^{k} a_i \ll (7 * i)$$

Where $\mathbf{a_i}$ and **n** are given by the following formula:

$$\begin{cases} k = \lceil (log_2 n)/7 \rceil \\ a_i = \left(n \gg (7 * i)\right) \boldsymbol{and}\ 0x7F \end{cases}$$

Then the number is represented on the wire as follows:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
|1|      a0     |
+---------------+
|1|      a1     |
+---------------+
~               ~
+---------------+
|0|      ak     |
+---------------+
```

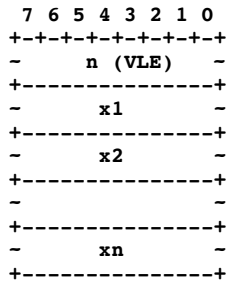
Notice the that most significant bit is used as a guard and is set to 1 whenever there is a reminder.

2.2.3.1.1 Sequence Encoding

A sequence type is an ordered and homogeneous set of elements. Example of sequence types are strings, lists, etc. Sequences encoded as a VLE representation of their length followed by the encoding of each element. Given a sequence **xs**:

$$\boldsymbol{xs} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots \boldsymbol{x_n}$$

Its wire representation is as follow:

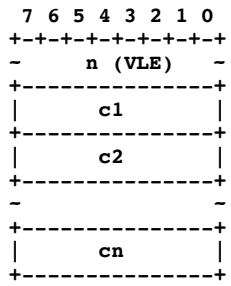


2.2.3.1.2 String Encoding

Strings are a special case of sequences for which the element type is a character. As a consequence, strings are encoded as a VLE representation of their length and the UTF8 array of bytes representing the string. Notice that wire representation should not include the string terminator for programming languages that require it (e.g. `\0` in C/C++). Thus given a string **s:**

$$s = c_1 c_2 \ldots c_n$$

Its wire representation is as follow:



2.2.4 Scouting

zenoh 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.

2.2.4.1 Scout Message

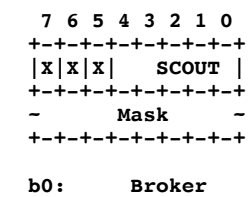
A node may send a `Scout` message to probe responses from specific kind of nodes it is interested in. The `Scout` message has the following structure:

```
struct Scout{
   vle mask;
};
```

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 `Scout` message is represented on the wire as follows:



```
b1:     Durability
b2:     Peer
b3:     Client
b4-b13: Reserved
```

Where the **SCOUT** message identifier is **0x01**.
The Mask is VLE encoded and the meaning of the various bits, once decoded, is described in the message above.

The scout message is best-effort and the default address scouting endpoint for a UDP/IPv4-based transport is **udp/239.255.0.1:7447**.

2.2.4.2 Hello Message

Upon receiving a `Scout` message a node matching the scout mask shall promptly respond with the `Hello` message described below.

