SiLA 2 Part (B) - Mapping Specification

Release v1.1 - 19 March 2022

Notices

License

SiLA 2 Version History

For the version history, please refer to Part (A) (Current Version).

Refer to the Structure of the SiLA 2 Specification for details about how the different documents are related.

SiLA 2 Working Group Members

Please refer to Part (A) (Current Version).

SiLA 2 Working Group Organization

Please refer to Part (A) (Current Version).

SiLA 2 Roadmap

Please refer to Part (A) (Current Version).

SiLA 2 Adoptions

Please refer to Part (A) (Current Version).

Status of The SiLA 2 Specification

Please refer to Part (A) (Current Version).

Table of Contents

SiLA Data Types

Abstract

This document describes the technical mapping of SiLA 2.

Introduction

Principles and Background

There were many Thoughts on Possible Base Protocols for SiLA 2 (document only accessible to members). The conclusion is that SiLA must map to a base technology & communication standard that will survive for a considerable amount of time. It needs to be accessible, robust and open.

Among many possibilities, a combination of these technologies is our favorite:

- HTTP/2 - as a base communication layer,
- A “REST”-like communication paradigm,
- Protocol Buffers as data structures, and
- An interface description language, such as RAML, OpenAPI

Combining all these technologies nicely fits with gRPC, a microservice architectural framework with many language bindings.

In summary, SiLA 2 is based on HTTP/2 and Protocol Buffers (specified by the wire format of gRPC). Note that this document focuses on the mapping to gRPC for the release candidate, but will be updated to relate more directly to the wire format in the future.

Introduction to gRPC

The gRPC Framework is an open-source, universal remote procedure call (RPC) framework. The framework is based on HTTP/2 for transportation and Protocol Buffers for the interface description. It is made available for a variety of programming languages, which allows for deployment on different platforms and devices.

The basic elements of Protocol Files are the following:

1. Service: Services are the fundamental element for defining remote procedure calls (RPC) using Protocol Buffers. A service contains a set of RPCs associated with that service.
2. RPC: RPCs are contained within a service and represent a functionality of that service. An RPC has a request message and a response message. Either request, response or both may be streams of messages.
3. Message: Messages contain information and they are the primary element for providing data and receiving data from an RPC. Messages can include simple data types (Integer, Double, etc.), but may also be nested and contain messages within themselves. Messages are serialized using Protocol Buffers.

REST vs. gRPC

Source: https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711

Protobuf vs. JSON

One of the biggest differences between REST and gRPC is the format of the payload. REST messages typically contain JSON. This is not a strict requirement but in practice the whole REST ecosystem is focused on JSON. With very few exceptions, REST APIs accept and return JSON.

gRPC, on the other hand, accepts and returns Protocol Buffer messages. From a performance point of view, Protobuf is a very efficient and packed format. JSON, on the other hand, is a textual format. JSON can be compressed, but then the benefit of having a textual format is lost.

HTTP/2 vs. HTTP 1.1

REST depends heavily on HTTP (usually HTTP 1.1) and the request-response model whereas gRPC uses the newer HTTP/2 protocol.

There are several problems that plague HTTP 1.1 that HTTP/2 fixes. Here are the major ones.

HTTP 1.1 Is Too Big and Complicated

HTTP 1.0 RFC 1945 is a 60-page RFC. HTTP 1.1 was originally described in RFC 2616, which consisted of 176 pages. However, later the IETF split it up into six different documents: RFC 7230, 7231, 7232, 7233, 7234, and 7235 with an even higher combined page count. HTTP 1.1 allows for many optional parts that contribute to its size and complexity.

Latency Issues

HTTP 1.1 is sensitive to latency. A TCP handshake is required for each individual request, and larger numbers of requests take a significant toll on the time needed to load a page. Ongoing improvements in available bandwidth do not solve these latency issues in most cases.

Head of Line Blocking

The restriction on the number of connections to the same domain (used to be just 2, today 6-8) significantly reduces the ability to send multiple requests in parallel.

With HTTP pipelining, requests can be sent while waiting for the response of a previous request, effectively creating a queue. But that introduces other problems. If the request gets stuck behind a slow request then response time will suffer.

There are also other concerns like performance and resource penalties when switching lines. At the moment, HTTP pipelining is not widely enabled.

How HTTP/2 Addresses the Problems

HTTP/2, which came out of Google's SPDY, maintains the basic premises and paradigms of HTTP:

- request-response model over TCP

- resources and verbs
- http:// and https:// URL schemas
- But the optional parts of HTTP 1.1 were removed.

To address the negotiating protocol due to the shared URL schema, there is an upgrade header. The HTTP/2 protocol being binary holds a great advantage as troubleshooting and constructing requests manually is far easier. The binary framing reduces complexity of handling frames in HTTP 1.1.

The major improvement of HTTP/2 is that it uses multiplexed streams. A single HTTP/2 TCP connection can support many bidirectional streams. These streams can be interleaved (no queuing), and multiple requests can be sent at the same time without a need to establish new TCP connections for each one. In addition, servers can now push notifications to clients via the established connection (HTTP/2 push).

Messages vs. Resources and Verbs

REST is built very tightly on top of HTTP. It does not just use HTTP as a transport, but embraces all its features and builds a consistent conceptual framework on top of it. In theory, this sounds great. In practice, it has been proven difficult to implement REST properly.

REST has been and is very successful, but most implementations do not fully adhere to the REST philosophy and use only a subset of its principles. The reason being that it's challenging to map business logic and operations into the strict REST world.

The conceptual model used by gRPC is to have services with clear interfaces and structured messages for requests and responses. This model translates directly from programming language concepts like interfaces, functions, methods, and data structures. It also allows gRPC to automatically generate client libraries.

Streaming vs. Request-Response

REST supports only the request-response model available in HTTP 1.x. But gRPC takes full advantage of the capabilities of HTTP/2 and lets you stream information constantly. There are several types of streaming.

Server-Side Streaming

The server sends back a stream of responses after getting a client request message. After sending back all its responses, the server’s status details and optional trailing metadata are sent back to complete on the server side. The client completes once it has all the server’s responses.

Client-Side Streaming

The client sends a stream of multiple requests to the server. The server sends back a single response, typically but not necessarily after it has received all the client’s requests, along with its status details and optional trailing metadata.

Bidirectional Streaming

In this scenario, the client and the server send information to each other in pretty much free form (except the client initiates the sequence). Eventually, the client closes the connection.

Strong Typing vs. Serialization

The REST paradigm does not mandate any structure for the exchanged payload. It is typically JSON. Consumers do not have a formal mechanism to coordinate the format of requests and responses. The JSON must be serialized and converted into the target programming language both on the server side and client side. The serialization is another step in the chain that introduces the possibility of errors as well as performance overhead.

The gRPC service contract has strongly typed messages that are converted automatically from their Protocol Buffer representation to your programming language of choice both on the server and on the client.

JSON, on the other hand, is theoretically more flexible because you can send dynamic data and do not have to adhere to a rigid structure.

= = = START OF NORMATIVE PART = = =

Structure of the SiLA 2 Specification

[COMPLETE; as of 0.1]

The SiLA 2 specification is a multi part specification:

- Part (A) - Overview, Concepts and Core Specification (current version): contains the user requirements specification of SiLA 2. It describes **what** SiLA would like to achieve.
  It describes the core of SiLA 2 including the Features Framework in details, but does not map to a specific implementation. This document deals with:
  - Overview of the design goals
  - SiLA 2 ·Features· specification
  - SiLA 2 ·Features· design rules
  - SiLA 2 ·Features· development and balloting process
  - Error handling and ·SiLA Data Types·
  - Security and Authentication
  - ·SiLA Server Discovery· and ·SiLA Feature Discovery·
- Part (B) - Mapping Specification (current version of **this document**): describes **how** the user requirements shall be implemented. The mapping specification document describes the specific mapping to a technology and an actual implementation
- Part (C) - Standard Features Index (current version): The Standard Features Index document is an index to ·Features· that are either standardized or currently being discussed to become standardized.

Terminology and Conformance Language

[COMPLETE; as of 0.1]

Unless otherwise noted, the entire text of this specification is normative. Exceptions include:

- Notes
- Sections explicitly marked non-normative
- Examples and their commentary
- Informal descriptions of details formally and normatively stated elsewhere (such informal descriptions are typically introduced by phrases like "Informally, ..." or "It is a consequence of ... that ...")

Explicit statements that some material is normative are not implying that other material is non-normative, other than the items mentioned in the list just described.

Special terms are defined at their point of introduction in the text. For example:

[Definition: Term] a **Term** is something used with a special meaning. The definition is labeled as such and the **Term** it defines is displayed in boldface. The end of the definition is not specially marked in the displayed or printed text. Uses of defined **Terms** are links to their definitions, set off with middle dots, for instance ·Term·.

Normative text describes one or both of the following kinds of elements:

- Vital elements of the specification

- Elements that contain the conformance language keywords as defined by RFC2119 "Key words for use in RFCs to Indicate Requirement Levels"

Informative text is potentially helpful to the user, but dispensable. Informative text can be changed, added, or deleted editorially without negatively affecting the implementation of the specification. Informative text does not contain conformance keywords.

All text in this document between "START OF NORMATIVE PART" and "END OF NORMATIVE PART" is, by default, normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 "Key words for use in RFCs to Indicate Requirement Levels".

Architecture

[COMPLETE; as of 0.1]

SiLA 2 runs over HTTP/2 (SiLA ·Connection·) and uses Protocol Buffers to serialize payload data (the data transported between ·SiLA Client· and ·SiLA Server·).

SiLA 2 relies on the wire format specified by gRPC to implement the ·Connection· and transport of data.

Therefore, any implementation of SiLA 2 MAY use the gRPC libraries to implement ·SiLA Clients· and ·SiLA Servers·, but MAY also be implemented from scratch without using gRPC, as long as the implementations strictly behaves as and uses the wire format as specified for gRPC.

Client-Server Communication

[COMPLETE; as of 1.1]

SiLA, as well as HTTP/2, are based on a client - server architecture. The server provides a set of services and clients can use these services. In SiLA the server is termed ·SiLA Server· and the client is referred to as ·SiLA Client·. ·Features· are the services provided by the ·SiLA Server· and the ·SiLA Client· uses the services (·Features·) of a ·SiLA Server·.

Two ·Connection Methods· are distinguished:

1. ·Client-Initiated Connection Method·: All services provided by the ·SiLA Server· MUST be exposed on one specific socket (i.e. port and IP address), that the ·SiLA Client· can connect to.
2. ·Server-Initiated Connection Method·: All services provided by the ·SiLA Server· MUST be served inside a bi-directional gRPC streaming of protocol buffer request and response messages, that the ·SiLA Server· has established with the ·SiLA Client·.

Connection

[COMPLETE; as of 0.1, updates: 1.1]

As stated in Part A, the ·Connection· is either established by the ·SiLA Client· (·Client-Initiated Connection Method·) or the ·SiLA Server· (·Server-Initiated Connection Method·). The

·Connection· can be closed by both, the ·SiLA Client· and the ·SiLA Server· and they MUST be both fully tolerant to closing and dropping of the ·Connection· at any time.

According to Part A, ·Connections· must always be secured by encryption.

·Client-Initiated Connection Method·

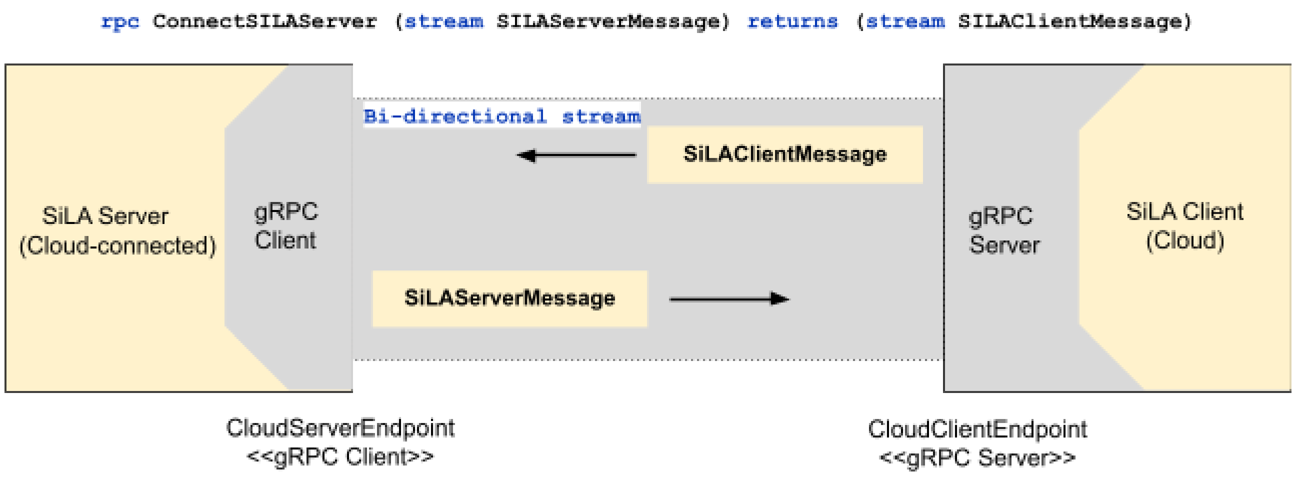
[COMPLETE; as of 0.1, updates: 1.1]

With the ·Client-Initiated Connection Method·, the ·Connection· is established by the ·SiLA Client· calling gRPC services on the ·SiLA Server·.

