WDM Next Protocol Specification

v0.4.11 3/8/2017

1. Change history

Version	Date	Notes
0.1	1/13/2016	Initial draft
0.2	1/30/2016	1. Change name to WDM Next. 2. Add data model section. 3. Add more description of error cases, add flow for queued events, address other review comments. 4. Merge message format into this document.
0.2.1	2/2/2016	1. Added a requirement for the publisher to send Update response only after it has seen Notification responses for all important traits. 2. Change Publisher State to Subscription State according to review comments.
0.2.2	2/8/2016	1. No Subscribe Confirm from Publisher to Client 2. Mandatory timeout for Subscription request and response
0.2.3	2/10/2016	Revert timeout for Subscription request and response to become optional
0.2.4	2/17/2016	1. Added warning that the one way subscription state could be removed soon. 2. Added stronger statement about Subscription response bring clients to reasonable alignment.
0.3.0	2/19/2016	1. Change name from WDMDataElementData to WDMDataElementMergeData. 2. Remove Subscription State message because now it belongs to another Weave profile TBD. WARNING: We're about to add two flags in DataElement and add more description about their use in View responses and Notification requests.
0.3.1	2/19/2016	Remove TLV Array as an option to be the root of WDM data model
0.4.0	4/8/2016	1. Remove support for queued traits, hence removed `WDMDataElementDiscontiguous` 2. Add event delivery feature. Add chapter and reference to explain event delivery. Modified Subscribe Request, Subscribe Response, and Notification Request to support events.

		3. Notification rejection has been changed to not use data elements, and not conveying the current data on the client. 4. Change all profile tags to context specific tags, and mandate encoding of them to be in “tag order”
0.4.1	4/25/2016	Add SystemTimestamp, rename Timestamp to UTC timestamp.
0.4.2	4/26/2016	1. Address review comments 2. Enforce implicit profile in TLV to be set to Dictionary Key profle 3. Explicitly allow unknown TLV tags in Events
0.4.3	5/6/2016	1. Remove `DataElementReasonForRejection` 2. Add support for version update but empty data in notification requests. Note: Excessive event related material will be removed shortly as they will be available in event-specific documents [5]
0.4.4	5/23/2016	LastVendedEventIdList removed from SubscribeRequest.
0.4.5	5/26/2016	A few tag definitions for events in Subscribe Request added.
0.4.6	6/6/2016	Make path list optional in subscribe requests, for some clients might just need events
0.4.7	6/29/2016	1. Replace security token with WDM:Authenticator 2. Create specific messages for custom command request and response 3. Add new encoding and parsing rules for profile tags 4. Add link to WDM Request Authentication [7]
0.4.8	7/11/2016	Added definition for command response to include version and wrapper for custom-defined response information.
0.4.9	9/30/2016	Introduced non-backward-compatible change to revert both polarity and name for tag DataElementLastForCurrentChange to DataElementPartialChange Removed the change abortion feature.
0.4.10	2/14/2017	Introduced support for deleting dictionary elements.
0.4.11	3/8/2017	Change context tag value DataElementDeletedDictionaryKeyList from 11 to 9, so it would always appear in front of the DataElementMergeData tag in a properly sorted message. Section 6.4 has also been modified to allow both to present.

2. Overview

This document describes the protocol and message format in Weave Data Management (WDM).

The most common interactions among devices and with Nest services are through the WDM protocol. The WDM protocol manages real-time data publication, command processing, and eventing among network hosts.

These data are specified in predefined schemas as traits. Traits are logical groupings of state, events, and commands for specific aspects of an application plane.

2.1. Related documents

These additional documents must be read for a full view of the protocol that is used in current Nest products:

1) [Trait Design Guidelines] (x-ref) for this generation of Nest products (TBD but must contain critical information/constraints for interoperation with the products)
2) [Weave Interface Define Language] (x-ref) (TBD)

2.2. Diagram legends

Many diagrams in this document, especially sequence diagrams, use the legends defined in [UML 2.0](http://www.omg.org/spec/UML/2.0/). The most commonly used legends are described below.

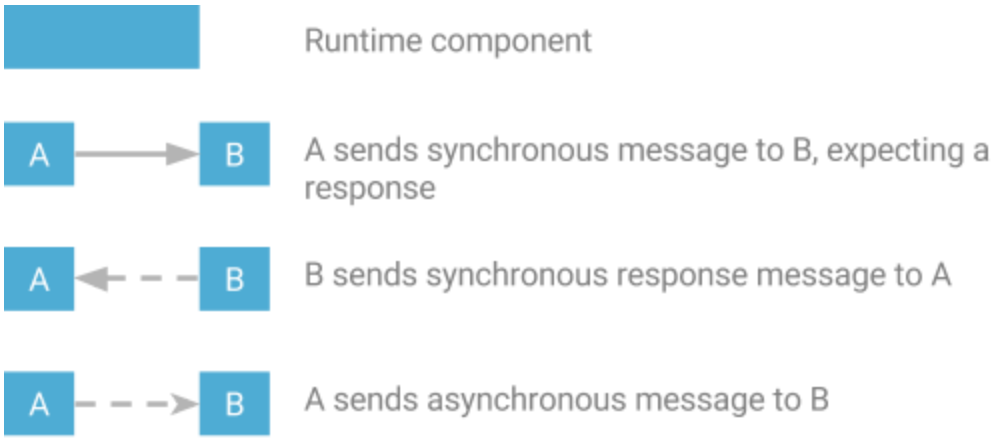

Figure 1. Legends used in many diagrams

2.3. Weave type-length-value (TLV) tuples

- Path type: **< >**
- Structure type: **{ }**
- Dictionary type: **{ }**
- Array type: **[ ]** Note that all elements contained by an array must have `AnonymousTag`.
- Tag **=** Description of value. Should begin with either “Optional” or “Mandatory”
- Key
- **/*** comments ***/**

3. Architecture overview and typical deployment

Figure 2. Basic components in the WDM protocol

WDM is a state synchronization protocol that is optimized to efficiently align a set of clients with the publisher.

A publisher has a number of data sets called trait instances that can be changed by internal and external causes. The publisher is responsible for keeping the data secure, responding to various requests, and sending notifications to clients when data is changed.

Clients can issue “view” commands to see the latest/current version of any data, “update” commands to change some of the data, “subscribe” commands to maintain long-term data streams and get notified about changes, or a custom WDM command using all the flexibility that Weave provides.

Precisely preserving every change is not a major goal of WDM; hence, changes are allowed to be collapsed together. Redundant data could be sent to save memory space on the publisher. On the other hand, the protocol emphasizes the eventual consistency on data among clients subscribing to the same publisher, and the efficiency for both over-the-wire data size and on-device memory space.

3.1. Resources, trait profiles, and trait instances

A resource is a grouping of trait instances and interfaces that represent a logical or physical entity, such as a device, structure, or user. The resource fully describes the capabilities and behaviors of the entity it represents. The same resource definition can be implemented by a Weave device, or by a logical entity in the sense of cloud services. The resource ID can be the Weave node ID for a device or the structure ID for the cloud service.

If not specified, the resource ID is the Weave node ID of the route target Weave node.

A trait is like a class definition in object-oriented programming languages, while a trait instance is like an object instance of the class. There can be multiple instances of the same trait, like multiple temperature sensors of the same type, on a resource.

3.2. Versions

For each trait instance, there is an associated version observable to its clients. To a client, the version number can be compared for order, as with unsigned integers, but the internal format or meaning in the difference between version numbers is opaque. This loose definition enables a publisher to use timestamps and other mechanisms for version numbers.

There is a hard requirement for the WDM publisher to be able to maintain the total order among all version numbers it has generated for a trait instance, even across system reboots. This is needed to guarantee correctness of data synchronization.

3.3. Weave TLV and WDM adoption

Weave TLV [4] is used in many places within WDM. Most message payloads, including change records sent over the wire, are encoded in this format. The data hosted on publishers is also modeled using this format. This means that although the data itself may or may not be stored in a specific encoding, all the operations have to be expressible with the WDM adoption of Weave TLV. Weave TLV is flexible in what can be contained and in what order, but WDM provides some guidelines and rules.

These rules apply to both message payloads and the data hosted on publishers:

1. Order in a TLV container is relevant. Unless otherwise described, all elements must appear in a fixed order as specified in this specification.
2. Unless explicitly marked as *extensible* in the documentation, an unrecognized tag or incompatible data type in message payloads is considered a protocol error. This implies we do not intend to extend the protocol by just adding new tags into existing message payloads.

There are further modeling restrictions for data hosted on publishers, which can be found in later sections and in [Trait Design Guidelines] (x-ref).

3.4. Transport and exchange context

WDM supports TCP and the Weave Reliable Messaging (WRM) protocol.

When TCP is used, all messages in the same connection are inherently serialized.

When WRM is used, the next message in the same exchange context cannot be sent until acknowledgement for the previous message is received.

A notification request can only be sent out after a publisher has received a response for the previous notification request from the client. Normally each notification request initiates its own exchange context, but in the subscribe response scenario, all notification requests are in the same exchange context as the subscribe request and subscribe response.

In all cases, the WDM protocol has an explicit response message for each request.

3.5. Security

The WDM protocol relies on Weave security for secrecy as well as authenticity and integrity checks. Additional layers of security can be added to further verify end-to-end security if part of the network topology doesn't communicate with Weave. More detail can be found in [7].

- Important information in a request message from a client is signed with a known key. At most, one authenticator is supplied with each request message to be verified at the publisher's side. The accepted authenticators are TBD. TLV tag, data type, and schema used by an authenticator is defined by its associated profile.

The authenticator could be one of these: [TBD]

Whether messages from a publisher, like responses and notifications, should be signed as well is currently being defined. This mechanism doesn't add secrecy on top of Weave security.

4. Model for data hosted on publishers

WDM models the data hosted on a publisher as a Weave TLV tree. The root of the tree must be an anonymous TLV Structure. The use of TLV Array in this data model is discouraged, for there is no way to convey partial data change in the current version of WDM, leading to much less efficient encoding when only part of the array is changed.

4.1. Paths

Path is a special TLV element that is used to point to a specific node in the hosted data. The detailed format and examples can be found in section 6, Message Format. The main limitation for paths in the current WDM implementation is that a path cannot point into any individual element in a TLV array. This limitation enforces the entire array to be treated as a single leaf value in change records sent over the wire.

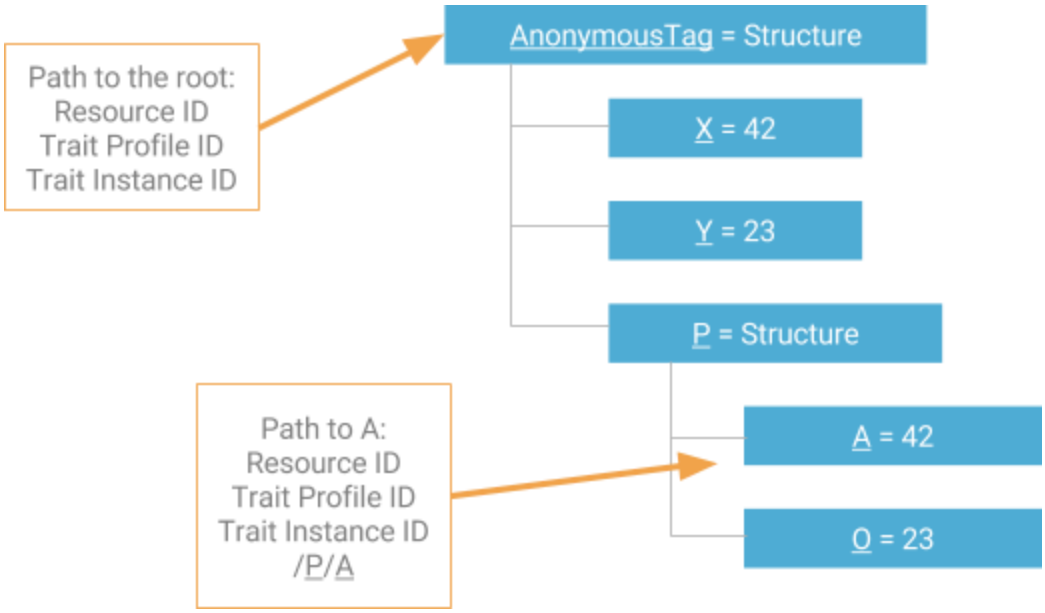

Figure 3. Paths pointing to different subtrees of TLV data

4.2. Dictionary

The dictionary is a WDM-specific concept built on top of the TLV structure container. A plain structure can only have a fixed schema, while a dictionary can have a variable number of elements of some fixed schema. All elements in a dictionary must carry tags under the Dictionary Key profile.

Since there is no marker in Weave TLV to differentiate the tags, a recipient must know the schema of the TLV tree to determine whether a structure is plain or a dictionary.

The root TLV structure cannot be a dictionary because we cannot express deletion of any element from the root. This is discussed in more detail later.

There might be some limitations on the tags of elements contained by a dictionary in certain implementations.

Operations on dictionaries are explored below in section 5.2. Data change model.

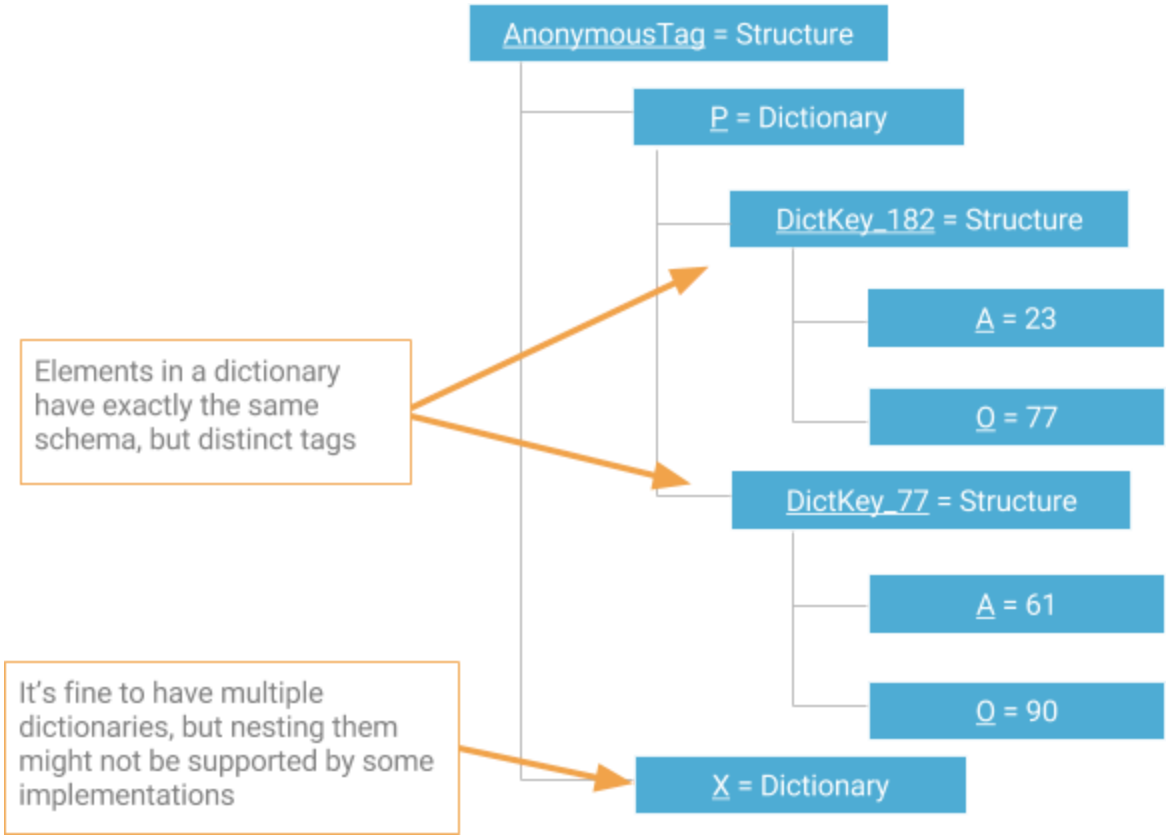

Figure 4. Limitations on dictionaries

5. Supported dialogues

5.1. Overview

A detailed message definition can be found in in Section 6, Message Format.

Table 1: *Supported dialogues*

Dialogue	Use
Subscription	Relatively long-term relationship in which a client is notified when requested data has been changed.
View	Retrieve the latest version of requested data without the burden of

	maintaining a subscription.
Update	Request to modify some data with the option of expiry time and additional context/arguments.
Custom WDM Commands	Request to perform some operation that might have observable WDM effects, with the option of expiry time, additional context/arguments, and a result.

5.2. Data change model

WDM uses several strategies to describe changes to data. The strategies described in this subsection are equally applicable to WDM Notify messages and WDM Update messages. We begin by noting that the path mechanism, along with the structured values, provides multiple ways to address the same information. For example, Figure 5 below, captures two possible encodings of setting the element `/P.A = 57`.

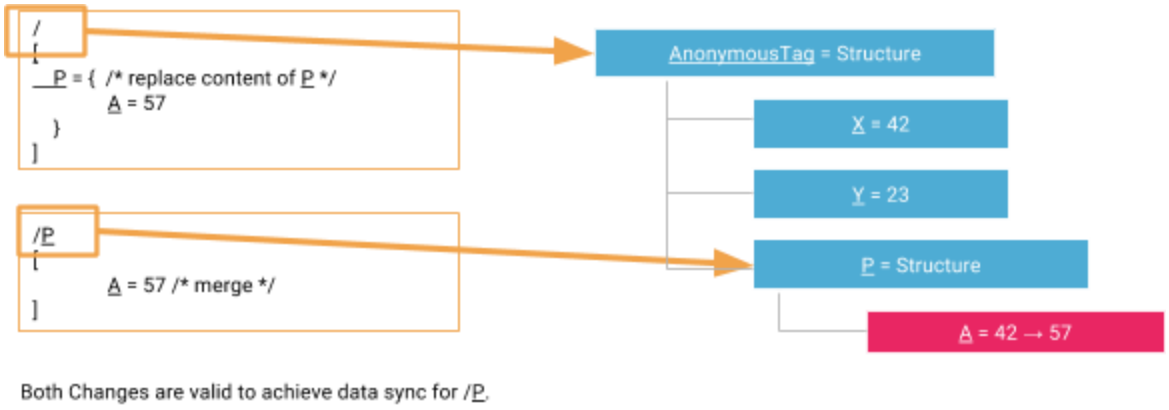

Figure 5: Multiple representations of a change for the same data item

In the first example, the path being changed is `/`, and the value being set is the complete value of structure `P`. In the second example, the path being modified is `/P` and the value being set is the integer `A`. WDM exploits the two different encodings to provide additional flexibility and to minimize the changes transmitted by the protocol.