```
struct Hello{
   vle mask
   Locators locators;
   Properties properties;
};
```

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

properties: the node properties, this is just a list of name value pairs.

The Properties are present iff P==1. Where the **HELLO** message identifier is **0x02**.

2.2.5 Session Management

Once scouting is completed, or alternatively if scouting information is configured statically, an zenoh node can start establishing 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.

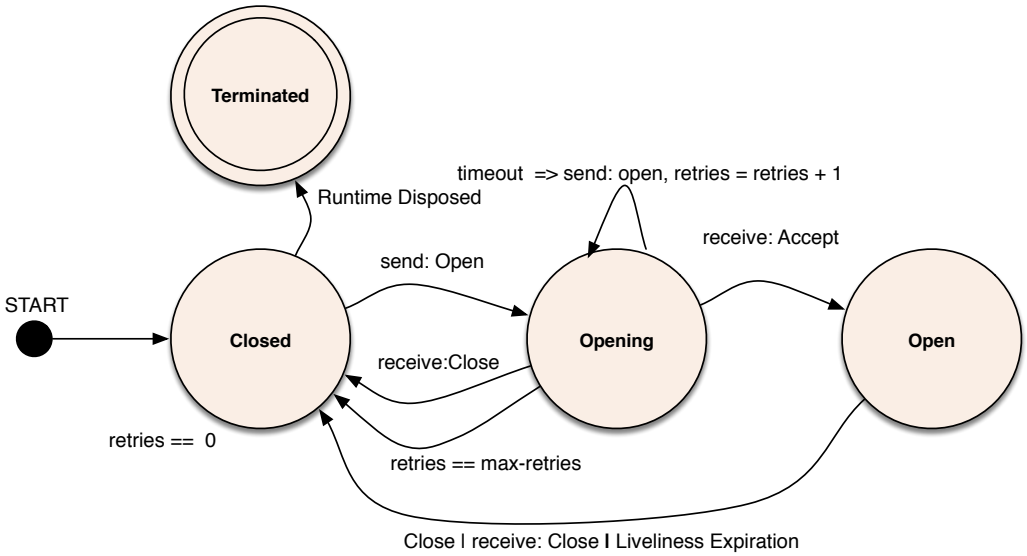


Figure 4 – Session management state machine for the session initiator.

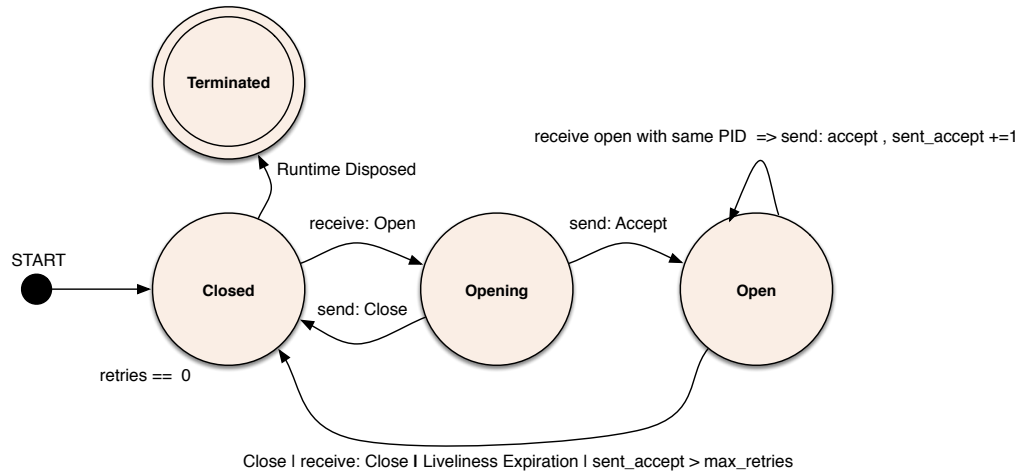


Figure 5 – Session management state machine for the agent.

2.2.5.1 Session Establishment

The protocol exchange for opening a session between a zenoh entities is described in the sequence diagram described in Figure 6. As represented in Figure 6a, to establish a session an zenoh application sends an Open message to the agent which in turn responds with an Accept if the section can be established or with a Close otherwise (see Figure 6b).

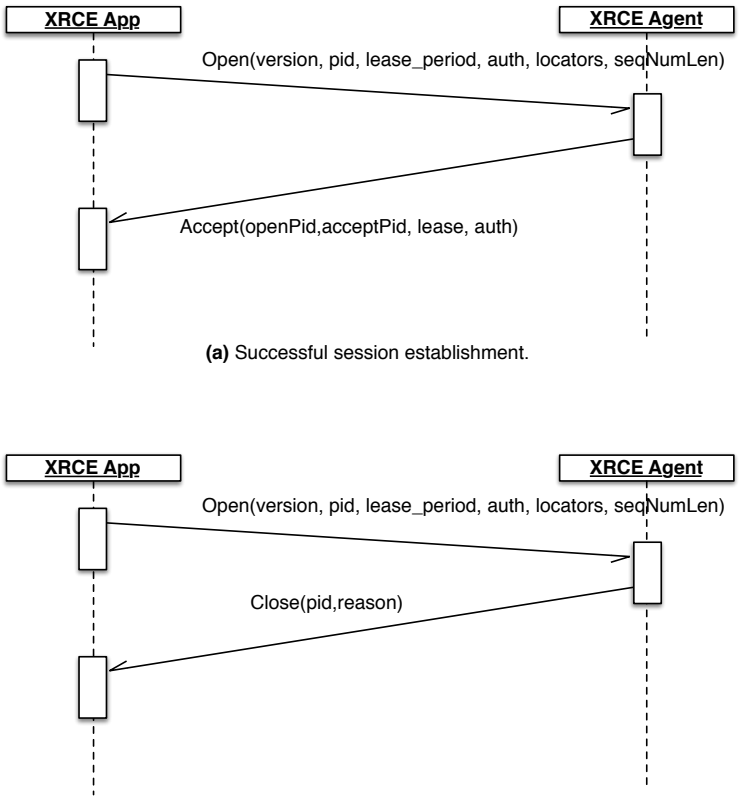


*Figure 6 – **Successful session establishment.***

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 agent.

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**.

2.2.5.2 Open Message

The Open message has the following structure:

```
struct Open {
  u8 version;
  bytes pid;
  vle lease;
  bytes auth;
  Locators locators;
  u8? snsize;
};
```

Its attributes have the following meaning:

version: The protocol version.

pid: The participant id, represented with a number of bytes that assures unicity 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.

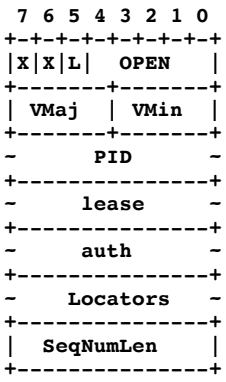
lease: This is the period – expressed in hundreds of milliseconds – for which the accepting party should assume the application is alive, even if no traffic has been received. A lease period of 0 (zero), represents an infinite lease. There is no specified upper bound to a valid lease duration, but implementations 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)

auth: This is a sequence of bytes containing some authentication information, such as a user name and password, etc. The content is not specified and left to the implementation.

locators: Additional locators that can be used to communicate with this participant.

snsize: This optional attribute can be used to negotiate the sequence number length. By the default the protocol uses 14-bit sequence numbers, but these can be extended or reduced by negotiating it. Notice that as sequence number are VLE encoded, this only serves to define the range of sequence number that can be expected from a participant. This is important in order to properly deal with sequence number roll-over. The **L** flag shall be set iff seqNumLen is present.

The Open message shall be represented on the wire as follows:



Where the **OPEN** message identifier is **0x03**.

2.2.5.3 Accept Message

The Accept message has the following structure:

```
struct Accept {
  bytes openPid;
  bytes acceptPid;
  vle lease;
  bytes auth;
};
```

Its attributes have the following meaning:

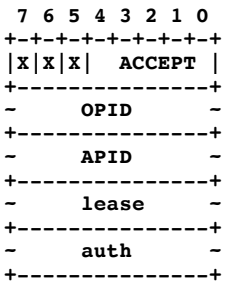
openPid: The id of the participant that sent the open message.

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

lease: see the **lease** field of the Open message .

auth: This is a sequence of bytes containing some authentication information, such as a user name and password, etc. The content is not specified and left to the implementation.

The Accept message shall be represented on the wire as follows:



Where the **ACCEPT** message identifier is **0x04**.

2.2.5.4 Close Message

The Close message has the following structure:

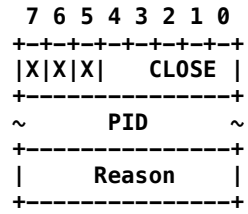
```
struct Close {
  bytes pid;
  u8 reason;
};
```

Its attributes have the following meaning:

pid: The pid corresponding to the entity whose Open message is being rejected.

reason: The error code explaining why the request to open a session was rejected.

The Close message shall be represented on the wire as follows:



Where the **CLOSE** message identifier is **0x05**.

Predefined codes for **Reason** are:

- 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

2.2.5.5 Session Termination

The protocol exchange for gracefully closing a session between a zenoh application and an agent is described in the sequence diagram described in Figure 7 . As represented in Figure 7, to close a session an zenoh application (agent) sends a Close message to the agent (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.

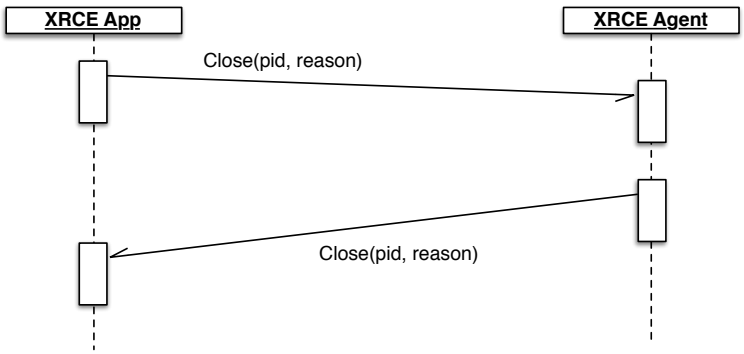


*Figure 7 – **Session termination.***

2.2.5.6 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.

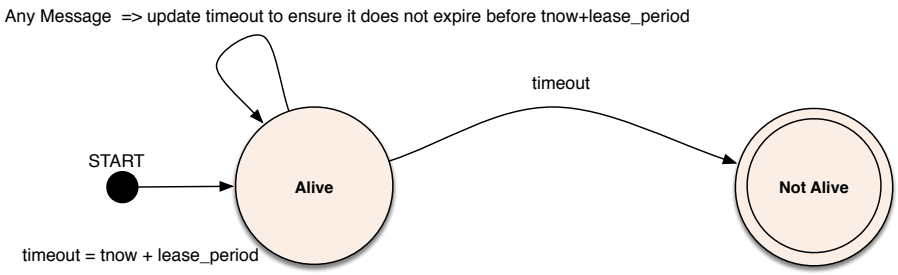


*Figure 8 – **Liveliness state machine.***

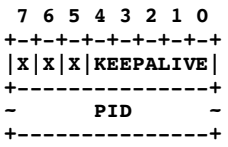
As indicated by the liveliness state machine any protocol message renews the lease, that said it is important to remark that when two entities communicate there are two sides of the session that need to be maintained.

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

