Drone Coordination Protocol (DCP) and VarDis Protocol Specification

Andreas Willig, University of Canterbury

October 1, 2022

Contents

---

1 Version and Contributors

1.1 DCP Protocol Version

- Version 1.0, October 1, 2022

1.2 Document Version

- Version 0.1 (July 29, 2022): created initial version
- Version 0.2 (October 1, 2022): first complete draft for protocol version 1.0

1.3 Contributors

- Andreas Willig, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand
- Sam Pell, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand

---

2 Change Log

2.1 Changes since publication of Version 1.0

- October 17, 2022: VarDis: Changed meaning of `VARDISPAR_MAX_SUMMARIES` configuration variable to now include an option to disable generation of summaries when this variable is set to zero, see Configurable Parameters, Payload Construction, Composing `IETYPE-SUMMARIES`.

---

3 Context and Aims

When operating a (connected) swarm or formation of unmanned aerial vehicles (UAVs) or drones, the swarm members will have to communicate wirelessly with each other for purposes of coordination (e.g. to decide where the formation should move), or avoidance of physical drone collisions. To address the requirement of collision avoidance we assume that each drone frequently transmits messages carrying its own position, speed, heading and possibly other operational information to its local neighbourhood, typically using a local broadcast. Neighboured drones can use this information to estimate the trajectory of the sender and predict impending collisions. This mode of operation is very similar to the case of vehicular networks, where each vehicle frequently transmits basic safety messages to its neighbourhood, often at a rate of 10 Hz, to reduce uncertainty about the position of neighbours to a tolerable level.

Under this arrangement, the network carries a constant background load of periodically transmitted packets, which from hereon we refer to as **beacons**. The Drone Coordination Protocol stack (DCP) described in this document provides such a functionality of frequent beacon transmission, and builds on it to not only let drones broadcast safety information to single-hop neighbours, but also to provide a mechanism allowing the replication of information throughout the entire (generally multi-hop) network by *piggybacking* it onto beacons. In other words, we use the periodic beacons as a vehicle for a flooding process instead of flooding each piece of information in separate packets through the network. These separate packets would otherwise compete with beacons for channel access and channel resources, and would also lead to significantly higher packet overheads, particularly for small information payloads as is often the case for coordination-related information. Having each node repeating piggybacked information a configurable number of times leads, together with the inherently available spatial diversity, to a high degree of redundancy and therefore high reliability. This overall functionality of information replication throughout the entire network is very useful for global coordination and task allocation purposes.

The VarDis (Variable Dissemination) protocol, which is a key part of DCP, provides this beacon-based flooding mechanism and offers to applications the abstraction of a **variable**. Such a variable is intended to be a piece of information

to be shared amongst all drones in a swarm in a reliable (and hopefully fast) fashion, the goal being to make sure that all stations very quickly have the same view on a variable after it has changed. Variables can be created, read, updated and deleted (CRUD). Currently, the variables provided by VarDis are restricted in that only the station that created a variable is also allowed to update and delete it, whereas any other node is only allowed to read the current value of the variable (and query some of its attributes). Variables can be dynamically created and removed. We refer to the collection of all variables as the **real-time database**. A key property of the variable abstraction provided by VarDis is that a node reading a variable will only have access to the most recent variable value received through an update, but not to the history of all updates that have happened in the past.

The DCP and in particular the VarDis protocol can operate over a general wireless multi-hop network. Its design is largely independent of the underlying wireless technology, as long as it provides a local broadcast facility and each station can be allocated a unique identifier like for example its MAC address. For better reading flow, we will often use WiFi as the underlying technology, which clearly fulfills these two foundational requirements.

This document describes the DCP and VarDis in a way that is largely ignorant of implementation matters, like for example the allocation of activities to processes, interprocess communication and management of race conditions / mutual exclusion, memory management, or the precise way in which service primitives are being exchanged between different protocol entities.

---

4 Basic Data Types and Conventions

In this section we describe our assumptions and conventions regarding the use of data types, variables and packet field names in DCP, insofar as they are relevant beyond the scope of just one of the component protocols. In addition, each of the constituent protocols of DCP introduces its own data types, variables and field names.

4.1 Naming and Typesetting Conventions

- We are going to be explicit about variables, packet field names and their respective types. Any variable, field name or type is typeset like `this`.
- We consistently start the names of variables or packet field names with a lower-case letter, like for example `length`, whereas types always start with an uppercase letter, like for example `String`. For the remainder of the name we often adopt the camel case convention.
- For data types that will be included in packets (referred do as **transmissible data types**) we will usually indicate their width (as a number of bits)

and, where relevant, their precise representation. For data types not to be included in packets (**non-transmissible data types**) we will not provide any details about their representation and leave it to the implementation.

4.2 Basic Non-Transmissible Data Types

The following basic data types are used inside an implementation but are not to be included in any packet transmission. They are generally system- and programming-language dependent.

- `String` is a data type for human-readable strings.
- `Int` is a data type for signed integer values, having a system-dependent width.
- `UInt` is a data type for unsigned integer values, having a system-dependent width.
- `Bool` is the Boolean data type with values `true` and `false`.

4.3 Basic Transmissible Data Types

The following basic data types can be used inside an implementation but are also included as fields in packets. Where relevant, we generally assume that these data types are transmitted in network byte order.

- Stations have unique identifiers, which we refer to as **node identifiers**, and these are of fixed-length data type `NodeIdentifier` that is not specified here in detail. Such an identifier can be administratively assigned or the implementer chooses to use a data type already provided by the underlying wireless technology, like for example a 48-bit WiFi MAC address. A beacon sender includes its identfier in every beacon and a receiving node can extract that identifier from a received beacon to identify the sender. Furthermore, each of the DCP protocols is able to retrieve the own node identifier.
- Nodes have access to a precise source of physical time (e.g. GPS). Timestamps are represented using a fixed-length data type `TimeStamp` that is not specified here in detail, but which we assume to allow to have a resolution of 1ms or better, small enough to differentiate between two update operations to the same variable.
- When a human-readable string is to be transmitted, then the data type `TString` is used, which refers to a zero-terminated string.

4.4 Queues and Lists

In some places the DCP constituent protocols need to keep runtime data organized in (first-in-first-out) queues or lists. As is common in modern programming languages, we treat queues and lists as abstract parameterizable data types that can have elements of a paticular element type `T`. We refer to the type of a queue

with element type `T` by the type name `Queue<T>`, and similarly we refer to the type of a list made up of elements of type `T` as `List<T>`.

The `Queue` data type generally supports the following operations:

- `qTake()` removes the head-of-line element from the queue and returns it (or signals that the queue is empty). It does not take any parameters.
- `qPeek()` returns the value of the head-of-queue element, but without removing it from the queue (or signals that the queue is empty). It does not take any parameters and does not modify the queue.
- `qAppend()` takes a value of the element type `T` as a parameter and appends it to the end of the queue.
- `qRemove()` takes a value of element type `T` as a parameter and removes all elements with the same value from the queue.
- `qIsEmpty()` returns a value of type `Bool` telling whether or not the queue is empty. It does not take any parameters and does not modify the queue.
- `qExists()` takes a value of element type `T` as parameter and returns `true` if the queue contains an entry with value equal to the parameter, and `false` otherwise.
- `qLength()` takes no parameters and returns the number of elements currently stored in the queue.

The `List` data type generally supports the following operations:

- `lInsert()` inserts a new element of element type `T` into a list.
- `lRemove()` takes a value of element type `T` as a parameter and removes all elements with the same value from the list.
- `lIsEmpty()` returns a value of type `Bool` telling whether or not the list is empty. It does not take any parameters and does not modify the list.
- `lExists()` takes a value of element type `T` as parameter and returns `true` if the list contains an entry with value equal to the parameter, and `false` otherwise.
- `lLength()` takes no parameters and returns the number of elements currently stored in the list.

4.5 Other Conventions

- We will refer to drones, nodes and stations interchangeably.
- We will specify interfaces offered by protocols through generic service primitives, i.e. for a particular service `S` there might be a `request`, a `response` or an `indication` primitive. We do not prescribe the precise mechanism through which service primitives are being exchanged, this is implementation-dependent. However, when describing the processing of a `S.request` primitive for a service `S`, we will frequently use the keyword `return`, followed by some status code. This indicates that processing of the `S.request` primitive stops and that a `S.response` primitive carrying the indicated status code shall be returned to the entity generating the

`S.request` primitive.

---

5 DCP Architecture and Protocol Overview

We break the DCP system down into a number of sub-protocols. Here we introduce these sub-protocols and the environment they operate in.

The **underlying wireless bearer** (UWB) is not part of the DCP stack proper, but refers to an underlying wireless technnology implementing at least a physical (PHY) and medium access control (MAC) sublayer, and offering a local broadcast facility.

The DCP stack itself consists of three protocols. The **beaconing protocol** (BP) sits on top of the UWB, whereas the **State Reporting Protocol** (SRP) and the **Variable Dissemination Protocol** (VarDis) operate in parallel on top of BP. In this section we give a high-level overview over each of these four protocols, their detailed specifications are then given in Sections BP, SRP and VarDis.

Finally, by **applications** we refer to any entity using services offered by the BP, SRP or VarDis protocols. These entities can be applications or other protocols, and they use BP / SRP / VarDis services through their respective interfaces.

5.1 The Underlying Wireless Bearer (UWB)

The UWB includes as a minimum a physical and a MAC layer. We do not prescribe any specific technology or protocol stack for the UWB but only make a reasonably minimal set of assumptions about its capabilities and behaviour:

- It provides a local broadcast capability, i.e. it is possible to send a single packet in one transmission to an entire local single-hop neighbourhood. We do not assume the UWB to provide any kind of control over the size of this neighbourhood or the particular set of neighbours reachable – this is in general influenced by physical layer parameters such as the transmit power, the modulation and coding scheme, antenna directivity and other factors outside the scope of the DCP. These broadcast packets are not acknowledged. A UWB entity also has the ability to receive such broadcasts from neighboured nodes and handing them over to the BP.
- The UWB provides a facility for protocol multiplexing, i.e. it is possible for the UWB to distinguish packets belonging to DCP (the BP, to be precise) from packets belonging to other protocol stacks, e.g. IPv4 or IPv6.
- The UWB has a known maximum packet size, which is available to the BP. At the discretion of DCP the actual size of beacon packets can be varied dynamically, e.g. to strike a balance between maintaining a small overhead ratio and avoiding channel congestion.

- The UWB protects packets with an error-detecting or error-correcting code. We assume that the error-detection capability is practically perfect. As a result, none of the DCP protocols (BP, SRP, VarDis) will need to include own checksum mechanisms.

5.2 The Beaconing Protocol (BP)

At the lowest level of the DCP we have the **Beaconing Protocol** (BP), which is responsible for periodically sending and receiving beacon packets through the UWB, and for exchanging **client payloads** (or simply payloads) as byte arrays with arbitrary BP client protocols. Such client protocols operate on top of BP and use its services, examples include the SRP and VarDis. The BP can include payloads from several client protocols, or multiple payloads for the same client protocol, in the same beacon.

The BP sits on top of the UWB and uses its ability to locally broadcast beacons and receiving such local broadcasts from neighboured nodes.