WDM uses the following terms in dealing with the change operations:

- "*replace*" refers to replacing the existing data on a client's replica with data provided in the change set.
- "*merge*" refers to replacing data with the same tag with new value.

As suggested in Figure 5, the first operation is a *replace*, and the second operation is a *merge*.

WDM provides the semantics called “one-level-merge”: it applies the merge strategy at the first level pointed by the path in a WDM data element. Lower levels are always replaced. Figure 6 below explores the equivalent changes in a more complex structure. Note that in the replace strategy, the change contains not only the changed leaf values, but also the unchanged element (`I`).

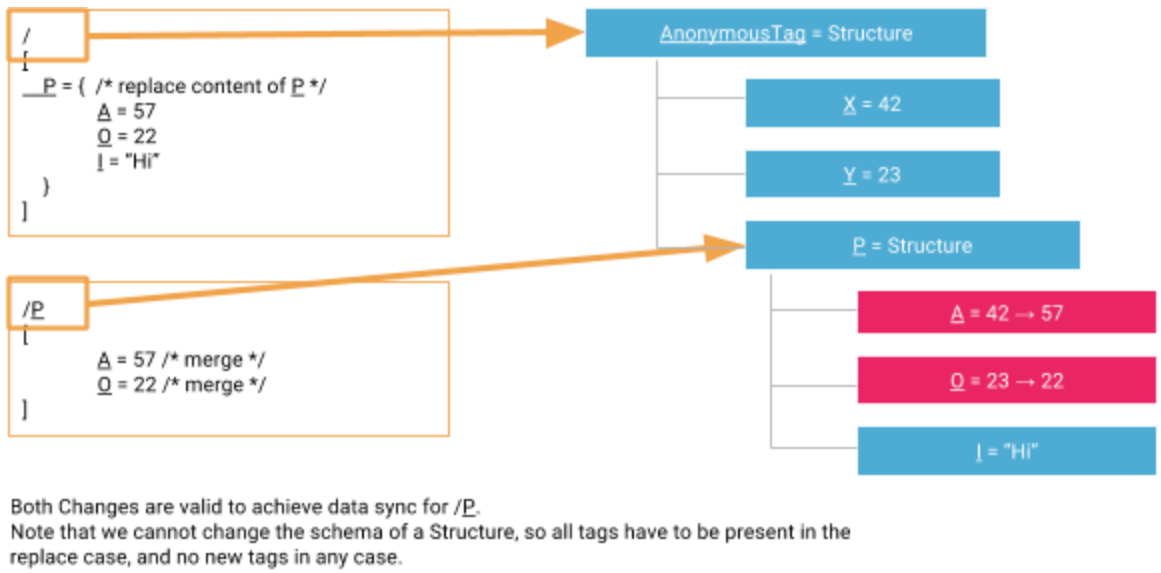

Figure 6: Multiple equivalent representations for a more complex change

This mechanism uses the depth in tree transversal to determine the scheme used, so a client might receive one level higher than what it has subscribed for. This strategy is useful in describing changes that cannot be described using the merge strategy, for example, in situations where the portions of the element subscribed to (or the element in its entirety) are deleted.

5.2.1. Changes in dictionaries

The overarching WDM strategy of one-level-replace applies to both the dictionaries and dictionary entries. When the path in the changeset points to the parent of the dictionary, and the changelist contains the tag of the dictionary, the contents of the dictionary are replaced with the value provided (see Figure 8). When the path points to the dictionary itself, the merge strategy is applied to the dictionary elements: items with new key tags are added into the dictionary and items with existing key tags are replaced with new values. (see Figure 7 below).

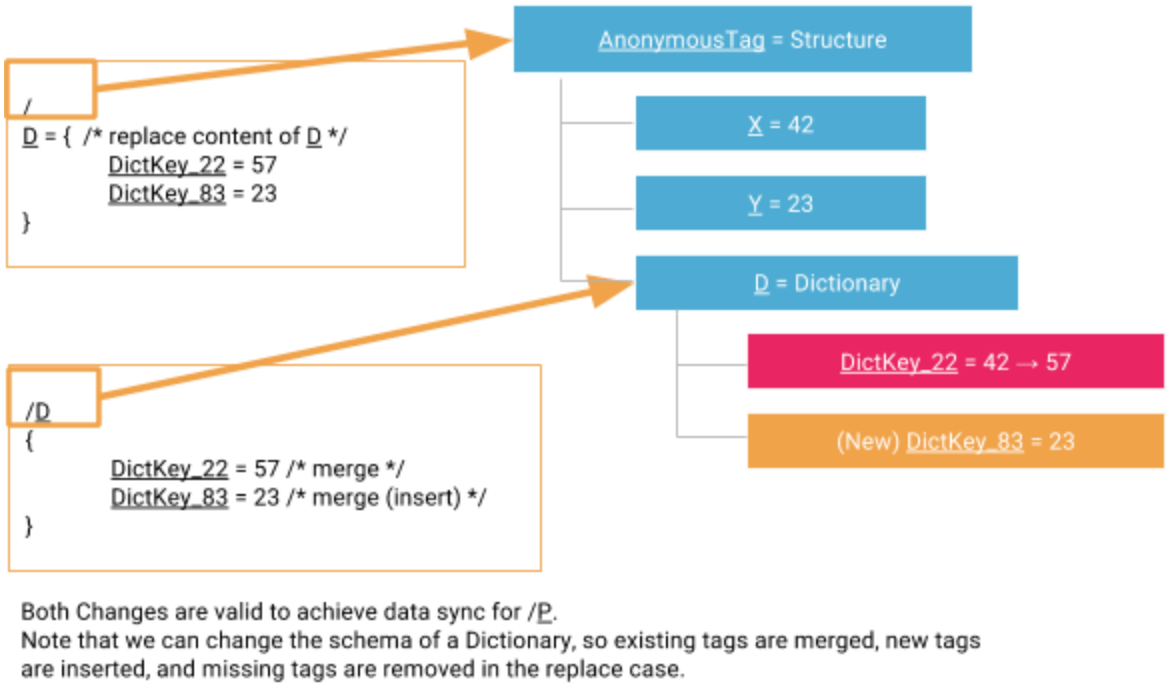

Figure 7: Change the content of a dictionary

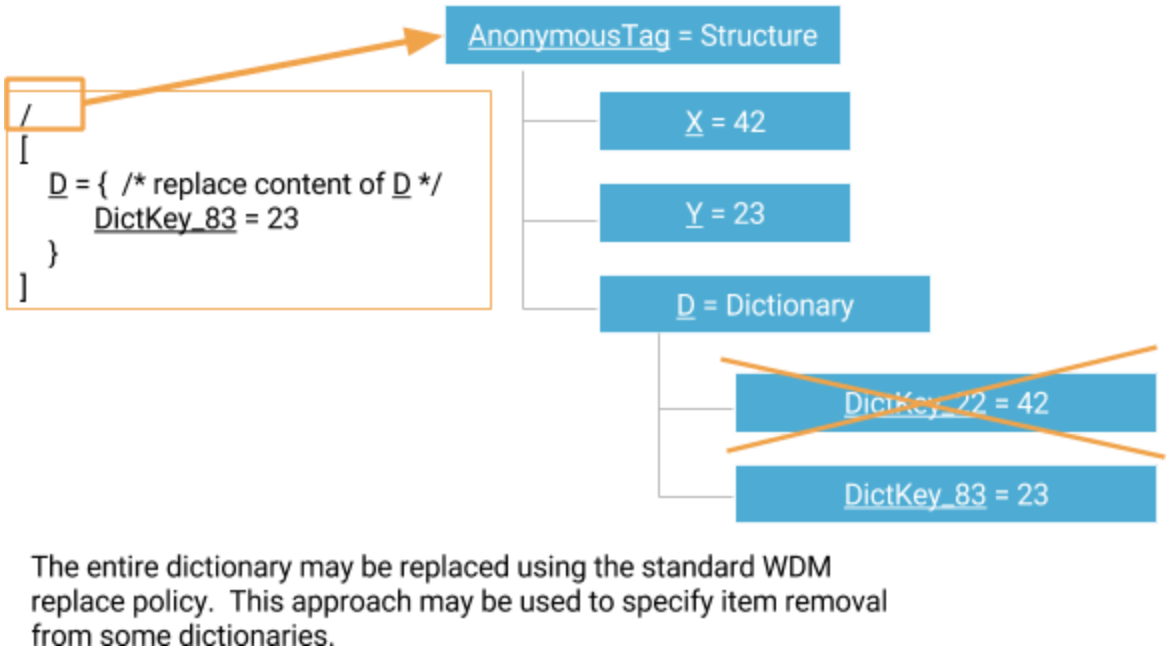

Figure 8: Replacing an entire dictionary.

By their nature, dictionaries store variable collections of data, and the number of items in the dictionary is unspecified in schema. In some cases, the number of elements in the dictionary can be large. Removing an element from a large dictionary would be costly without specialized support: removal of an element cannot be expressed using a merge strategy, and the replace strategy would dictate that the changeset contains all the items remaining in the dictionary. To support the compact dictionary item removal, WDM changeset contains specialized support: instead of the merge list, the changeset for a dictionary path may contain a list of keys to be removed. Figure 9 below shows that operation.

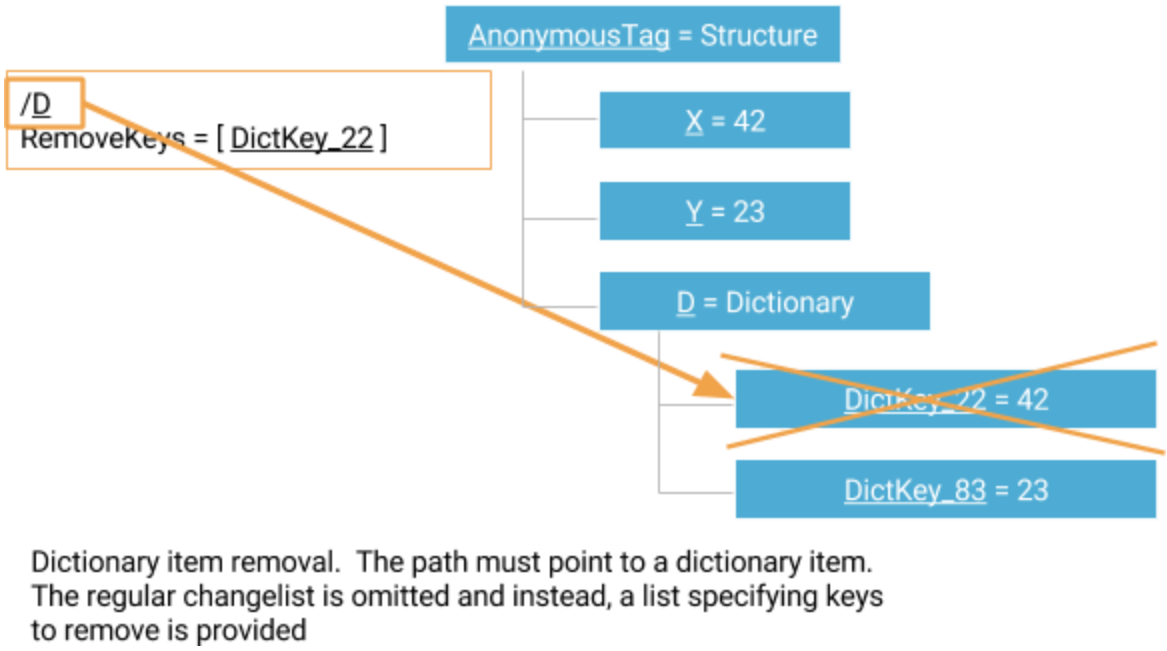

Figure 9: Dictionary item removal

5.2.2. Changes in arrays

As has been noted, TLV arrays are treated as a leaf elements. Any changes in any element, such as the addition and deletion of elements, results in replacing the entire array. There is no merge operation defined for arrays. Figure 9 below demonstrates that operation.

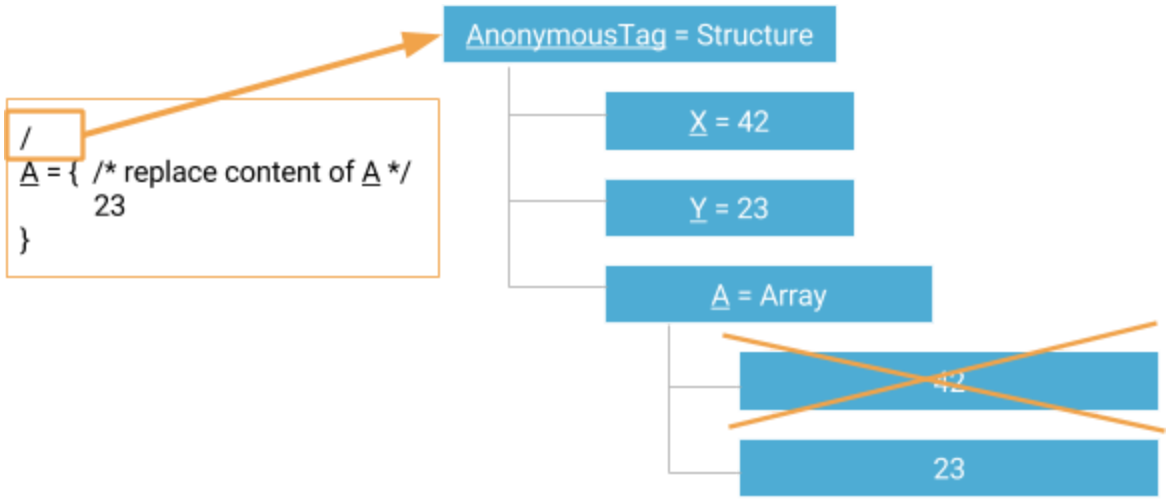

Figure 9: Modify content of an array

5.2.3. Changes in the root path

There is no way to delete elements directly contained by the root container: at the top level only merge operation is permitted and the root element has no parent.

5.3. Subscription

Table 2: *Messages related to WDM subscriptions*

Message	Use
Subscribe Request	Sets up a subscription.
Subscribe Response	Confirms subscription setup is complete.
Notification Request	Sends change records to a client.
Notification Response	Confirms the change record is accepted.
Subscribe Cancel Request	Tears down the subscription.
Subscribe Cancel Response	Confirms subscription is torn down.
Subscribe Confirm Request	Indicates subscription is still alive and asks for confirmation from the peer.

Subscribe Confirm Response	Confirms subscription is still alive.

5.3.1. One way subscription

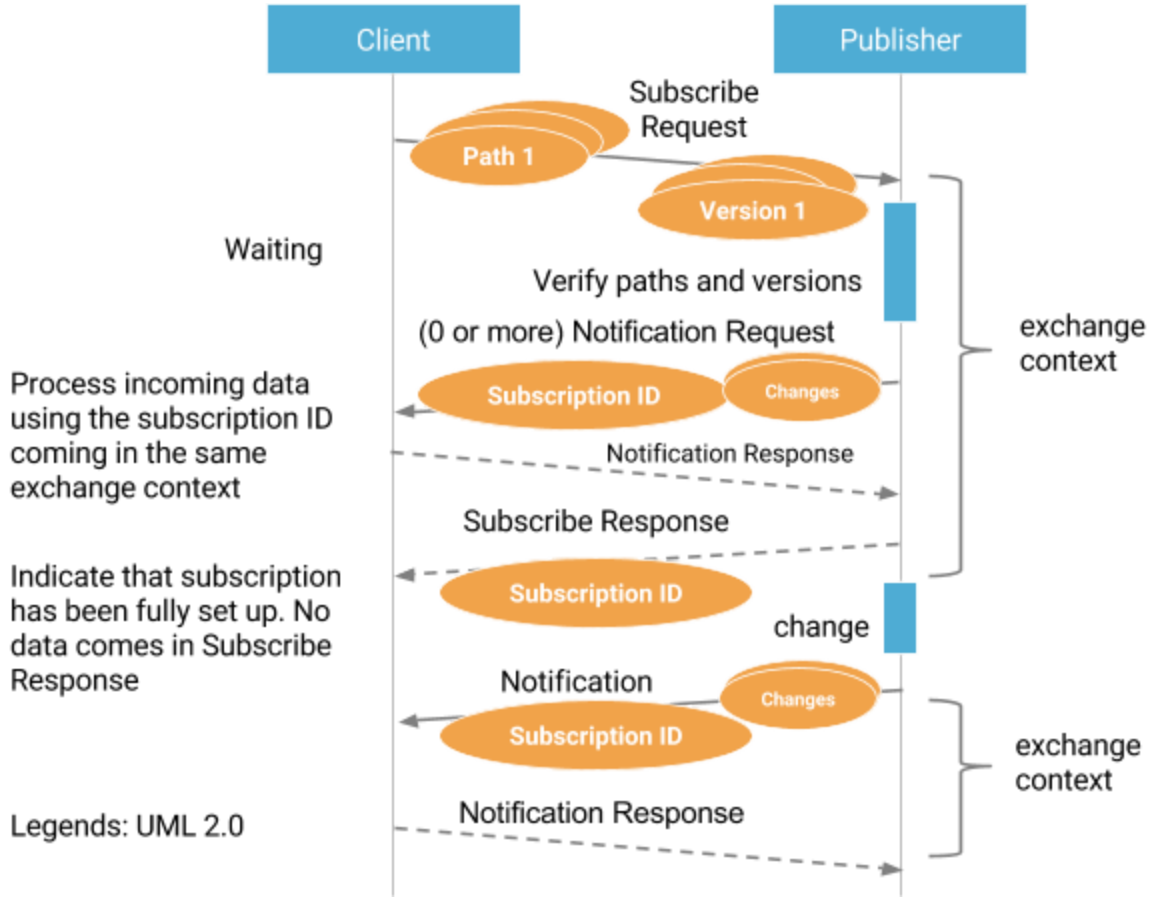

Figure 10: Message flow for one way subscription

A client sends a subscribe request to a publisher with a list of paths the client is interested in and a list of versions the client already has for each path. The publisher either returns a status report on error or starts sending notification requests with data and versions in them followed by a subscribe response.

The notification requests before the subscribe response should bring the client to a reasonably aligned state, which means the publisher believes the client has all current versions.

All notification requests and the final subscribe response contain a subscription ID, which is a number specific to the publisher. The generation mechanism should be designed so the chances for subscriptions on the same publisher having the same ID is very low, even across

system reboots. 64-bit random numbers generated by a secure random number generator are recommended.

All notification requests carry one or more changes, either complete or partial. Changes could be a superset of what has actually been changed, but must be constrained by the paths listed in the original subscribe request.