·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

With the ·Server-Initiated Connection Method·, the ·Connection· is established by calling the ‘**`connectSiLAServer`**’ gRPC service on the ·SiLA Client·, see the ·Connection Configuration Service Feature·.


rpc ConnectSILAServer (stream SILAServerMessage) returns (stream SILAClientMessage)
Bi-directional stream
SiLAClientMessage
SiLAServerMessage
SiLA Server (Cloud-connected)
gRPC Client
gRPC Server
SiLA Client (Cloud)
CloudServerEndpoint <<gRPC Client>>
CloudClientEndpoint <<gRPC Server>>

This service opens a bidirectional read-write stream, where both sides send a sequence of ·SiLA Client Messages· and ·SiLA Server Messages·.

protobuf

```
// from SiLACloudConnector.proto
service CloudClientEndpoint {
    rpc ConnectSiLAServer (stream SILAServerMessage)
                 returns (stream SILAClientMessage ) {}
}
```

SiLA Server Message

[COMPLETE; as of 1.1]

[Definition: SiLA Server Message] The **SiLA Server Message** is used for all gRPC messages sent from the ·SiLA Server· to the ·SiLA Client· through the bidirectional read-write stream. A **SiLA Server Message** has a specific purpose as defined below.

Message Field	Meaning

requestUUID	Unique ID to map response(s) to request. Copied from the request of the SiLA Client.
<oneof message>	One of the SiLA Server Messages mentioned below.

The requestUUID is a UUID, copied from the respective SiLA Client Message, e.g. “53336864-2da2-4b23-93c4-c1d8537645d6”.

·SiLA Server Message· Type	Purpose
`CommandResponse`	Result message of an CommandExecution message
`ObservableCommandConfirmation`	Confirmation of an observable CommandInitiaition message, contains information about execution.
`ObservableCommandExecutionInfo`	Subscription message to the ExecutionInfo of an Observable Command
`ObservableCommandIntermediateResponse`	Subscription message to the intermediate results of an ObservableCommand
`ObservableCommandResponse`	Result message of an ObservableCommand
`GetFCPAffectedByMetadataResponse`	Response for ·SiLA Client Message· GetFCPAffectedByMetadataRequest
`PropertyValue`	Response for ·SiLA Client Message· PropertyRead
`ObservablePropertyValue`	Subscription message to the values of an ObservableProperty
`CreateBinaryResponse`	Response for ·SiLA Client Message· CreateBinary
`UploadChunkResponse`	Response for ·SiLA Client Message· UploadChunk
`DeleteBinaryResponse`	Response for ·SiLA Client Message· DeleteBinary
`GetBinaryInfoResponse`	Response for ·SiLA Client Message· GetBinaryInfo
`GetChunkResponse`	Response for ·SiLA Client Message· GetChunk
`BinaryTransferError`	Error that can occur during a ·Binary Transfer·
`SiLAError`	Error that can occur during processing commands
`SiLAError`	Error that can occur during processing properties

protobuf

```
// from SiLACloudConnector.proto
message SILAServerMessage {

   string requestUUID = 1;

   oneof message {
     CommandResponse commandResponse = 2;
     ObservableCommandConfirmation observableCommandConfirmation = 3;
     ObservableCommandExecutionInfo observableCommandExecutionInfo = 4;
     ObservableCommandIntermediateResponse intermediateResponse = 5;
     ObservableCommandResponse observableCommandResponse = 6;
     GetFCPAffectedByMetadataResponse metadataResponse = 7;
     PropertyValue propertyValue = 8;
     ObservablePropertyValue observablePropertyValue = 9;
     CreateBinaryResponse createBinaryResponse = 10;
     UploadChunkResponse uploadChunkResponse = 11;
     DeleteBinaryResponse deleteBinaryResponse = 12;
     GetBinaryInfoResponse getBinaryInfoResponse = 13;
     GetChunkResponse getChunkResponse = 14;
     BinaryTransferError binaryTransferError = 15;
     SiLAError commandError = 16;
     SiLAError propertyError = 17;
   }
}
```

SiLA Client Message

[COMPLETE; as of 1.1]

[Definition: SiLA Client Message] The **SiLA Client Message** is used for all gRPC messages sent from the ·SiLA Client· to the ·SiLA Server· through the bidirectional read-write stream.

Message Field	Meaning
requestUUID	Unique ID used to map response(s) to request. Every request should contain a new ID.
<oneof message>	One of the SiLA Client Messages mentioned below.

The cloudRequestUUID is a UUID created by the ·SiLA Client·.

·SiLA Client Message· Type	Purpose
CommandExecution	unobservable command execution Possible responses: ● CommandResponse ● SiLAError
CommandInitiation	observable command initiation Possible responses: ● ObservableCommandConfirmation ● SiLAError
CommandExecutionInfoSubscription	subscription to execution info Possible responses: ● ObservableCommandExecutionInfo ● SiLAError

CancelCommandExecutionInfoSubscription	cancel execution info subscription Possible responses: ● ObservableCommandIntermediateResponse ● SiLAError
CommandIntermediateResponseSubscription	subscription to intermediate results Possible responses: ● ObservableCommandIntermediateResponse ● SiLAError
CancelCommandIntermediateResponseSubscription	cancel intermediate result subscription
CancelCommandExecutionInfoSubscription	Cancel execution info subscription
CommandGetResponse	retrieve result Possible responses: ● ObservableCommandResponse ● SiLAError
GetFCPAffectedByMetadataRequest	Possible responses: ● GetFCPAffectedByMetadataResponse
PropertyRead	get property Possible responses: ● PropertyValue ● SiLAError
PropertySubscription	subscription to property Possible responses: ● ObservablePropertyValue ● SiLAError
CancelPropertySubscription	cancel property subscription
CreateBinaryUploadRequest	Create binary upload request Possible responses: ● CreateBinaryResponse ● BinaryTransferError
UploadChunkRequest	Upload chunk request Possible responses: ● UploadChunkResponse ● BinaryTransferError
DeleteBinaryRequest	Delete binary request Possible responses: ● DeleteBinaryResponse ● BinaryTransferError
GetBinaryInfoRequest	Get binary info request Possible responses:

	• GetBinayInfoResponse • BinaryTransferError
GetChunkRequest	Get chunk request Possible responses: • GetChunkResponse • BinaryTransferError

protobuf

```
// from SiLACloudConnector.proto
message SILAClientMessage {

   string requestUUID = 1;

   oneof message {
     CommandExecution commandExecution = 2;
     CommandInitiation commandInitiation = 3;
     CommandExecutionInfoSubscription infoSubscription = 4;
     CancelCommandExecutionInfoSubscription cancelInfoSubscription = 5;
     CommandIntermediateResponseSubscription responseSubscription = 6;
     CancelCommandIntermediateResponseSubscription
cancelIntermediateResponseSubscription = 7;
     CommandGetResponse commandGetResponse = 8;
     GetFCPAffectedByMetadataRequest metadataRequest = 9;
     PropertyRead propertyRead = 10;
     PropertySubscription propertySubscription = 11;
     CancelPropertySubscription cancelPropertySubscription = 12;
     CreateBinaryUploadRequest createBinaryUploadRequest = 13;
     DeleteBinaryRequest deleteBinaryRequest = 14;
     UploadChunkRequest upoadChunkRequest = 15;
     GetBinaryInfoRequest getBinaryInfoRequest = 16;
     GetChunkRequest getChunkRequest = 17;
   }
}
```

Subscription

[COMPLETE; as of 0.1]

In contrast to traditional subscription schemes, there is no broker involved in SiLA and the subscription is peer-to-peer concerning subscription to ·Observable Properties· or ·Observable Command· progress.

The subscription is implemented by using a server side streaming RPC that can only be canceled by the ·SiLA Client·. By default the ·SiLA Server· is required to send any change in the requested data on a best effort basis.

The ·SiLA Server· is also expected to always send the current value of the requested data upon the initial subscription request.

High Level Mapping Overview

[COMPLETE; as of 1.1]

The mapping overview should provide a general idea of how the SiLA concepts are mapped to the gRPC framework for both, the ·Client-Initiated Connection Method· and the ·Server-Initiated Connection Method·.

A detailed description of how the SiLA 2 core and ·Features· are mapped to the gRPC framework is provided in the Feature Implementation section.

SiLA Definition	gRPC Mapping
Command Parameter, Command Response, Intermediate Command Response, Property Value, SiLA Data Type	Mapped to Protocol Buffer messages.

Mapping in Case of The ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1, updates: 1.1]

For the ·Client-Initiated Connection Method·, ·Features· are defined as services in gRPC. ·Features· have ·Commands· and ·Properties·, which are implemented using RPCs in the gRPC Framework.

SiLA Definition	gRPC Mapping
Command	Executing ·Commands· of a ·Feature· are implemented using a single (for ·Unobservable Commands·) or multiple (for ·Observable Commands·) RPCs per ·Command·, defined within a service within a {·Feature Identifier·}.proto file. There are specific mappings for ·Unobservable Commands· and ·Observable Commands·.
Property	Reading ·Properties· are implemented using a single RPC for ·Unobservable Properties·, and a streaming RPC for ·Observable Properties·. All RPCs are defined within a service within a {·Feature Identifier·}.proto file.
Validation Errors, Execution Errors, Framework Errors	Mapped to a specific SiLA error Protocol Buffer message serialized into the status message of the gRPC error model.
SiLA Client Metadata	Mapped to a Protocol Buffer message containing the ·SiLA Client Metadata· content and an RPC returning the ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties·, which the ·SiLA Client Metadata· applies to.

Mapping in Case of The ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

For the ·Server-Initiated Connection Method·, ·Features· do not have a direct representation on gRPC level. The ·Commands· and ·Properties· can be accessed by their ·Fully Qualified Identifiers· through sending ·SiLA Client Messages· and receiving ·SiLA Server Messages·.

SiLA Definition	gRPC Mapping
Command	Executing ·Commands· of a ·Feature· are implemented by sending a single (for ·Unobservable Commands·) or multiple (for ·Observable Commands·) ·SiLA Client Messages· per ·Command·. There are specific mappings for ·Unobservable Commands· and ·Observable Commands·.
Property	Reading ·Properties· are implemented using a single ·SiLA Client Messages· for ·Unobservable Properties·, and multiple ·SiLA Client Messages· for ·Observable Properties·.
Validation Errors, Execution Errors, Framework Errors	Mapped to a specific SiLA error Protocol Buffer message and sent through a ·SiLA Server Message·.
SiLA Client Metadata	Mapped to a Protocol Buffer message containing the ·SiLA Client Metadata· content and an ·SiLA Client Messages· for retrieving ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties·, which the ·SiLA Client Metadata· applies to.

Nomenclature

[COMPLETE; as of 0.1]

This specification document directly relates to gRPC definitions, therefore these definitions use the proto 3 syntax.

Protocol Buffer definitions are always titled with **protobuf** as block code and formatted with `Consolas` font, like this:

protobuf

```
service GreetingProvider {
    rpc SayHello (SayHello_Responses) returns (SayHello_Responses) {}
}

message SayHello_Parameters {
    String Name = 1;
    String Type = 2;
}

message SayHello_Responses {
    String Greeting = 1;
}
```

Remote Procedure Calls (RPCs) defined using gRPC are just called RPC in this document.

Replacing SiLA Terms in Protocol Buffer Definitions

SiLA ·Terms· between "<" and ">", e.g. <·Command Identifier·> MUST be resolved and replaced by the respective value they are referring to, according to the definition of the ·Term· and the mapping indicated in this document.