The BP interface to client protocols allows client protocols to register and unregister a unique client protocol identifier with the BP, which the latter uses to identify and distinguish payloads in beacons. Once such a client protocol identifier has been registered, a client protocol entity will be able to generate variable-length payloads (which from the perspective of BP are just blocks of bytes without any internal structure) for transmission, such that neighboured nodes will receive these blocks under the same client protocol identifier. Furthermore, in the reverse direction BP delivers received payloads to their respective client protocols (as indicated by their client protocol identifier).

5.3 The State Reporting Protocol (SRP)

The **State Reporting Protocol** (SRP) is a client protocol of the BP. In the transmit direction, the application frequently retrieves safety-related information from the system (e.g. position, speed, heading) and hands them over as a record of type `SafetyData` to the SRP – the specifics of this data type are outside the scope of this document, but it has a fixed and well-known length. When the SRP prepares a block for transmission by the BP, it always only includes the information from the most recent `SafetyData` record it has received. Furthermore, the SRP adjoins meta information like a timestamp (of type `TimeStamp`) and a SRP sequence number to the `SafetyData` record, which allows a receiving node to assess how old the last SRP information received from the sender is.

On the receiving side, the SRP receives extended SRP `SafetyData` records from neighboured nodes and uses these to build and maintain a **neighbour table**, which registers all neighboured nodes and their extended `SafetyData`. The information contained in the neighbour table can then be used by applications to predict trajectories of other drones, forecast collisions and so on. The neighbour

table entries are furthermore subject to a soft-state mechanism with configurable timeout.

5.4 The Variable Dissemination Protocol (VarDis)

The **Variable Dissemination** (VarDis) protocol is a second client protocol of the BP. VarDis allows applications to create, read, update and delete (CRUD) so-called variables, which are disseminated by VarDis into the entire network by piggybacking them onto beacons. A key goal is to achieve a consistent view after any operation modifying a variable (create, update, delete) across the entire network as quickly and reliably as possible. To achieve this, the protocol leverages spatial diversity and also offers the option of having information about each modifying operation being repeated a configurable number of times by each node receiving it.

In the current DCP version update and delete operations are restricted to the node that created the variable. The protocol also includes mechanisms for nodes to detect missing information (like new variables or missed updates) and to query neighbors for this missing information.

The interface between VarDis and an application offers all operations necessary for creating, reading, updating and deleting variables, as well as auxiliary operations like listing the currently known variables or retrieving meta-information about variables.

---

6 The Beaconing Protocol (BP)

This section describes the beaconing protocol (BP).

6.1 Purpose

The BP entity on a node frequently effects the transmission of **beacons** to a local neighbourhood, using the services of the UWB. These beacons can include zero or more **client payloads** or simply payloads, which refer to variable-length byte blocks generated by a client protocol entity on the sending node, up to a (configurable) maximum payload size. Furthermore, the BP receives such beacons from local neighbours and delivers their payloads to their respective client protocols.

The BP does not offer any guarantees about successful transmission of client protocol payloads. Furthermore, as the criteria for deciding which payload(s) to include in the next outgoing beacon is left to implementations, there is also no guarantee that any client protocol payload will be included in a beacon within any fixed amount of time. Implementers can choose decision criteria that provide such guarantees at least for some client protocols.

6.2 Key Data Types and Definitions

6.2.1 Data Types

- The transmissible data type `BPProtocolId` (short for protocol identifier) is a 16-bit unsigned integer value uniquely identifying a client protocol. The following values are pre-defined:
  - `BP_PROTID_SRP = 0x0001` represents the SRP client protocol.
  - `BP_PROTID_VARDIS = 0x0002` represents the VarDis client protocol.
- The transmissible data type `BPLength` is a 16-bit unsigned integer value indicating the length of a payload block. A payload is assumed to be a contiguous block of bytes without any further internal structure of relevance to BP.
- The non-transmissible data type `BPQueueingMode` has three distinguishable values, indicating how payloads handed over by client protocols for transmission are being buffered in the BP:
  - `BP_QMODE_QUEUE` means that payloads enter a first-in-first-out queue of indefinite length. Any payload is transmitted in a beacon only once, then dropped from the queue.
  - `BP_QMODE_ONCE` means that the BP places client payloads handed down from client protocols into an internal buffer holding at most one payload. When a new payload arrives before the buffer contents have been transmitted, the buffer is over-written with the new payload. Once a buffered payload has been transmitted, it is removed from the buffer (i.e. it is only transmitted once).
  - `BP_QMODE_REPEAT` is similar to `BP_QMODE_ONCE`, but the payload is not dropped from the buffer after transmission, i.e. it may be transmitted repeatedly (until the client protocol hands over a new payload).
  - No other values are allowed.
- The non-transmissible data type `BPBufferEntry` holds all the data needed to describe a payload handed down from a client protocol for transmission. It is a record with the following fields:
  - `length` of type `BPLength` contains the length of the payload as a number of bytes.
  - `payload` is a byte array containing the actual payload. It has exactly as many bytes as indicated by the `length` field.
- The non-transmissible data type `BPClientProtocol` holds all the information the BP maintains about a client protocol. It is a record with the following fields:
  - `protocolId` of type `BPProtocolId` contains the unique protocol identifier for the client protocol.

  - `protocolName` of type `String` is the human-readable name of the client protocol.
  - `maxPayloadSize` of type `BPLength` specifies the maximum allowable payload size for this client protocol.
  - `queueMode` of type `BPQueueingMode`
  - `timeStamp` of type `TimeStamp` contains the local time at which the client protocol has been registered.
  - `queue` of type `Queue<BPBufferEntry>` is the queue holding payloads needed when `queueMode` equals `BP_QMODE_QUEUE`.
  - `bufferOccupied` of type `Bool` is needed for `queueMode` being either `BP_QMODE_ONCE` or `BP_QMODE_REPEAT`.
  - `bufferEntry` of type `BPBufferEntry` is needed for `queueMode` being either `BP_QMODE_ONCE` or `BP_QMODE_REPEAT`.

- The transmissible data type `BPPayloadBlockHeader` holds all the header information for an individual payload. It is a record with the following fields:
  - `protocolId` of type `BPProtocolId` contains the unique protocol identifier of the registered client protocol generating the payload.
  - `length` of type `BPLength` contains the length of the payload as a number of bytes.

6.2.2 Client Protocol List

The BP maintains a variable `currentClientProtocols` of type `List<BPClientProtocol>`. In addition to the standard operations on list (see Section Queues and Lists) this list also supports the following operation:

- `lProtocolExists()`, which takes as parameter a value of type `BPProtocolId` and returns a `Bool` indicating whether the list contains an entry of type `BPClientProtocol` with `protocolId` field equal to the paramter (result is `true`) or not (result `false`).
- `lRemoveProtocol()`, which takes as parameter a value of type `BPProtocolId` and removes all entries from the list for which their `protocolId` field is equal to the parameter.
- `lLookupProtocol()`, which takes as parameter a value of type `BPProtocolId` and returns the first entry from the list for which its `protocolId` field is equal to the parameter (if it exists).

6.3 Interface

6.3.1 Registration and De-Registration of Client Protocols

Before a client protocol entity is able to transmit payloads in beacons or receive payloads from incoming beacons for its own processing, it must register itself with the BP. At the end of its operation the client protocol entity can de-register.

Transmitting and receiving payloads for the registered client protocol is only possible while the registration is active.

Different client protocols are distinguished through their `BPProtocolId` values.

6.3.1.1 Service `BP-RegisterProtocol` The purpose of this service is to allow a client protocol entity to register itself with the local BP entity, so that the client entity can submit and receive payloads.

The service user invokes this service by using the `BP-RegisterProtocol.request` primitive, which carries the following parameters:

- `protId` of type `BPProtocolId` is the protocol identifier under which the new client protocol should be registered.
- `name` of type `String` is a human-readable name for the new client protocol.
- `maxPayloadSize` of type `BPLength` specifies the maximum allowable size for a payload for this client protocol.
- `queueingMode` of type `BPQueueingMode` specifies the queueing mode to be applied for this client protocol.

After receiving the `BP-RegisterProtocol.request` service primitive, the BP performs at least the following actions:

```
1.     If (currentClientProtocols.lProtocolExists(protId) == true) then
          stop processing, return status code BP-STATUS-PROTOCOL-ALREADY-REGISTERED
2.     If (maxPayloadSize > (BPPAR_MAXIMUM_PACKET_SIZE - (sizeof(BPPayloadBlockHeader) + 4))
          stop processing, return status code BP-STATUS-ILLEGAL-MAX-PAYLOAD-SIZE
3.     Let newent : BPClientProtocol with
           newent.protocolId      =  protId
                 .protocolName    =  name
                 .maxPayloadSize  =  maxPayloadSize
                 .queueMode       =  queueingMode
                 .timeStamp       =  current system time
                 .queue           =  empty queue
                 .bufferOccupied  =  false
                 .bufferEntry     =  empty buffer
4.     currentClientProtocols.lInsert(newent)
5.     return BP-STATUS-OK
```

6.3.1.2 Service `BP-DeregisterProtocol` The purpose of this service is to allow a client protocol entity to revoke a protocol registration with BP, so that no payloads are being received or transmitted any longer.

The service user invokes this service by using the `BP-DeregisterProtocol.request` primitive, which carries the following parameters:

- `protId` of type `BPProtocolId` is the protocol identifier of the protocol to be de-registered.

After receiving the `BP-DeregisterProtocol.request` service primitive, the BP performs at least the following actions:

```
1.     If (currentClientProtocols.lProtocolExists(protId) == false) then
          stop processing, return status code BP-STATUS-UNKNOWN-PROTOCOL
2.     de-allocate buffers / queues as appropriate for chosen queueMode
3.     currentClientProtocols.lRemoveProtocol(protId)
4.     return BP-STATUS-OK
```

An implementation must guarantee that after finishing processing the `BP-DeregisterProtocol.request` primitive no payloads from the requesting client protocol is included into beacons anymore.

6.3.2 Querying Registered Protocols

6.3.2.1 Service `BP-ListRegisteredProtocols` This service returns a list of all currently registered protocols. For each protocol its record of type `BPClientProtocol` is returned.

The service user invokes this service by using the `BP-ListRegisteredProtocols.request` primitive, which carries no parameters.

After receiving the `BP-ListRegisteredProtocols.request` primitive, the BP performs returns a `BP-ListRegisteredProtocols.response` primitive, which carries as a parameter the current value of the `currentClientProtocols` variable (as registered while processing the `BP-RegisterProtocol.request` primitive).

6.3.3 Payload Exchange

The following services govern the exchange of payloads between the BP and a registered client protocol.

6.3.3.1 Service `BP-ReceivePayload` With this service the BP indicates to a client protocol that a payload for this protocol has been received in an incoming beacon. The payload is handed over as is to the client protocol without further processing by the BP.

In particular, when the BP instance has received a payload for a currently registered protocol, it generates a `BP-ReceivePayload.indication` primitive for the client protocol, which carries two parameters:

- `protId` of type `BPProtocolId` holds the protocol identifier of the payload, which is equal to the protocol identifier registered by the client protocol.
- `length` of type `BPLength` holds the length of the received payload in bytes.
- `value` is a block of bytes of length `length`, which contains the received payload.