Once the subscribe request is received, the publisher can start sending notification requests containing the change that needs to be applied to the existing copy of the trait instance data the client already has to maintain consistency.

After the subscribe response is sent or received, the subscription is considered alive and can be torn down through the subscribe cancel request from any party. If something goes wrong on the publisher's side before the subscribe response is sent or received, a Status Report can be sent in response to the original subscribe request.

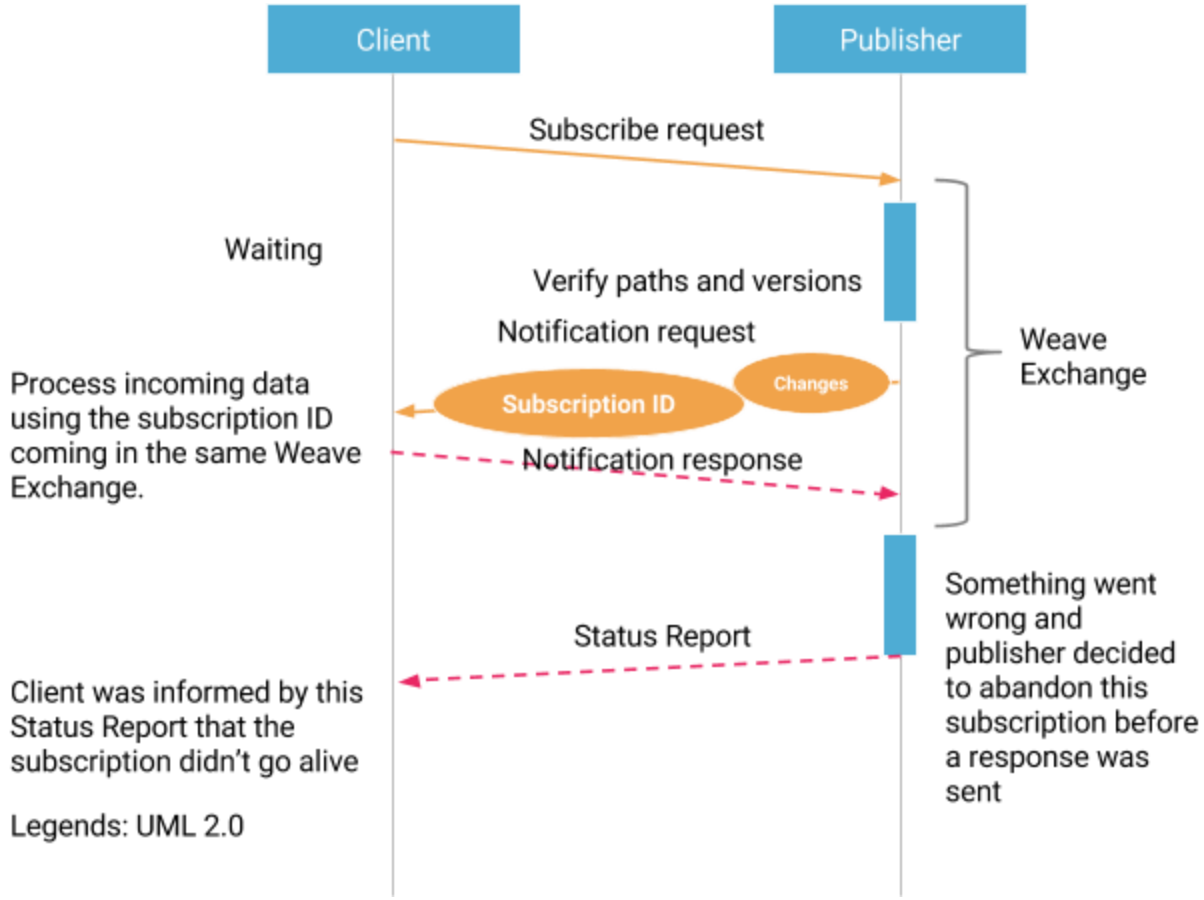

Figure 11: Publisher aborts subscription setup after sending notification requests

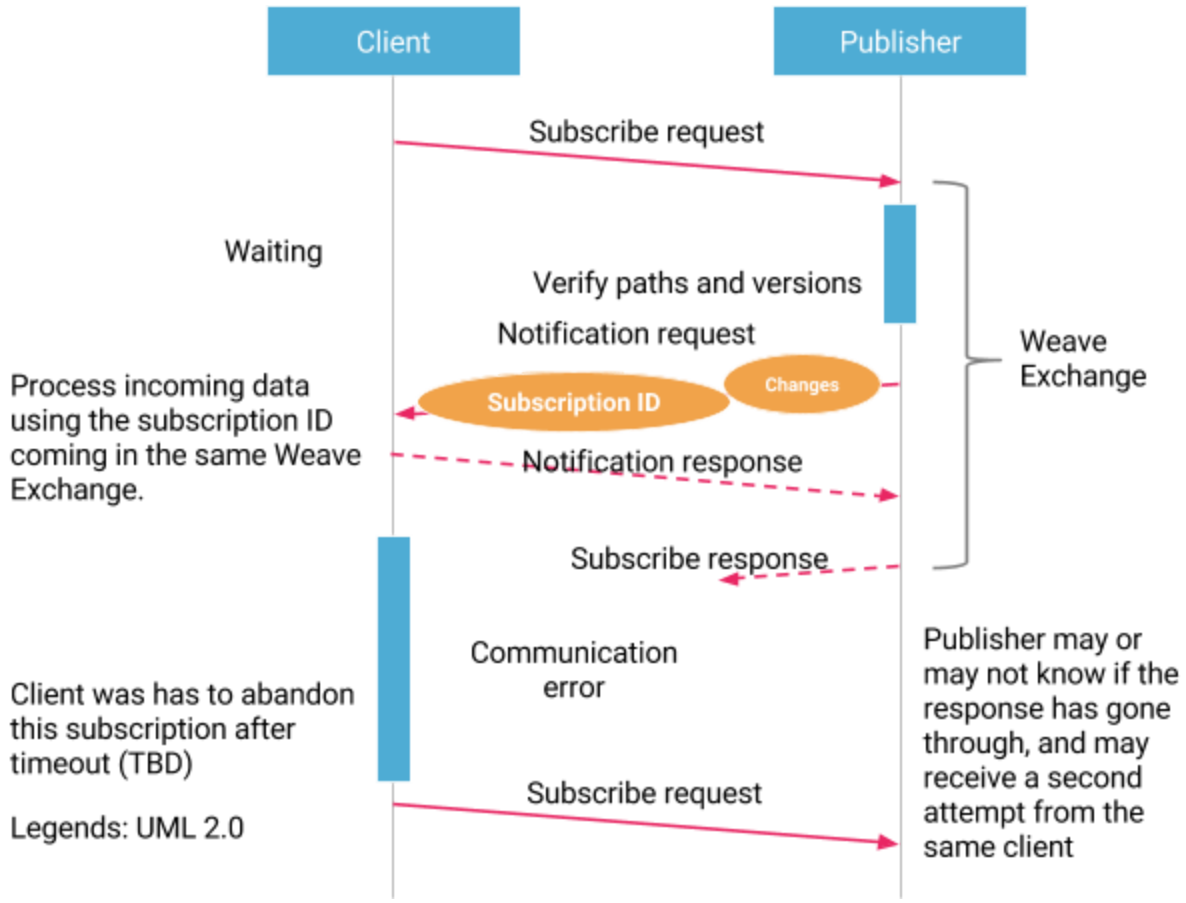

Figure 12: Client recovers from communication error during subscription setup

The time a client has to wait before abandoning a subscription is TBD. It means that any kind of error, including communication difficulties in the early stage of subscription setup, could cause longer delays. For example, a notification request could fail to arrive causing the publisher to abandon the subscription. The client would have to wait before trying again.

The publisher may or may not know if a subscribe response goes through successfully, especially if the link is TCP. The publisher may actually see another attempt from the same client when the previous one is still considered alive. This scenario is discussed in the liveness section.

5.3.2. Canceling subscriptions

The following diagram describes the process flow for cancelling a description.

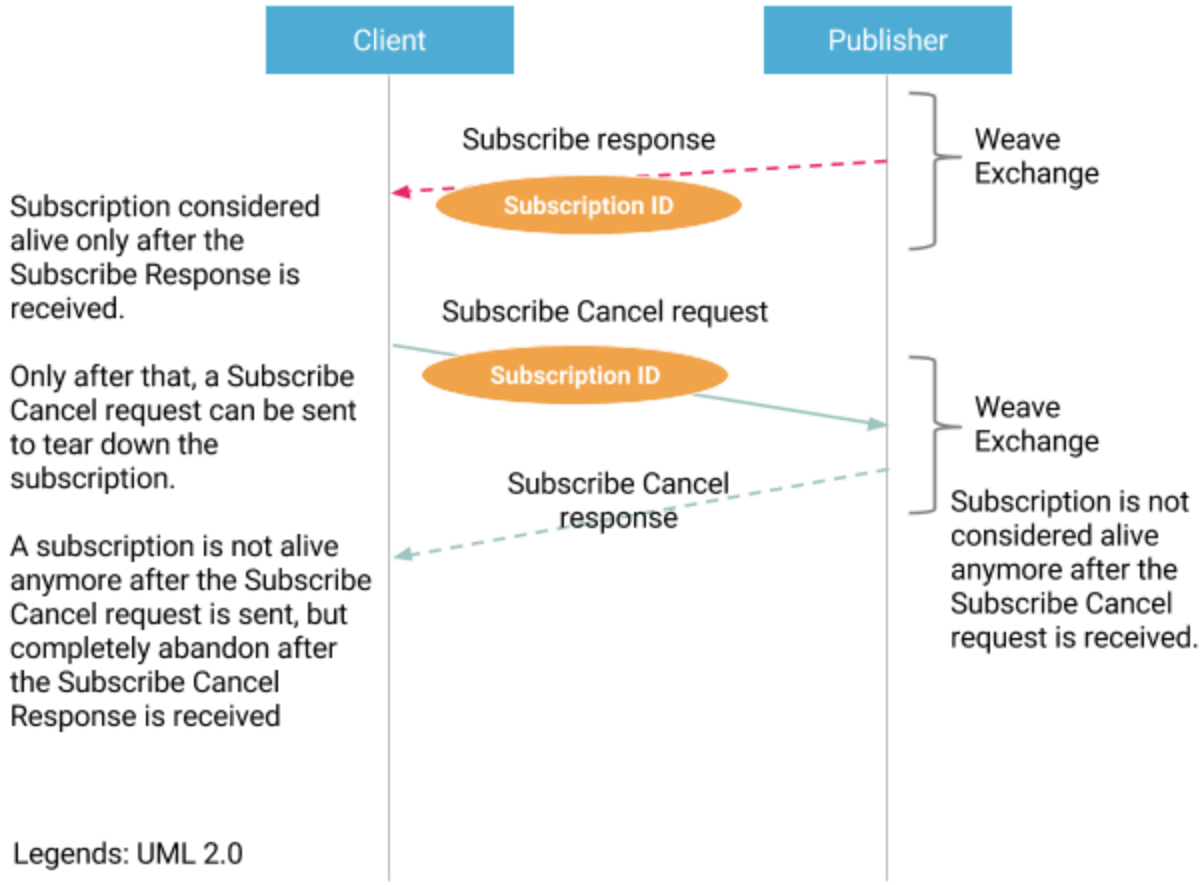

Figure 13: The process flow for cancelling a subscription

5.3.3. Notifications and changes

Notification requests are sent only after a subscribe request is received, but could be sent both before and after a Subscribe response is sent. Once a Subscribe Cancel request is sent or received, a publisher should not send more notification requests carrying the subscription ID.

The notifications sent before a Subscribe response are sent in the same Weave exchange as both the subscribe request is received and the subscribe response is sent. Each notification sent after the Subscribe response must have a new Weave exchange.

Every notification request carries one or more changes that contain information that is applied on top of the existing version for this particular trait instance on the receiver end to be one step closer to alignment. Every change describes anything from the exact changes from one specific version to another as well as all the data of the specific version, all pruned by the path list of the original subscribe request. Note that we have an exception case to communicate version changes without observable data alternation to a certain client, which will be described later in this section.

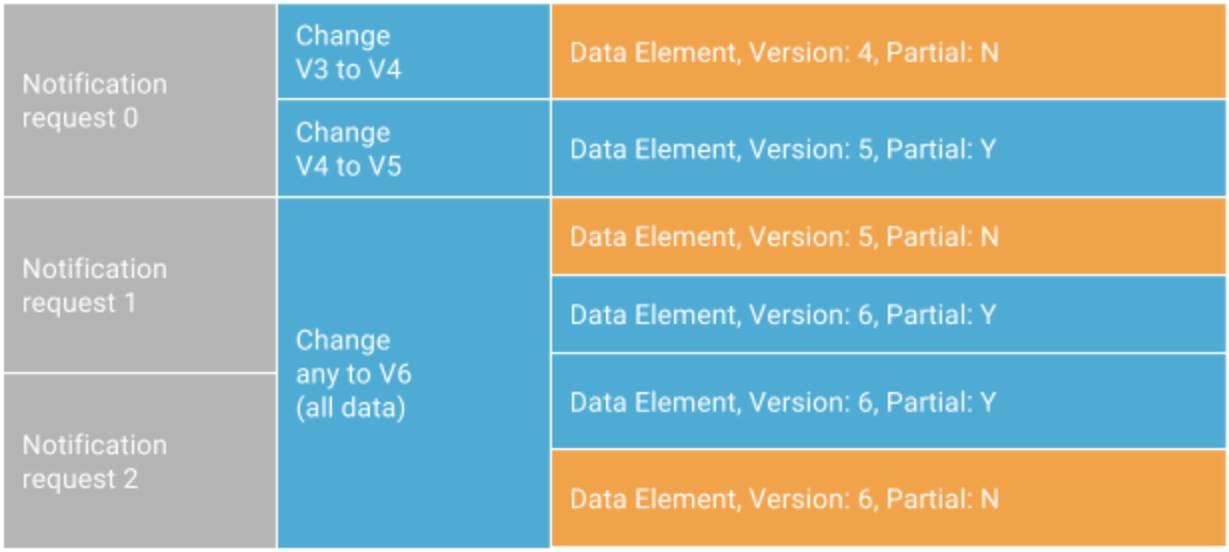

Figure 14. Changes consist of data elements that can be distributed among notifications

The change might describe data that is more than what has been actually changed on the publisher since the previous version, and could be more than what has been changed since the existing version on the client side. The change must be pruned by the path list in the subscribe request. A change has all the data if it describes everything a client cares about for that particular version no matter what prior information the client might possess. A change with all the data for one client might not be a change to another client, since every client could be interested in different parts of the same trait instance. A change with all the data must contain everything a client needs to replace the entire trait instance no matter how many paths the client has subscribed to for the specific trait instance.

Changes are only logical and do not have a physical presence in the message payload or protocol significance. The beginning of a change for a specific trait instance is implied by a data element carrying a new version for the trait instance. The ending of a change is indicated by a data element with the `DataElementPartialChange` flag set to false.

For a client, an incomplete change indicates what it already received and does not yet provide a coherent view of a version of the trait instance. If a publisher runs out of buffer space in a multi-notify sequence and must interrupt current change, it can choose to cancel or abort the current subscription instead. The client would eventually re-subscribe and receive the most up-to-date view of the traits then.

To simplify the implementation, a publisher must send all data elements in the same change before another change can begin. The next change can describe any trait instance, but data elements of different changes must not mix. Changes for the same trait instance must be sent in version-ascending order, which means a client can apply the changes in the same order as they are received.

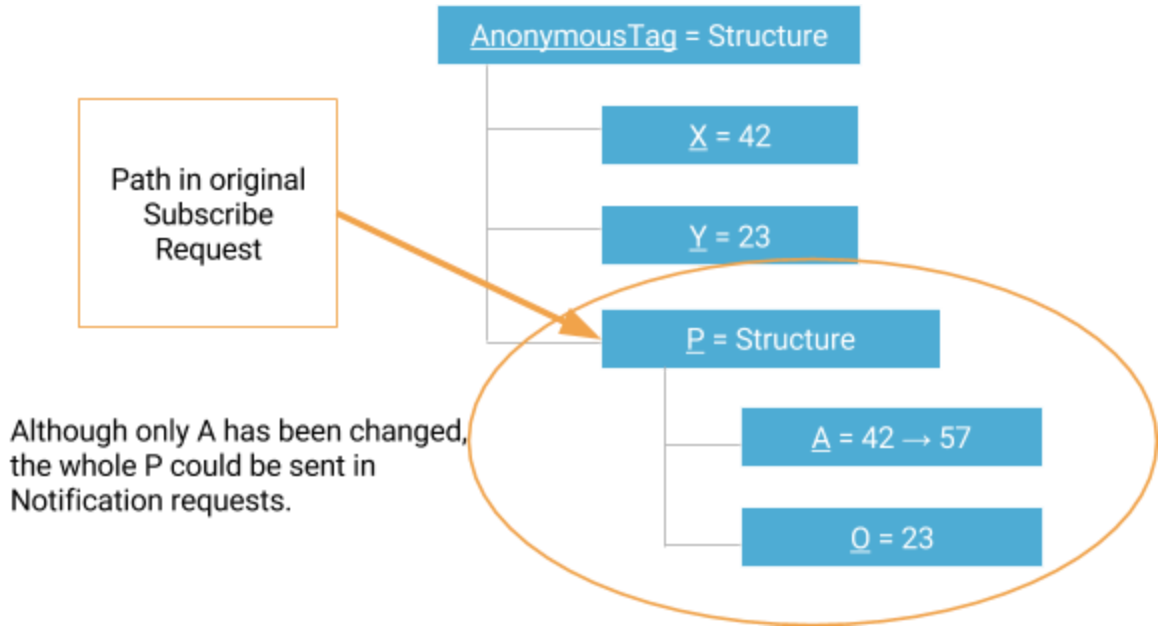

Figure 15: More than exact changes can be sent over in notifications

Sometimes, the data change is in a trait instance is outside of subscription of a client. To that particular client, it only senses version of the whole trait instance has changed, but not data alternation. Occasionally the publisher simply wants to bump the version but not change any data for some administrative reasons. To communicate this situation, a publisher can send a single data element, which has the path pointing to the root of this trait instance, and the data is an empty TLV structure. A client should be able to pick up the version change no matter which part it actually subscribes to.

5.3.4. Version list in subscribe requests and changes

The elements in the version list of the Subscribe request are versions that a client has immediately before the subscription setup. A client can choose to send NULL indicating it doesn't have any prior versions. The publisher would then send changes to completely cover the path lists in the subscription request.