```
struct KeepALive {
  bytes pid;
```

```
};
```

Where the **pid** attributes represent the participant identifier of the node that wants to renew the lease. The KeepALive message shall represented on the wire as follows:



Where the **KEEPALIVE** message identifier is **0x10**.

2.2.5.7 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 an agent 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 agent.

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.

2.2.6 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 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.

2.2.6.1 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 … 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.

2.2.6.2 Declare Message

The message used to declare entities has the following format:

```
typedef sequence<Declaration> Declarations

struct Declare {
  bool sync;
```

```
  bool committed;
  vle sn;
  Declarations declarations;
};
```

Its attributes have the following meaning:

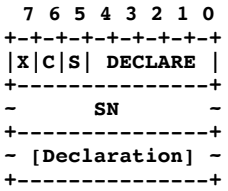
sync: As this message is reliable this attribute controls the synch flag S which when **1** asks for receiving an ack immediately.

committed: This attribute corresponds to the C flag, and when 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 reliable and shall be represented on the wire as follows:



The **DECLARE** message identifier is **0x06.**

2.2.6.3 Declarations

As we saw above, a `Declare` message contains a sequence of declarations. Below we describe the list of supported declarations.

```
 enum SubscriptionModeId {
  PushModeId         =  0x01,
  PullModeId         =  0x02,
  PeriodicPushModeId =  0x03,
  PeriodicPullModeId =  0x04
 };

union SubscriptionMode switch (SubscriptionModeId) {
case PushModeId:
case PushModeId:
case PeriodicPushModeId:
  vle temporalOrigin;
  vle period;
  vle duration;
case PeriodicPullMode:
  vle temporalOrigin;
  vle period;
  vle duration;
};

enum DeclarationId  {
  ResourceDeclId = 0x01,
  PublisherDeclId = 0x02,
  SubscriberDeclId = 0x03,
  SelectionDeclId = 0x04,
  BindingDeclId = 0x05,
  CommitDeclId = 0x06,
  ResultDeclId = 0x07
};
```

```
union Declaration switch (DeclarationId) {
case ResourceDeclId:
  vle rid;
  Resource resource;
  Properties? properties;
  bool forget;

case PublisherDeclId:
  vle id;
  Properties? properties;
  bool forget;

case SubscriberDeclId:
  vle id;
  SubscriptionMode mode;
  Properties? properties;
  bool forget;

case SelectionDeclId:
  vle selId;
  string query;
  Properties? properties;
  bool global;
  bool forget;

case BindingDeclId:
  vle oldId;
  vle newId;
  bool global;

case CommitDeclId:
  u8 id;

case ResultDeclId:
  u8 commitId;
  u8 status;
  vle? id;
};
```

Resource Declaration. The resource declaration corresponds to the `ResourceDeclId` clause in the `Declaration` type defined above. 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.
>
> **forget:** when true indicates that this resource has to be "forgotten" in other terms removed from the system; the **F** flag shall be set iff **forget**.

The resource declaration is represented on the wire as follows:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
|X|F|P|RESOURCE |
+---------------+
~      RID      ~
+---------------+
~    Resource   ~
+---------------+
~  [Property]   ~
+---------------+
```

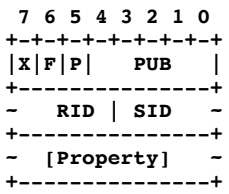
The **RESOURCE** message identifier is **0x01.**

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. 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.
>
> **forget:** when true indicates that this resource has to be "forgotten" in other terms removed from the system; the **F** flag shall be set iff **forget**.

The resource declaration is represented on the wire as follows:

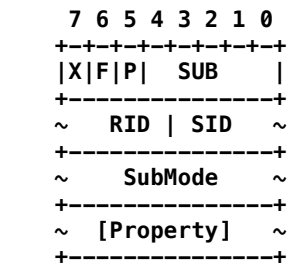


The **PUB** message identifier is **0x02.**

Subscriber Declaration. The subscriber declaration corresponds to the SubscriberDeclId clause in the Declaration type defined above. 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:
>
> - **push:** updates shall be pushed to this subscriber;
> - **pull:** data shall be stored until the subscriber pulls it (using **PULL** messages);
> - **periodic push:** 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, …
> - **periodic pull:** 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 2^{14}ms, 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.
>
> **forget:** when true indicates that this resource has to be "forgotten" in other terms removed from the system; the **F** flag shall be set iff **forget**.