Any received payload for a currently registered protocol is handed over only once to its client protocol and cannot be retrieved afterwards.

6.3.3.2 Service `BP-TransmitPayload` With this service a registered client protocol can hand down a payload to BP for transmission in one of the next beacons. No guarantees are given on when (or if) the payload is transmitted or whether it will be successfully received by any neighbour.

The client protocol invokes this service by submitting a `BP-TransmitPayload.request` primitive. This primitive carries the following parameters:

- `protId` of type `BPProtocolId` holds the protocol identifier of the client protocol generating the payload.
- `length` of type `BPLength` holds the length of the payload to be transmitted in bytes.
- `payload` is a block of bytes of length `length` containing the payload to be transmitted.

After receiving the `BP-TransmitPayload.request` primitive, the BP performs at least the following actions:

```
1.     If (currentClientProtocols.lProtocolExists(protId) == false) then
          stop processing, return status code BP-STATUS-UNKNOWN-PROTOCOL
2.     Let protEntry = currentClientProtocols.lLookupProtocol(protId)
3.     If (length > protEntry.maxPayloadSize) then
          stop processing, return status code BP-STATUS-PAYLOAD-TOO-LARGE
4.     If (protEntry.queueMode == BP_QMODE_QUEUE) then
          If (length > 0) then
             Let newent : BPBufferEntry with
                 newent.length  =  length
                        .payload =  payload
             protEntry.queue.qAppend(newent)
             possibly trigger generation of beacon
             stop processing, return BP-STATUS-OK
          else
             stop processing, return BP-STATUS-EMPTY-PAYLOAD
5.     If (    (protEntry.queueMode == BP_MODE_ONCE)
           || (protEntry.queueMode == BP_MODE_REPEAT)) then
          If (length > 0) then
             protEntry.bufferEntry.length   = length
             protEntry.bufferEntry.payload  = payload
             protEntry.bufferOccupied       = true
             possibly trigger generation of beacon
          else
             protEntry.bufferOccupied       = false
             protEntry.bufferEntry.length   = 0
             protEntry.bufferEntry.payload  = null
          stop processing, return BP-STATUS-OK
```

6.3.4 Payload Transmission Indication

6.3.4.1 Service `BP-PayloadTransmitted` This service is used to allow BP to signal to a currently registered client protocol that one of its payloads has just been transferred from their queue or buffer into a bearer payload (see Section Packet Format), generated during the process of assembling the bearer payload.

When a client payload is added to the bearer payload, then BP issues a 'BP-PayloadTransmitted.indication' primitive with the following parameters:

- `protId` of type `BPProtocolId` holds the protocol identifier of the payload, which is equal to the protocol identifier registered by the client protocol.

6.3.5 Querying Number of Buffered Payloads

6.3.5.1 Service `BP-QueryNumberBufferedPayloads` This service allows a client protocol to query how many of its payloads are currently available for transmission to the BP.

The client protocol invokes this service by submitting a `BP-QueryNumberBufferedPayloads.request` service primitive, which carries only one parameter

- `protId` of type `BPProtocolId` holds the protocol identifier of the requesting client protocol.

After receiving the `BP-QueryNumberBufferedPayloads.request` service primitive, the BP performs at least the following actions:

```
1.     If (currentClientProtocols.lProtocolExists(protId) == false) then
          stop processing, return status code BP-STATUS-UNKNOWN-PROTOCOL
2.     Let protEntry = currentClientProtocols.lLookupProtocol(protId)
3.     If (protEntry.queueMode == BP_QMODE_QUEUE) then
          stop processing, return status code BP-STATUS-OK and value of protEntry.queue.qLer
4.     If (protEntry.bufferOccupied == true) then
          stop processing, return status code BP-STATUS-OK and value 1
       else
          stop processing, return status code BP-STATUS-OK and value 0
```

6.3.6 Configurable Parameters

These can be set by client protocols or by station management entities through a suitably defined management interface. Parameter changes are assumed to take effect immediately and will apply to all subsequently generated beacons and service invocations.

Here we specify only mandatory parameters that any BP implementation needs to support. These are:

- `BPPAR_BEACON_PERIOD`: refers to the (long-term average) beacon period
- `BPPAR_MAXIMUM_PACKET_SIZE`: specifies the maximum packet size of beacons (which can include several payloads). This parameter must never be

larger than the value of the `maxPacketSize` variable, which is initialized at startup and contains the maximum packet size allowed by the UWB.

6.4 Packet Format

The BP prepares a block of data to be handed over to the underlying wireless bearar (UWB, see Section Architecture), we refer to this block of data as the **bearer payload**. We refer to the payloads generated and processed by the client protocols (which are embedded into the bearer payload) as client payloads or sometimes simply as payloads.

The bearer payload is made up of one or more **payload blocks** following each other without gap. A payload block wraps a client payload, it consists of a value of type `BPPayloadBlockHeader`, immediately followed by the actual client payload as a contiguous sequence of bytes.

The combined length of all payload blocks included in the bearer payload must not exceed the value given in the `BPPAR_MAXIMUM_PACKET_SIZE` parameter (see Section Configurable Parameters). When no client payload is available at the time the BP attempts to construct a beacon, then no beacon is being generated.

6.5 Initialization, Runtime and Shutdown

During initialization of the BP the list of currently registered client protocols (variable `currentClientProtocols`) is initialized as the empty list.

Furthermore, BP retrieves from the UWB the maximum allowed packet size and stores this in the global variable `maxPacketSize`. The value of the configurable `BPPAR_MAXIMUM_PACKET_SIZE` is initialized to the `maxPacketSize` value.

6.6 Transmit Path

The timing of beacon transmissions is not specified – beacons can be generated by the BP and submitted to the UWB for example strictly periodically or with some random jitter; their generation might additionally be triggered by the arrival of a new client payload. However, for a fixed value of the `BP_BEACON_PERIOD` parameter it is expected that the number of beacons transmitted during a time period T converges to (T/`BP_BEACON_PERIOD`) as T grows large.

When preparing a beacon packet, the BP inspects all available payload queues and buffers (fields `queue`, `bufferOccupied` and `bufferEntry` of the currently registered protocols in the list `currentClientProtocols`, see Section Service TransmitPayload) in an unspecified, implementation-dependent order. It is also left to the implementation to decide how many and which payload blocks (see Section Packet Format) are being added into a bearer payload, as long as the combined length of payload blocks does not exceed the value of the `BP_MAXIMUM_PACKET_SIZE` parameter.

Let clientProtocol of type BPClientProtocol be a client protocol entry in currentClientProtocols currently under consideration for adding a payload to the bearer payload. The following rules apply:

- If clientProtocol.queueMode equals BP_QMODE_ONCE or BP_QMODE_REPEAT, then the payload contained in clientProtocol.bufferEntry can only be added to the bearer payload when the clientProtocol.bufferOccupied flag is true and when the payload length clientProtocol.bufferEntry.length of the payload stored in the buffer is small enough so that adding the resulting payload block to the bearer payload does not make the latter exceed the maximum bearer payload length (BP_MAXIMUM_PACKET_SIZE). When the BP does add the payload stored in the buffer, it will afterwards set the clientProtocol.bufferOccupied flag to false when the clientProtocol.queueMode flag supplied has value BP_QMODE_ONCE, whereas the bufferOccupied flag remains true when queueMode is equal to BP_QMODE_REPEAT.
- If clientProtocol.queueMode equals BP_QMODE_QUEUE, a payload can only be added if clientProtocol.queue.qIsEmpty() returns false and the length field of the head-of-queue element of type BPBufferPayload (obtained through clientProtocol.queue.qPeek()) indicates that the payload length is small enough so that adding the resulting payload block to the bearer payload does not make the bearer payload exceed the maximum bearer payload length (BP_MAXIMUM_PACKET_SIZE). When the BP indeed does add a payload from the queue, then it calls clientProtocol.queue.qTake() to remove this head-of-line payload from the queue.

In any case, if a payload of a client protocol is being added to the bearer payload, it must inform the client protocol by sending a BP-PayloadTransmitted.indication service primitive (see Service BP-PayloadTransmitted), with the protId parameter being set to the value of the protocolId field of the variable clientProtocol.

6.7 Receive Path

When the UWB hands over the bearer payload of a received beacon to the BP, it parses the contained payload blocks sequentially. Recall that each payload block consists of a header of type BPPayloadBlockHeader followed by the actual payload as a sequence of bytes, without any gap (see Section Packet Format). We initialize a variable index to zero, where generally index refers to the byte position in the bearer payload at which the next payload block starts.

In the following, we refer to the payload block starting at index index as pblock, to its header as pblock.header (of type BPPayloadBlockHeader) and the actual payload data as pblock.payload. The receiving BP performs at least the following steps:

```
1.     let index      =  0
```

```
           bearerlen  =  length of overall bearer payload
           transId    =  node identifier of node sending the beacon
2.     If (transId == own node identifier) then
          stop processing, drop beacon
3.     While (index < bearerlen)
3.a.      let protId        = pblock.header.protocolId
              plength       = pblock.header.length
              blocklength   = plength + sizeof(BPPayloadBlockHeader)
3.b.      index = index + blocklength
3.c.      If (currentClientProtocols.lProtocolExists(protId) == true) then
             send BP-ReceivePayload.indication primitive prim to client protocol, with
                 prim.protId  = protId
                 prim.length  = plength
                 prim.value   = pblock.payload
```

---

7 The State-Reporting Protocol (SRP)

This section describes the DCP state-reporting protocol (SRP).

7.1 Purpose

The state-reporting protocol (SRP) provides frequent transmission of safety-related information about a drone (such as its position, speed, heading and any other safety-related data) to neighboured drones, inside of beacons. In the other direction, it receives those safety transmissions from neighboured drones and consolidates them into a **neighbour table**, which can be used by applications to check, for example, the risk of an imminent collision. The size of the neighbourhood is determined by external factors such as the chosen transmit power or properties of the propagation environment, and is not under the control of this protocol. The neighbour table entries are equipped with a soft-state mechanism. The safety-related information has strictly single-hop scope.

The SRP operates on top of the beaconing protocol (see Section Beaconing Protocol) as a client protocol.

7.2 Key Data Types and Definitions

7.2.1 Data Types

- The transmissible data type `SRPSequenceNumber` is a 32-bit unsigned integer. Sequence numbers are included in SRP payloads.
- The transmissible data type `SafetyData` refers to all the data that a drone wants to transmit to its neighbours, for example its current position, speed and direction, or some information about its short-term trajectory. The

precise details of this data type are implemention-dependent and therefore the generation of values is left to applications, but `SafetyData` records have a fixed size (given by `sizeof(SafetyData)`.