Table 3: *Meaning of different value for version in subscribe requests*

Matching element in version list	**Publisher behavior**
NULL	The publisher must send the current version as a whole.
Version V	1) If the current version is still V → no change is sent.

	2) If the current version is VX > V → send complete VX to replace V 3) If the current version is VY < V, which means V must be invalid, this subscription request is invalid.
0	Version 0 is currently reserved and must not be used in any trait instance. The minimal version can only be 1.

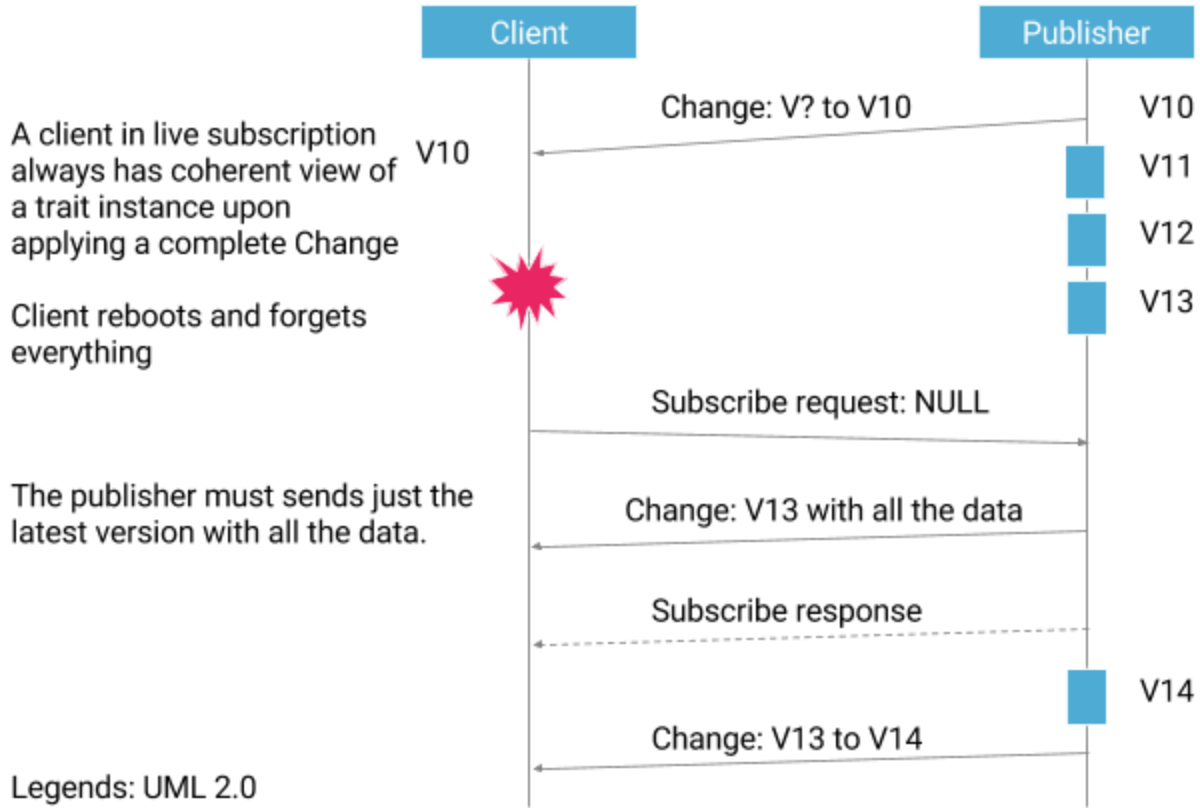

Figure 16: Client requests only the latest version

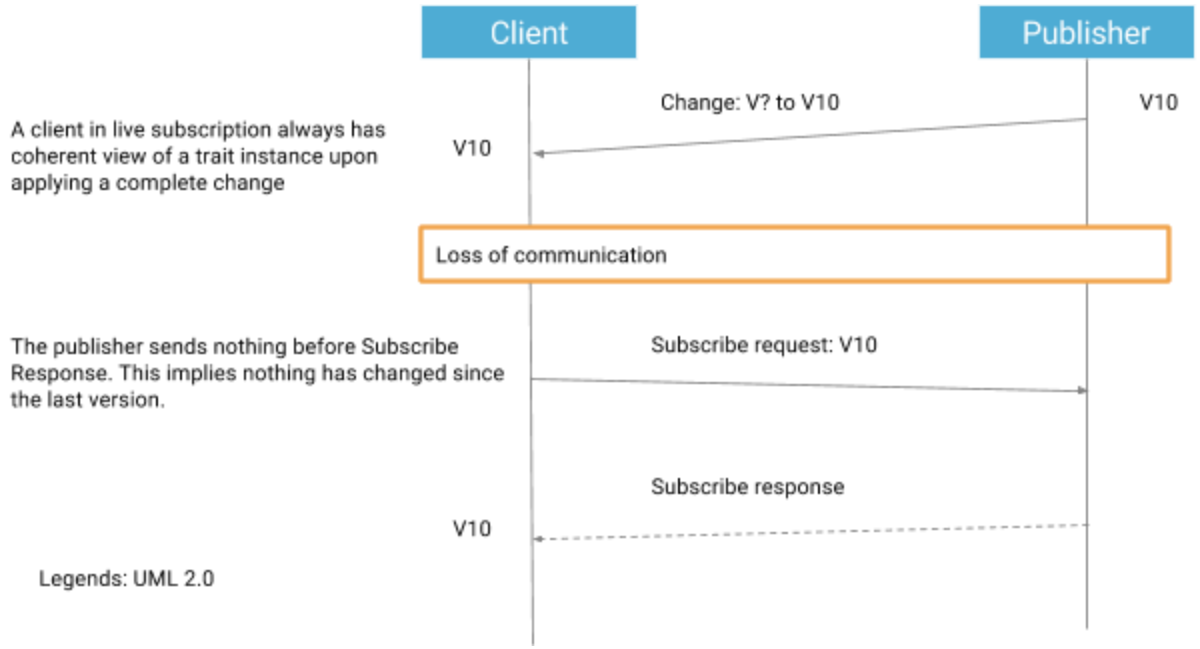

Figure 17: Nothing changed on the publisher's side

5.3.5. Liveness and liveness check

Liveness for a subscription is important for a client since a client might rely on incoming notifications to alert itself about a change that just happened. Imagine a security console relying on notifications from a window sensor to alert it that a window is being opened. If a communication outage prevents the window sensor from sending out the notification, the security console could miss the critical event that some window has been opened. A client needs to confirm the liveness of the connection from time to time subject to the needs of application.

Once a subscription is determined to be dead, a client usually should try to re-establish a new one. A publisher should choose to serve the latest subscription from the same client if it cannot serve more subscriptions. It's not necessary to explicitly send a subscribe cancel request to the subscription that is being abandoned.

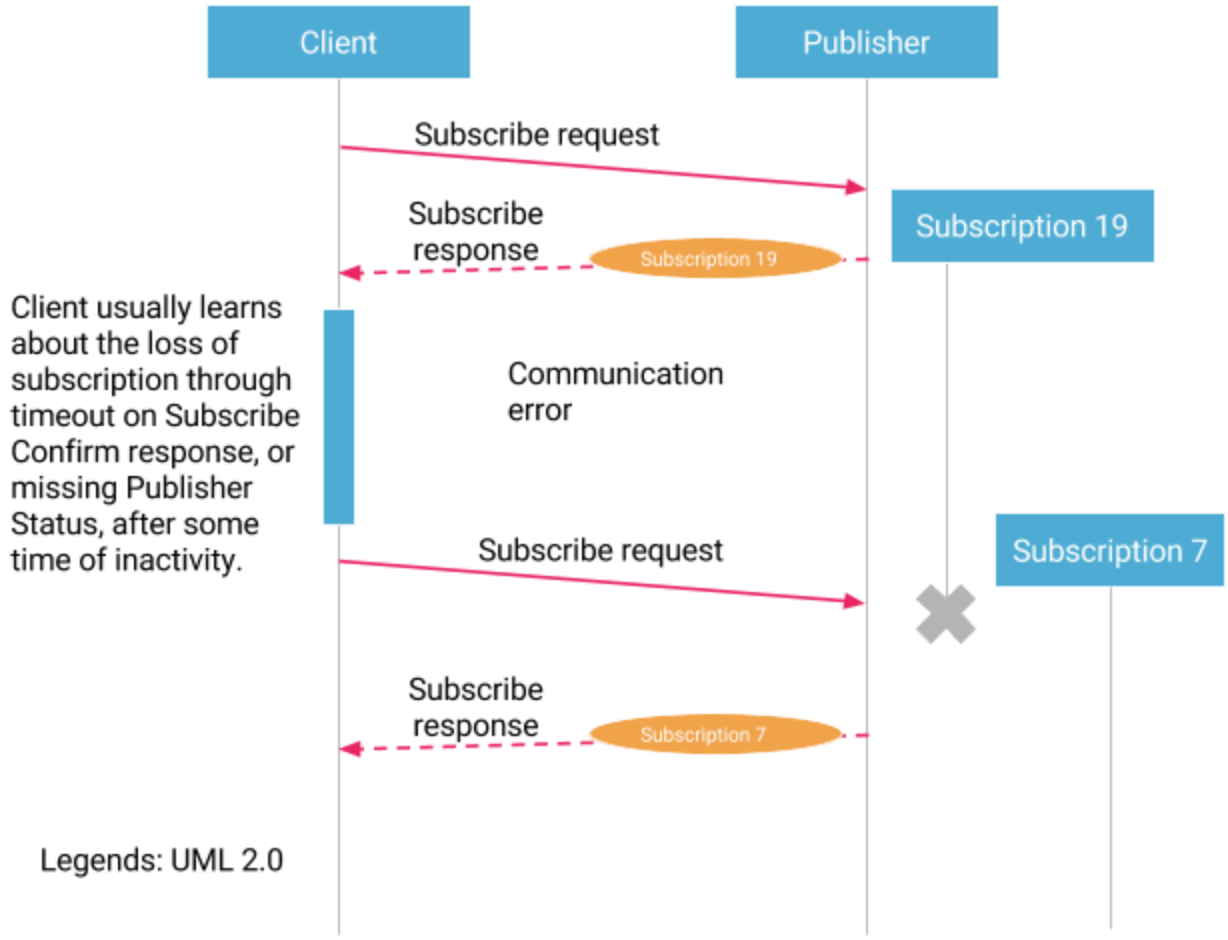

Figure 18: Client tries to recover via a new subscription

Liveness can be confirmed by various methods. Successful exchange of any message containing the correct subscription ID can be used to prove the subscription is still alive on both ends. The most common method is notification requests of a trait instance, which are exchanged when data changes. A Subscribe Confirm request can also be inserted by a client in the absence of data changes. Note that there is no direct support for liveness validation from a publisher.

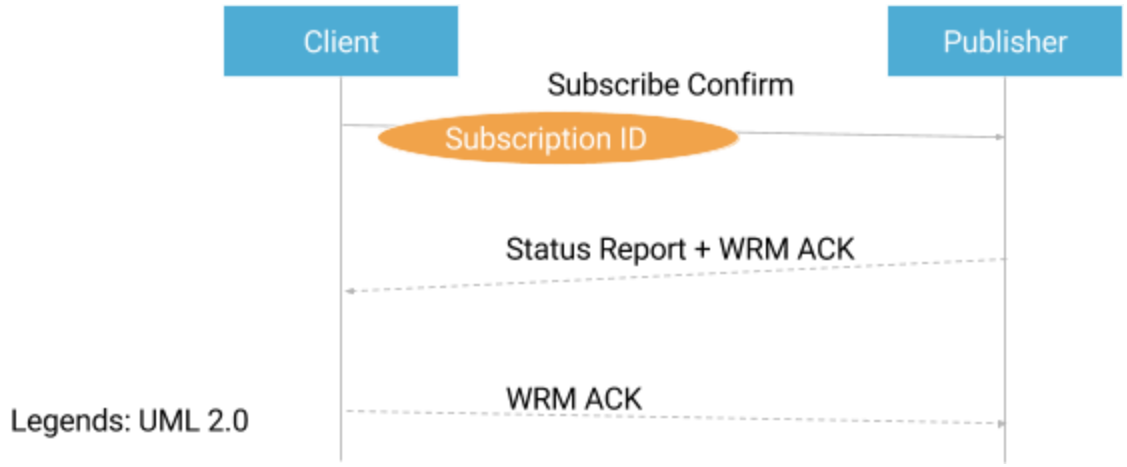

Figure 19: Message flow for Subscribe Confirm

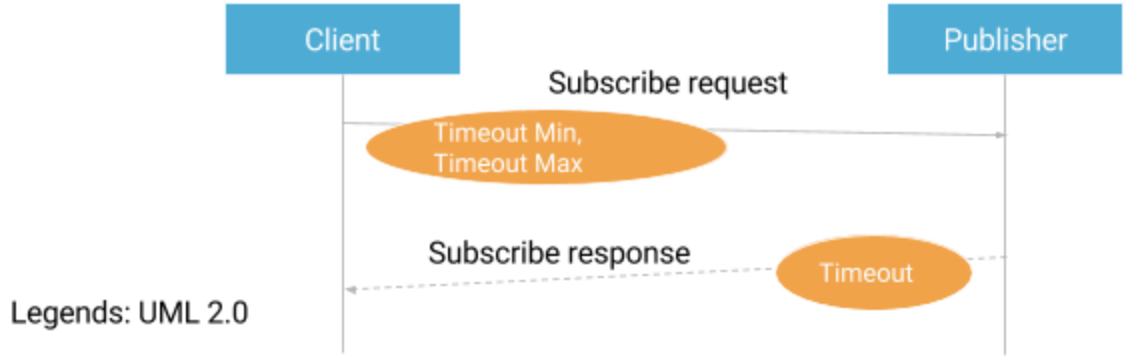

Figure 20: Timeout negotiation during subscription setup

If the publisher requires some upper bound on time between liveness validations, it can express this in a Subscribe response with a timeout. The client would then be responsible for sending a Subscribe Confirm request whenever the inactivity lasts longer than that threshold. During the subscription setup, the client can optionally send the range of time periods it can support by sending Subscribe Confirm requests, given its power and network budget, so the publisher can make a more reasonable choice on the timeout.

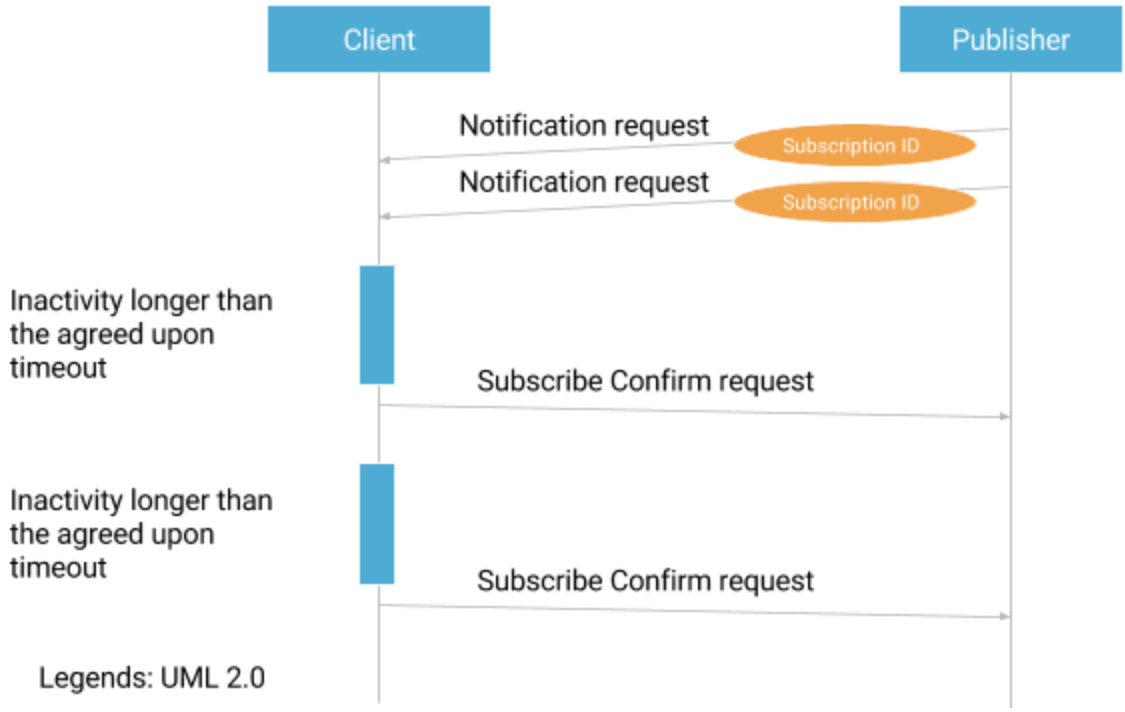

Figure 21: Inserting subscribe confirm request during periods of inactivity

5.3.6. Mutual subscription and bounded liveness

Mutual subscription happens between a pair of entities subscribing to each other sharing the same subscription ID and liveness. This is usually used between a device and the Cloud service.

Since both entities are publishers and clients at the same time, we use “initiator” to indicate the party that initiates the first Subscribe request, and “responder” to indicate party that sends the second Subscribe request. By convention, a device initiates the subscription toward the Cloud service. The detailed parameters, especially the timeout specs, can be found in section 6, Message Format.