The resource declaration is represented on the wire as follows:

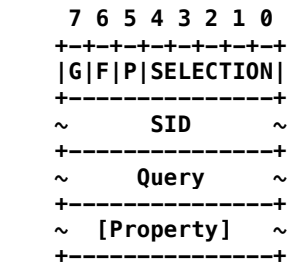


The **SUB** message identifier is **0x03.**

Selection Declaration. The Selection declaration corresponds to the SelectionDeclId clause in the Declaration type defined above. 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; the **P** flag shall be set iff properties are present in the message.
>
> **forget:** when true indicates that this resource has to be "forgotten" in other terms removed from the system; the **F** flag shall be set iff **forget**.
>
> **global:** when true indicates that **sid** is a global selection id; the **G** flag shall be set iff **global**.

The selection declaration is represented on the wire as follows:



The **SELECTION** message identifier is **0x04.**

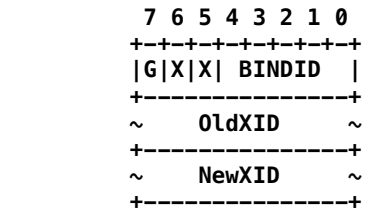
Binding Declaration. The Binding declaration corresponds to the BindingDeclId clause in the Declaration type defined above. This declaration is used to change the id associated with a selection, this is typically done with a selection has associated multiple local id and the agent want to be able to multicast data belonging to the selection to the interested parties.

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; the **G** flag shall be set iff **global**.

The binding declaration is represented on the wire as follows:



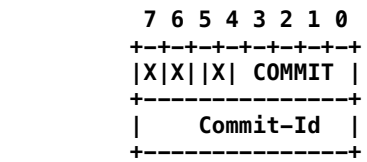
The **BINDID** message identifier is **0x05.**

Commit Declaration. The Selection declaration corresponds to the CommitDeclId clause in the Declaration type defined above. The commit declaration is used to indicate that the receiving party should process all the received declaration as an atomic lot. In other terms, received declarations are not evaluated until a commit is received and they succeed as a lot or fail as a lot.

The attributes of this declaration have the following meaning:

id: the identifier associated with this transaction.

The commit declaration is represented on the wire as follows:



The **COMMIT** message identifier is **0x06.**

Result Declaration. The Selection declaration corresponds to the ResultDeclId clause in the Declaration type defined above. The result declaration is used to communicate the result of applying a lot of declarations.

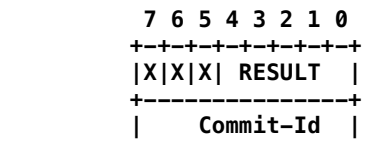
The attributes of this declaration have the following meaning:

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. The **id** field shall be omitted iff **status** = 0.

The commit declaration is represented on the wire as follows:



```
+---------------+
|    STATUS     |
+---------------+
~    RID | SID  ~
+---------------+
```

The **RESULT** message identifier is **0x07.**

2.2.7 Data Exchange

As a result of a session establishment between two zenoh runtimes, a conduit, composed by two logical channels, is established. The two conduit's 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 a 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.

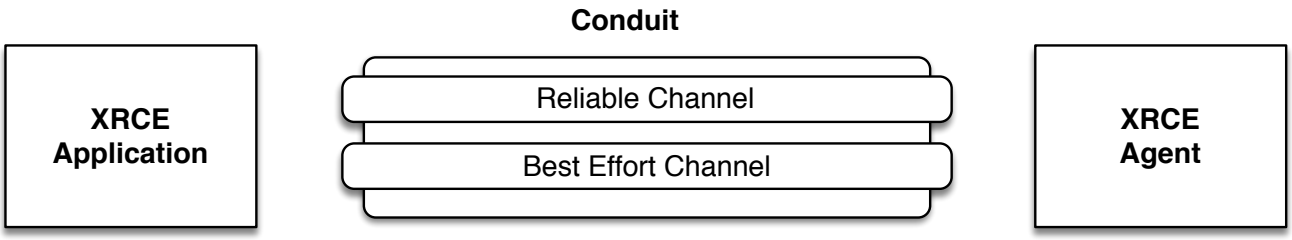


Figure 9 – ***Logical communication channels between the zenoh application and the agent.***

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.

2.2.7.1 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 to create another pair of channels with their own sequence number that can be used to either unicast or multicast data. The conduit associated with a message is identified by a message marker. The message marker has the following structure:

```
struct Conduit {
   vle id;
};
```

The **id** attribute represents the conduit identifier. The highest supported conduit id is implementation-defined, and messages received on a conduit with an unsupported conduit id shall be ignored.