Example

For a ·Command Identifier· of "MyCommand", the following Protocol Buffer definition

protobuf

```
message <·Command Identifier·>_Parameters {
   …
}
```

becomes:

protobuf

```
message MyCommand_Parameters {
   …
}
```

Handling Multiplicity of Values of the Same Term

Some terms may have multiple values in a specific context. These multiples are indexed with an integer number starting from 1. The letter "N" indicated the last index.

For example, as multiple ·Command Parameters· can exist as part of a ·Command·, there are multiple ·Command Parameter Identifiers· as a consequence:

protobuf

```
message {Command Identifier}_Parameters {
    <·Command Parameter Data Type· 1> <·Command Parameter Identifier· 1> = 1;
    <·Command Parameter Data Type· 2> <·Command Parameter Identifier· 2> = 2;
    …
    <·Command Parameter Data Type· N> <·Command Parameter Identifier· N> = N;
}
```

Leaving Out Non-Relevant Part in Protocol Buffer Definitions

The horizontal ellipsis "…" indicates further content of the Protocol Buffer definition not relevant in the current context, that is specified elsewhere in this document. For example:

protobuf

```
service <·Feature Identifier·> {
    …
}
```

Informational Feature Elements

[COMPLETE; as of 0.1]

Some elements of the ·Feature Definitions· have no direct mapping with regards to the implementation:

- ·Display Name·: No implication on the implementation, these serve only a human readable format. This is the case for all ·Display Names· used in a ·Feature·, e.g. for ·Commands· or ·Parameters·.
- ·Description·: Defines how the implementation should behave but does not have a direct influence on the mapping.

Feature Implementation

[COMPLETE; as of 0.1]

The Feature Implementation Guideline defines the process of implementing a ·Feature· in gRPC. This guideline gives instructions conforming with the SiLA standard to translate the abstract ·Feature Definition Language· into the Protocol Buffer files, the interface definition language of gRPC. Additionally guidelines on the expected behavior are given.

·Features· comprise ·Commands· and ·Properties·, for which the specific implementation guidelines are provided in the following sections.

Proto Files

[COMPLETE; as of 0.1]

The standard is defined with Protobuf version 3 syntax (see Language Guide (proto3)), which, on the top of the file, requires:

protobuf

```
syntax = "proto3";
```

Framework Definitions

[COMPLETE; as of 0.1]

Some definitions (such as ·SiLA Data Types·) will be uniform across all Protocol Buffer definitions. It is RECOMMENDED to import the respective Protocol Buffer files provided by SiLA provided on GitLab.

The Protocol Buffer compiler can resolve dependencies from different sources by adding it to its "proto_path". The service requests should be cross-compatible if other Protocol Buffer definitions are written that comply with the definitions in this document.

Additional Framework definitions that are necessary for the commands or errors respectively are defined in the chapter describing these concepts.

Feature Identifier

[COMPLETE; as of 0.1]

A ·Feature· maps to a service in gRPC, which is defined in the corresponding Protocol Buffer file. Each ·Feature Definition· Protocol Buffer file MUST contain the definition of only a single ·Feature·. The ·Feature Identifier· is used as the name of the gRPC service as well as the name of the Protocol Buffer file the ·Feature· is defined in.

Note that Protocol Buffer files are restricted to only containing one gRPC service.

The Protocol Buffer file should be named “{Feature Identifier}.proto”.

protobuf

```
service <·Feature Identifier·> { … }
```

Example

[COMPLETE; as of 0.1]

·Feature Identifier·: “LockController”

·Feature Definition· file: “LockController.sila.xml”

Protocol Buffer file: “LockController.proto”

protobuf

```
service LockController { … }
```

Protobuf Package

[COMPLETE; as of 0.1]

[Definition: Name of the Protocol Buffer Package] SiLA 2 Part A specifies how to fully qualify ·Identifiers·. SiLA 2 Part A also requires versioning and defines how backwards and forwards versioning should work. In order to implement these requirements, all Protocol Buffer files should specify a **Name of the Protocol Buffer Package** that SHALL be composed as follows.

The following components are used to create the ·Name of the Protocol Buffer Package·:

- SiLA generation: Needs to be “sila2”.
- ·Originator·: Taken as is.
- ·Category·: Taken as is.
- ·Feature Identifier·: Converted to all lowercase, i.e. “A” -> “a”, “B” -> “b”, … “Z” -> “z”.
- ·Feature Version·: The string “v” (small letter “v”) followed by the ·Major Feature Version· of the ·Feature· (not including the ·Minor Feature Version·).

The elements of the ·Name of the Protocol Buffer Package· are separated by a “.” (dot) and the ·Name of the Protocol Buffer Package· is written in lowercase only, see Protocol Buffers package names.

The ·Name of the Protocol Buffer Package· is included in the Protocol Buffer file using the `package` keyword. In gRPC, the package name is used to avoid name clashes between protocol message types and the usage of package names has different implications on the implementation depending on the programming language.

Example

[COMPLETE; as of 0.1]

This:

- SiLA generation: always “sila2”
- ·Originator·: org.silastandard
- ·Category·: core
- ·Feature Identifier·: silaservice
- ·Feature Version·: 1 (only major version)

will result in:

protobuf

```
package sila2.org.silastandard.core.silaservice.v1;
```

Commands

[COMPLETE; as of 0.1]

There are two different ·Command· execution behaviors defined in Part A: ·Unobservable Commands· and ·Observable Commands·.

Parameters and Responses

[COMPLETE; as of 0.1]

Because it is not possible in gRPC to define multiple input or output parameters (this is intentionally, by design of Protocol Buffers), multiple ·Command Parameters· or ·Command Responses· are bundled into one gRPC message each.

Command Parameters gRPC Mapping

[COMPLETE; as of 0.1]

Command Parameters gRPC Mapping Over the ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

protobuf

```
message <·Command Identifier·>_Parameters {
    <·Command Parameter Data Type· 1> <·Command Parameter Identifier· 1> = 1;
    <·Command Parameter Data Type· 2> <·Command Parameter Identifier· 2> = 2;
    …
    <·Command Parameter Data Type· N> <·Command Parameter Identifier· N> = N;
}
```

All ·Command Parameter· fields are mandatory, the ·SiLA Server· MUST throw a ·Validation Error· if the fields are missing.

In case there are no ·Command Parameters· defined, an empty Protocol Buffer message with no fields MUST be used.

Command Parameters gRPC Mapping Over·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The ·Unobservable Command· is executed by sending a ·SiLA Client Message· of type CommandExecution, containing a CommandParameter Protocol Buffer message like this:

Message Field	Meaning
metadata	The list of ·SiLA Client Metadata·, if any. For details refer to Over “Server-initiated” Connection.
parameters	This is the serialized form of the `<·Command Identifier·>_Parameters` gRPC message as defined in Command Parameters gRPC Mapping Over the ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message CommandParameter {
    repeated Metadata metadata = 1;
    bytes parameters = 2;
}
```

Command Response gRPC Mapping

[COMPLETE; as of 0.1]

Command Response gRPC Mapping Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

protobuf

```
message <·Command Identifier·>_Responses {
    <·Command Response Data Type· 1> <·Command Response Identifier· 1> = 1;
    <·Command Response Data Type· 2> <·Command Response Identifier· 2> = 2;
    …
    <·Command Response Data Type· N> <·Command Response Identifier· N> = N;
}
```

In case there are no ·Command Responses· defined in the ·Feature Definition·, an empty Protocol Buffer message with no fields MUST be used, like this:

protobuf

```
message <·Command Identifier·>_Responses {
}
```

In case of an error on the RPC execution, the ·Command Responses· are invalid and MUST be ignored by the ·SiLA Client·.

Command Response gRPC Mapping Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

There is a specific mapping for ·Unobservable Commands· as well as for ·Observable Commands· that is described in the respective chapters below, see Command Result Retrieval Over ·Server-Initiated Connection Method· and Retrieving Intermediate Command Responses Over ·Server-Initiated Connection Method·.

Example

[COMPLETE; as of 0.1]

The two ·Command Parameters· for the "SendLetter" ·Command·, "Name" and "Address" will result in (gRPC mapping in the case of "Client-Initiated" ·Connection·):

protobuf

```
message SendLetter_Parameters {
    String Name = 1;
    String Address = 2;
}
```

One ·Command Response· called "Greeting" for the "SayHello" ·Command· will result in:

protobuf

```
message SayHello_Responses {
    String Greeting = 1;
}
```

Executing Unobservable Commands

[COMPLETE; as of 0.1, updates: 1.1]

Executing Unobservable Commands Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

The ·Unobservable Command· execution is implemented using a single RPC named like the ·Command Identifier·.

protobuf

```
rpc <·Command Identifier·> (<·Command Identifier·>_Parameters) returns (<·Command
Identifier·>_Responses) {}
```

Note that there is no potential naming conflict between the naming of the RPC and the encapsulating service, they can be named in the same way.

Example

[COMPLETE; as of 0.1]

A complete example for a simple ·Command· with the ·Command Identifier· “MoveSample”, inside a ·Feature· named “RobotMoveController ” looks like:

protobuf

```
service RobotMoveController {
    rpc MoveSample (MoveSample_Parameters) returns (MoveSample_Responses) {}
}

message MoveSample_Parameters {
    Integer PlateSiteA = 1;
    Integer PlateSiteB = 2;
}

message MoveSample_Responses {
    Real Precision = 2;
}
```

Executing Unobservable Commands Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The ·Unobservable Command· is executed by sending a ·SiLA Client Message· of type CommandExecution.

Message Field	Meaning
fullyQualifiedCommandId	The ·Fully Qualified Command Identifier· of the ·Command· to be executed.
commandParameter	The CommandParameter Protocol Buffer message, see Command Parameters gRPC Mapping Over ·Server-Initiated Connection Method·

protobuf

```
// from SiLACloudConnector.proto
message CommandExecution {
   string fullyQualifiedCommandId = 1;
   CommandParameter commandParameter = 2;
}
```

The ·Command Response· of an ·Unobservable Command· is returned by sending a ·SiLA Server Message· of type CommandResponse.

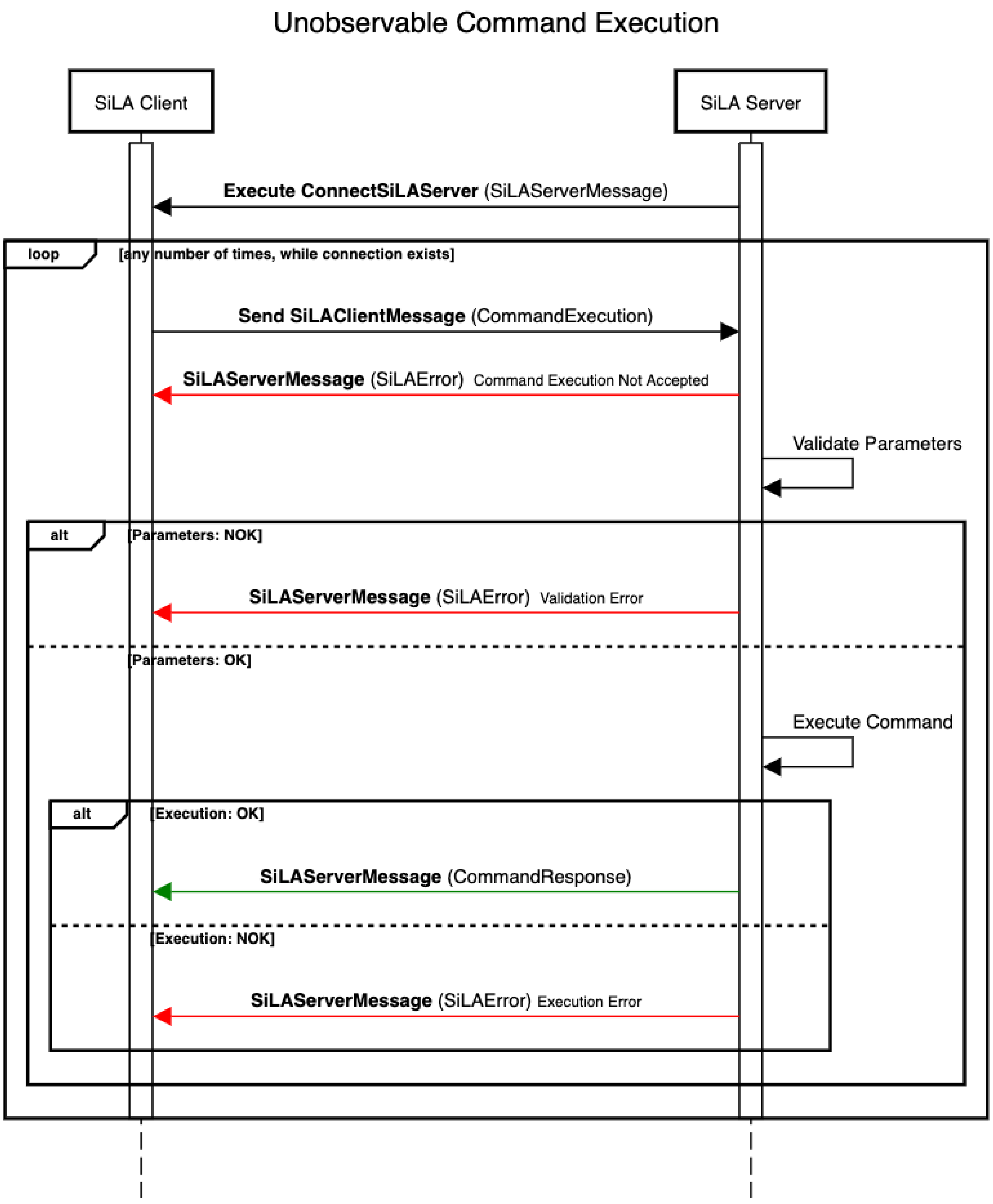
Message Field	Meaning
result	The result of the command execution as bytes. This is the serialized form of the `<·Command Identifier·>_Responses` gRPC

	message defined in Command Response gRPC Mapping Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message CommandResponse {
   bytes result = 1;
}
```

The following sequence diagram illustrates the ·Unobservable Command· execution over the ·Server-Initiated Connection Method·.