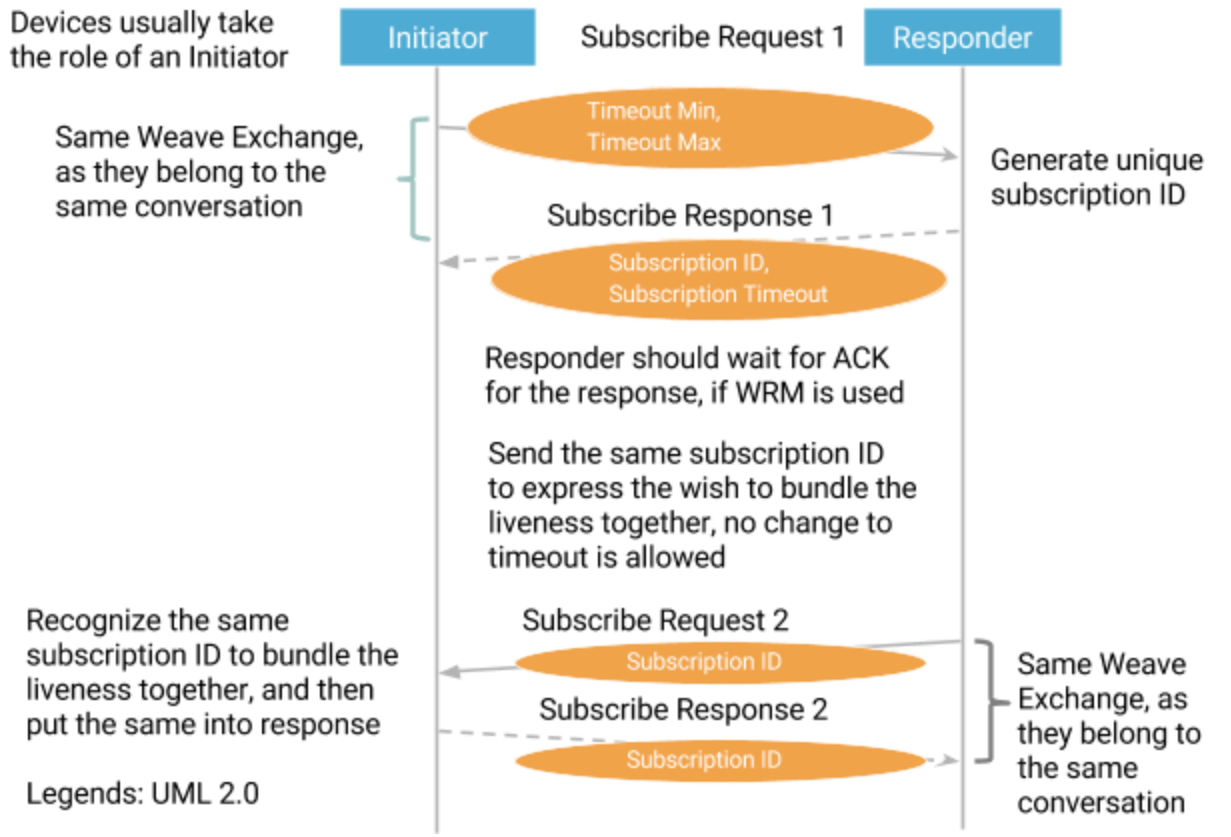

Figure 22: Message flow for mutual subscription

A liveness check can be initiated on both sides for either subscription. Since they share the same subscription ID, the liveness of one subscription proves the liveness of the other. Note that the subscription ID has to be unique on both ends, which is a stronger requirement than in one-way subscriptions. An initiator can cancel the subscription when it sees a conflicting subscription ID on its side and re-initiate a new subscription.

A liveness check time limit negotiation is done the same way as a subscription, only it is allowed to happen in the first subscription. The second subscription must not change the timeout threshold. More specifically, `SubscribeTimeOutMin` and `SubscribeTimeOutMax` in Subscribe Request 2 are ignored by Initiator. `SubscribeTimeOut` can be absent in Subscribe Response 2 and the timeout shall still be the same for both subscriptions. If Response actually specifies `SubscribeTimeOut` in Subscribe Response 2, the value must be the same as in Subscribe Response 1.

The initiator is responsible for inserting subscribe confirm requests when there is no other activities on any direction for long time. The responder would cancel subscriptions on timeout after inactivity. The initiator would also cancel its subscriptions if the subscribe confirm request fails.

5.4. Views

A view can be seen as an instant subscription, which gives a one-time result identical to the view response. A client would specify the paths it is interested in. No current version is provided in the request, so a publisher must always populate the response with the latest version for all paths requested.

Since view response doesn't assume a merge is possible, the data elements in the view response must not be applied using the "one level merge" scheme but rather the "replace" scheme.

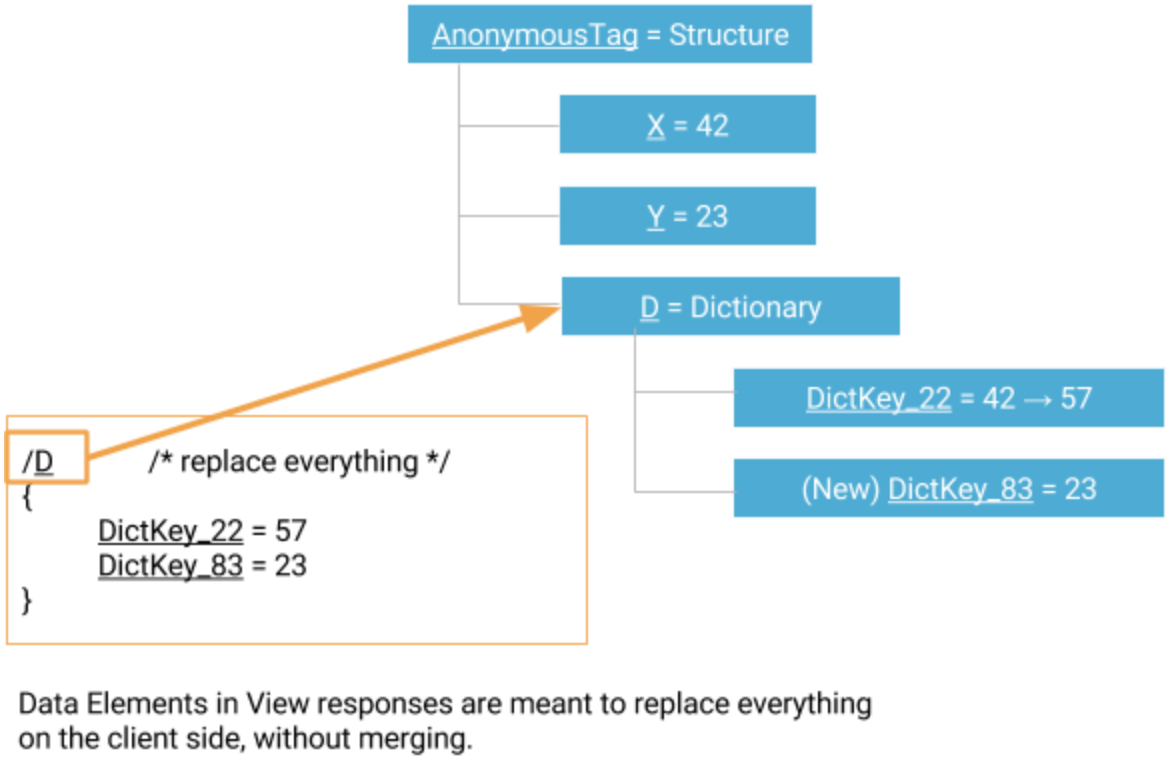

Figure 23: *Replace scheme used in a view response*

5.5. Update

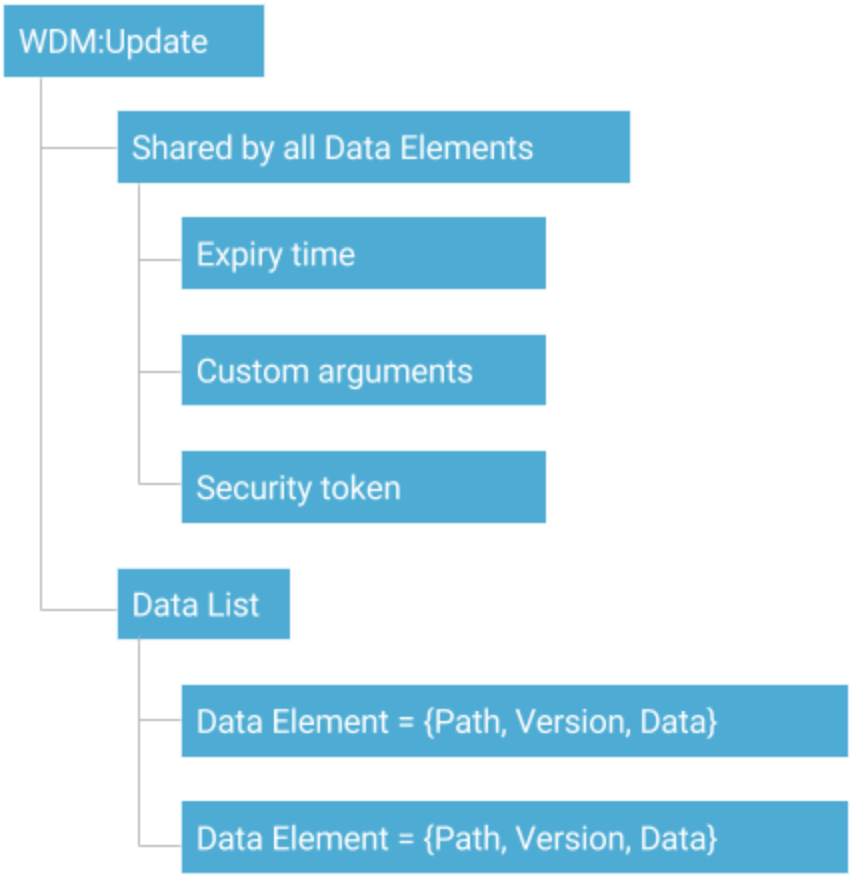

Figure 24: Payload of update request

An update request is sent from a client to a publisher. One or more data elements in the request are the unit for processing at the publisher. Data elements intended for the same trait instance are guaranteed to be applied in a serial manner, but the order is undefined. There is no guarantee for data elements intended for different trait instances.

Aside from the actual data, some parameters can also be provided as context. Parameters including security token, expiry time, and custom arguments are shared among all data elements, while the version is unique for each data element. Details for these parameters are discussed in later sections. The application of data elements follows the “one level merge” scheme described for notification requests. A publisher may choose to increase the version of a certain trait instance after any number of data elements have been applied to it, as long as no client gets an incoherent view of it.

Clients must expect the target trait instances to be in some partially updated status when an update request is rejected. This is because data elements are processed individually, and the

publisher may or may not be able to perform a rollback when an error is detected while processing a data element that is sent later.

5.5.1. Version for optimistic locking

Including a version in a data element specifies the change is only applicable if the current version of the target trait instance is as specified. If not, this request will be rejected. This can be used to avoid modification or action on the trait instance over an unknown state.

Assume there is a command to bump a traffic light to the next state instead of setting the state to a specific state. If there is another client also trying to bump the state, the result could be that the state is bumped twice.

5.5.2. Expiry time

Due to delays at various layers, it's difficult to predict exactly when a publisher would begin processing a command. As an example, an "open the door" command could reach the publisher 20 minutes after it was sent, in which case the door would be opened long after the user has given up and left. For most commands that have a UI element, a time period of several seconds is probably the longest time interval we should allow before the command is considered invalid.

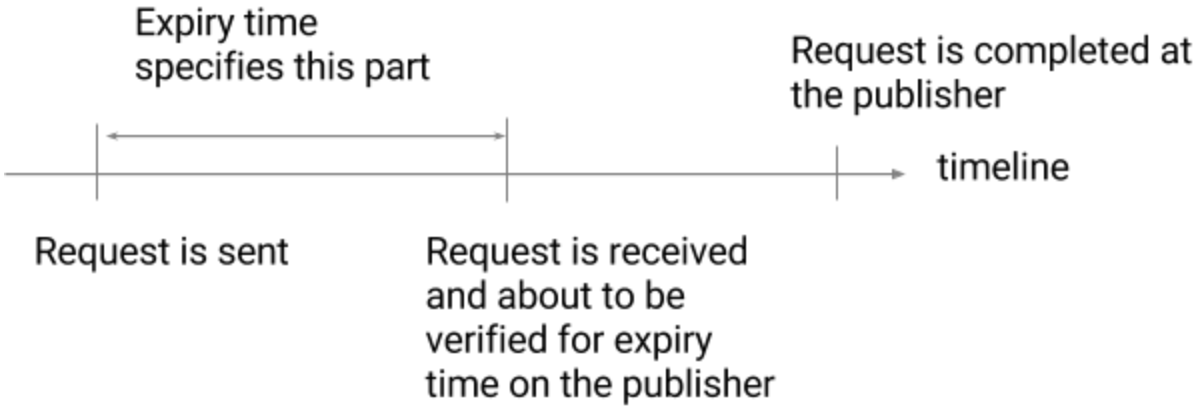

Figure 25: *Expiry time*

For expiry time, there are a group of error codes that indicate the reason and network location (proxy or destination publisher) when a timeout occurs. This information is described in later chapters.

5.5.3. Custom arguments

Custom arguments are outside of the scope of this document since the use and schema of this information is application specific. The WDM profile only specifies that these arguments are

optional in the payload of an update request under a specific tag. The information is treated as shared context while processing all data elements.

Since custom arguments are shared by all data elements in an update, which might point to different trait instances, it makes more sense for custom arguments in an update to carry profile tags instead of context-specific tags, or even be anonymous. This is designed to allow handlers for all affected trait instances to be able to tell what the arguments mean. On the other hand, arguments for a specific custom WDM command would be interpreted by one specific profile implementation, and hence can be interpreted with less ambiguity.

5.5.4. WDM subscription

Update requests usually generate new versions for one or more trait instances. For example, a “change current track to #8” request could result in a change in the device status as “current track is #8”. With a WDM subscription, the client could potentially observe the change and update its UI.

WDM is not designed to deliver edges of changes reliably, which means state change events could be merged with each other and become undetectable. As an example, if a door is closed again soon after the “change door to open” command finishes, an observer could potentially lose the transition from close to open.

There is a version number carried in the response for each data element that indicates the last version a client needs to wait for regarding any direct change in that trait instance. After receiving that version, whether an edge is detected or not, a client might want to cancel the subscription to save resources.

5.4.5. In progress message

For long running update operations, a publisher can choose to send an “In progress” message in the same Weave exchange, indicating it’s still working on the update. There can be only one of these messages in the Weave exchange. Whether the message used and when to send it is specific to each application. An example can be found in the next section.

5.5.6. Message flow overview

The figures below depict the flow of messages in two typical cases: a short-running case and a long-running case. The client is also usually a subscriber to the target trait instance, so notifications will be sent back to it if the request causes property changes in the target trait instance.

The version delivered in notifications may or may not be the same as the version carried in the response. The version in notifications indicates the latest version, while the version in responses indicates the last version the client needs to wait for. In the case of V2 > V2’, the client might

need to wait longer for the effects to be delivered. In the case of V2 < V2', the observable changes to this particular trait instance probably have already been collapsed with other changes.

A publisher must not send the update response before the notification responses for all the directly modified traits have been received from the requesting client.

A command might have side effects on other trait instances, but the versions are not communicated in the response. This means a client would have no direct indicator of when to stop monitoring notifications or cancel the subscription other than a timeout.

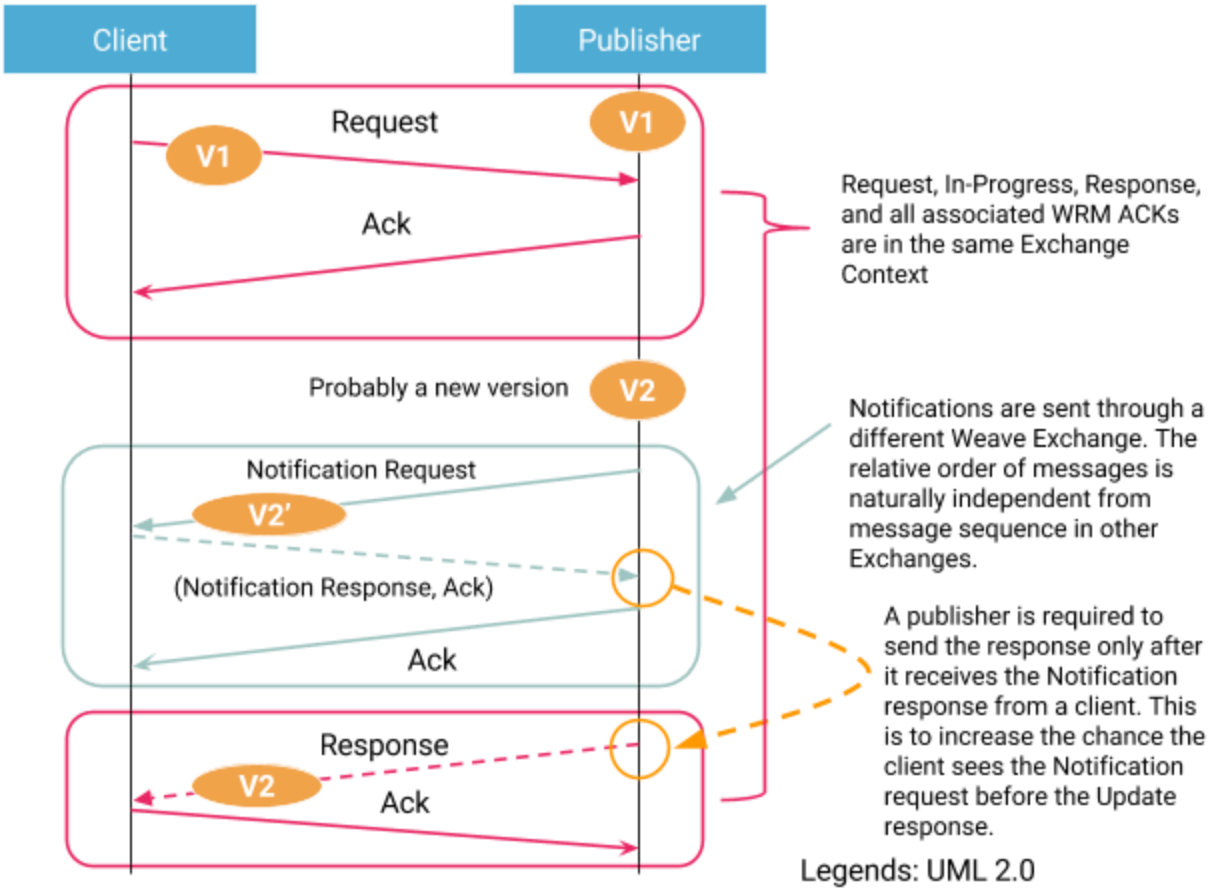

Figure 26: Message flow for update

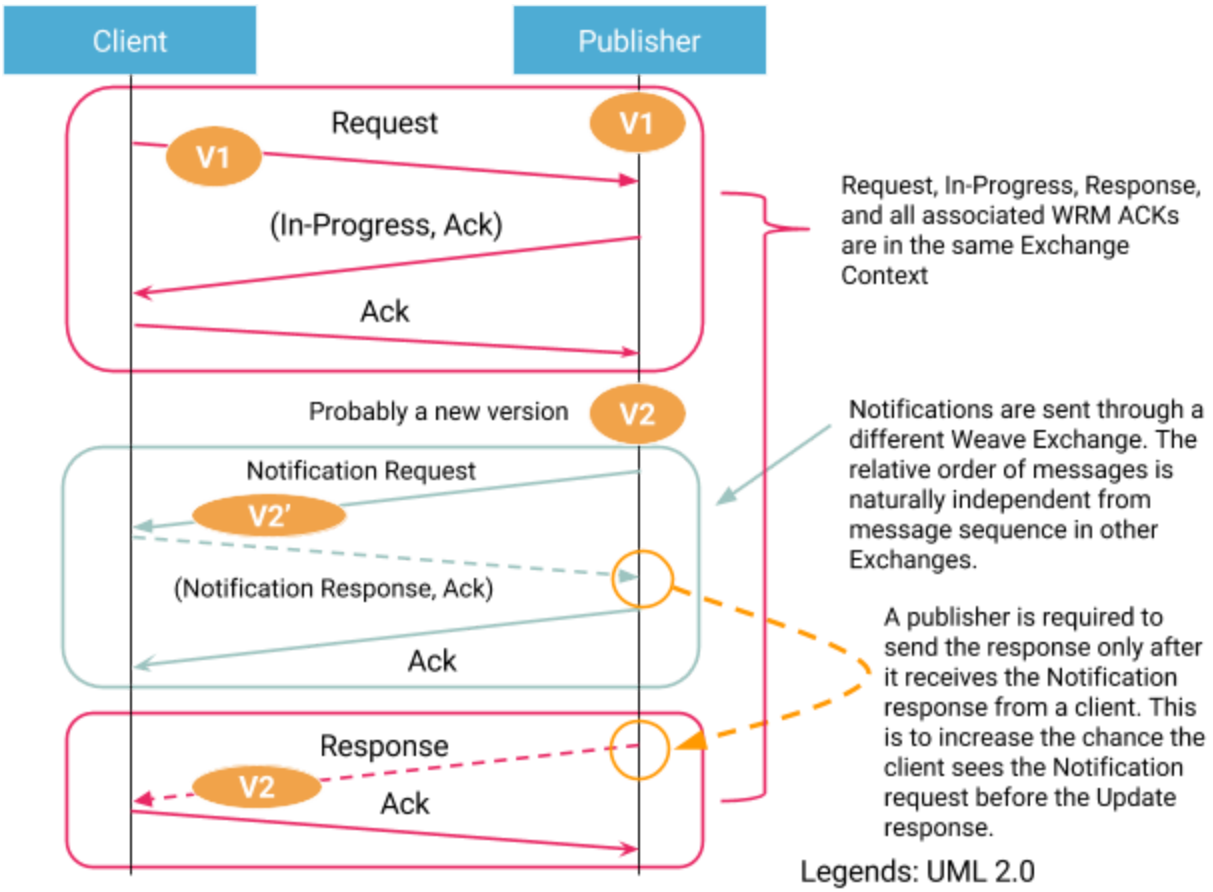

Figure 27: Message flow for update with an In Progress message

5.5.7. Door lock as an update request

The request to update some data could be seen as a command to perform some sort of operation as well. Successfully updating data means the operation has been performed without issues.