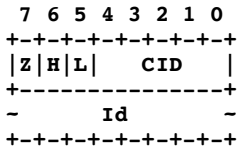
The conduit marker is represented on the wire as follows:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
|Z|H|L|   CID   |
+---------------+
~      Id       ~
+---------------+
|X|X|X| MessageId|
+----------------+
~  Message Body  ~
+---------------+
```

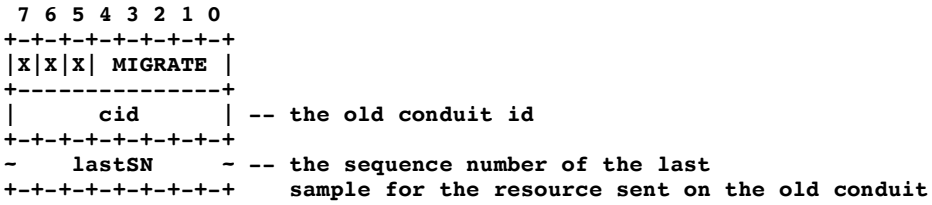
Where **CID** shall have the value **0x13**. Z=1 indicates that the compressed format is used, in this case the **Id** is not present and the **HL** flags represent the conduit id diminished by one. In other terms HL=00 represent the conduit 1, HL=01 represent the conduit 2 and so on. This allows representing four conduit Ids with a single byte. The compact representation is an optimisation and implementations are free to always set Z=0 and express 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!

As a final remark, recall that the conduit marker is added 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 constitute a whole.

In other terms, on the wire a marker will always be prepended to the message it targets, thus in the case of conduit, the marked message shall be framed as follows:



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



Where **MIGRATE** shall have the value **0x14**

.

2.2.7.2 Best-Effort Channel (BC)

2.2.7.2.1 Channel Specification

The Best-Effort Channel (BC) supports the following primitives:

$$send\colon Channel \times Msg \rightarrow Channel$$
$$receive\colon Channel \rightarrow Msg$$

The semantic of the channel operations are as follows:

> **send:** given a sequence of messages m_0, m_1, … 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:
>
> $$\begin{cases} s_o < s_i < \ldots < s_h \\ s_i \in \{x \in \mathbb{N}_0 \colon 0 \leq x \leq k\} \end{cases}$$

In other terms, the zenoh BC 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.

2.2.7.2.2 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.

2.2.7.3 Reliable Channel (RC)

2.2.7.3.1 Channel Specification

The Reliable Channel (RC) supports the following primitives:

$$r\_send: Channel \times Msg \rightarrow Channel$$
$$r\_receive: Channel \rightarrow Msg$$

The semantic of the channel operations are as follows:

`open`: creates a new Reliable Channel (RC);

`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, … m_k, sent over the channel using the **r_send** primitive, the channel will deliver (on the other end) exactly the sequence:

$$m_0, m_1, \dots, m_k$$

In other terms, the zenoh RC shall not drop messages nor deliver them out of order. This semantics 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.

2.2.7.3.2 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.

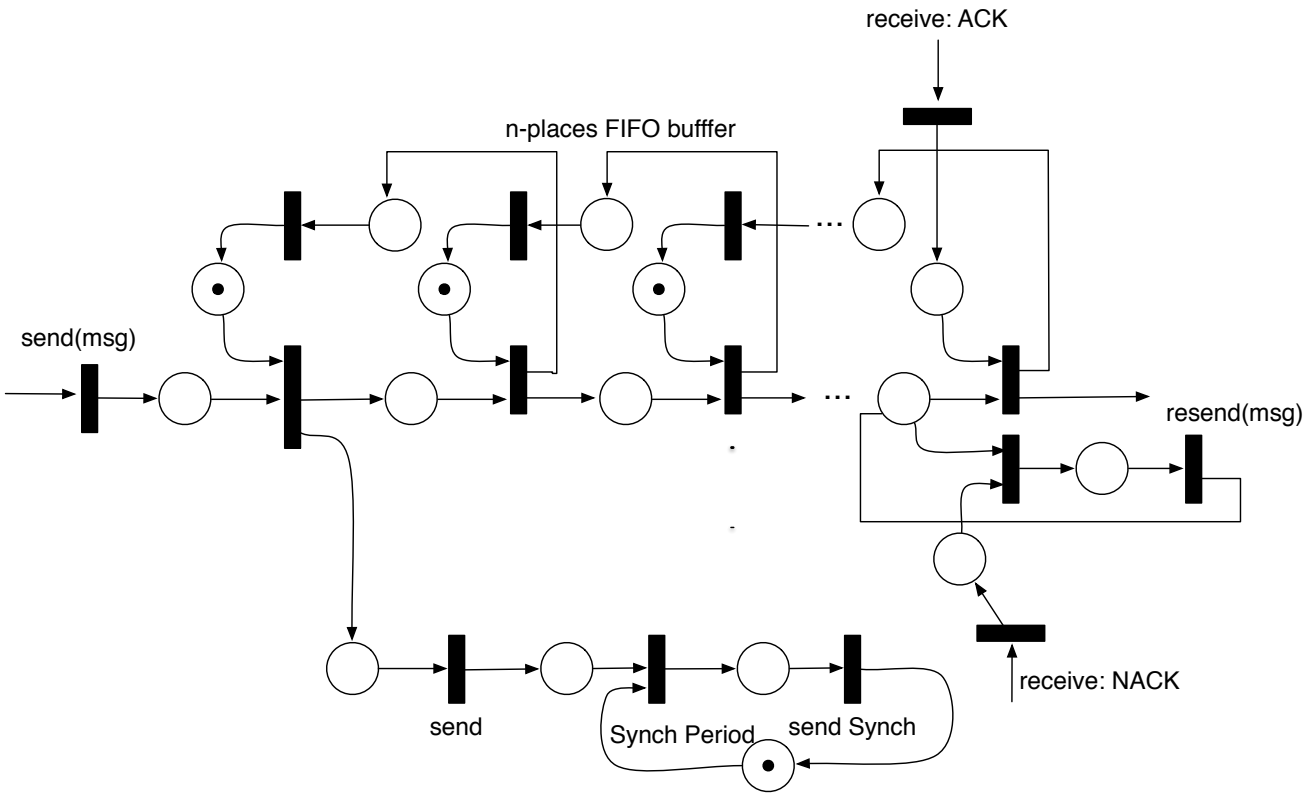


Figure 10 – Petri Net describing the Reliable Channel behaviour.

Synch Messages. 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:

```
struct Synch {
   bool reliable;
   bool sync;
   vle  sn;
   vle? n;
};
```

Where the message attributes have the following meaning:

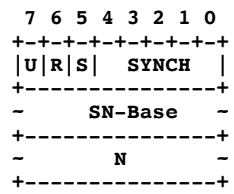
reliable: set if this concerns the reliable channel, clear if it concerns the unreliable channel; the **R** flag shall be set iff **reliable**.

sync: if set, the message shall be acknowledged promptly; the **S** flag shall be set iff **sync**.

n: optional number of unacknowledged messages; **n** present $\Leftrightarrow 0 < \mathbf{n} < 2^{\mathbf{snsize}-1}-1 \wedge$ **reliable**; the **U** flag shall be set iff **n** is present.

sn: the sequence number of the next message that will be sent on this channel — that is, the transmit window contains sequence numbers **sn-1**, **sn**-2,…,**sn**-**n** if **n** is present and the next sequence number will be **sn**

The synch message has the following wire representation:



Where **SYNCH** shall have the value **0x0E**.

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 an unbounded 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.

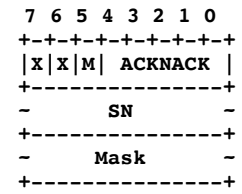
The structure of the message is the following:

```
struct AckNack {
   vle sn;
   vle? Mask
};
```

Where the attributes have the following meaning:

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

mask: if present the message is a NACK for the messages with sequence numbers (SN+*i*) s.t. $(\lfloor 2^{-i}m \rfloor - 2\left\lfloor \lfloor 2^{-i}m \rfloor / 2 \right\rfloor \neq 0)$, where *m* = 2 mask+1. The **M** flag shall be set iff **mask** is present.



Where **ACKNACK** shall have the value **0x0F**.

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

2.2.8 Writing Data

After a publisher has been successfully created and the declaration of a matching data reader has been received, a zenoh implementation shall start to send data to the matching entity(s). To avoid any ambiguity, a 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.

2.2.8.1 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:

```
struct WriteData {
  bool reliable;
  bool sync;
  vle sn;
  Resource resource;
  bytes payload;
};
```

Where the attributes have the following meaning:

reliable: set if this concerns the reliable channel, clear if it concerns the unreliable channel; the **R** flag shall be set iff **reliable**.

sync: if set, the message shall be acknowledged promptly; the **S** flag shall be set iff **sync**.

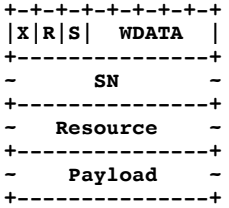
sn: the sequence number for this message.

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.

The WriteData message is represented on the wire as follows:



Where **WDATA** shall have the value **0x09**.

2.2.8.2 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:

```
struct StreamData {
  bool reliable;
  bool sync;
  vle sn;
  vle id;
  vle? prid;
  bytes payload;
};
```

Where the attributes have the following meaning:

reliable: set if this concerns the reliable channel, clear if it concerns the unreliable channel; the **R** flag shall be set iff **reliable**.

sync: if set, the message shall be acknowledged promptly; the **S** flag shall be set iff **sync**.

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; the **P** flag shall be set iff **pub_id** is present.

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.

The StreamData message is represented on the wire as follows:

```
+-+-+-+-+-+-+-+-+
|A|R|S|  SDATA  |
+---------------+
~      SN       ~
+---------------+
~   RID | SID   ~
+---------------+
~      PRID     ~
+---------------+
~    Payload    ~
+---------------+
```

Where **SDATA** shall have the value **0x07**.

2.2.8.3 Batch Data

zenoh supports message batching for the same resource/selection. The BatchedData message further improves the wire efficiency of the protocol. The structure of this message is as follows:

```
struct BSData {
   vle? id;
   bytes payload;
};
typedef sequence<BSData> BSPayload;

struct BatchedStreamData {
   bool reliable;
   bool sync;
   vle sn;
```

```
    vle id;
    BSPayload payload;
};
```

Where the attributes have the following meaning:

reliable: set if this concerns the reliable channel, clear if it concerns the unreliable channel; the **R** flag shall be set iff **reliable**.

sync: if set, the message shall be acknowledged promptly; the **S** flag shall be set iff **sync**.

sn: the sequence number for this message.

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

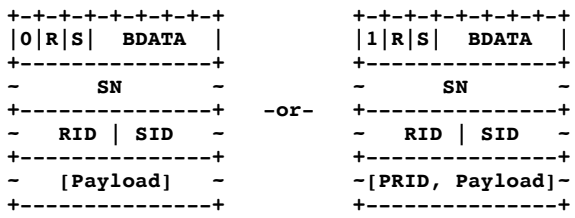
payload: the batched data; the **P** flag shall be set iff the batched data includes the resource ids to which the data belongs.

Notice that the minimal overhead in this case is:

$$overhead = \frac{3+n}{n}$$

which gets very quickly close to 1 as the number **n** of batched message grows beyond a few.

The `BatchedData` message is represented on the wire in either of two forms:



Where **BDATA** shall have the value **0x08**.

2.2.9 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.

2.2.9.1 Pull Message

Pull is a reliable message with the following structure:

```
struct Pull {
   bool sync;
   vle sn;
   vle id;
   vle pull_id;
   vle? max_samples;
};
```

Its attributes have the following meaning:

sync: if set, the message shall be acknowledged promptly; the **S** flag shall be set iff **sync**.

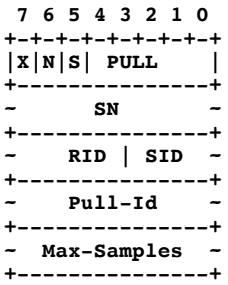
sn: the message sequence number.

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

pull_id: an id used to identify this pull session. This way any state required can be cached on the broker side. Implementations at least support 0 ≤ **pull_id** < 16.

max_samples: optionally the maximum number of samples that this node is willing to receive before having to issue another pull; the **N** flag shall be set iff **max_samples** is present. Implementation shall at least support 0 < max_samples < 2^{16}.

The Pull message shall be represented on the wire as follows:



Where the **PULL** message identifier is **0x0b.**

2.2.10 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:

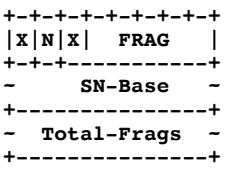
```
struct Frag {
   vle sn_base;
   vle? n;
};
```

Where the attributes have the following meaning:

sn_base: 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. The **N** flag shall be set if **n** is present.

The `Frag` message shall be represented on the wire as follows:



Where the **FRAG** message identifier is **0x12.**

2.2.11 Roundtrip-Time Estimation

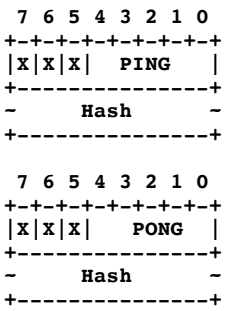
Latency estimation is essential in order to do proper reliability window management. As such the zenoh message provides a couple of messages to estimate the roundtrip-time between to nodes. These messages are called Ping and Pong respectively and their structure is shown below.

```
struct Ping {
  vle hash;
};

struct Pong {
  vle hash;
};
```

On receipt of a `Ping` a `Pong` with the same hash shall be sent in response with the contents of the **hash** attribute copied from the **hash** attribute of the `Ping` message. An implementation shall at least support hashes < 2^{16}.

The `Ping` and the `Pong` message layout shall be as defined below:



Where the **PING** message identifier is **0x0C** and the **PING** message identifier is **0x0D**.

2.3 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}** 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 associated resource **R** is derived as follows:

- The Topic QoS as well as the topic type information is written as a property of the Resource. The serialisation format is 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 prefixing all except the first character by a forward slash “/”, replacing each occurrence of a “*” wildcard by “**” and each appearance of a “?” wildcard by “*”, and collapsing any sequence of repeated of “/**” by a single occurrence of it.

- The resource name is then concatenation of the string **“xrce://”**, 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 a zenoh resource into a DDS topic, is trivial as all the information required is provided as part of the resource properties. Those information, will be created by the agent to create the proper DDS entities to ensure interoperability between DDS and zenoh.

2.4 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.

3 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.