- The transmissible data type `ExtendedSafetyData` acts as a wrapper around the type `SafetyData`. It contains additional fields, and the SRP is actually concerned with the transmission of values of type `ExtendedSafetyData` and their wholesale storage in the neighbour table. An `ExtendedSafetyData` record contains the following fields:
  - `sData` of type `SafetyData` is the actual safety data record wrapped into the present `ExtendedSafetyData` record.
  - `nodeId` of type `NodeIdentifier` contains the node identifier of the node that generated the `ExtendedSafetyData` record.
  - `tStamp` of type `TimeStamp` (see Section Data Types) is generated by the SRP when a `SafetyData` record is handed down by applications for transmission
  - `seqno` of type `SRPSequenceNumber` is a sequence number associated with the present `ExtendedSafetyData` record. Two successive `ExtendedSafetyData` records generated in response to two `SafetyData` records received from the application must have distinct sequence numbers.
- The non-transmissible data type `NeighbourTableEntry` record has the following fields:
  - `nodeId` of type `NodeIdentifier`
  - `extSD` of type `ExtendedSafetyData` is the last value of type `ExtendedSafetyData` received from the drone with identifier `nodeId`.
  - `receptionTime` of type `Timestamp` contains the local timestamp of the time the last value in `extSD` has been received, as generated by the SRP entity upon successfully processing a received `BP-ReceivePayload.indication` primitive. Assuming sufficiently well synchronized local clocks among neighboured nodes, the difference between this timestamp and the timestamp contained in `extSD.tStamp` gives an indication of the delay introduced by the underlying BP and UWB.

7.2.2 The Neighbour Table

The non-transmissible `NeighbourTable` data type is a collection of a number of `NeighbourTableEntry` records, together with some pre-defined operations on this collection. These include:

- `ntLookup(nodeId)`: the parameter `nodeId` is of type `NodeIdentifier` and refers to the identifier of the node for which a full `NeighbourTableEntry` is to be found in the table. This operation can end unsuccessful if the table does not contain an entry for which its `nodeId` field equals the parameter,

in which case a `NULL` value is returned. Alternatively, if such an entry exists, it returns this (unique) entry.
- `ntListAllNodeIds()`: returns a list of all `nodeId` fields for all `NeighbourTableEntry` records currently in the neighbour table, in no guaranteed order.

There is a single variable `neighbourTable` of type `NeighbourTable`. The neighbour table only contains records for neighboured nodes, not for the hosting node itself.

An implementation might provide additional, more specific functions for querying the contents of the `NeighbourTable`, e.g. listing entries in order of increasing distance. Applications can use these functions for example to assess the risk of a collision in a given time window, or to assess the amount of uncertainty in the position data of a neighbour from the available timestamps.

7.2.3 The Sequence Number

The SRP entity maintains a global variable called `currentSequenceNumber` of type `SRPSequenceNumber`.

7.3 Interface

SRP offers its services to the applications through the service interface described in the following.

Whenever an application submits a service request primitive, SRP must check whether it is currently registered as a client protocol with BP. If not, then SRP must reject the service request primitive, i.e. send back a response primitive with status code 'SRP-STATUS-INACTIVE' to the application.

7.3.1 Service SRP-TransmitSafetyData

The purpose of this service is to allow an application to submit a new `SafetyData` record for transmission.

The service user invokes this service by using the `SRP-TransmitSafetyData.request` primitive, which carries the following parameters: - `sData` of type `SafetyData`

After receiving the `SRP-TransmitSafetyData.request` primitive the SRP performs at least the following actions:

```
1.     Let extsd : ExtendedSafetyData with
           extsd.sData   =  sData
                .nodeId  =  own node identifier
                .tStamp  =  current system time
                .seqno   =  currentSequenceNumber
2.     increment currentSequenceNumber modulo (MAXVAL(SRPSequenceNumber)+1)
3.     Prepare BP-TransmitPayload.request primitive req with
```

```
            req.protId  = BP_PROTID_SRP
               .length  = sizeof(ExtendedSafetyData)
               .payload = extsd
4.      Submit req to BP.
5.      stop processing, return SRP-STATUS-OK
```

7.3.2 Neighbour Table Access

The way applications access the neighbour table is considered as implementation-dependent and not specified in this document.

7.3.3 Configurable Parameters

- `SRPPAR_NEIGHBOUR_TABLE_TIMEOUT` in milliseconds: if a node doesn't receive a new `ExtendedSafetyData` from a particular neighbour for this many milliseconds, then the neighbour table entry will be dropped. Default value: 3,000 (corresponding to 3 seconds).

7.4 Packet Format

The payload transmitted and received by SRP are all of type `ExtendedSafetyData`.

7.5 Initialization, Runtime and Shutdown

The SRP will have to perform certain actions besides processing packets or responding to service requests from the application.

7.5.1 Initialization

During the initialization, the SRP will need to register itself as a client protocol with the underlying BP before it can process any `SRP-TransmitSafetyData.request` primitives. To achieve this registration, the SRP prepares a `BP-RegisterProtocol.request` service primitive (see Service BP-RegisterProtocol) with the following parameters:

- `protId` is set to `BP_PROTID_SRP`
- `name` is set to "SRP – State Reporting Protocol Vx.y" where 'x' and 'y' refer to the present version of SRP.
- `maxPayloadSize` is set to `sizeof(ExtendedSafetyData)`. Implementations may opt to add an additional safety margin.
- `queueingMode` is set to `BP_QUEUEING_MODE_REPEAT`, meaning that the BP will always transmit the last `ExtendedSafetyData` record that the SRP has handed over.

The variable `neighbourTable` of type `NeighbourTable` is initialized as being empty.

The variable `currentSequenceNumber` of type `SRPSequenceNumber` is initialized to zero.

7.5.2 Runtime

The SRP entity traverses the neighbour table `neighbourTable` at a frequency chosen such that during any time interval of length `SRPPAR_NEIGHBOUR_TABLE_TIMEOUT` (see Configurable Parameters) at least five traversals are carried out. During each traversal the SRP entity first obtains the current system time and then checks for each entry `ent` of type `ExtendedSafetyData` contained in `neighbourTable` whether the difference between the current system time and the (receiver-generated) timestamp `ent.receptionTime` exceeds the value of the `SRPPAR_NEIGHBOUR_TABLE_TIMEOUT` parameter. If so, the neighbour table entry is dropped from the table.

7.5.3 Shutdown

During the shutdown, the SRP will de-register itself from the BP, by issueing the service primitive `BP-DeregisterProtocol.request` with the `protId` parameter set to `BP_PROTID_SRP`.

7.6 Receive Path

The receive path is activated when the underlying BP signals the reception of an SRP payload with the `BP-ReceivePayload.indication` primitive (see Service BP-ReceivePayload). In this case, referring to the received primitive as `payloadIndication`, the SRP performs at least the following actions:

```
1.     If (    (payloadIndication.protId =/= BP_PROTID_SRP)
            || (payloadIndication.length =/= sizeof(ExtendedSafetyData)) then
          stop processing.
2.     Let ext : ExtendedSafetyData = payloadIndication.payload
3.     If (ext.nodeId == own node identifier) then
          stop processing.
4.     Let ent : NeighbourTableEntry with
           ent.nodeId         =  ext.nodeId
              .extSD          =  ext
              .receptionTime  =  current system time
          add ent to neighbourTable or replace existing entry for ext.nodeId
          stop processing.
```

Implementations may choose to extend the data type `NeighbourTableEntry` by further fields and add appropriate processing here. For example, the sequence numbers contained in values of type `ExtendedSafetyData` can be used to estimate packet loss rate between the neighbour and the the current node.

---

8 The VarDis (Variable Dissemination) Protocol

This section describes the Variable Dissemination (VarDis) protocol.

8.1 Purpose

On the highest level, VarDis is concerned with the maintenance of a **real-time database**, which is simply defined to be a collection of **variables**. The variable concept of VarDis supports the four key operations of create, read, update and delete (CRUD). Each variable is created by exactly one node, and that same node is the only node that is allowed to modify the variable – we occasionally refer to this node as the **producer** of that variable. All other nodes are only allowed to read the current variable value, we refer to them as the **consumers**. The role of the producer does not change during the lifetime of the variable, and the producer is the only node allowed to update its value or delete the variable.

A key responsibility of VarDis is to disseminate updates to the variable as quickly and reliably as possible in the network and to keep the real-time database in a consistent state across all nodes. The key vehicle for that, and the feature that distinguishes VarDis from other approaches, is that dissemination of variable updates relies on piggybacking them on frequently transmitted beacons (with beacon transmission provided by the underlying BP, for which VarDis is a client protocol) instead of flooding updates separately.

The real-time database itself is dynamic in the sense that variables can be created or removed at runtime.

The real-time database is not tied to any notion of persistency, and there is no guarantee that any node will be able to track all updates made to a variable. Furthermore, the moniker "real-time" should not be interpreted strictly, as no guarantees on dissemination speed or reliability will and can be given. It merely captures an aspiration to be both fast and reliable.

8.2 Key Data Types and Definitions

8.2.1 Variables

We start by defining some data types related to variables.

- The transmissible data type `VarId` (short for variable identifier) is an 8-bit unsigned integer. Each variable needs to have a unique identifier, but uniqueness must be ensured by developers, it is not strictly enforced by VarDis.
- The transmissible data type `VarLen` (short for variable length) is an 8-bit integer describing the length of a variables value in memory as a number of bytes. VarDis treats a variable as just a block of bytes of given length. Note that the length of (the values of) a variable is not fixed over time, at different times a variable can have values of different lengths. The variable length must not exceed a configurable maximum variable length (parameter `VARDISPAR_MAX_VALUE_LENGTH`, see Configurable Parameters).
- The transmissible data type `VarRepCnt` is an 8-bit unsigned integer, which specifies for a variable in how many distinct beacons a node should re-

peat information pertaining to that variable (create or delete operations, updates).

- The transmissible type `VarSpec` (short for variable specification) represents a record collecting all the metadata or attributes describing a variable. All these attributes remain fixed throughout its lifetime. Furthermore, VarDis does not keep track of the type of a variable, it just treats the value of a variable as a block of bytes without further structure. Any interpretation of a variable (including its association with a datatype) is handled by the applications. A `VarSpec` has the following fields:
  - `varId` of type `VarId`: unique numerical variable identifier.
  - `prodId` of type `NodeIdentifier`: the node identifier of the variable producer.
  - `repCnt` of type `VarRepCnt`: its repetition count.
  - `descr` of type `TString`: a human-readable description of the variable.
- The transmissible data type `VarSeqno` is an 8-bit unsigned integer, to be used in a circular fashion. Each producer of a variable maintains a separate local sequence number for that variable. This sequence number is incremented upon every new update operation and included in the disseminated update. Sequence numbers are used by nodes to check whether they have the most recent value of a variable.
- The transmissible data type `VarUpd` (short for variable update) is generated by the producer of a variable when its application writes to it, and the update is then disseminated by VarDis to all nodes in the network. A `VarUpd` record includes the following fields:
  - `varId` of type `VarId`: identifier of the variable
  - `seqno` of type `VarSeqno`: a sequence number as managed by the producer, such that successive updates have distinct sequence numbers.
  - `length` of type `VarLen`: gives the length of the variables value as a number of bytes.
  - `value`: The value of the variable as a block of bytes. The length of this block of bytes is the same as the length given in the `length` field for this variable.
- The transmissible data type `VarCreate` (short for variable creation) collects all the information the producer of a variable disseminates upon creation of a variable. It contains the following fields:
  - `spec` of type `VarSpec` contains all the metadata describing the variable.
  - `upd` of type `VarUpd` contains the initial value of the variable.
- The transmissible data type `VarSumm` (short for variable summary) is used to let nodes include variable summaries in their beacons to let neighbours know what is the most recent update to a variable they have received

(either from applications if it is the producer, or from other nodes if it is a consumer). A `VarSumm` includes the following fields:

- `varId` of type `VarId`: identifier for a variable that the sending node has in its real-time database (i.e. an existing variable).
- `seqno` of type `VarSeqno`: the seqno of the most recent variable update (of type `VarUpd`) that the sender has received.