Unobservable Command Execution
SiLA Client
SiLA Server
Execute ConnectSiLAServer (SiLAServerMessage)
loop
[any number of times, while connection exists]
Send SiLAClientMessage (CommandExecution)
SiLAServerMessage (SiLAError) Command Execution Not Accepted
Validate Parameters
alt
[Parameters: NOK]
SiLAServerMessage (SiLAError) Validation Error
[Parameters: OK]
Execute Command
alt
[Execution: OK]
SiLAServerMessage (CommandResponse)
[Execution: NOK]
SiLAServerMessage (SiLAError) Execution Error

Executing Observable Commands

[COMPLETE; as of 0.1]

·Observable Commands· are implemented using three (or four, respectively, in case of ·Intermediate Command Responses·) distinct RPCs. All RPCs MUST be implemented for an ·Observable Command·, except for “{Command Identifier}_Intermediate” that only has to be implemented if there are ·Intermediate Command Responses· defined in the ·Feature Definition·.

The ·Command· execution is triggered using the RPC for Command Initiation. Within the response to this RPC, the ·SiLA Client· receives a “CommandConfirmation” message, which is used to identify the ·Command· execution. The ·Command Execution UUID· out of the "CommandConfirmation" message is required to call the other three RPCs.

These other RPCs are used to

- monitor the ·Command· execution by checking the execution status,
- retrieve ·Intermediate Command Responses· during the ·Command· execution (in case they are defined for the respective ·Command·) and
- retrieve the ·Command Response· generated by the ·Command· execution.

Command Initiation

[COMPLETE; as of 0.1, updates: 1.1]

Command Initiation Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

The name of the RPC to request the ·Command· execution MUST be equal to the ·Command Identifier·. The request message for the ·Command· execution method MUST be named "<·Command Identifier·>_Parameters" and contain the required ·Command Parameters·.

The return message of the gRPC method MUST be a Protocol Buffer message in the form of the "CommandConfirmation", as defined in the Framework Definitions for Observable Commands. It MUST contain the ·Command Execution UUID·, which uniquely identifies the ·Command· execution and optionally the ·Lifetime of Execution· (gRPC message field "lifetimeOfExecution") that defines the duration the returned ·Command Execution UUID· MUST be valid for.

RPC to initiate a command execution.

protobuf

```
rpc <·Command Identifier·> (<·Command Identifier·>_Parameters) returns
(CommandConfirmation) {}
```

Command Initiation Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

A ·Observable Command· execution is initiated by sending a ·SiLA Client Message· of type CommandInitiation.

Message Field	Meaning
fullyQualifiedCommandId	The ·Fully Qualified Command Identifier· of the ·Command· to be executed. e.g. "org.silastandard/examples/ObservableGreetingProvider/v1/Command/SayHello"
commandParameter	The CommandParameter Protocol Buffer message, see Command Parameters gRPC Mapping via ·Server-Initiated Connection Method·

protobuf

```
// from SiLACloudConnector.proto
message CommandInitiation {
    string fullyQualifiedCommandId = 1;
    CommandParameter commandParameter = 2;
}
```

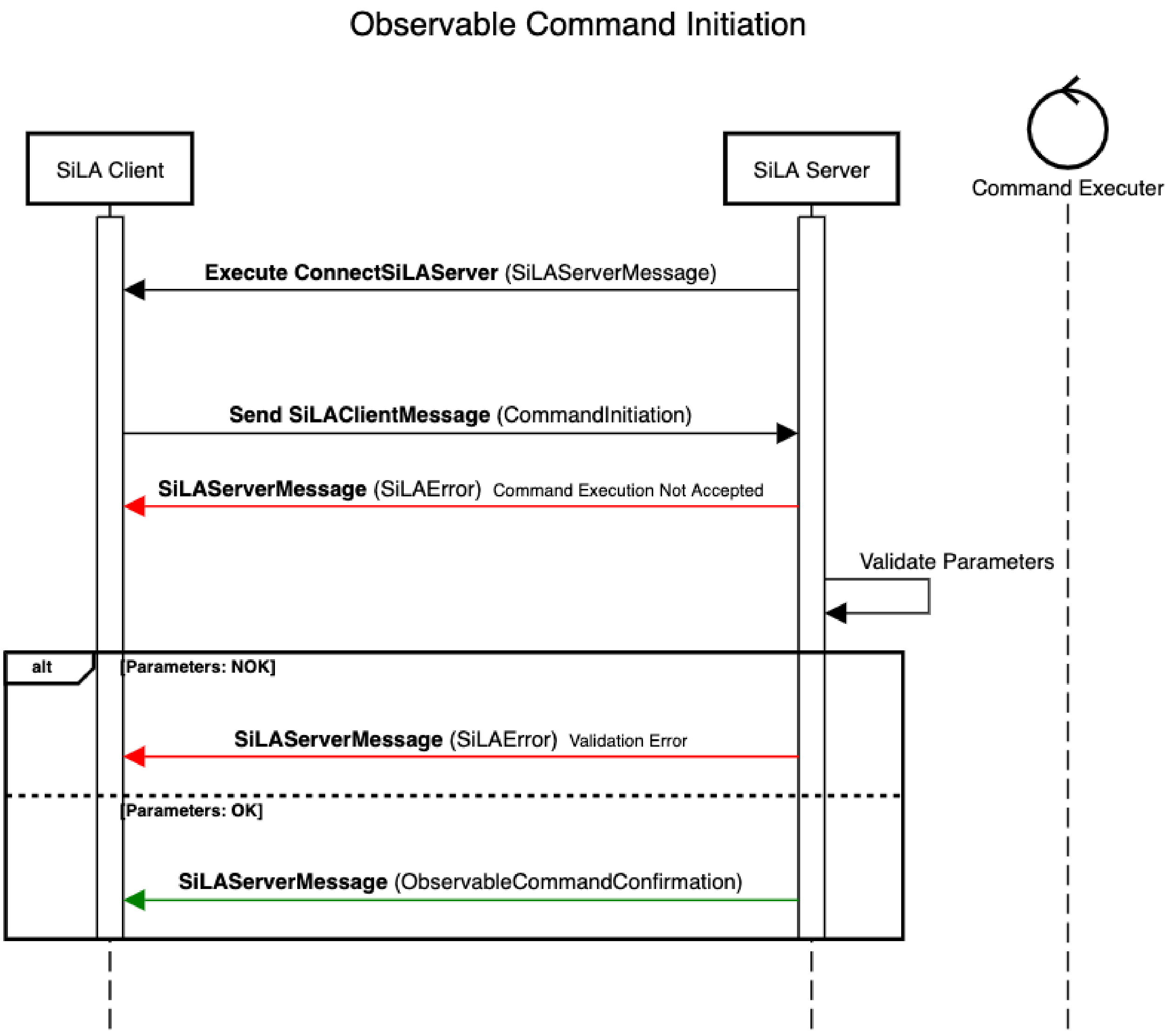
The ·Observable Command· Initiation is returned by sending a ·SiLA Server Message· of type ObservableCommandConfirmation.

Message Field	Meaning
commandConfirmation	The "CommandConfirmation", as defined in the Framework Definitions for Observable Commands. For details about the CommandConfirmation Protocol Buffer message, refer to Command Initiation Over ·Client-Initiated Connection Method· .

protobuf

```
// from SiLACloudConnector.proto
message ObservableCommandConfirmation {
    CommandConfirmation commandConfirmation = 1;
}
```

The following sequence diagram illustrates the ·Observable Command· initiation over the ·Server-Initiated Connection Method·.


Observable Command Initiation
SiLA Client
SiLA Server
Command Executer
Execute ConnectSiLAServer (SiLAServerMessage)
Send SiLAClientMessage (CommandInitiation)
SiLAServerMessage (SiLAError) Command Execution Not Accepted
Validate Parameters
alt
[Parameters: NOK]
SiLAServerMessage (SiLAError) Validation Error
[Parameters: OK]
SiLAServerMessage (ObservableCommandConfirmation)

Observing Command Execution

[COMPLETE; as of 0.1, updates: 1.1]

Observing Command Execution Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

The RPC used to monitor the state of the ·Command· execution MUST be named “<·Command Identifier·>_Info”. For the identification of the ·Command· execution, the ·Command Execution UUID· MUST be transmitted in the request message. The method MUST return a stream of messages (formatted like the “ExecutionInfo” message according to Framework Definitions for Observable Commands) containing the current ·Command Execution Status· and optionally the ·Progress Info·, the ·Estimated Remaining Time· and an updated ·Lifetime of Execution·.

The “ExecutionInfo” message MUST contain a field “commandStatus” that MUST have one of the following values “waiting”, “running”, “finishedSuccessfully”, “finishedWithError”.

If ·Progress Info· is provided, the field “progressInfo” MUST contain the current estimated progress of the ·Command· execution, where 0.0 = *not started* and 1.0 = *completed*.

If the ·Estimated Remaining Time· is provided, the field "estimatedRemainingTime" MUST contain the estimated remaining ·Command· execution time as a duration relative to the point in time it has been sent by the ·SiLA Server·.

If a ·Lifetime of Execution· has been returned within the response of the initial ·Command· call, the field "updatedLifetimeOfExecution" MUST contain the new lifetime as a duration relative to the point in time it has been sent by the ·SiLA Server·. The resulting absolute time MUST NOT be less than prior returned lifetimes.

Instead of subscribing to the ·Command· execution state, a ·SiLA Client· MAY just read the state one time. To achieve this, the returned stream has to be canceled after it has been read the first time.