The assumption is the command has two directly observable results: the response indicating the result of command execution, and the notifications indicating the state of the deadbolt right after the command is executed.

Following the design principle, the notification should come from the same trait instance, which means the trait implementing the ‘door lock’ command should also publish the ‘state of the dead bolt’.

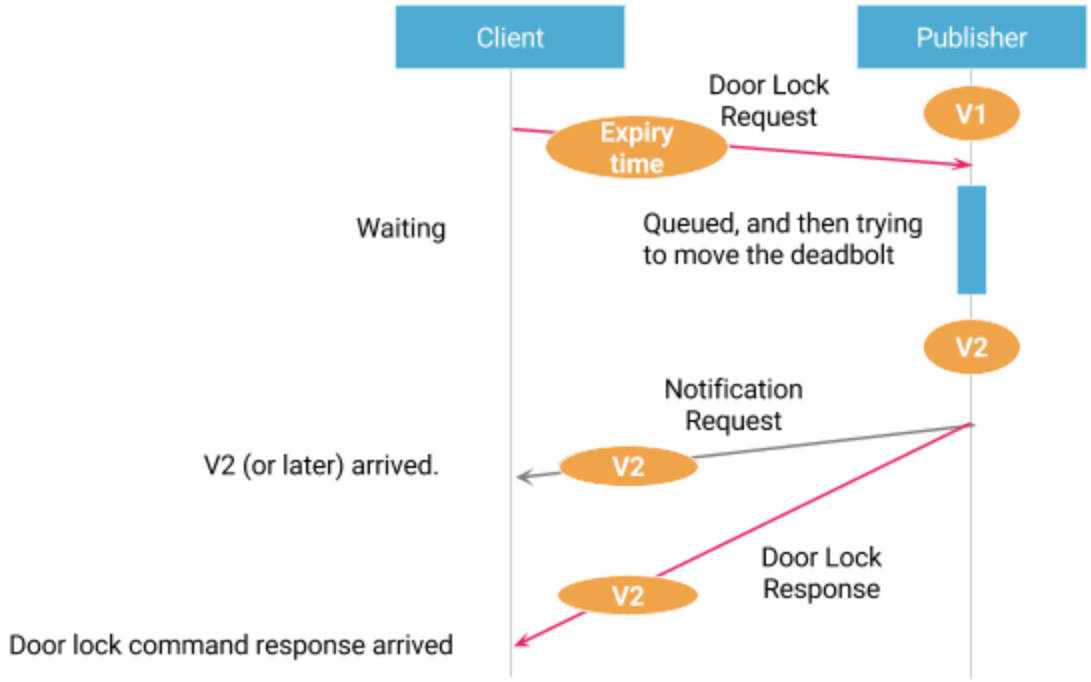

Figure 28: Update response comes after notification

It's possible that the notification of the same or later version actually arrives later than the response. The client can choose to wait for the notification if necessary.

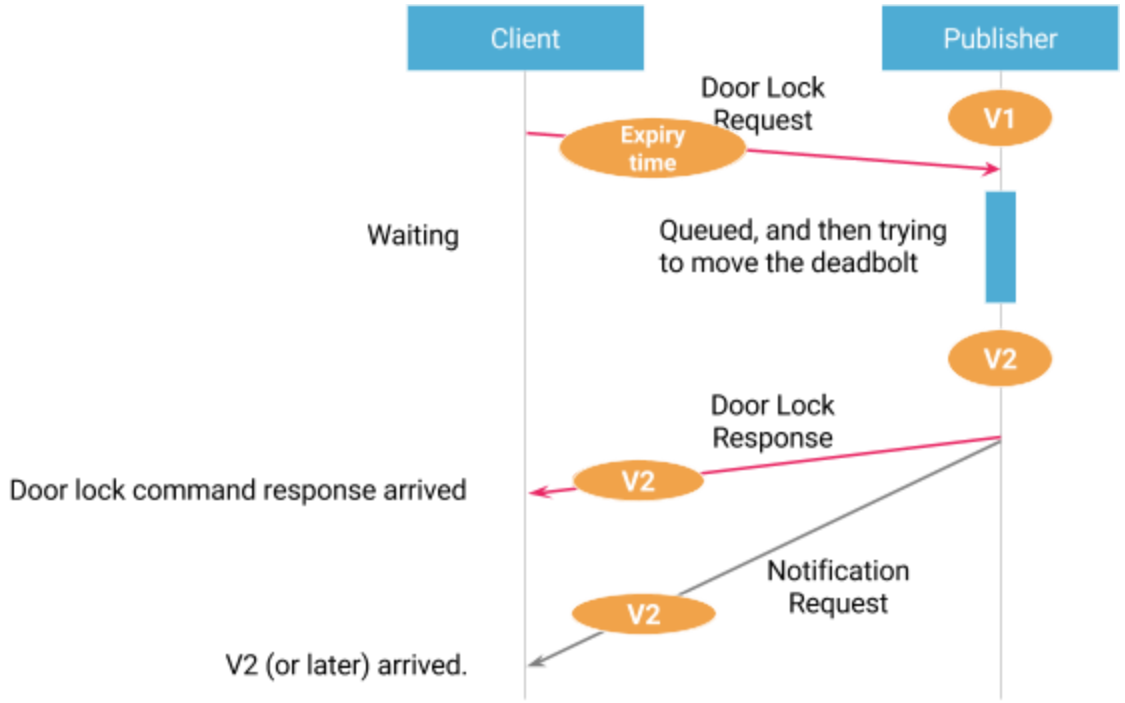

Figure 29: Update response comes before notification

In case there is no state change associated with this command, the version carried in the response is the ‘current’ version that the client should already be on with a live subscription.

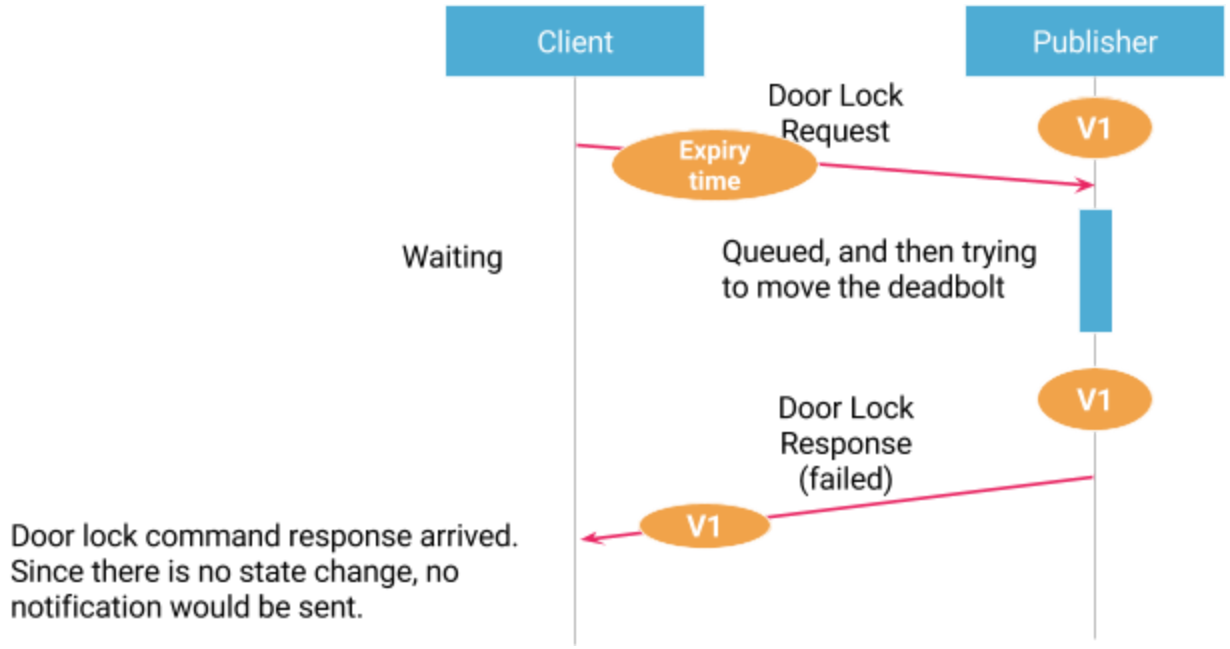

Figure 30: Update response indicates failure

Table 4: *Scenarios for update responses*

	Command succeeded	**Command failed**
No version is carried in the response	Must not happen. Response could carry the current version, which is the same as before this command took action, indicating there has been no state change	
Notification with same or later version arrived before response	Notification probably indicates the door is locked, and response provides extra assurance that the command request was granted and the door was locked at least once between the command and response.	Notification probably indicates the door is still not locked, and response provides some kind of explanation.
Notification with same or later version not arrived yet	Response already indicates the door was locked at least once between the command and response. It's probably unnecessary to wait for this notification.	Response already indicates the door wasn't locked because of this command. Whether the door is actually locked or not may or may not be answered by this yet to come notification.

For a simple client without a subscription, the response alone would be enough to indicate the operation has been completed.

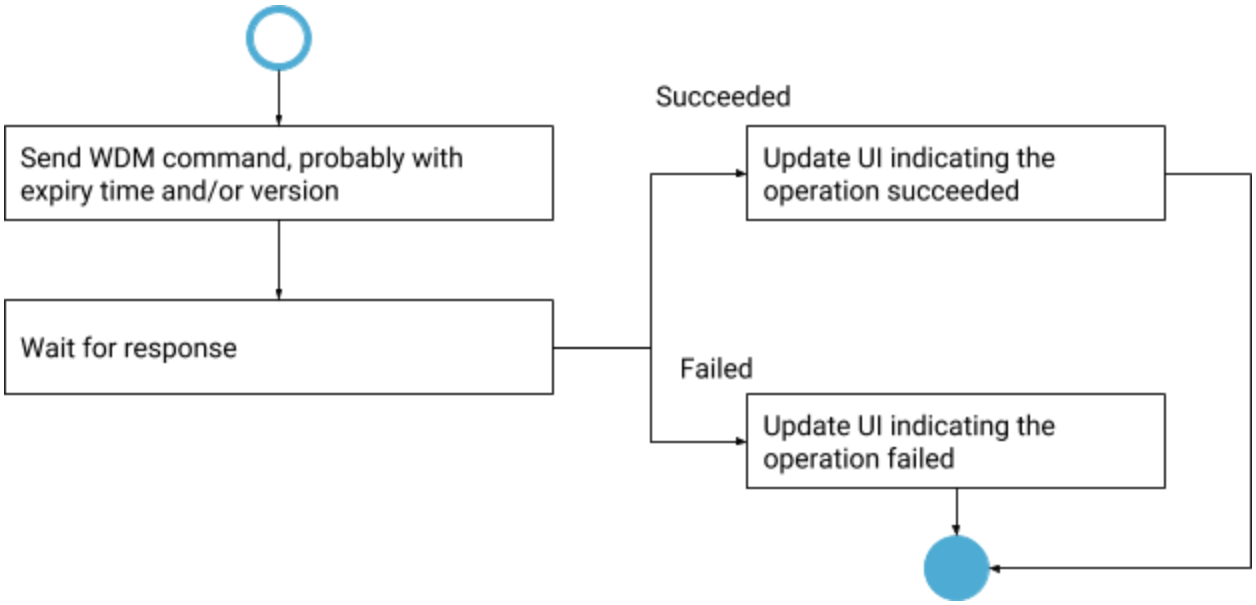

Figure 31: Flow chart for update command

For a more capable client with a subscription, the notifications could provide more of a real-time update as the operation proceeds.

WDM can collapse edges, so the notifications received may represent the result of multiple operations from a number of sources. As an example, say a ‘lock the door’ command could succeed but the notification still indicates the door is unlocked. This might mean some other client, even a human operator, tried to unlock the door right after the command took action. Whether the client should pay attention to these contradictory pieces of information is application-specific.

The version is only for the target trait instance for a command. There is no guarantee on the cohesiveness with other trait instances the client might also be subscribing to.

5.6. Custom WDM commands

Custom WDM commands look similar to Update requests, but they do not carry data lists. The purpose of a custom command is to pass a command type ID and optional arguments to a trait instance, for some operation that may or may not lead to change in properties of that trait instance. As the name implies, it is like a remote procedure call wrapped under WDM profile.

The assignment of command type ID in each profile and actual data schema of command arguments and response must be further defined by each individual command. Similar to other

Weave requests, a Status Report could be sent in reply to indicate simple failures like no memory or access denied. A command response message must be returned if the command executed okay.

5.7. Events

Multiple changes to a trait instance could be ‘collapsed’ and loses their individual details. For example, multiple changes to door status could be collapsed to just the final status of either open or close, disregard of how many status changes have happened. Other happenings on the resource are important to note and communicate but do not result in a change of trait state. When capturing history of the happenings on the device is important, WDM provides a mechanism called Events to address it. The detailed rationale behind the data design of events is captured in [5]. Here we just touch on a few key points that impact the WDM protocol.

Events can be uniquely identified through a tuple of `{Source, Importance, Event ID}`, and Event IDs are guaranteed to be generated sequentially within the same resource and importance level. The global uniqueness of the tuple `{Source, Importance, Event ID}` makes it possible to implement event aggregation and deduplication. The sequential nature of Event IDs makes it straightforward to detect loss on the subscriber side. An event can also be marked as ‘related to’ another one, which makes grouping of events easier.

Clients can only access events through subscriptions. Transmission of the event stream often requires multiple messages. There's no straightforward way to extend View to support event transfer, since `ViewResponses` are single messages without any opportunity for chunking or streaming of data. Consequently, at this time, accessing event stream using the View command is not supported.

5.7.1. Event data definition

In a fully expanded form, all events have the following fields:

- **UTCTimestamp** - The time when event was generated in UTC milliseconds.
- **SystemTimestamp** - The time when event was generated in milliseconds. The origin of the timestamp does not need to be synchronized to UTC. The field is optional; when omitted, it defaults to the value of Timestamp
- **Event source** - The node ID where the event was generated.
- **Event ID** - The sequential, non-repeating number of the event. Event IDs are sequential per Event Importance, start at 0, are incremented sequentially. The Event ID is persisted across reboots; in case of an unexpected reboot it jumps forward by a sufficiently large number to indicate a gap in the event delivery. Event IDs require similar properties to WDM versions and Weave Message IDs, and may leverage the same libraries on embedded devices.

- **Event Importance** - Importance of the event. All implementations must support `LOG_PRODUCTION` importance to provide guaranteed buffering volume to the critical events. Implementations must support at least one additional Importance level to provide additional data collection and debugging information. In constrained environments, implementations may collapse non-production importance levels (`LOG_INFO` and `LOG_DEBUG`) into a single event ID sequence, such that only a single additional EventID sequence and a single buffer are used beyond the `LOG_PRODUCTION` requirements.
- **Related Event ID** - The Event ID from the same Event Source that this event is related to. When the event is not related to any other events, Related Event ID is shall be equal to Event ID, and may be omitted.
- **Related Event Importance** - Event Importance of the Related Event ID. When this event and the related event are of the same importance, the field may be omitted.
- **Event Resource** - The ID of the resource that the generated event pertains to. When the event resource is equal to the event source, it may be omitted.
- **Event Trait** - Trait profile ID. It is formed by concatenation of the `vendor_id` and id attributes from `wdl.trait` options
- **Event Type** - The type of this event; the number is equal to the `wdl.event.id` option in the IDL definition.
- **Event Data** - The event data itself.

5.7.2. Event delivery in WDM

Events are delivered as a part of the WDM `Notify()` message, subject to the following rules:

1. Events are TLV-packed into a separate field called `EventList` list.
2. Events within each importance level are always transmitted from oldest to most recent. Events within each importance level are always encoded from oldest to most recent.
3. Events of higher importance should be transmitted first, so as to minimize the chance of overflowing the high importance queues. If all events to be transmitted fit within a single `Notify()` message, implementations may choose to send interleaved event sequences for all importance level as long as rule 2 is not violated.
4. Events may be delivered within a `Notify()` message that contains an empty `DataList` element.

When an event is overwritten within the history buffer and a subscription was unable to deliver that event to the subscriber, it should not tear down the subscription. Strict requirements on the `EventID` ordering guarantee that the subscriber will be able detect the gap in the event delivery.