8.2.2 Information Elements

The VarDis payload of a beacon packet is made up of one or more **information elements** (IE), which hence are transmissible data types. Each information element is a type-length-value triple as follows:

- The field `ieType` (for information-element type) specifies the particular type of information element being considered.
- The field `ieLen` (for information-element length) specifies the length of the value part of the IE as a number of bytes.
- The field `ieVal` (for information-element value) is an array of bytes with a structure suitable for the given `ieType`. There are as many bytes in the array as the `ieLen` field specifies.

The precise encoding of the `ieType` and `ieLen` fields is implementation-dependent, but it is suggested to ensure that these two fields in total use up either 8 or 16 bits.

A VarDis payload transmitted in a beacon can contain at most one of each of the following types of information elements, as long as their combined size fits within the allowed payload size (cf. parameter `VARDISPAR_MAX_PAYLOAD_SIZE`, see Section Configurable Parameters).

In the following we describe the key information elements used by VarDis. We leave out the computation of the `ieLen` field, which should not present any difficulties. In some cases the IE payload is a list of elements of another fixed-size data type `T`. In these cases the receiver should always check that the `ieLen` field has a value that is an integer multiple of `sizeof(T)`.

8.2.2.1 `IETYPE-SUMMARIES` (`ieType` = 1) The `ieVal` is a list of `VarSumm` records, where each `VarSumm` includes a variable identifier (of type `VarId`) and a sequence number (of type `VarSeq`), representing the latest sequence number received for that variable. A neighbour receiving this information element can compare the received summaries with the contents of its own real-time database and check whether it misses any variables, only has them in outdated versions, or has itself newer versions than the sending neighbour.

8.2.2.2 `IETYPE-UPDATES` (`ieType` = 2) The `ieVal` is a list of `VarUpd` records. The transmitter includes the latest variable values and sequence numbers it has stored in its real-time database (but not necessarily for all the variables in the

database). The intention is that the receiver will itself repeat that update as many times in distinct future beacons as indicated by the `repCnt` field of the corresponding variable specification (type `VarSpec`), to further propagate the update in the network. This is conditional on this update being truly new to the receiver, i.e. having a sequence number strictly larger than the sequence number it previously had stored for the same variable.

8.2.2.3 `IETYPE-REQUEST-VARUPDATES` (`ieType` = 3) The `ieVal` consists of a list of pairs of type (`VarId`, `VarSeqno`), which can be represented by elements of type `VarSumm`. The intention is that the sending node requests any / all of its neighbours to send `VarUpd` records for the requested `VarId` some time in the future, in as many distinct future beacons as given by the `repCnt` field of the corresponding variable specification (`VarSpec`), provided that the neighbours have strictly more recent versions of the requested variables than indicated by the included sequence number.

8.2.2.4 `IETYPE-REQUEST-VARCREATES` (`ieType` = 4) The `ieVal` consists of a list of `VarId` values. The intention is that the sending node requests any / all of its neighbours to send `VarCreate` records for each of the requested variables (each of which combines a `VarSpec` record and a `VarUpd` record), which they will embed into `IETYPE-CREATE-VARIABLES` information elements some time in the future, in as many distinct beacons as indicated by the `repCnt` parameter of the corresponding variable specifications (of type `VarSpec`).

8.2.2.5 `IETYPE-CREATE-VARIABLES` (`ieType` = 5) The `ieVal` consists of a list of `VarCreate` records, which themselves are pairs of variable specifications (`VarSpec`) and initial values and sequence numbers (`VarUpd`). The intention is that the sending node notifies the neighbour that new variables (together with their initial values) have been added to the real-time database, and for the neighbour to further repeat the `VarCreate` records in as many distinct future beacons as indicated in the `repCnt` field of the variable specification (`VarSpec`).

8.2.2.6 `IETYPE-DELETE-VARIABLES` (`ieType` = 6) The `ieVal` is a list of `VarId` values. The intention is that the sending node notifies the neighbour that the given variables are to be deleted from the real-time database, and for the neighbour to further repeat the `VarId` values in as many distinct future beacons as indicated in the `repCnt` field of the variable specification (`VarSpec`).

8.2.2.7 Global vs. Local Dissemination The three information elements `IETYPE-SUMMARIES`, `IETYPE-REQUEST-VARUPDATE` and `IETYPE-REQUEST-VARSPEC` have single-hop scope, i.e. the sender wishes to reach only its immediate neighbours. These information elements are not included by the receivers in their own beacons for further dissemination.

In contrast, the `IETYPE-UPDATES`, `IETYPE-CREATE-VARIABLES` and `IETYPE-DELETE-VARIABLES` information elements have global scope. In particular, when a receiver recognizes

that the data contained in any of these information elements is either truly new (i.e. the receiver has not received it before) or truly updated (i.e. the receiver only has a strictly older value of a variable), then it will in turn include the received data (of type `VarUpd`, `VarCreate` or `VarId`) in as many distinct future beacons as indicated by the `repCnt` parameter of the corresponding variable specification (`VarSpec`), to help with further propagation in the network.

8.2.3 Real-Time Database

The real-time database is a collection of variables, with `VarId`'s used as a key. For each `VarId` there exists a database entry of the non-transmissible type `DBEntry`, containing the following elements:

- `spec` of type `VarSpec`: contains the variable specification for the variable, as received in an `IETYPE-CREATE-VARIABLES` information element. The `VarSpec` includes the identifier of the variable (of type `VarId`).
- `length` of type `VarLen`: the length in bytes of the current value of the variable.
- `value` of type byte array: the current value of the variable.
- `seqno` of type `VarSeqno`: the sequence number corresponding to the current value of the variable. The `length`, `value` and `seqno` fields have been received in the last valid `IETYPE-UPDATES` (carrying a `VarUpd` for this variable) or `IETYPE-VARCREATE` (carrying a `VarCreate` record) elements received for that variable.
- `tStamp` of type `TimeStamp`: gives the last time (in local system time) that a valid `VarUpd` or `VarCreate` for this variable has been received and processed.
- A counter `countUpdate` of type `VarRepCnt`, indicating how many more times a recently received update (`VarUpd`) for this variable is to be included into distinct beacons.
- A counter `countCreate` of type `VarRepCnt`, indicating how many more times a recently received `VarCreate` for this variable is to be included into distinct beacons.
- A counter `countDelete` of type `VarRepCnt`, indicating how many more times a recently received instruction to delete the variable (it's `VarId`) is to be included into distinct beacons.
- A flag `toBeDeleted` of type `Bool` to indicate whether the variable is marked for deletion (`true`) or not (`false`). A variable deletion is not immediately carried out in the moment an `IETYPE-DELETE-VARIABLES` element is processed, but only after the deletion instruction has been repeated in beacons sufficiently often. This flag is to be initialized with `false`.

The real-time database supports three main operations:

- `RTDB.lookup()`, which takes as parameter a value of type `VarId` and either returns the unique database entry `ent` (of type `DBEntry`) for which

`ent.spec.varId` equals the given `VarId` value, or an indication that no such entry exists.

- `RTDB.remove()`, which takes as parameter a value of type `VarId`. If the database contains an entry `ent` for which `ent.spec.varId` equals the given `VarId` value, then the entry is removed from the database, otherwise nothing happens.
- `RTDB.update()`, which takes as parameter a value of type `DBEntry`, referred to as `newent`. If the database contains no entry `ent` for which `ent.spec.varId == newent.spec.varId`, then `newent` is added to the database, otherwise the old entry `ent` is replaced by `newent`.

8.2.4 Queues

A beacon can contain only one instance of each type of information elements. To manipulate and specify the contents of the information elements in future beacons, for each `ieType` VarDis maintains a number of variables of type `Queue<VarId>` (see Section Queues and Lists). In addition to the standard operations on queues, in VarDis we make use of the following two additional operations:

- `qDropNonexistingDeleted()` drops all `VarId` values from the given queue for which either `RTDB.lookup()` indicates that the corresponding variable does not exist in the real-time database or the database entry for the variable (of type `DBEntry`) has the `toBeDeleted` flag set to `true`.
- `qDropNonexisting()` drops all `VarId` values from the given queue for which `RTDB.lookup()` indicates that the corresponding variable does not exist in the real-time database.

A VarDis entity maintains the following queues at runtime, all of type `Queue<VarId>`:

- `createQ`: contains `VarId` values for all the variables for which further repetitions of the `VarCreate` information records have to be included in future beacons (inside of `IETYPE-CREATE-VARIABLES` information elements).
- `deleteQ`: contains `VarId` values for all the variables for which further repetitions of the `VarId` value for deletion have to be included into future beacons (inside of `IETYPE-DELETE-VARIABLES` information elements).
- `updateQ`: contains `VarId` values for all the variables for which further repetitions of their most recent value and sequence number (`VarUpd` records) have to be included in future beacons (inside of `IETYPE-UPDATES` information elements).
- `summaryQ`: like `updateQ`, but with summaries (type `VarSumm`) instead of updates.
- `reqUpdQ`: contains `VarId` values for all the variables for which a `VarSumm` (inside an `IETYPE-REQUEST-VARUPDATE` information element) is to be transmitted in a future beacon, to request a `VarUpd` from neighboured nodes.
- `reqCreateQ`: like `reqUpdQ`, but for `VarId` values used to request `VarCreate` records from neighboured nodes.

8.3 Interface

An application is mainly concerned with maintaining and accessing the real-time database (RTDB), which is a collection of variables. The interaction between an application and the real-time database happens through the following pre-defined services. For each service we specify a request primitive, which the application uses to request a service, and a response primitive, through which the local VarDis entity provides its response to the service request to the local application. We furthermore sketch how the VarDis entity processes a service request.

Whenever an application submits a service request primitive, VarDis must check whether it is currently registered as a client protocol with BP. If not, then VarDis must reject the service request primitive, i.e. send back a response primitive with status code 'VARDIS-STATUS-INACTIVE' to the application.

8.3.1 Describing Database Contents

8.3.1.1 Service `RTDB-DescribeDatabase` The application issues a `RTDB-DescribeDatabase.request` primitive. This primitive carries no further parameters. The intention is that the application is being provided with a list of `VarSpec` records for all variables currently in the RTDB.

The VarDis entity responds with a `RTDB-DescribeDatabase.response` primitive. This primitive carries as data a list of the `VarSpec` records of all variables currently in the real-time database, including those that have their `toBeDeleted` flag set in their respective `DBEntry` records.

8.3.1.2 Service `RTDB-DescribeVariable` The application issues a `RTDB-DescribeVariable.request` primitive, which carries a value of type `VarId` as a parameter. The intention is that the application is being provided with the full entry of the RTDB for that variable (of type `DBEntry`), or with a signal that the variable does not exist.

If the requested variable does not exist in the real-time database, then the VarDis entity responds with a `RTDB-DescribeVariable.response` primitive carrying the status code `VARDIS-STATUS-VARIABLE-DOES-NOT-EXIST`. If the variable does exist, the VarDis entity responds with the same primitive, but now carrying the status code `VARDIS-STATUS-OK` and the `DBEntry` record for this variable as parameters.