RPC to subscribe to commands execution status:

protobuf

```
rpc <·Command Identifier·>_Info (CommandExecutionUUID) returns (stream ExecutionInfo)
{}
```

Observing Command Execution Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The subscription to the ·Observable Command· execution info is executed by sending a ·SiLA Client Message· of type CommandExecutionInfoSubscription.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.

protobuf

```
// from SiLACloudConnector.proto
message CommandExecutionInfoSubscription {
    CommandExecutionUUID executionUUID = 1;
}
```

The ·Command Execution Info· of an ·Observable Command· is returned by sending one or more ·SiLA Server Messages· of type ObservableCommandExecutionInfo. Multiple messages can be sent unless the subscription is canceled.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.
executionInfo	The "ExecutionInfo" Protocol Buffer message, as described in Observing Command Execution Over ·Client-Initiated Connection Method·

protobuf

```
// from SiLACloudConnector.proto
message ObservableCommandExecutionInfo {
   CommandExecutionUUID executionUUID = 1;
   ExecutionInfo executionInfo = 2;
}
```

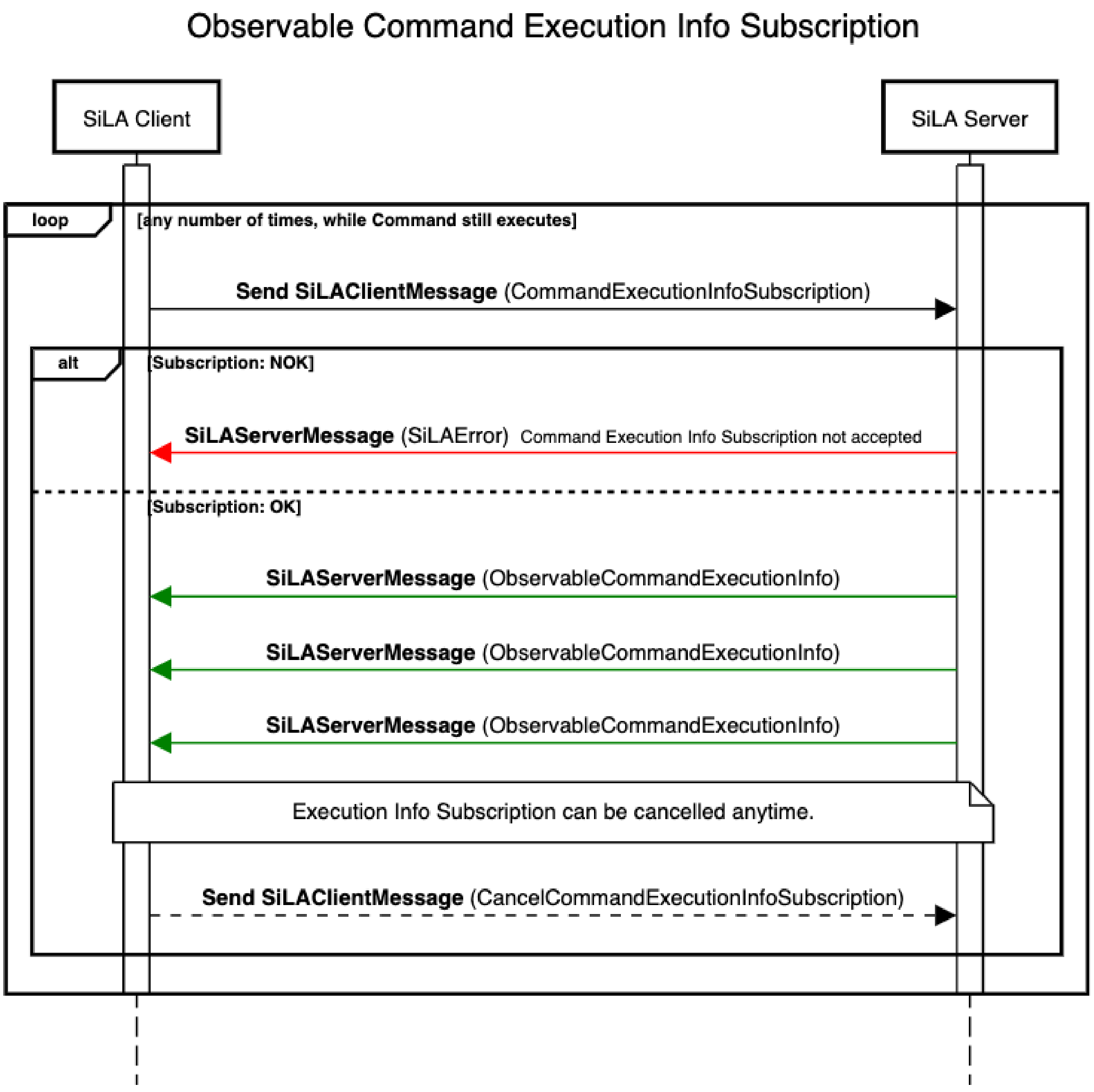
The subscription of the ·Observable Command· execution info is canceled by sending a ·SiLA Client Message· of type CancelCommandExecutionInfoSubscription.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.

protobuf

```
// from SiLACloudConnector.proto
message CancelCommandExecutionInfoSubscription {
    CommandExecutionUUID executionUUID = 1;
}
```

The following sequence diagram illustrates the ·Observable Command· execution info over the ·Server-Initiated Connection Method·.


Observable Command Execution Info Subscription
SiLA Client
SiLA Server
loop
[any number of times, while Command still executes]
Send SiLAClientMessage (CommandExecutionInfoSubscription)
alt
[Subscription: NOK]
SiLAServerMessage (SiLAError) Command Execution Info Subscription not accepted
[Subscription: OK]
SiLAServerMessage (ObservableCommandExecutionInfo)
SiLAServerMessage (ObservableCommandExecutionInfo)
SiLAServerMessage (ObservableCommandExecutionInfo)
Execution Info Subscription can be cancelled anytime.
Send SiLAClientMessage (CancelCommandExecutionInfoSubscription)

Retrieving Intermediate Command Responses

[COMPLETE; as of 0.1, updates: 1.1]

Retrieving Intermediate Command Responses Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

If a ·Command· specifies ·Intermediate Command Responses·, a ·SiLA Client· can subscribe to retrieve ·Intermediate Command Responses· during the ·Command· execution. The according RPC MUST be named “<·Command Identifier·>_Intermediate”. The ·Command Execution UUID· of the ·Command· execution MUST be transmitted in the request message and the method MUST return a stream of “<·Command Identifier·>_IntermediateResponses” messages containing the ·Intermediate Command Response· data.

RPC to subscribe to command intermediate responses:**protobuf**

```
rpc <·Command Identifier·>_Intermediate (CommandExecutionUUID) returns (stream
<·Command Identifier·>_IntermediateResponses) {}

message <·Command Identifier·>_IntermediateResponses {
    <·Intermediate Command Response Data Type· 1> <·Intermediate Command Response
Identifier· 1> = 1;
    <·Intermediate Command Response Data Type· 2> <·Intermediate Command Response
Identifier· 2> = 2;
    …
    <·Intermediate Command Response Data Type· N> <·Intermediate Command Response
Identifier· N> = N;
}
```

As ·Intermediate Command Responses· are optional, the presence of this RPC and accompanying messages are only required if ·Intermediate Command Responses· are specified in the ·Feature Definition·.

Retrieving Intermediate Command Responses Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The subscription to the ·Observable Command· intermediate responses is executed by sending a ·SiLA Client Message· of type CommandIntermediateResponseSubscription.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.

protobuf

```
// from SiLACloudConnector.proto
message CommandIntermediateResponseSubscription {
    CommandExecutionUUID executionUUID = 1;
}
```

The ·Observable Command· intermediate responses are returned by sending a ·SiLA Server Message· of type ObservableCommandIntermediateResponse.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.
result	This is the serialized form of the `<·Command Identifier·>_IntermediateResponses` gRPC message as defined in chapter Retrieving Intermediate Command Responses Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message ObservableCommandIntermediateResponse {
    CommandExecutionUUID executionUUID = 1;
    bytes result = 2;
}
```

The subscription to the ·Observable Command· intermediate response is canceled by sending a ·SiLA Client Message· of type CancelCommandIntermediateResponseSubscription.

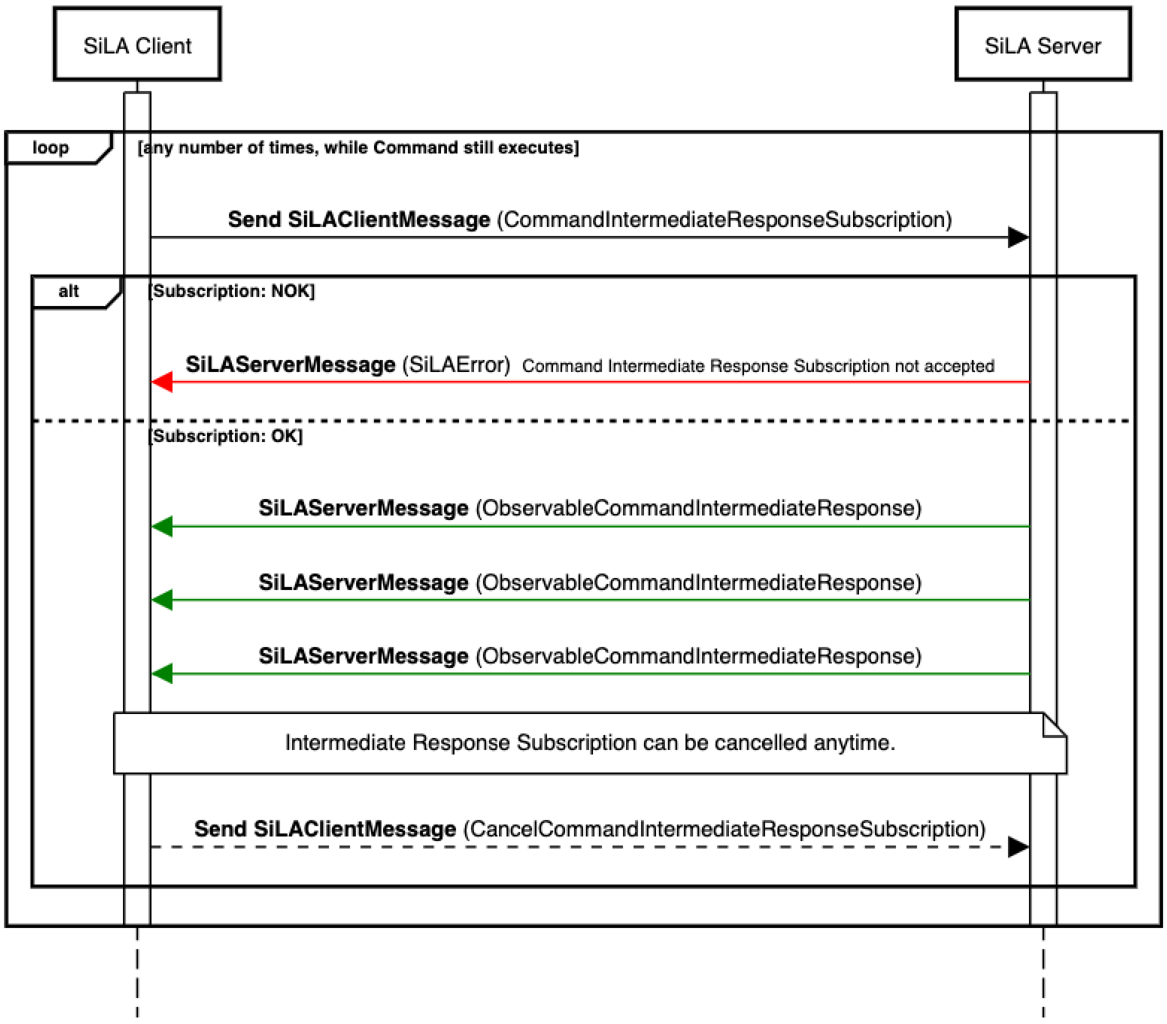
Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.

protobuf

```
// from SiLACloudConnector.proto
message CancelCommandIntermediateResponseSubscription {
    CommandExecutionUUID executionUUID = 1;
}
```

The following sequence diagram illustrates the ·Observable Command· intermediate response subscription.

Observable Command Intermediate Response Subscription


SiLA Client
SiLA Server
loop
[any number of times, while Command still executes]
Send SiLAClientMessage (CommandIntermediateResponseSubscription)
alt
[Subscription: NOK]
SiLAServerMessage (SiLAError) Command Intermediate Response Subscription not accepted
[Subscription: OK]
SiLAServerMessage (ObservableCommandIntermediateResponse)
SiLAServerMessage (ObservableCommandIntermediateResponse)
SiLAServerMessage (ObservableCommandIntermediateResponse)
Intermediate Response Subscription can be cancelled anytime.
Send SiLAClientMessage (CancelCommandIntermediateResponseSubscription)

Command Result Retrieval

[COMPLETE; as of 0.1, updates: 1.1]

Command Result Retrieval Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

The result of the ·Command· execution can be retrieved after it has been completed. The according RPC MUST be named “<·Command Identifier·>_Result”. The ·Command Execution UUID· MUST be transmitted in the request message and the ·Command Response· message MUST be named “<·Command Identifier·>_Responses” and contain all ·Command Response· data.

In case, the ·Command· execution has not finished yet, when the ·Command· result request is sent, a ·Command Execution Not Finished Error· MUST be returned.

In case an error occurs during the ·Command· execution. The “ExecutionInfo” message will contain a “commandStatus” value of “finishedWithError” and an ·Execution Error· MUST be thrown when the ·Command· result request is sent.

RPC to obtain command response, once the command is completed:

protobuf