When the event publisher does not natively support time synchronization, it may provide the `SystemTimestamp`. It then falls on the client to compute the UTC-corrected timestamp.

5.7.3. Event forwarding

A key feature of the design is that each event has a global ID. As a result, event delivery along multiple paths is possible, and an observer will be able to distinguish duplicate events. The Figure below shows an example scenario of Pinna events being forwarded to Salt via Flintstone.

When a publisher forwards events from a different event source, the forwarding shall be subject to the following rules:

- The publisher must apply the event delivery logic from Event Delivery to forwarded events.
- The publisher should prioritize its own events over the forwarded events.
- The publisher must begin the sequence of forwarded events with the fully expanded eventID. This minimizes the subscription bookkeeping that the subscriber must keep.

5.7.4. Event loss detection

Event loss is certain, however, the design goals are to minimize the loss, provide clear semantics around event loss and, provide mechanisms for detecting event loss along with indications of loss volume and reasons for the loss. Event loss occurs under three primary circumstances:

- The device generates events faster than the it can offload the events to all of its subscribers and is forced to drop events.
- The device crashes before it has a chance to offload its event history.
- An intermediate node, such as Flintstone, that is normally responsible for forwarding events, experiences one of the above problems, for example, it crashes or it is unable to send events fast enough.

Event loss is detected whenever the subscriber observes non-continuous `EventID`s for a specific combination of (`EventSource:EventImportance`). When the `EventSource` is the publisher of the event sequence, the subscriber must conclude that there is an event loss. When the `EventSource` is different from the publisher, the subscriber must conclude that the publisher observed an event loss from the (`EventSource:EventImportance`) stream.

The event source is obligated to persist the `EventID`s for each distinct `EventImportance`.

6. Message format

For all TLV structures described in this chapter, TLV elements with context-specific tags must appear in “tag order”, which means an element with smaller tag value must appear before an element with larger tag value. TLV elements with profile tags must appear after all TLV elements with context-specific tags, but can appear in any order among themselves.

TLV elements with unknown context-specific tags are seen as protocol error, if appear in payloads of any WDM message, unless they are in the custom section of Data or Event.

TLV elements with unrecognized profile tags are only allowed to appear in payloads of WDM Update Request and Command Request messages. They are ignored if not understood by the receiving node. This is to enable further extension of authentication and authorization mechanisms.

Also, the Implicit Profile for encoding and decoding must be fixed at Dictionary Key profile for all TLV elements.

6.1. Profile ID and message types

6.1.1. Profile ID

The profile identifier field of the Weave application header shall have a value of 0x0B for all WDM messages.

6.1.2. Message type

The message type field of the Weave application header shall have one of the following set of values for WDM frames.

Table 5: *Message types*

type	message
0x00-0x06	WDM v1 messages *(deprecated)*
0x07-0x0F	Reserved
0x10-0x16	WDM v2 messages *(deprecated)*
0x17-0x1F	Reserved
0x20	View request
0x21	View response
0x22	Update request
0x23	In progress
0x24	Subscribe request

0x25	Subscribe response
0x26	Subscribe cancel request
0x27	Subscribe confirm request
0x28	Notification request
0x29	Command request
0x2A	Command response
0x2B-0xFF	reserved

6.2. Status codes

Table 6: *Status codes*

Profile ID: Value	Name	Comments
WDM: 0x01	Deprecated	Used in WDM V2
WDM: 0x02-0x12	Reserved	
WDM: 0x13-0x18	Deprecated	Used in WDM V2
WDM: 0x19-0x1F	Reserved	
WDM: 0x20	`InvalidValueInNotification`	Response for notification request
WDM: 0x21	`InvalidPath`	Response for view and subscribe requests
WDM: 0x22	`ExpiryTimeNotSupported`	Response for update requests and also custom WDM commands
WDM: 0x23	`NotTimeSyncedYet`	Response for update requests and also custom WDM commands
WDM: 0x24	`RequestExpiredInTime`	Response for update requests and also custom WDM commands
WDM: 0x25	`VersionMismatch`	Response for update requests and also custom WDM commands
WDM: 0x26	`GeneralProtocolError`	

WDM: 0x27	`GeneralSecurityError`	
WDM: 0x28	`InvalidSubscriptionID`	Response to any request that carries an invalid subscription ID
WDM: 0x29	`GeneralSchemaViolation`	

6.3. Primitive TLV elements

No extra elements are allowed in any of the TLV containers described in this section. Not all optional elements have default values.

6.3.1. Path

Path is a special TLV element, which is ordered in interpretation and can contain a duplication of tags. The first element points to the root of the trait instance, while other elements mark the tags to look for at each level down. Allowing the duplication of tags is necessary for it to be able to follow multiple layers of tags that could collide with each other.

Table 7: *Tags in TLV elements for paths*

Name **Profile ID: Tag Value or** **CS (Context Specific): Tag Value**	**Tag Value**	**Description**
CS: `PathInstanceLocator`	1	Instance locator portion of a path
CS: `PathResourceID`	1	ID for the resource in instance locator
CS: `PathTraitProfileID`	2	ID for the profile in instance locator
CS: `PathTraitInstanceID`	3	ID for the instance in instance locator

Figure 32: Listing: Path example schema

```
<
   /*
   The first element in a Path must be a descriptor for where to find
   the ‘root’ for this trait instance. We call this Instance Locator.
   */
   PathInstanceLocator = {
      PathResourceID = Optional, any value of any type.
```

```
      PathTraitProfileID = Mandatory, 32-bit unsigned integer,
        Profile ID of the trait

      PathTraitInstanceID = Optional, any value of any type.
   }

   /*
   Property path section. Tag for each element marks the tag to look
   for at this particular level, while the value must be NULL.
   */
   Example_1stLevel = NULL
   Example_2ndLevel = NULL
   Example_3rdLevel = NULL
>
```

A shorter form for the same path is shown below.

```
<{ResourceID = ..., TraitInstanceID = ..., TraitProfileID = ...}, / 1stLevel /
2ndLevel / 3rdLevel>
```

6.3.2. Path list

Since a path list is an array, and every element in an array must be anonymous, you should ignore the notation of AnonymousTag for elements in a path list.

Figure 33: Listing: Path list example schema

```
[
   <1st Path>
   <2nd Path>
   <3rd Path>
   ...
]
```

6.3.3. Version list

Since a version list is an array, and every element in an array must be anonymous, you should ignore the notation of AnonymousTag for elements in a version list.

Note that there must be some matching path list or data list to associate these versions with. If a version is not available or not applicable for any path, the version value for that entry must be replaced with a NULL.

Figure 34: Listing: Version list example schema

```
[
   AnonymousTag = Unsigned integer value or NULL to be associated
   with the 1st element in the matching Path List or Data List
   AnonymousTag = Unsigned integer value or NULL to be associated
   with the 2nd element in the matching Path List or Data List
   AnonymousTag = Unsigned integer value or NULL to be associated
   with the 3rd element in the matching Path List or Data List
   ...
]
```

6.4. Data element

A data element marks a chunk of data in WDM, which contains path, version, and the actual data.

NULL is different from absence of data in Weave TLV. NULL must not be used in a value pointed by `WDMDataElementMergeData` to indicate absence of data at any path and version. If the intention is to express deletion of some element, the upper container, dictionary or array must be replaced.

Table 8: *Tags in TLV elements for data elements*

Name **Profile ID: Tag Value or** **CS (Context Specific): Tag Value**	**Tag Value**	**Description**
CS: DataElementPath	1	Path for this data element
CS: DataElementVersion	2	Version for the trait instance described in this data element
CS: DataElementPartialChange	3	True if there are more data elements is in the current Change.
	4-8	Reserved
CS: DataElementDeletedDictionaryKeyList	9	Optional, contains a list of keys to be deleted from the dictionary
CS: DataElementMergeData	10	Optional

Figure 35: Listing: data element schema

```
{
```

`DataElementPath` = Mandatory, a path

`DataElementVersion` = Optional, integer value. Version of the trait instance referred in the path

`DataElementPartialChange` = Optional, boolean value, default to false. Set to true in all but the last data element for the current change. This is used to mark the end of a change so data elements can be applied as a whole across Weave messages, mostly useful in notification requests. If there is only one data element in some change, this element can either be set to false explicitly or skipped.

`DataElementMergeData` = Optional, any value of any type. May only be absent if `DataElementDeletedDictionaryKeyList` is specified.

`DataElementDeletedDictionaryKeyList` = Optional. May only be present when the `DataElementPath` refers to a dictionary. When present, the element is a TLV array containing a set of dictionary keys to be deleted from the element pointed to by the `DataElementPath`. Note: the overall element is an array, so each entry is encoded as an anonymous tag. The value is an unsigned integer. Note: When both `DataElementMergeData` and `DataElementDeletedDictionaryKeyList` are present in the same `DataElement` structure, the deletion operation takes precedence.

```
}
```

6.4.1. Data list

Since a data list is an array, and every element in an array must be anonymous, you should ignore the notation of `AnonymousTag` for elements in a data list.

Figure 37: Listing: data list example

```
[
   {1st data element}
   {2nd data element}
   {3rd data element}
   ...
]
```

6.4.2. Events

To accommodate future extension in events, unknown TLV tags must be allowed.

Note: This section reproduces event fields and tags from the authoritative reference defined in [5] for completeness and copied to Table 9. The section is authoritative for WDM-specific extensions to the event data model, such as `EventDeltaSystemTime`, that provide additional compression applicable to the event stream presented in Table 9.

Note that elements in an `EventList` could be subjected to minor cross-element compression. Some fields can be omitted if they are either the same as or sequential to the previous element in the current `EventList`. More details can be found in section 6.4.3.

Timestamp is mandatory in an event. There are two possible choices for timestamps: UTC and System. UTC timestamp has a defined epoch and should be reasonably in sync with real world. System time does not have a defined epoch nor any requirement on accuracy. For the need of save over-the-wire packet size, each timestamp can be transmitted as delta to the immediately previous event.

Table 9: *Tags in TLV elements for events*

Name **Profile ID: Tag Value or** **CS (Context Specific): Tag Value**	**Tag Value**	**Description**
CS: `EventSource`	1	Resource ID for the source of this event
CS: `EventImportance`	2	Importance level of this event
CS: `EventID`	3	Sequential ID within this importance level
	4-9	Reserved
CS: `EventRelatedImportance`	10	Importance level of the related event of the same source
CS: `EventRelatedEventID`	11	ID of the related event of the same source
CS: `EventUTCTimestamp`	12	UTC-synchronized timestamp of when this event was generated. If the publisher does not support time synchronization, `EventSystemTimestamp` must be used instead.
CS: `EventSystemTimestamp`	13	System timestamp of when the event was generated. May be omitted when `EventUTCTimestamp` is present.
CS: `EventResourceID`	14	Resource ID for the subject of this event
CS: `EventTraitProfileID`	15	Profile ID for schema definition of this trait

CS: EventTraitInstanceID	16	Trait instance of the subject of this event
CS: EventType	17	Type defined within this trait
	18-29	Reserved
CS: EventDeltaUTCTime	30	Used to compress the UTC timestamp in the current EventList.
CS: EventDeltaSystemTime	31	Used to compress the system timestamp in the current EventList.
	32-49	Reserved
CS: EventData	50	Actual data for this event

Figure 38: Listing: event schema

{

EventSource = Optional, could be any value of any type, default to resource ID of the current publisher. The resource which generated this event.

EventImportance = Optional, unsigned integer value, default to the same value of the previous element in the current EventList.

EventID = Optional, unsigned integer value, default to the next unsigned integer to EventID of the previous element in the current EventList.

RelatedEventImportance = Optional, unsigned integer value, EventImportance of the event this one relates to. Note this event can only be generated by the same EventSource.

RelatedEventID = Optional, unsigned integer value, EventID of the event this one relates to. Note this event can only be generated by the same EventSource.

EventUTCTimestamp = Optional, unsigned integer value, number of milliseconds since 0:0:0 1/1/1970 UTC. This information is only for reference, as the clock on a publisher could be off, and adjusted abruptly or even backwards.

EventDeltaUTCTime = Optional, signed integer value, number of milliseconds since the previous event in the current EventList. Note this value could be zero or even negative.

EventSystemTimestamp = Optional, unsigned integer value, number of milliseconds from an arbitrary point in time, for example, since boot. This information is only for reference, as neither epoch nor linearity is guaranteed.

EventDeltaSystemTime = Optional, signed integer value, number of milliseconds since the previous event in the current EventList. Note this value could be zero or even negative.

EventResourceID = Optional, could be any value of any type, default to EventSource. The resource which this event is about. Note this field is only needed if the event was generated by EventSource, only to describe what was happening on another resource.

EventTraitProfileID = Mandatory, 32-bit unsigned integer value, trait profile ID which generates this particular event

EventTraitInstanceID = Optional, unsigned integer value, default to 0. EventResourceID-specific ID for trait instance which generates this particular event.

EventType = Optional, unsigned integer value, default to the same value of the previous element in the current EventList. Together with EventTraitProfileID, EventType provides a complete link to the schema required to verify or interpret the EventData type below.

EventData = Mandatory. The type is collectively defined by EventResourceID, EventTraitProfileID, EventType. EventTraitInstanceID could also affect the schema if there is any optionality. See [6] for reference. When there is no EventData associated with the event, implementations should set the value of this tag to NULL.

}

6.4.3. Event list

Within the WDMEventList field, the full event representation may be compressed, subject to the following rules:

- Events are ordered by EventImportance, highest importance first.
- First event in any EventImportance always carries the full (UTC/System)Timestamp and EventID.
- (UTC/System)Timestamp may be replaced with a Delta(UTC/System)Timestamp that relates the timestamp of the event to the previous timestamp in the list regardless of importance of the consecutive events in the stream. Presence of both (UTC/System)Timestamp and Delta(UTC/System)Timestamp should be considered an error; in that case, the implementations should ignore the Delta(UTC/System)Timestamp.
- If the EventID is sequential with respect to the EventID within the current importance event stream, it may be omitted.
- EventImportance may be omitted if it is the same as the EventImportance of the preceding event.
- If the ID of the publisher is equal to the SourceID of the event, the SourceID field may be omitted.

- `EventType` may be omitted if it is the same as the `EventType` of the preceding event. This optimization may be most useful for debug events.

The example below explores a compression scheme for a sequence of events.= Events are represented in a JSON notation, but the representation maps directly onto a Weave TLV encoding. Consider the event definitions from Section 5.6.1. augmented with an unrelated temperature reading (different importance level) and an unrelated liveness event. In the compression scenario below, the publisher is a Flintstone with node id `0x18B43000000BEEFF`; that Flintstone additionally monitors the liveness of a Pinna with a node ID `0x18B430000000CAFE`. The full representation of events (as they were generated) might look as follows. For brevity, in examples below, we collapse the tuple of `(EventProfileID, EventType)` into `EventType`, and omit the `EventTraitInstanceID` altogether.

```
{eventSource: 0x18B43000000BEEFF, eventImportance:
LOG_PRODUCTION, 0x18B43000000BEEFF, eventType:
TemperatureReading, eventData: {temperatureInC: 25}}}

{eventSource: 0x18B43000000BEEFF, eventImportance:
LOG_PRODUCTION, eventID: 1236, relatedEventID: 1236,
UTCtimestamp: 1459966878420, eventResource: 0x18B430000000CAFE,
eventType: Liveness, eventData: {status: Unreachable}}}

{eventSource: 0x18B43000000BEEFF, eventImportance:
LOG_PRODUCTION, eventID: 1237, relatedEventID: 1234,
UTCtimestamp: 1459966878520, eventResource: 0x18B43000000BEEFF,
eventType: Event1, eventData: {happening_finished: true}}}eventID:
1234, relatedEventID: 1234, UTCtimestamp: 1459966878020, eventResource:
0x18B43000000BEEFF, eventType: StartEvent, eventData: {}}}

{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_PRODUCTION, eventID:
1235, relatedEventID: 1234, UTCtimestamp: 1459966878120, eventResource:
0x18B43000000BEEFF, eventType: Event1, eventData: {happening: "It
happened"}}}

{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_INFO, eventID: 3469,
relatedEventID: 3469, UTCtimestamp: 1459966878220, eventResource:
```

Applying the rules for omitting the redundant fields per rules from Section 5.6.1, we would observe the following encoding.

```
{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_PRODUCTION,
eventID: 1234, UTCtimestamp: 1459966878020, EventType: StartEvent,
EventData: {}}}
```

```
{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_PRODUCTION,
eventID: 1235, relatedEventID: 1234, UTCtimestamp: 1459966878120,
EventType: Event1, EventData: {happening: "It happened"}}}

{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_INFO,
eventID: 3469, UTCtimestamp: 1459966878220, EventType:
TemperatureReading, EventData: {temperatureInC: 25}}}

{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_PRODUCTION,
eventID: 1236, UTCtimestamp: 1459966878420, eventResource:
0x18B430000000CAFE, EventType: Liveness, EventData: {status:
Unreachable}}}

{eventSource: 0x18B43000000BEEFF, eventImportance: LOG_PRODUCTION,
eventID: 1237, relatedEventID: 1234, UTCtimestamp: 1459966878520,
EventType: Event1, EventData: {happening_finished: true}}}
```

According to rules specified in this section, if the publisher observes that on this particular subscription it has delivered event 1233 for LOG_PRODUCTION and event 3468 for LOG_INFO, the encoding becomes:

```
{eventImportance: LOG_PRODUCTION, eventID: 1234, UTCtimestamp:
1459966878020, EventType: StartEvent, EventData: {}}}

{deltaUTCTimestamp: 100, relatedEventID: 1234, EventType: Event1,
EventData: {happening: "It happened"}}}

{eventResource: 0x18B430000000CAFE, deltaUTCTimestamp: 100,
EventType: Liveness, EventData: {status: Unreachable}}}

{relatedEventID: 1234, deltaUTCTimestamp: 100, EventType: Event1,
EventData: {happening_finished: true}}}

{eventImportance: LOG_INFO, eventID: 3469, UTCtimestamp:
1459966878220, EventType: TemperatureReading, EventData:
{temperatureInC: 25}}}
```

After the above events have been successfully delivered to the subscriber, that is. the publisher received the `NotificationResponse()` with a status code `CommonProfile:kStatus_Success`, the publisher updates its subscription state to reflect it has delivered event 1237 for `LOG_PRODUCTION` and event 3469 for `LOG_INFO`.

6.5. Subscription initiation

6.5.1. Subscribe request

This message is sent from a client to a publisher.

If present, `SubscriptionID` means that this request is the second part of a mutual subscription. Both parts of a mutual subscription share the same subscription ID and liveness status.