8.3.2 CRUD Operations on Variables

8.3.2.1 Service `RTDB-Create` An application wishes to create a variable and provides an initial value. If successful, then the current node automatically becomes the producer for this variable.

The application issues a `RTDB-Create.request` primitive, which carries the following parameters:

- `spec` of type `VarSpec`: the variable specification of the new variable. Its identifier (`spec.varId`) must be unique, i.e. there should be no active variable with the same `VarId` value currently in the real-time database.
- `descr` of type `TString`: a textual description of the variable, giving a human user information about the meaning or representation of a variable.
- `length` of type `VarLen`: The length of the initial value to be written.
- `value` of type byte array: The actual initial value, as a block of bytes. This has to be exactly as many bytes as given in the `length` parameter.

The `RTDB-Create.response` primitive is returned after the VarDis entity has finished processing the request. As a parameter it only includes a status code.

To process the request, the VarDis entity performs the following steps:

```
1.     If (RTDB.lookup(spec.varId) == true) then
          stop processing, return status code VARDIS_STATUS_VARIABLE_EXISTS
2.     If (descr.length > (VARDISPAR_MAX_DESCRIPTION_LENGTH-1)) then
          stop processing, return status code VARDIS_STATUS_VARIABLE_DESCRIPTION_TOO_LONG
3.     If (length > VARDISPAR_MAX_VALUE_LENGTH) then
          stop processing, return status code VARDIS_STATUS_VALUE_TOO_LONG
4.     If (length == 0) then
          stop processing, return status code VARDIS_STATUS_INVALID_VALUE
5.     If (    (spec.repCnt <= 0)
            || (spec.repCnt > VARDISPAR_MAX_REPETITIONS)) then
          stop processing, return status code VARDIS_STATUS_ILLEGAL_REPCOUNT
6.     Let newent : DBEntry with
           newent.spec         =  spec
           newent.length       =  length
           newent.value        =  value
           newent.seqno        =  0
           newent.tStamp       =  current system time
           newent.countUpdate  =  0
           newent.countCreate  =  spec.repCnt
           newent.countDelete  =  0
           newent.toBeDeleted  =  False
7.     RTDB.update(newent)
8.     createQ.qAppend (spec.varId)
9.     summaryQ.qAppend (spec.varId)
10.    return status code VARDIS_STATUS_OK
```

A successful completion of this service request only means that the variable has now been added to the real-time database **on the producer node**. It may take an unspecified amount of time before the new variable is known to all other nodes in the network.

The checking of uniqueness of a variable is quite limited. We can reliably discover the case where a variable is to be created twice on the same producer node, but there are other pathological cases we cannot immediately discover, and

the protocol makes no attempt to prevent, resolve or report such a situation. One example of such a situation is when two nodes at far away ends of a large multi-hop network want to create the same variable at more or less the same time. None of the two nodes will be aware of the other one's efforts, and quite likely it will be some intermediate nodes noticing a Create request for an already existing variable (and coming from a different producer node).

8.3.2.2 Service `RTDB-Delete` An application wishes to delete a variable. To be effective, this must be issued on the producer node, other nodes cannot delete a variable.

To request deletion of a variable, an application uses the `RTDB-Delete.request` primitive, which carries as a parameter the `VarId` value of the variable to be deleted, referred to as `varId` below.

The `RTDB-Delete.response` primitive is returned after the VarDis entity has finished processing the request. As a parameter it only carries a status code.

To process the request, the VarDis entity performs the following steps:

```
1.     If (RTDB.lookup(varId) == false) then
           stop processing, return status code VARDIS_STATUS_VARIABLE_DOES_NOT_EXIST
2.     Let ent = RTDB.lookup(varId)
3.     If (ent.spec.prodId =/= own node identifier) then
           stop processing, return status code VARDIS_STATUS_NOT_PRODUCER
4.     If (ent.toBeDeleted == true) then
           stop processing, return status code VARDIS_STATUS_VARIABLE_BEING_DELETED
5.     deleteQ.qAppend(varId)
6.     createQ.qRemove(varId)
7.     summaryQ.qRemove(varId)
8.     updateQ.qRemove(varId)
9.     reqUpdQ.qRemove(varId)
10.    reqCreateQ.qRemove(varId)
11.    ent.toBeDeleted   =  True
12.    ent.countDelete   =  ent.spec.repCnt
13.    ent.countCreate   =  0
14.    ent.countUpdate   =  0
15     RTDB.update(ent)
16.    return status code VARDIS_STATUS_OK
```

A successful completion of this primitive only means that the variable has now been deleted from the real-time database **on the producer node**. It may take an unspecified amount of time before it has been deleted from all other nodes in the network.

8.3.2.3 Service `RTDB-Update` An application on the producer node wishes to write a new value to the variable.

To achieve that, the application issues the `RTDB-Update.request` primitive, which carries three parameters:

- `varId` of type `VarId`: the identifier of the variable to be updated
- `length` of type `VarLen`: the length of the new variable value
- `value` of type byte array: The actual new variable value, as a block of bytes. This has to be exactly as many bytes as given in the `length` parameter.

The `RTDP-Update.response` primitive is returned after the VarDis entity has finished processing the request. As a parameter it only carries a status code.

To process the request, the VarDis entity performs the following steps:

```
1.     If (RTDB.lookup(varId) == false) then
           stop processing, return status code VARDIS_STATUS_VARIABLE_DOES_NOT_EXIST
2.     Let ent = RTDB.lookup(varId)
3.     If (ent.spec.prodId =/= own node identifier) then
           stop processing, return status code VARDIS_STATUS_NOT_PRODUCER
4.     If (ent.toBeDeleted == True) then
           stop processing, return status code VARDIS_STATUS_VARIABLE_BEING_DELETED
5.     If (varLen > VARDISPAR_MAX_VALUE_LENGTH) then
          stop processing, return status code VARDIS_STATUS_VALUE_TOO_LONG
6.     If (varLen == 0) then
          stop processing, return status code VARDIS_STATUS_INVALID_VALUE
7.     Increment ent.seqno modulo maximum seqno
8.     Set ent.length       = length
           ent.value        = value
           ent.countUpdate  = ent.spec.repCnt
           ent.tStamp       = current system time
9.     RTDB.update(ent)
10.    updateQ.qRemove(varId)
11.    updateQ.qAppend(varId)
12.    return status code VARDIS_STATUS_OK
```

A successful completion of this primitive only means that the variable has been updated **on the producer node**. It may take an unspecified amount of time before it has been updated on all other nodes in the network. Furthermore, there is no guarantee of order: if the producer node updates a variable at time t_1 with value v_1 and a short time later at time t_2 it updates it with value v_2, a given consumer node may receive both updates in the same order and apply both updates to its RTDB, or it receives the second update before the first one, in which case it will ignore the first update completely (as from the sequence numbers it is recognized as outdated after the second update has been processed).

8.3.2.4 Service `RTDB-Read` An application wishes to inquire the current value of a variable stored in its local real-time database.

The application issues the `RTDB-Read.request` primitive, which carries the

VarId value of the variable as its only parameter.

The RTDB-Read.response primitive is returned after the VarDis entity has finished processing the request. As a parameter it carries a status code and, if the status code is VARDIS_STATUS_OK, also the value of the variable (length field, data bytes) and the time stamp of the last update of the variable.

To process the request, the VarDis entity performs the following steps:

```
1.      If (RTDB.lookup(varId) == false) then
            stop processing, return status code VARDIS_STATUS_VARIABLE_DOES_NOT_EXIST
2.      Let ent = RTDB.lookup(varId)
3.      If (ent.toBeDeleted == True) then
            stop processing, return status code VARDIS_STATUS_VARIABLE_BEING_DELETED
4.      return status code VARDIS_STATUS_OK, ent.length, ent.value, ent.tStamp
```

8.3.3 Configurable Parameters

The configurable parameters of VarDis can be set by applications or station management entities through a suitably defined management interface. Parameter changes are assumed to take effect immediately and will apply to all subsequently generated beacons and service invocations.

Here we only specify mandatory parameters that any VarDis implementation needs to support. These are:

- VARDISPAR_MAX_VALUE_LENGTH is the maximum allowable length of a variable value. Default value is 32. The value must be larger than zero. A hard upper limit is given by the minimum of:
  - the maximum value representable in the VarLen data type
  - the maximum value representable in the ieLen component of any information element.
- VARDISPAR_MAX_DESCRIPTION_LENGTH is the maximum length of a textual variable description (of type TString) contained in a value of type VarSpec. Default value is 32. The value must be larger than zero. A hard upper limit is given by the maximum value representable in the ieLen component of any information element.
- VARDISPAR_MAX_REPETITIONS gives the maximally allowed number of repetitions for a variable update, variable creation or variable deletion that a node may include into its outgoing beacons. A hard upper bound for this parameter is 15. The value must be larger than zero.
- VARDISPAR_MAX_PAYLOAD_SIZE is the maximum allowable length of a payload generated by VarDis and handed over to the underlying BP for transmission. The value must be larger than zero.
- VARDISPAR_MAX_SUMMARIES is the maximum number of variable summaries (of type VarSumm) to be included in a payload in an IETYPE-SUMMARIES

information element. If this value is zero, then no IETYPE_SUMMARIES information element is being created.

- VARDISPAR_BUFFER_CHECK_PERIOD specifies the period of time between checks of the number of payloads in the BP buffer for VarDis (see Section Runtime). This value must be larger than zero.

The sum of VARDISPAR_MAX_VALUE_LENGTH and VARDISPAR_MAX_DESCRIPTION_LENGTH must be upper-bounded such that a VarCreate record together with the header of the IETYPE-CREATE-VARIABLES information element does not exceed VARDISPAR_MAX_PAYLOAD_SIZE.

8.4 Payload Format and Payload Construction Process

We discuss how a node (the "current node") composes the VarDis payload to be handed down for transmission to the underlying BP. The payload is composed by appending information elements until either no further elements are to be added, or the maximum allowed payload size (parameter VARDISPAR_MAX_PAYLOAD_SIZE, see Section Configuruable Parameters) will be exceeded. As each information element contains information describing its length (its ieLen field), it suffices to simply store information elements successively without gaps.

For each of the available types of information elements (see Information Elements) at most once instance is inserted into a payload. Insertion is considered in the following order and under the following conditions:

- First step: When createQ.qIsEmpty() indicates a non-empty queue, then generate an IETYPE-CREATE-VARIABLES information element with as many VarCreate records as are needed and may fit into the VarDis payload.
- Second step: When deleteQ.qIsEmpty() indicates a non-empty queue and there is sufficient space left in the VarDis payload, then generate an IETYPE-DELETE-VARIABLES information element with as many VarId values as are needed and may fit into the remaining VarDis payload.
- Third step: When updateQ.qIsEmpty() indicates a non-empty queue and there is sufficient space left in the VarDis payload, then generate an IETYPE-UPDATES information element with as many VarUpd records as are needed and may fit into the remaining VarDis payload.
- Fourth step: When summaryQ.qIsEmpty() indicates a non-empty queue and there is sufficient space left in the VarDis payload, then generate an IETYPE-SUMMARIES information element with as many VarSumm records as are needed and may fit into the remaining VarDis payload, but not exceeding the value of the VARDISPAR_MAX_SUMMARIES parameter (assuming this parameter is larger than zero).
- Fifth step: When reqCreateQ.qIsEmpty() indicates a non-empty queue and there is sufficient space left in the VarDis payload, then generate an IETYPE-REQUEST-VARCREATES information element with as many VarId values as are needed and may fit into the remaining VarDis payload.