```
rpc <·Command Identifier·>_Result (CommandExecutionUUID) returns (<·Command
Identifier·>_Responses) {}
```

Examples

The robot ·Command· “MoveSample” similarly to the example of an ·Unobservable Command·, implemented as an ·Observable Command·:

protobuf

```
service RobotMoveController {
      rpc MoveSample (MoveSample_Parameters) returns (CommandConfirmation) {}
      rpc MoveSample_Intermediate (CommandExecutionUUID) returns (stream
MoveSample_IntermediateResponses) {}
      rpc MoveSample_Info (CommandExecutionUUID) returns (stream ExecutionInfo) {}
      rpc MoveSample_Result (CommandExecutionUUID) returns (MoveSample_Responses) {}
}

message MoveSample_Parameters {
      Integer PlateSiteA = 1;
      Integer PlateSiteB = 2;
}

message MoveSample_IntermediateResponses {
      Real DrivenDistance = 1;
}

message MoveSample_Responses {
      Real Precision = 2;
}
```

Command Result Retrieval Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The result of the ·Observable Command· execution is retrieved by sending a ·SiLA Client Message· of type CommandGetResponse.

Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.

protobuf

```
// from SiLACloudConnector.proto
message CommandGetResponse {
    CommandExecutionUUID executionUUID = 1;
}
```

The result of an ·Observable Command· is returned by sending a ·SiLA Server Message· of type ObservableCommandResponse.

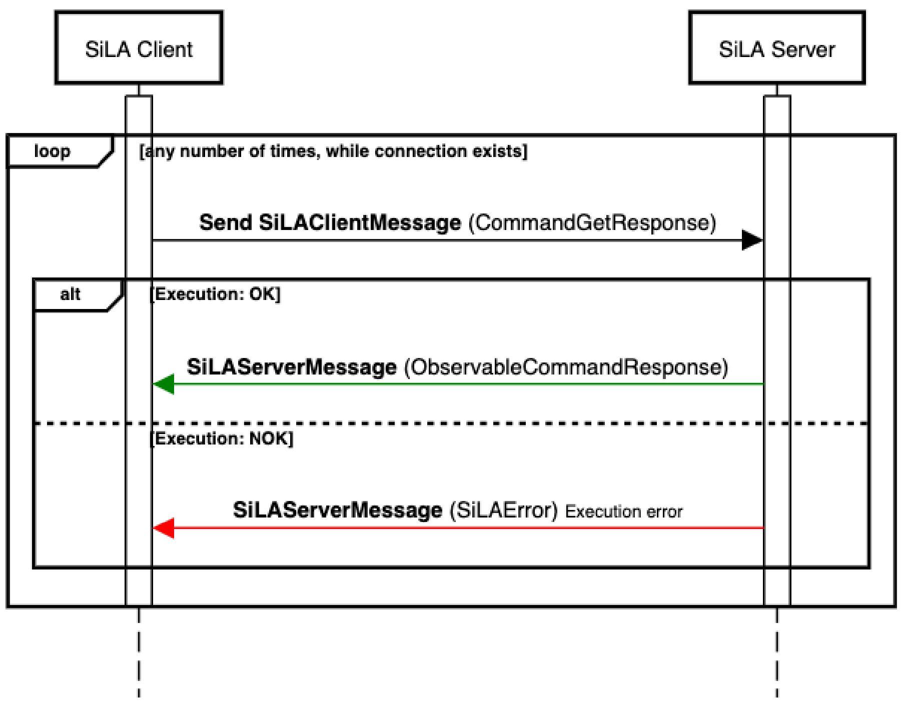
Message Field	Meaning
executionUUID	The ·Command Execution UUID·, as generated by ·SiLA Server·, as part of the CommandConfirmation Protocol Buffer message.
result	This is the serialized form of the `<·Command Identifier·>_Responses` gRPC message as defined in chapter Command Response gRPC Mapping Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message ObservableCommandResponse{
    CommandExecutionUUID executionUUID = 1;
    bytes result = 2;
}
```

The following sequence diagram illustrates the ·Observable Command· result retrieval.

Observable Command Result Retrieval


SiLA Client
SiLA Server
loop
[any number of times, while connection exists]
Send SiLAClientMessage (CommandGetResponse)
alt
[Execution: OK]
SiLAServerMessage (ObservableCommandResponse)
[Execution: NOK]
SiLAServerMessage (SiLAError) Execution error

Framework Definitions for Observable Commands

[COMPLETE; as of 0.1]

The framework definitions necessary for ·Observable Commands· are defined as:

protobuf

```
// from SiLAFramework.proto
message CommandExecutionUUID {
    string value = 1;
}

message CommandConfirmation {
    CommandExecutionUUID commandExecutionUUID = 1;
    Duration lifetimeOfExecution = 2;
}

message ExecutionInfo {
    enum CommandStatus {
        waiting = 0;
        running = 1;
        finishedSuccessfully = 2;
        finishedWithError = 3;
    }
    CommandStatus commandStatus = 1;
    Real progressInfo = 2;
    Duration estimatedRemainingTime = 3;
    Duration updatedLifetimeOfExecution = 4;
}
```

Note that the ·Command Execution UUID· MUST be encoded as a string as there exists no 128 bit integer in Protocol Buffers. The reason not to use bytes is to avoid compatibility problems in case of different endianness.

As defined for the ·Term· ·UUID·, SiLA always uses the ·UUID· in its string representation (e.g. “f81d4fae-7dec-11d0-a765-00a0c91e6bf6”), as specified by the formal definition of the ·UUID· string representation in RFC 4122. It is RECOMMENDED to always use lower case letters (a-f). In any case, comparisons of ·UUID· in their string representation must always be performed ignoring lower and upper case, i.e. “a” = “A”, “b” = “B”, … , “f” = “F”.

Defined Execution Errors

[COMPLETE; as of 0.1]

·Defined Execution Errors· do not have a direct mapping like e.g. a ·Command·. Check the Error Handling Chapter for the treatment of errors in gRPC.

Properties

[COMPLETE; as of 0.1]

The SiLA 2 Specification Part A defines ·Observable Properties· as well as ·Unobservable Properties·. The different implementation guidelines are provided below.

Unobservable Properties

[COMPLETE; as of 0.1, updates: 1.1]

Reading Properties Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

Mapping ·Unobservable Properties· to gRPC follows similar rules as ·Unobservable Commands·. The RPC parameter will always be an empty message. The response value will be a message with a single field, named like the ·Property Identifier· and the Protocol Buffer data type that corresponds to the ·SiLA Data Type· of the ·Property·.

protobuf

```
rpc Get_<·Property Identifier·> (Get_<·Property Identifier·>_Parameters) returns
(Get_<·Property Identifier·>_Responses) {}

message Get_<·Property Identifier·>_Parameters {}

message Get_<·Property Identifier·>_Responses {
    <·Property Data Type·> <·Property Identifier·> = 1;
}
```

Example

[COMPLETE; as of 0.1]

A simple ·Feature· with identifier "DeviceProvider" with one ·Unobservable Property· with identifier "DeviceName" results in:

protobuf

```
service DeviceProvider {
     rpc Get_DeviceName (Get_DeviceName_Parameters) returns
(Get_DeviceName_Responses) {}
}

message Get_DeviceName_Parameters {}

message Get_DeviceName_Responses {
    String DeviceName = 1;
}
```

Reading Properties Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

Reading of an ·Unobservable Property· is executed by sending a ·SiLA Client Message· of type PropertyRead.

Message Field	Meaning
fullyQualifiedPropertyId	The ·Fully Qualified Property Identifier· of the ·Property· to be read, e.g. "org.silastandard/examples/ObservableGreetingProvider/v1/Property/Timestamp"

propertyParameter	The <·Property Identifier·>_Parameters gRPC message as defined in chapter Reading Properties Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message PropertyRead {
   string fullyQualifiedPropertyId = 1;
   PropertyParameter propertyParameter = 2;
}
```

The response of an ·Unobservable Property· read is returned by sending a ·SiLA Server Message· of type PropertyValue.

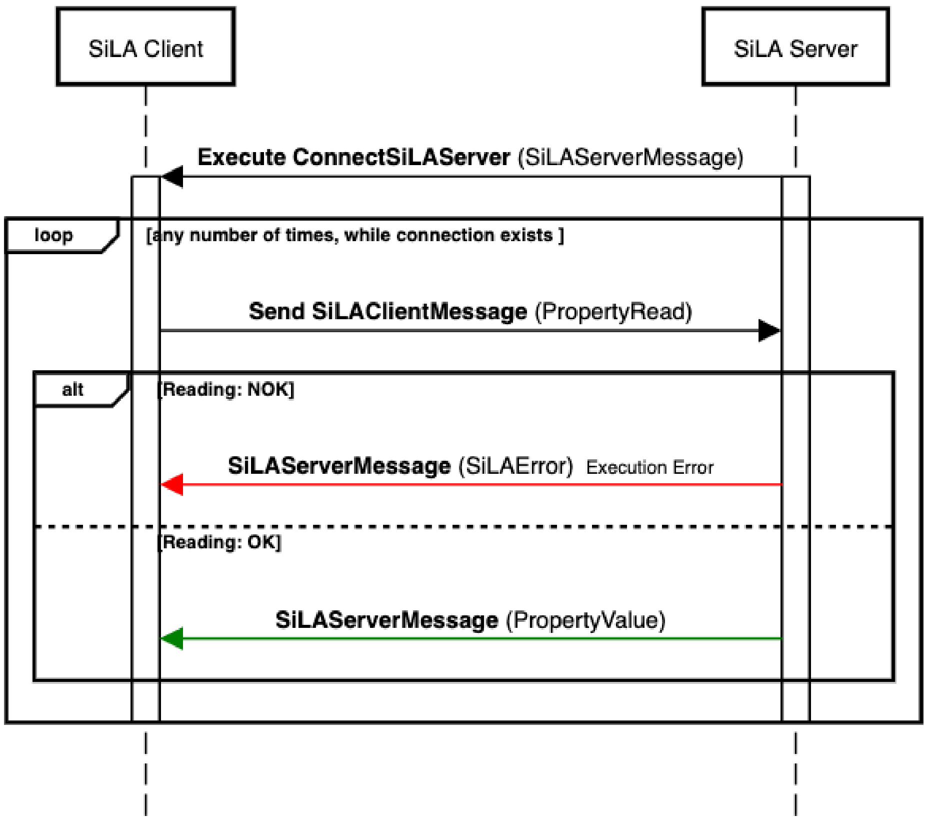
Message Field	Meaning
result	This is the serialized form of the <·Property Identifier·>_Responses gRPC message as defined in chapter Reading Properties Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message PropertyValue {
    bytes result = 1;
}
```

The following sequence diagram illustrates the ·Unobservable Property· read over a "Server-initiated" ·Connection·.

Reading a Property


SiLA Client
SiLA Server
Execute ConnectSiLAServer (SiLAServerMessage)
loop
[any number of times, while connection exists]
Send SiLAClientMessage (PropertyRead)
alt
[Reading: NOK]
SiLAServerMessage (SiLAError) Execution Error
[Reading: OK]
SiLAServerMessage (PropertyValue)

Observable Properties

[COMPLETE; as of 0.1, updates: 1.1]

Reading and Subscribing to Properties Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1]

·Observable Properties· map to one RPC.

·Observable Properties· are implemented using gRPC with a server side stream that the client can subscribe to. The RPC MUST be called with an empty message as input parameter. The server MUST respond with a stream of messages with a single field, named as the ·Property Identifier· and the Protocol Buffer data type that corresponds to the ·SiLA Data Type· of the ·Property·.

Note: To close the subscription, the client MUST cancel the gRPC stream.

Instead of subscribing to an ·Observable Property·, a ·SiLA Client· MAY just read the property value one time. To achieve this, the returned stream has to be canceled after it has been read the first time.

protobuf

```
rpc Subscribe_<·Property Identifier·> (Subscribe_<·Property Identifier·>_Parameters)
returns (stream Subscribe_<·Property Identifier·>_Responses) {}

message Subscribe_<·Property Identifier·>_Parameters {}
message Subscribe_<·Property Identifier·>_Responses {
    <·Property Data Type·> <·Property Identifier·> = 1;
}
```

Examples

[COMPLETE; as of 1.0]

A simple ·Feature· with identifier “TemperatureProvider” with one ·Observable Property· having identifier “CurrentTemperature” results in:

protobuf

```
service TemperatureProvider {
      rpc Subscribe_CurrentTemperature (Subscribe_CurrentTemperature_Parameters)
returns (stream Subscribe_CurrentTemperature_Responses) {}
}

message Subscribe_CurrentTemperature_Parameters {}

message Subscribe_CurrentTemperature_Responses {
      Real CurrentTemperature = 1;
}
```

Reading and Subscribing to Properties Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

Reading an Subscribing to an ·Observable Property· is executed by sending a ·SiLA Client Message· of type PropertySubscription.