`SubscribeTimeOutMin` and `SubscribeTimeOutMax`, if present, describe the range of the time period between the subscribe confirm request messages a client can support to send to the publisher in number of seconds.

`VersionList`, if present, describes the current version for the trait instance pointed by every path in the path list held at the client. If any of the versions is NULL, or there are conflicting versions of the same targeted trait instance, the publisher will assume the client doesn't have any prior versions of the targeted trait instance. If not present, the publisher will assume the client doesn't have a prior version of any trait instance.

A client is responsible for removing redundancy in the path list, otherwise, redundant data could be sent to the client in subsequent communications. For example, if a path already points to the root of some trait instance, there shouldn't be another path pointing to a subtree of the same trait instance. While this behavior is not seen as an error, it wastes resources.

Elements `SubscribeToAllEvents` and `LastObservedEventIdList` are designed for event delivery. `SubscribeToAllEvents`, if true, indicates the client's intention to receive all events generated by this publisher. There is no filtering or scoping defined in this version of the protocol. `LastObservedEventIdList` is an array of tuples describing, for each event importance, what is the last event ID and resource ID this client has received from this publisher. If a particular combination of (`SourceID`, `EventImportance`, `EventID`) is not represented in the list, the publisher will assume that the subscriber requested (SourceID, `EventImportance`, 0), such as events for that particular `SourceID`, `EventImportance` from the beginning of the event buffer.

Table 10: *TLV elements in the payload of Subscribe request*

Tag	Value	Description
CS: `SubscriptionID`	1	ID for this subscription
CS: `SubscribeTimeOutMin`	2	Timeout in seconds

CS: SubscribeTimeOutMax	3	
CS: SubscribeToAllEvents	4	True if this client demands to receive all events generated locally with all events proxied through this node
CS: LastObservedEventIdList	5	List of previously observed events
	6-19	Reserved
CS: PathList	20	Combined, records in both arrays should specify the path to subscribe to, and for any particular path, which version is current in the client's cache.
CS: VersionList	21	
CS: SourceID	1	Combined, these three properties uniquely determines the last event ever observed from this particular source
CS: EventImportance	2	
CS: EventID	3	

Figure 39: Listing: Subscribe request payload example schema

```
AnonymousTag = {
```

SubscriptionID = Optional, unsigned integer value

SubscribeTimeOutMin = Optional, unsigned integer value, seconds. Default is 1 second if absent.

SubscribeTimeOutMax = Optional, unsigned integer value, seconds. Default is 2147483647 seconds if absent.

SubscribeToAllEvents = Optional, boolean. Default is false if absent.

LastObservedEventIdList = Optional, must contain at least one element if present [

Elements in this array are anonymous structures of the same schema.

{

SourceID = Optional, could be any type and any value, default to the resource ID of the publisher. This is for the resource which generated the last observed event described in this element.

EventImportance = Mandatory, unsigned integer value. Every importance level must only appear at most once in LastObservedEventIdList.

EventID = Mandatory, unsigned integer value.

}

...

```
   ]
   PathList = Optional, must contain at least one path if present [
      <1st path>
      <2nd path>
      <3rd path>
      ...
   ]
   VersionList = Optional, a Version List
}
```

6.5.2. Subscribe response

The Subscribe response message is sent from a publisher to a client usually following a series of fragmented subscribe response messages, all in the same exchange context where the original subscribe request entered.

`SubscriptionID` is an unsigned integer generated at the publisher. The recommended value is 64-bit generated by a secure random number source. The reason why is to avoid various types of conflict and can be seen as a unique ID on both the publisher and the client, even in the mutual subscription cases.

If present, `SubscribeTimeOut` means the publisher will consider this subscription dead if there is no any activity regarding the subscription in that time period. That implies a client should also monitor the activity and send additional subscribe confirm request messages occasionally to avoid termination of the subscription. To make a reasonable choice for this value, a publisher should consider using `SubscribeTimeOutMin` and `SubscribeTimeOutMax` in the subscribe request. If this value is not acceptable by the client, it could cancel the subscription.

In the absence of `SubscribeTimeOut`, the publisher will not terminate the subscription just because of periods of inactivity.

Regardless of the timeout setting, the client is still free to send subscribe confirm requests when it has to verify the liveness of the subscription. The publisher is not allowed to use this mechanism.

The array `LastVendedEventIdList` contains information for every combination of resources and event importances this publisher can deliver now. The event ID belongs to either the last vended event from this publisher, or the last re-published event from the resource ID. If the publisher hasn’t published any event of a particular importance, or if a publisher hasn’t re-published any event of some importance for some publisher, there would be no elements describing those combinations.

`PossibleLossOfEvents` is a best effort made by the publisher to warn a client about the possibility of event loss between subscriptions. This is a global setting and not directly associates with any event source or importance.

Table 11: *TLV elements in the payload of Subscribe response*

Tag	Value	Description
CS: `SubscriptionID`	1	ID for this subscription
CS: `SubscribeTimeOut`	2	Timeout in seconds
	3-9	Reserved
CS: `PossibleLossOfEvents`	10	True if there could have been some loss of event
CS: `LastVendedEventIdList`	11	Array of last vended event IDs by this
CS: `SourceID`	1	Combined, these three properties uniquely determines the last event ever vended from a particular source
CS: `EventImportance`	2	
CS: `EventID`	3	

Figure 40: Listing: Subscribe response payload example schema

`AnonymousTag = {`

`SubscriptionID` = Mandatory, unsigned integer value

`SubscribeTimeOut` = Optional, integer value, seconds

`PossibleLossOfEvents =` Optional, boolean value. Default to false if absent.

`LastVendedEventIdList` = Optional but could only be present in subscription with events. Must not be empty when present.

[

Elements in this array are anonymous structures of the same schema.

{

`SourceID` = Optional, could be any type and any value, default to the resource ID of the publisher. This is for the resource which generated the last observed event described in this element.

`EventImportance =` Mandatory, unsigned integer value. Every importance level can only appear once in `LastVendedEventIdList`.

`EventID =` Mandatory, unsigned integer.

```
      }
      ...
   ]
}
```

6.6. Subscribe cancellation

6.6.1. Subscribe cancel request

This message can be sent from any party in a one-way or mutual subscription.

Table 12: *TLV elements in the payload of Subscribe cancel request*

Tag	Value	Description
CS: SubscriptionID	1	ID for subscription to be canceled

Figure 41: Listing: Subscribe cancel request payload example/schema

```
AnonymousTag = {
   SubscriptionID = Mandatory, unsigned integer value
}
```

6.6.2. Subscribe cancel response (status report)

On success, a status code of `CommonProfile:kStatus_Success` [2] should be in the status code.

On error, the most common status code could be `WDM:InvalidSubscriptionID`.

No matter what the status code is, the subscription is considered as terminated.

6.7. Subscribe liveness

6.7.1. Subscribe confirm request

This message can only be sent from a client to a publisher in a one-way or mutual subscription.

Table 13: *TLV elements in the payload of Subscribe confirm request*

Tag	Value	Description

CS: SubscriptionID	1	ID for subscription to be confirmed

Figure 42: Listing: payload for subscribe confirm request

```
AnonymousTag = {
   SubscriptionID = Mandatory, unsigned integer value
}
```

6.8. Subscribe confirm response (status report)

On success, a status code of `CommonProfile:kStatus_Success` should be in the status code.

On error, status code would indicate the reason for this failure, and the subscription is considered as terminated without an explicit cancel request.

6.9. Notification of changes

6.9.1. Notification request

All data elements in the data list don't have to belong to the same trait instance, but all changes regarding a version of any trait instance must be communicated in a single notification request.

Table 14: *TLV elements in the payload of notification request*

Tag	Value	Description
CS: `SubscriptionID`	1	
	2-9	Reserved
CS: `DataList`	10	
	11-19	Reserved
CS: `PossibleLossOfEvent`	20	
CS: `UTCTimestamp`	21	
CS: `SystemTimestamp`	22	
CS: `EventList`	23	

Figure 43: Listing: Notification request payload example/schema

```
AnonymousTag = {
   SubscriptionID = Mandatory, unsigned integer value
   DataList = Optional, must contain at least one data element if present [
      <1st data element>
      <2nd data element>
      <3rd data element>
      ...
   ]
   UTCTimestamp = Optional, unsigned integer. Number of milliseconds
   since 0:0:0 1/1/1970 UTC. UTC timestamp at the publisher's side,
   when the request was generated.
   SystemTimestamp = Optional, unsigned integer. Number of
   milliseconds since some arbitrary epoch. System timestamp at the
   publisher's side, when the request was generated.
   PossibleLossOfEvent = Optional, boolean value. Default to false if absent.
   EventList = Optional, must contain at least one event if present [
      <1st Event>
      <2nd Event>
      <3rd Event>
      ...
   ]
}
```

6.9.2. Notification response (status report)

On success, a status code of `CommonProfile:kStatus_Success` [2] must be in the status code.

If the client cannot accept the new value of any element in any data element in the Notification request, it can set the status code to `WDM:InvalidValueInNotification`, and populate the TLV structure with a list of rejection records. In every record, the client can optionally indicate the path and version being rejected, and the reason for rejection with an application specific error code. Note this should not be considered as fatal to the subscription, but the client becomes out of sync with the publisher for the data element mentioned in the rejection record.

Table 15: *TLV elements in the payload of Notification response*

Tag	Value	Description
CS: `RejectionList`	1	Array of rejection records
CS: `RejectPath`	1	Path being rejected. Copied from the notification request
CS: `RejectVersion`	2	Version being rejected. Copied from the notification request
CS: `RejectReason`	3	Reason for rejection

Figure 44: Listing: Additional information on invalid data example/schema

```
AnonymousTag = {
    RejectList = Optional, array of rejection records. must contain at least one element if
    present
    [
        {
            RejectPath = Mandatory, a path copied from the notification request being
            rejected
            RejectVersion = Mandatory, the version copied from the notification request
            being rejected
            RejectReason = Optional, Unsigned integer value for the reason of rejection.
            This value is application specific, probably status code defined by the trait's profile.
        }
        ...
    ]
    /* Additional tags can be added into this structure */
}
```

On any other error the subscription is considered as terminated without an explicit subscribe cancel request.

6.10. View

6.10.1. View request

The payload for a View request must be an anonymous TLV structure, which must contain one TLV element: the path list. Any extra TLV element is seen as protocol error. An empty path list is considered a protocol error as well.

Table 16: *TLV elements in the payload of View request*

Tag	Value	Description
CS: PathList	1	Array of paths

Figure 45: Listing: View request payload example/schema

```
AnonymousTag = {
   PathList = Mandatory [
      <1st Path>
      <2nd Path>
      <3rd Path>
      ...
   ]
}
```

6.10.2. View response

On error there should be a status report indicating the reason for failure. Some WDM specific errors are listed in earlier section, including `WDM:InvalidPath`.

On success, the payload shall contain a data list with the same number of entries in the original path list. Order of elements in the data list does not have to be the same as in the path list, but every path in the original path list must has a matching data element in the response.

Table 17: *TLV elements in the payload of View response*

Tag	Value	Description
CS: DataList	1	Array of data elements

Figure 46: Listing: View response payload example/schema

```
AnonymousTag = {
```

```
   DataList = Mandatory [
      <1st data element>
      <2nd data element>
      <3rd data element>
      ...
   ]
}
```

6.11. Update

6.11.1. Update request

If present, `ExpiryTime` indicates the intention to limit till when the request could be queued at various layers. A client should consider how inaccurate a publisher's clock could be, and adjust the expiry time accordingly. It's always possible that a publisher's clock doesn't allow it to honor this request with very high accuracy.

If present, `Argument` contains any number of TLV elements of any type. The tags used for these elements shall be profile tags. This is because the updating of all data elements of many profiles/traits would share the same set of arguments.

`DataList` is mandatory and must contain at least one data element. In each Data Element there is an optional `DataElementVersion`, indicating the intended version of this particular section of update described by this data element. Note that a publisher is allowed to apply elements in the data list in any order.

There is no transaction machinery in WDM updates. When an error is encountered at applying some data element, it's usually impossible to revert the changes already applied in earlier data elements. A client must assume an update could fail partially when it receives an error result or timeouts.

Table 18: *TLV elements in the payload of Update request*

Tag	Value	Description
CS: ExpiryTime	1	Indicates the intention to limit till when the request could be queued at various layers. A client should consider how inaccurate a publisher's clock could be, and adjust the expiry time accordingly. It's always possible that a publisher's clock doesn't allow it to honor this request with very high accuracy.

	2-9	Reserved
CS: Argument	10	Contains any number of TLV elements of any type. The tags used in this container do not subject to the same restrictions as in Update request and can be context-specific, for the content is parsed by the trait instance only.
	11-19	Reserved
CS: DataList	20	Data to update
Authenticator		Optional. The authenticator could be one of the accepted profile-specific tags and the type and schema would be defined by that.

Figure 47: Listing: Update request payload example/schema

```
AnonymousTag = {
    ExpiryTime = Optional, signed integer value, microseconds since 0:0:0 1/1/1970 UTC
    Argument = Optional {
        Example_Argument_1 = Any value of any type, to be referenced when updating all
        data elements
        Example_Argument_2 = Any value of any type, to be referenced when updating all
        data elements
    }
    DataList = Mandatory, must contain at least one data element [
        <1st data element>
        <2nd data element>
        <3rd data element>
        ...
    ]
    Authenticator = Optional.
}
```

6.11.2. In progress

There is no payload defined for this message.

6.11.3. Update response (status report)

The status report contains three segments:

- 32 bit profile ID for both the status code and TLV additional data
- 16 bit status code
- Anonymous TLV structure containing additional information

The TLV structure can contain any custom information in any order, but a Version List is mandatory when the status code is `CommonProfile:kStatus_Success` [2].

Table 19: *TLV elements in the payload of Update response*

Tag	Value	Description
CS: `VersionList`	1	Version list used to reflect the current versions of paths in the matching update request.

Figure 48: Listing: Additional information on success example/schema

```
AnonymousTag = {
    VersionList = Mandatory, a Version List.
    /* Additional tags can be added into this Structure */
}
```

Note that `VersionList` should contain exactly the same number of entries as in `DataList` of the update request, in the same order. It's allowed if some of the elements are NULL in this list, if the paths in matching update request is invalid or the version cannot be shared with this client. Otherwise, the version list should be populated with current versions of the paths, whether the update request succeeded or not.

6.12. Custom WDM command

This command request and response pair is designed to convey "commands" for specific profile, sharing some concept with other WDM message like path, expiry time, and authenticator. The need for WDM-defined command request and response message types comes from easing design of message routing and processing.

6.12.1. Command request

Table 20: *TLV elements in the payload of custom WDM commands*

Tag	Value	Description

CS: Path	1	Provides more information about the operation that needs to be taken, in addition to Profile ID and command type that is already in the payload.
CS: CommandType	2	Unsigned integer, type id of this command, in the scope of profile specified by Path
CS: ExpiryTime	3	Indicates the intention to limit till when the request could be queued at various layers. A client should consider how inaccurate a publisher's clock could be, and adjust the expiry time accordingly. It's always possible that a publisher's clock doesn't allow it to honor this request with very high accuracy.
CS: MustBeVersion	4	If the trait instance is not on this version, this command shall be rejected.
	5-19	Reserved
CS: Argument	20	This TLV structure contains any number of TLV elements of any type. The tags used in this container do not subject to the same restrictions as in command request and can be context-specific, for the content is parsed by the trait instance only.
Authenticator		Optional. The authenticator could be one of the accepted profile-specific tags and the type and schema would be defined by that.

Figure 49: Listing: Custom WDM command payload example/schema

AnonymousTag = {

Path = Mandatory, additional information about the operation if present

CommandType = Mandatory, unsigned integer. unique command type id within the profile specified in Path

ExpiryTime = Optional, signed integer value, microseconds since 0:0:0 1/1/1970 UTC

MustBeVersion = Optional, unsigned integer value

Argument = Optional, TLV structure {

Example_Argument_1 = Any value of any type

Example_Argument_2 = Any value of any type

}

Authenticator = Optional

}

6.12.2. In progress

There is no payload defined for this message.

6.12.3. Command response

Response message to a Custom Command could either a Status Report message, or a Command Response message. A Status Report message is used whenever there is some error detected in any of the layers the command flows through, and a full schema-conformant Command Response message is not possible. The typical Weave Status Report of status code and optionally Weave error code are carried in a Status Report message.

A Command Response message indicates the command has reached the application layer and the “response” section conforms to pre-agreed schema for that command request. Note that the definition of “success” is command-specific, and hence receiving a Command Response message doesn’t necessarily mean the command succeeded. Version information is mandatory no matter the result of the execution. It means either the current version or the version that reflects the effect of this command.

Table 20: *TLV elements in the payload of Command response*

Tag	Value	Description
CS: Version	1	Current version of the trait instance as referenced by the path in command request. Since there can be only one path in the request, only one version is present in the response.
CS: Response	2	This TLV structure contains any number of TLV elements of any type. The tags used in this container do not subject to the same restrictions as in command request and can be context-specific, for the content is parsed by the trait instance only.

Figure 50: Listing: Additional information on success example/schema

```
AnonymousTag = {
   Version = Mandatory, unsigned integer.
   Response = Optional, TLV structure {
      Example_Response_1 = Any value of any type
      Example_Response_2  = Any value of any type
   }
```

```
    /* Additional tags can be added into this Structure */
}
```

7. Reference

1. Weave: Data Management Profile (V2, the previous version of WDM) https://docs.google.com/a/nestlabs.com/document/d/1Hm_trFIvlsGJpnNzzctBudrnR8AOO-4_JrfliJL0Z1Y
2. Weave Status Report Profile https://docs.google.com/a/nestlabs.com/document/d/14MFf1ev2l_-5zfpNey4pxbC2-m8MANuXkaP7W0eZ36Y
3. Weave TLV Format https://docs.google.com/document/d/1VWG79nUK8C9NaVk8BoDYTr4F1ZJyjSu_pAIRfKUvWgY
4. Trait design guidelines for this generation of Nest products (TBD)
5. Delivery of Events in WDM 0.4.0 https://docs.google.com/a/nestlabs.com/document/d/1Vkeps-3i3PGYXUPD5mZCwQLr4sT_dlWuQY4LXr5BT9U
6. Event Definition in IDL 0.4 https://docs.google.com/a/nestlabs.com/document/d/1Tx-_l6Ib9eXfvLv4fC_KDIkeHa1WXXK4hxz7qLq7-aY
7. WDM Request Authentication https://docs.google.com/a/nestlabs.com/document/d/1BGFRJZWbTLjAViaUB10-FzG8a5egKOlEaLKF6srdBN4/edit?usp=sharing