- Sixth step: When `reqUpdQ.qIsEmpty()` indicates a non-empty queue and there is sufficient space left in the VarDis payload, then generate an `IETYPE-REQUEST-VARUPDATES` information element with as many (`VarId`, `VarSeqno`) pairs as are needed and may fit into the remaining VarDis payload.

When none of these steps produces an information element, then VarDis will not generate a BP payload.

We next describe the process of composing each of these types of information elements. It is assumed throughout that there is sufficient space for placing the respective information element.

8.4.1 Composing the `IETYPE-SUMMARIES` Element

The process of composing the `IETYPE-SUMMARIES` information element includes the following steps:

```
1.     summaryQ.qDropNonexistingDeleted()
2.     If (    (summaryQ.qIsEmpty())
            || (no space to add one VarSumm)
            || (VARDISPAR_MAX_SUMMARIES == 0)) then
           stop processing, return empty IETYPE-SUMMARIES element
3.     Let numSummaries = 0
           firstVarId   = summaryQ.qTake()
4.     Add VarSumm for firstVarId to IETYPE-SUMMARIES element
5.     summaryQ.qAppend (firstVarId)
6.     While (    (summaryQ.qPeek() =/= firstVarId)
               && (numSummaries < VARDISPAR_MAX_SUMMARIES)
               && (space available for one further VarSumm))
           nextVarId = summaryQ.qPeek()
           summaryQ.qTake()
           summaryQ.qAppend(nextVarId)
           numSummaries = numSummaries+1
           Add VarSumm for nextVarId to IETYPE-SUMMARIES element
7.     return the IETYPE-SUMMARIES element
```

The preceding procedure effectively adds summaries to the `IETYPE-SUMMARIES` element as long as the queue contains (existing) variables that haven't been added to the information element yet. We stop when we want to add the starting element again, when we have exhausted the maximum number of summaries (`VARDISPAR_MAX_SUMMARIES`) or when we run out of space. At the start, we remove those `VarId` values from `summaryQ` for which either the underlying variable does not exist in the real-time database or has been marked for deletion. But as long as a variable exists and is valid, summaries will be included in payloads.

8.4.2 Composing the IETYPE-CREATE-VARIABLES Element

The process of composing the IETYPE-CREATE-VARIABLES information element includes the following steps:

```
1.      createQ.qDropNonexistingDeleted()
2.      If (    (createQ.qIsEmpty())
             || (no space to add one VarCreate)) then
            stop processing, return empty IETYPE-CREATE-VARIABLES element
3.      While (    (createQ.qIsEmpty() == false)
                && ((createQ.qPeek()).countCreate == 1)
                && (enough space to add one VarCreate))
            Let firstVarId = createQ.qTake()
                firstVar   = RTDB.lookup(firstVarId)
            Add VarCreate (off firstVar entry) to IETYPE-CREATE-VARIABLES element
4.      If (    (createQ.qIsEmpty() == true)
             || (no space to add one VarCreate)) then
            stop processing, return the constructed IETYPE-CREATE-VARIABLES element
5.      Let firstVarId = createQ.qTake()
            firstVar   = RTDB.lookup (firstVarId)
6.      Decrement firstVar.countCreate
7.      Add VarCreate (off firstVar entry) to IETYPE-CREATE-VARIABLES element
8.      createQ.qAppend(firstVarId)
9.      While (    (createQ.qIsEmpty() == false)
                && (createQ.qPeek() =/= firstVarId)
                && (enough space to add one VarCreate))
            Let nextVarId = createQ.qPeek()
                nextVar   = RTDB.lookup(nextVarId)
            createQ.qTake()
            Add VarCreate (off nextVar entry) to IETYPE-CREATE-VARIABLES element
            Decrement nextVar.countCreate
            if (nextVar.countCreate > 0) then
                createQ.qAppend(nextVarId)
10.     Return the constructed IETYPE-CREATE-VARIABLES element
```

The preceding procedure adds VarCreate records to the IETYPE-CREATE-VARIABLES information elements as long as there are elements available that have not already been added to this element and that are small enough to fit into the remaining space. The repetition counter for each element is decremented, and the element is appended again if it is still larger than zero.

8.4.3 Composing the IETYPE-UPDATES Element

This follows almost exactly the same process as for the IETYPE-CREATE-VARIABLES element, with the following substitutions:

- Replace createQ by updateQ
- Replace IETYPE-CREATE-VARIABLES by IETYPE-UPDATES

- Replace `countCreate` by `countUpdate`
- Replace `VarCreate` by `VarUpd`

8.4.4 Composing the `IETYPE-DELETE-VARIABLES` Element

This follows almost exactly the same process as for the `IETYPE-CREATE-VARIABLES` element, with the following substitutions:

- Replace `createQ` by `deleteQ`
- Replace `IETYPE-CREATE-VARIABLES` by `IETYPE-DELETE-VARIABLES`
- Replace `countCreate` by `countDelete`
- Replace `VarCreate` by `VarId`
- Replace `qDropNonexistingDeleted()` by `qDropNonexisting()`

Furthermore, when `countDelete` counts down to zero, then the variable should be completely removed from the RTDB.

8.4.5 Composing the `IETYPE-REQUEST-VARUPDATES` Element

The process of composing the `IETYPE-REQUEST-VARUPDATES` information element includes the following steps:

```
1.     reqUpdQ.qDropNonexistingDeleted()
2.     If (    (reqUpdQ.qIsEmpty())
            || (no space to add one VarSumm)) then
           stop processing, return empty IETYPE-REQUEST-VARUPDATES element
3.     While (    (reqUpdQ.qIsEmpty() == false)
               && (enough space to add one VarSumm))
           Let nextVarId = reqUpdQ.qTake()
           Add VarSumm for nextVarId to IETYPE-REQUEST-VARUPDAES element
4.     Return the constructed IETYPE-REQUEST-VARUPDATES element
```

The preceding procedure effectively adds summaries to the `IETYPE-REQUEST-VARUPDATES` element as long as the queue contains (existing) variables and there is enough space. Note that a request for a VarUpd is only transmitted once.

8.4.6 Composing the `IETYPE-REQUEST-VARCREATES` Element

This follows exactly the same process as for the `IETYPE-REQUEST-VARUPDATES` element, with the following substitutions:

- Replace `reqUpdQ` by `reqCreateQ`
- Replace `IETYPE-REQUEST-VARUPDATES` by `IETYPE-REQUEST-VARCREATES`
- Replace `VarSumm` by `VarId`

8.5 Initialization, Runtime and Shutdown

VarDis will have to perform certain actions at runtime besides processing received packets, preparing beacon payloads or responding to service requests from

applications.

8.5.1 Initialization

During initialization, VarDis will need to register itself as a client protocol with the underlying BP. To achieve this registration, VarDis prepares a `BP-RegisterProtocol.request` service primitive (see Service BP-RegisterProtocol) with the following parameters:

- `protId` is set to `BP_PROTID_VARDIS`.
- `name` is set to "VarDis – Variable Dissemination Protocol Vx.y" where 'x' and 'y' refer to the present version of VarDis.
- `maxPayloadSize` is set to the value of the `VARDISPAR_MAX_PAYLOAD_SIZE` configuration parameter.
- `queueingMode` is set to `BP_QMODE_ONCE`.

Furthermore, VarDis performs the following additional actions during initialization:

- The real-time database `RTDB` is initialized to be empty, i.e. not holding any variables.
- All queues (`createQ`, `deleteQ`, `updateQ`, `summaryQ`, `reqUpdQ`, `reqCreateQ`) are initialized to be empty.

8.5.2 Runtime

The default process by which VarDis generates payloads to transmit is of a reactive nature: VarDis waits until the underlying BP has indicated that a payload has just been included in a beacon (see Section Service BP-PayloadTransmitted), upon which VarDis receives the `BP-PayloadTransmitted.indication` primitive. In response to this primitive, VarDis then generates the next payload by following the process described in Section Payload Construction, and submitting the resulting payload (if any) to the BP by invoking the `BP-TransmitPayload.request` primitive as follows:

- The `protId` parameter is set to `BP_PROTID_VARDIS`.
- The `length` parameter is set to the length of the new payload in bytes.
- The `value` parameter is set to be the new payload.

However, an additional process is needed in case a `BP-PayloadTransmitted.indication` is being lost or in order to initiate the transmission of the very first payload. To achieve this, VarDis checks periodically (with the period given by the `VARDISPAR_BUFFER_CHECK_PERIOD` parameter, see Section Configurable Parameters) the number of payloads that the BP currently holds for VarDis. It does so by invoking the `BP-QueryNumberBufferedPayloads.request` service primitive with the `protId` parameter set to `BP_PROTID_VARDIS`. If the value received in the `BP-QueryNumberBufferedPayloads.response` primitive is valid and equal to zero, then VarDis generates a beacon payload and, if there is any, hands it

over to the BP by submitting a `BP-TransmitPayload.request` primitive as above.

8.5.3 Shutdown

When the operation of VarDis is ending, VarDis performs at least the following actions:

- It de-registers itself as a client protocol with BP, by preparing a `BP-DeregisterProtocol.request` primitive with the `protId` parameter set to `BP_PROTID_VARDIS`.
- The real-time database `RTDB` is completely cleared.
- All queues (`createQ`, `deleteQ`, `updateQ`, `summaryQ`, `reqUpdQ`, `reqCreateQ`) are set back to be empty.

8.6 Transmit Path

The process ensuring that payloads are being generated at appropriate times is described in Section Runtime.

8.7 Receive Path

We discuss how a node (the "current node") processes the VarDis payload of a beacon received from a neighbour (the "neighbour node") and handed to VarDis by the BP.

8.7.1 Processing Order of Information Elements

The received VarDis payload contains a number of information elements of the various types introduced before (Section Information Elements). As a very first step, the current node extracts all information elements from the payload, and checks for each information element whether its type is one of the types of information elements supported by the implementation. If not, the information element is silently discarded.

The valid information elements contained in the payload must be processed in the following order:

- First step: process `IETYPE-CREATE-VARIABLES` elements.
- Second step: process `IETYPE-DELETE-VARIABLES` elements.
- Third step: process `IETYPE-UPDATES` elements.
- Fourth step: process `IETYPE-SUMMARIES`, `IETYPE-REQUEST-VARCREATES` and `IETYPE-REQUEST-VARUPDATES` in any convenient order.

The rationale for processing `IETYPE-CREATE-VARIABLES` and `IETYPE-DELETE-VARIABLES` elements before `IETYPE-UPDATES` is that a producer node might in the same payload not only include the information element for creating the variable, but also an update for that same variable. However, as updates are only allowed

to be carried out for existing variables, the 'Create' instruction should be processed first, otherwise the update will be rejected. For a similar reason `IETYPE-UPDATES` elements should be processed after `IETYPE-DELETE-VARIABLE` elements, as updates are not carried out while a variable is in the process of deletion.