Message Field	**Meaning**
fullyQualifiedPropertyId	The ·Fully Qualified Property Identifier· of the ·Property· to be read, e.g. "org.silastandard/examples/ObservableGreetingProvider/v1/Property/Timestamp"
propertyParameter	The `<·Property Identifier·>_Parameters` gRPC message as defined in chapter Reading Properties Over ·Client-Initiated Connection Method·.

protobuf

```
// from SiLACloudConnector.proto
message PropertySubscription {
   string fullyQualifiedPropertyId = 1;
   repeated Metadata metadata = 2;
}
```

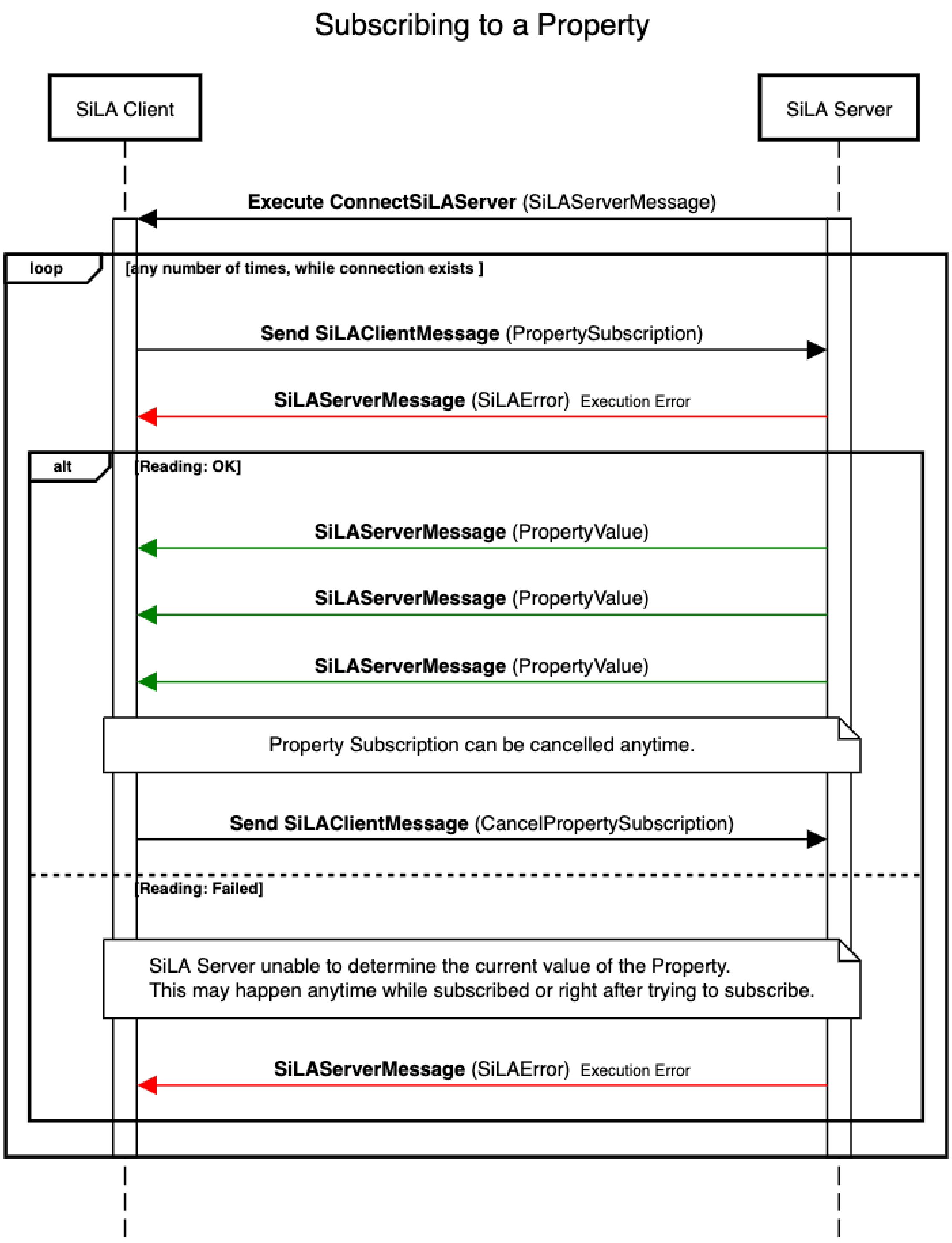
The response of the ·Observable Property· subscription is returned by sending one or more ·SiLA Server Message· of type PropertyValue described in Reading Properties over “server-initiated” Connection.

A ·Observable Property· subscription is canceled by sending a ·SiLA Client Message· of type CancelPropertySubscription.

protobuf

```
// from SiLACloudConnector.proto
message CancelPropertySubscription {
}
```

The following sequence diagram illustrates the ·Observable Property· subscription over a “Server-initiated” ·Connection·.


Subscribing to a Property
SiLA Client
SiLA Server
Execute ConnectSiLAServer (SiLAServerMessage)
loop
[any number of times, while connection exists]
Send SiLAClientMessage (PropertySubscription)
SiLAServerMessage (SiLAError) Execution Error
alt
[Reading: OK]
SiLAServerMessage (PropertyValue)
SiLAServerMessage (PropertyValue)
SiLAServerMessage (PropertyValue)
Property Subscription can be cancelled anytime.
Send SiLAClientMessage (CancelPropertySubscription)
[Reading: Failed]
SiLA Server unable to determine the current value of the Property.
This may happen anytime while subscribed or right after trying to subscribe.
SiLAServerMessage (SiLAError) Execution Error

SiLA Client Metadata

[COMPLETE; as of 0.2, updates: 1.1]

SiLA Client Metadata Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.2]

A ·SiLA Client Metadata· definition is mapped to a Protocol Buffer message containing the ·SiLA Client Metadata· content and an RPC returning the ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties·, which the ·SiLA Client Metadata· applies to.

The Protocol Buffer message contains only one field of the defined SiLA Type with the identifier equal to the ·Metadata Identifier·.

The RPC is named “Get_FCPAffectedByMetadata_” followed by the ·Metadata Identifier·, where “FCP” is an abbreviation and stands for ·Features·, ·Commands·, ·Properties·. The response is a list of ·Fully Qualified Identifiers· of ·Features·, ·Commands· and ·Properties· for which the ·SiLA Client Metadata· is expected as part of the respective RPCs.

If the response contains a ·Fully Qualified Feature Identifier·, all ·Commands· and ·Properties· of the respective ·Feature· are affected by the ·SiLA Client Metadata·.

SiLA Client Metadata gRPC Mapping

A Protocol Buffer message describing the ·SiLA Client Metadata· looks like:

protobuf

```
message Metadata_<·Metadata Identifier·> {
   <·Metadata Data Type·> <·Metadata Identifier·> = 1;
}
```

An RPC defining a property that returns all ·Features·, ·Commands· and ·Properties· (as ·Fully Qualified Identifiers·, see above) that are affected by the ·SiLA Client Metadata·:

protobuf

```
rpc Get_FCPAffectedByMetadata_<·Metadata Identifier·>
(Get_FCPAffectedByMetadata_<·Metadata Identifier·>_Parameters) returns
Get_FCPAffectedByMetadata_<·Metadata Identifier·>_Responses {}

message Get_FCPAffectedByMetadata_<·Metadata Identifier·>_Parameters {}

message Get_FCPAffectedByMetadata_<·Metadata Identifier·>_Responses {
   repeated String AffectedCalls = 1;
}
```

Sending SiLA Client Metadata Over gRPC

·SiLA Client Metadata· is sent as Custom-Metadata in the Binary-Header format as specified in gRPC over HTTP/2 framing.

Each ·SiLA Client Metadata· is added to the Custom-Metadata arbitrary set of key-value pairs, conforming to the production rule "Binary-Header" as specified in gRPC over HTTP/2 framing. The key must be conforming to the rule "{Header-Name "-bin" }" and the binary value must be Base64 encoded (according to RFC 4648). Note that some gRPC implementations might provide the Base64 de- and encoding in a transparent way, as soon as a Custom-Metadata with a key suffixed by "-bin" is used.

·SiLA Client Metadata· is restricted in size: HTTP2 implementations might limit the total header size to a few KB and ·Binary Transfer· is not possible for ·SiLA Client Metadata·, so the encoded base64-message must contain the full values and all Metadata values transferred with a single message must not violate the header size limit of the particular HTTP2 implementation.

Header names (Custom-Metadata key) MUST be equal to the respective ·Fully Qualified Metadata Identifier· (converted as indicated below), prefixed with "**`sila-`**" and suffixed with "**`-bin`**".

Due to limitations in gRPC Header Names (production rule "Header-Name"), the ·Fully Qualified Metadata Identifier· needs to be converted as follows:

- Convert all "**`/`**" to "**`-`**" (hyphen)
- Convert all uppercase letters ("**`A`**", "**`B`**", … "**`Z`**") to lowercase ("**`a`**", "**`b`**", … "**`z`**")

Example:

The following ·Fully Qualified Metadata Identifier·:

```
org.silastandard/core/SiLAService/v1/Metadata/AuthorizationToken
```

after conversion becomes:

```
org.silastandard-core-silaservice-v1-metadata-authorizationtoken
```

hence the gRPC header name becomes:

```
sila-org.silastandard-core-silaservice-v1-metadata-authorizationtoken-bin
```

The Custom-Metadata value is a serialized, gRPC-encoded Protocol Buffer message of the ·SiLA Client Metadata· ·SiLA Data Type· in binary format, see SiLA Client Metadata gRCP Mapping.

To serialize and deserialize a value of a ·SiLA Client Metadata·, the same mapping to gRPC messages as for any ·SiLA Data Type· (into a gRPC message) is used. Then the gRPC library's functions are to be used to serialize or deserialize the value into a binary format or back. The value in the binary format will then be transferred as the Custom-Metadata binary value.

Examples

[COMPLETE; as of 1.0]

A LockController ·Feature· defining a ·SiLA Client Metadata· "LockIdentifier" will result in:

protobuf

```
service LockController {
      rpc Get_FCPAffectedByMetadata_LockIdentifier
(Get_FCPAffectedByMetadata_LockIdentifier_Parameters) returns
(Get_FCPAffectedByMetadata_LockIdentifier_Responses) {}
      …
}

message Get_FCPAffectedByMetadata_LockIdentifier_Parameters {}

message Get_FCPAffectedByMetadata_LockIdentifier_Responses {
      repeated String AffectedCalls = 1;
}

message Metadata_LockIdentifier {
      String LockIdentifier = 1;
}
```

SiLA Client Metadata Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The retrieval for the list of ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties· that are affected by the ·SiLA Client Metadata· is executed by sending a ·SiLA Client Message· of type GetFCPAffectedByMetadataRequest.

Message Field	Meaning
fullyQualifiedMetadataId	The ·Fully Qualified Metadata Identifier·, for which all ·Features· / ·Commands· / ·Properties· that are affected by the ·SiLA Client Metadata· should be retrieved.

protobuf

```
// from SiLACloudConnector.proto
message GetFCPAffectedByMetadataRequest {
    string fullyQualifiedMetadataId = 1;
}
```

The list of ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties· is returned by sending a ·SiLA Server Message· of type GetFCPAffectedByMetadataResponse.

Message Field	Meaning
affectedCalls	The list of ·Fully Qualified Identifiers· of all ·Features· / ·Commands· / ·Properties· that are affected by the ·SiLA Client Metadata·.

protobuf

```
// from SiLACloudConnector.proto
message GetFCPAffectedByMetadataResponse {
    repeated string affectedCalls = 1;
}
```

The Protocol Buffer message used for sending ·SiLA Client Metadata· over ·Server-Initiated Connection Method· is defined as follows.