Furthermore, it is also helpful to process `IETYPE-UPDATES` before `IETYPE-SUMMARIES`. To see why, imagine that a neighbour includes both a summary and an update for the same variable in the same payload, showing the same sequence number. Assume that the receiver only possesses an older sequence number at the time of reception. If the receiver were to process the summary first, it would notice that it only has that variable in an outdated version and might itself instigate transmission of a `IETYPE-REQUEST-VARUPDATES` element for that variable. But that would be un-necessary.

Finally, while the receive path extracts the information elements out of the received payload, it needs to perform a range of checks on the IE's, including the check of whether the received IE is of known type and, for those IE types where the payload is a list of elements of a given data type `T`, whether the `ieLen` field has a value that is an integer multiple of `sizeof(T)`. If any of these sanity checks for an information element fails, then the element shall be silently dropped and processing continues with the next element, if any.

8.7.2 Processing `IETYPE-CREATE-VARIABLES` Elements

The contents of an `IETYPE-CREATE-VARIABLES` information element is a list of `VarCreate` records. For a specific such record `rcvdVC` of type `VarCreate` the receiving VarDis instance performs the following steps:

```
1.     Let spec : VarSpec    = rcvdVC.spec
           upd : VarUpd      = rcvdVC.upd
           varId : VarId     = spec.varId
2.     If (RTDB.lookup (varId) == true) then
          stop processing
3.     If (spec.prodId == own node id) then
          stop processing
4.     Let newent : DBEntry with
           newent.spec         =  spec
           newent.length       =  upd.length
           newent.value        =  upd.value
           newent.seqno        =  upd.seqno
           newent.tStamp       =  current system time
           newent.countUpdate  =  0
           newent.countCreate  =  spec.repCnt
           newent.countDelete  =  0
           newent.toBeDeleted  =  False
5.     RTDB.update(newent)
```

```
6.      createQ.qAppend(varId)
7.      summaryQ.qAppend(varId)
8.      reqCreateQ.qRemove(varId)
```

8.7.3 Processing IETYPE-DELETE-VARIABLES Elements

The contents of an IETYPE-DELETE-VARIABLES information element is a list of VarId values. For each included VarId value varId the receiving VarDis instance performs the following steps:

```
1.      If (RTDB.lookup(varId) == false) then
            stop processing
2.      Let ent = RTDB.lookup(varId)
3.      If (ent.toBeDeleted == True) then
            stop processing
4.      If (ent.spec.prodId == own node id) then
            stop processing
5.      ent.toBeDeleted = True
6.      ent.countUpdate = 0
7.      ent.countCreate = 0
8.      ent.countDelete = ent.spec.repCnt
9.      RTDB.update(ent)
10.     updateQ.qRemove(varId)
11.     createQ.qRemove(varId)
12.     reqUpdQ.qRemove(varId)
13.     reqCreateQ.qRemove(varId)
14.     summaryQ.qRemove(varId)
15.     deleteQ.qRemove(varId)
16.     deleteQ.qAppend(varId)
```

8.7.4 Processing IETYPE-UPDATES Elements

The contents of an IETYPE-UPDATES information element is a list of VarUpd records. For each included VarUpd record upd the receiving VarDis instance performs the following steps:

```
1.      If (RTDB.lookup(upd.varId) == false) then
            If (reqCreateQ.qExists(upd.varId) == false) then
                reqCreateQ.qAppend(upd.varId)
            stop processing
2.      Let ent = RTDB.lookup(upd.varId)
3.      If (ent.toBeDeleted == True) then
            stop processing
4.      If (ent.spec.prodId == own node id) then
            stop processing
5.      If (upd.seqno == ent.seqno) then
            stop processing
```

```
6.      If (upd.seqno is strictly older than ent.seqno) then
            if (updateQ.qExists(upd.varId) == false) then
                updateQ.qAppend(upd.varId)
                ent.countUpdate = ent.spec.repCnt
                RTDB.update(ent)
            stop processing
7.      ent.length        =  upd.length
8.      ent.value         =  upd.value
9.      ent.seqno         =  upd.seqno
10.     ent.tStamp        =  current system time
11.     ent.countUpdate   =  ent.spec.repCnt
12.     RTDB.update(ent)
13.     if (updateQ.qExists(upd.varId) == false) then
            updateQ.qAppend(upd.varId)
14.     reqUpdQ.qRemove(upd.varId)
```

8.7.5 Processing IETYPE-SUMMARIES Elements

The contents of an IETYPE-SUMMARIES information element is a list of VarSumm records. For each included VarSumm record summ the receiving VarDis instance performs the following steps:

```
1.      If (RTDB.lookup(summ.varId) == false) then
            If (reqCreateQ.qExists(summ.varId) == false) then
                reqCreateQ.qAppend(summ.varId)
            stop processing
2.      Let ent = RTDB.lookup(summ.varId)
3.      If (ent.toBeDeleted == true) then
            stop processing
4.      If (ent.spec.prodId == own node id) then
            stop processing
5.      If (summ.seqno == ent.seqno) then
            stop processing
6.      If (summ.seqno is strictly older than ent.seqno) then
            if (updateQ.qExists(summ.varId) == false) then
                updateQ.qAppend(summ.varId)
                ent.countUpdate = ent.spec.repCnt
                RTDB.update(ent)
            stop processing
7.      if (reqUpdQ.qExists(summ.varId) == false) then
            reqUpdQ.qAppend(summ.varId)
```

8.7.6 Processing IETYPE-REQUEST-VARUPDATES Elements

The contents of an IETYPE-REQUEST-VARUPDATES information element is a list of (VarId, VarSeqno) pairs. For each such pair (varId, seqno) the receiving VarDis instance performs the following steps:

```
1.      If (RTDB.lookup(varId) == false) then
            stop processing
2.      Let ent = RTDB.lookup(varId)
3.      If (ent.toBeDeleted == True) then
            stop processing
4.      If (ent.seqno is not strictly larger than seqno) then
            stop processing
5.      ent.countUpdate  =  ent.spec.repCnt
6.      RTDB.update(ent)
7.      if (updateQ.qExists(varId) == false) then
            updateQ.qAppend(varId)
```

8.7.7 Processing `IETYPE-REQUEST-VARCREATES` Elements

The contents of an `IETYPE-REQUEST-VARCREATES` information element is a list of `VarId` values. For each such value `varId` the receiving VarDis instance performs the following steps:

```
1.      If (RTDB.lookup(varId) == false) then
            stop processing
2.      Let ent = RTDB.lookup(varId)
3.      If (ent.toBeDeleted == true) then
            stop processing
4.      ent.countCreate  =  ent.spec.repCnt
5.      RTDB.update(ent)
6.      If (createQ.qExists(varId) == false) then
            createQ.qAppend(varId)
```

---

9 Known Issues, Shortcomings and Comments

9.1 Beaconing Protocol

9.1.1 Known Issues

- The interface does not yet allow BP to signal memory shortage issues, e.g. when a client protocol has chosen a queue as buffering mode and no more payload can be added to the queue.

9.1.2 Potential Future Features

- Allow the BP to dynamically change the allowable beacon size, e.g. to respond to congestion.
- When there are several higher-layer protocols and their combined payload exceeds the maximum beacon size, we might wish to be able to statically

prioritize higher layer protocols, to make sure the higher-priority payloads are always included. Alternatively we could go for round-robin.

- Offer a management facility to enable and disable the beacon transmission process.
- There is no runtime liveness checking of client protocols and no proper cleanup. That could be a soft-state mechanism for registered protocols, together with additional status codes
- Add something like a 'network identifier' to the BP header, so that we can reliably separate two swarms in close vicinity

9.2 SRP

9.2.1 Potential Future Features

- Offer a management facility to activate and deactivate the SRP
- The sequence numbers in the extended state records / neighbour table entries can potentially be used to form a packet loss rate estimate.

9.3 VarDis

9.3.1 Known Issues

- The behaviour when two different nodes want to create variables with the same VarId is not yet convincing, particularly when this happens at about the same time on two far-away nodes, so that neither creator knows the intention of the others.
- The VarDis interface does not allow to signal any memory management issues (e.g. when it is running out of storage when adding new variables)
- Unclear what should happen when a new VarCreate arrives for a variable that is currently in the process of being deleted on a consumer node. That could happen when the producer follows a delete immediately with another create. Right now a receiving node will ignore the new create until the variable has completely disappeared. Would we need sequence numbers for variable existence periods?
- Changing the `VARDISPAR_MAX_PAYLOAD_SIZE` parameter at runtime might require a new registration with BP (which is not currently done), as currently this is the only way for client protocols to hand down their max payload size to BP.
- It is unclear whether it is necessary to call `qDropNonexisting` whenever an information element is created for a payload.
- When receiving and processing a variable summary and noticing that the summary advertises a more recent sequence number, currently the following

update-request includes the older sequence number of the receiving node, not the more recent sequence number received in the summary. That may lead to too many outdated updates being sent by neighbours.

9.3.2 Potential Future Features

- The protocol does not contain any mechanism to allow a consumer check the liveness of the producer of a variable and remove variables for which the producer has died (though a 'dead variable' does not require much bandwidth). We could add a separately configurable lifetime or timeout to each variable specification, such that if a node does not hear from the producer for that amount of time then the variable is dropped. A timeout could be infinite to forbid dropping. When a node drops a variable, it might flood that information into the network to force others to drop it as well, but that could be problematic if a disconnected node drops everything and then reconnects, asking everyone else to drop variables.
- A more bandwidth-efficient summary mechanism, in particular for variables with low update rates.
- Add support for multiple writers to the same variable, who possibly write concurrently. When very precise timestamps are available, these could be included in updates and precedence being decided based on them.
- Transmit update and create requests more than once. And include in such requests the node identifier of the preferred sender (presumably the node from which we have learned the need to get the updates), to avoid having the entire neighbourhood responding.
- Introduce separate information element types for variables with fixed length (only included in their creation record). For such fixed-length variables the IE header does not need to include length information, but if a node receives such a fixed-length IE for a variable it does not yet know about, the remainder of the packet may become unparse-able.
- Include a mechanism that allows a node not to include recent updates when it has overheard them often enough (the update will still be reflected in the summaries), or to make a probabilistic decision as to whether or not to include updates.
- Create a separate `MAX_SIZE` type of parameter for each of the information element types.
- Single-writer scenario: when a consumer node realizes that the same VarId has been created by different nodes, it could itself flood an "Abort" message into the entire network, deleting both instances of the variable.
- Consider the handling of timestamps, allow for updates to carry the timestamp of the producer and have consumers being able to use that timestamp and their local timestamp to tell propagation time

- While responding to `RTDB-Create.request` and `RTDB-Update.request` primitives we currently stop procesing when `length`. We could allow this to indicate a NULL value.
- When a node is switched on lately, it will have to learn about all variables and their values, and the current mechanism (through summaries and the occasional update) might be slow. A database synchronization procedure like in OSPF could be added, provided this can be more efficient.
- Add services and logic to re-initialize and suspend / resume VarDis. During suspension no service requests shall be accepted and no beacons be transmitted or received(?).