Message Field	Meaning
fullyQualifiedMetadataId	The ·Fully Qualified Metadata Identifier· of the ·SiLA Client Metadata·.
value	This is the serialized form of the **`Metadata_<·Metadata Identifier·>`** gRPC message as defined in chapter SiLA Client Metadata gRCP Mapping.

protobuf

```
// from SiLACloudConnector.proto
message Metadata {
   string fullyQualifiedMetadataId = 1;
   bytes value = 2;
}
```

SiLA Data Type Mapping

[COMPLETE; as of 0.1]

Each ·SiLA Data Type· is translated to a Protocol Buffer message type, this chapter describes how to do this translation.

SiLA Basic Types

[COMPLETE; as of 0.1]

The mapping for ·SiLA Basic Types· is the following:

SiLA Definition	gRPC Mapping

·SiLA String Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message String {`
	`   string value = 1;`
	`}`

·SiLA Integer Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Integer {`
	`   int64 value = 1;`
	`}`

·SiLA Real Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Real {`
	`    double value = 1;`
	`}`

·SiLA Boolean Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Boolean {`
	`    bool value = 1;`
	`}`

·SiLA Binary Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Binary {`
	`    oneof union {`
	`        bytes value = 1;`
	`        string binaryTransferUUID = 2;`
	`    }`
	`}`
	See SiLA Binary Type.

·SiLA Date Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Date {`
	`    uint32 day = 1;`
	`    uint32 month = 2;`
	`    uint32 year = 3;`
	`    Timezone timezone = 4; // MUST NOT be null`
	`}`

·SiLA Time Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Time {`
	`    uint32 second = 1;`
	`    uint32 minute = 2;`
	`    uint32 hour = 3;`
	`    Timezone timezone = 4; // MUST NOT be null`
	`}`

·SiLA Timestamp Type·	**protobuf**
	`// from SiLAFramework.proto`
	`message Timestamp {`
	`    uint32 second = 1;`
	`    uint32 minute = 2;`
	`    uint32 hour = 3;`
	`    uint32 day = 4;`
	`    uint32 month = 5;`
	`    uint32 year = 6;`
	`    Timezone timezone = 7; // MUST NOT be null`
	`}`

·SiLA Any Type·	See SiLA Any Type .

Framework Definitions

[COMPLETE; as of 0.1]

Duration

[COMPLETE; as of 0.1]

[Definition: Duration] A **Duration** represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month".

protobuf

```
// from SiLAFramework.proto
message Duration {
    int64 seconds = 1;
    int32 nanos = 2;
}
```

Timezone

[COMPLETE; as of 0.1]

[Definition: Timezone] A **Timezone** represents a signed, fixed-length span of time represented as a count of hours and minutes as an offset from UTC.

This is used for the ·SiLA Date Type·, the ·SiLA Timestamp Type· and the ·SiLA Time Type· which all need to provide a timezone.

protobuf

```
// from SiLAFramework.proto
message Timezone {
    int32 hours = 1;
    uint32 minutes = 2;
}
```

SiLA Binary Type

[COMPLETE; as of 1.0]

The ·SiLA Binary Type· is used to send arbitrary binary data in two ways: either directly as payload within the gRPC message or using the ·Binary Transfer· mechanism.

The ·SiLA Server· or ·SiLA Client· implementation MUST decide which way to use according the rule defined for ·Binary Transfer·. Depending on the way chosen, the gRPC message needs to be handled differently.

Binary Transfer

[COMPLETE; as of 1.0, updates: 1.1]

The Protocol Buffer (.proto) “bytes” type can only hold a limited number of bytes. Hence SiLA introduces a safe mechanism to transfer larger binary data by introducing the ·Binary Transfer· mechanism.

[Definition: Binary Transfer] **Binary Transfer** is the process of transferring larger amounts of binary data between a ·SiLA Client· and a ·SiLA Server·. The **Binary Transfer** mechanism is required for handling binary data larger than 2 MiB in size. If a ·SiLA Server· does not support **Binary Transfer** a ·Binary Upload Failed Error· has to be thrown. Note that the **Binary Transfer** is only applicable in the context of ·Command Parameters·, ·Command Responses·, ·Intermediate Command Responses· or ·Properties·.

Two different behaviors are described for a ·Binary Transfer·:

1. [Definition: Binary Upload]: **Binary Upload** is the process of sending binary data from the ·SiLA Client· to the ·SiLA Server·.
2. [Definition: Binary Download]: **Binary Download** is the process of sending binary data from the ·SiLA Server· to the ·SiLA Client·.

[Definition: Binary Chunk] In both ·Binary Upload· and ·Binary Download·, the binary data is split into several **Binary Chunks**. Each **Binary Chunk** will be uploaded or downloaded individually. A Binary Chunk MUST not be larger than 2 MiB in size.

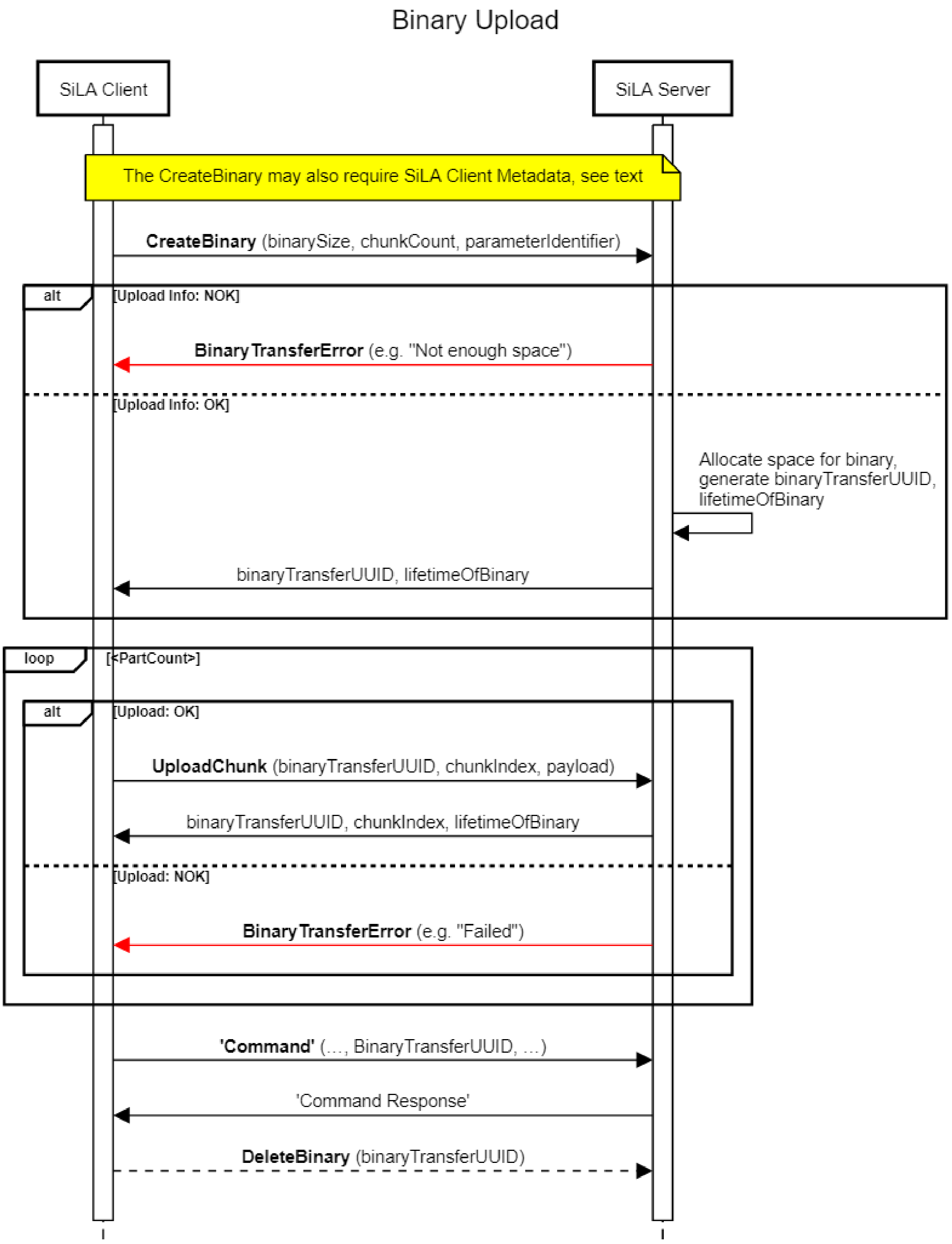
For each ·Binary Transfer·, both a ·Binary Transfer UUID· and a ·Lifetime of Binary· is created and maintained.

[Definition: Binary Transfer UUID] A **Binary Transfer UUID** is a ·UUID· referring to specific binary data of a ·Binary Transfer·. It MUST be unique within one instance of a ·SiLA Server· and its lifetime (·Lifetime of a SiLA Server·).

[Definition: Lifetime of Binary] The **Lifetime of Binary** is the duration during which a ·Binary Transfer UUID· is valid. The **Lifetime of Binary** is always a relative duration with respect to the point in time the ·SiLA Server· sent the **Lifetime of Binary** to the ·SiLA Client·. It is the responsibility of the ·SiLA Client· to account for potential transmission delays from the ·SiLA Server· to the ·SiLA Client·.

Binary Upload Over ·Client-Initiated Connection Method·

[COMPLETE; as of 1.0]


Binary Upload
SiLA Client
SiLA Server
The CreateBinary may also require SiLA Client Metadata, see text
CreateBinary (binarySize, chunkCount, parameterIdentifier)
alt
[Upload Info: NOK]
BinaryTransferError (e.g. "Not enough space")
[Upload Info: OK]
Allocate space for binary, generate binaryTransferUUID, lifetimeOfBinary
binaryTransferUUID, lifetimeOfBinary
loop
[<PartCount>]
alt
[Upload: OK]
UploadChunk (binaryTransferUUID, chunkIndex, payload)
binaryTransferUUID, chunkIndex, lifetimeOfBinary
[Upload: NOK]
BinaryTransferError (e.g. "Failed")
'Command' (..., BinaryTransferUUID, ...)
'Command Response'
DeleteBinary (binaryTransferUUID)

The ·Binary Upload· is divided into three phases:

1. Creating the storage for the binary data with associated ·Binary Transfer UUID·,
2. uploading ·Binary Chunks· of the binary data and
3. using the binary data by sending the ·Binary Transfer UUID· as a ·Parameter· to a ·Command·.

During the creation of the storage for the binary data, the implementer must check that there is enough resources (such as disk space) to handle the upload and storage of the binary data.

The uploading of the ·Binary Chunks· can be performed in an arbitrary order and is considered complete only when all the ·Binary Chunks· have been received by the ·SiLA Server·.

Create a Binary (Step 1)

[COMPLETE; as of 1.0]

As a first step, storage space is allocated based on the size of the binary data and on how many ·Binary Chunks· it will be divided in.

The ·SiLA Client· MUST provide the ·Fully Qualified Command Parameter Identifier· that the uploaded data is going to be used for. Further, the ·SiLA Client· must provide the request headers for every ·SiLA Client Metadata· that affects the associated ·Command·. The ·SiLA Server· MAY enforce that only binaries uploaded for a given ·Command· can be effectively used with this ·Command·.

The ·SiLA Server· MUST validate if the required space is available to save the binary data. If the validation is successful, the ·SiLA Server· MUST generate a ·Binary Transfer UUID· associated with the reserved storage and an associated ·Lifetime of Binary·.

Upload Binary Chunks (Step 2)

[COMPLETE; as of 1.0]

The ·Binary Upload· can be started by the ·SiLA Client· within the given ·Lifetime of Binary·. It is the responsibility of the ·SiLA Server· implementer that the ·Lifetime of Binary· is chosen within a reasonable interval.

The ·SiLA Client· uploads the ·Binary Chunk· with the associated ·Binary Transfer UUID· and ·Binary Chunk· index. The index is an integer from 0 to N-1, where N is the total number of chunks. The ·SiLA Server· has to acknowledge each upload with the current ·Lifetime of Binary· that MAY be different than the initial time given at the creation.

Issue a Command (Step 3)

[COMPLETE; as of 1.0]

After all promised ·Binary Chunks· have been uploaded, the ·Command· can be issued with the associated ·Binary Transfer UUID· as the ·Parameter· to the ·Command·.

Optionally Delete Binary Data

[COMPLETE; as of 1.0]

Additionally, the ·SiLA Server· MUST provide means to delete the Binary after use. If not deleted, the ·Binary Transfer UUID· MUST be valid until it expires.

Protocol Buffer Definition for Binary Upload

[COMPLETE; as of 1.0]

The following Protocol Buffer definition MUST be implemented to offer the ·Binary Upload· service:

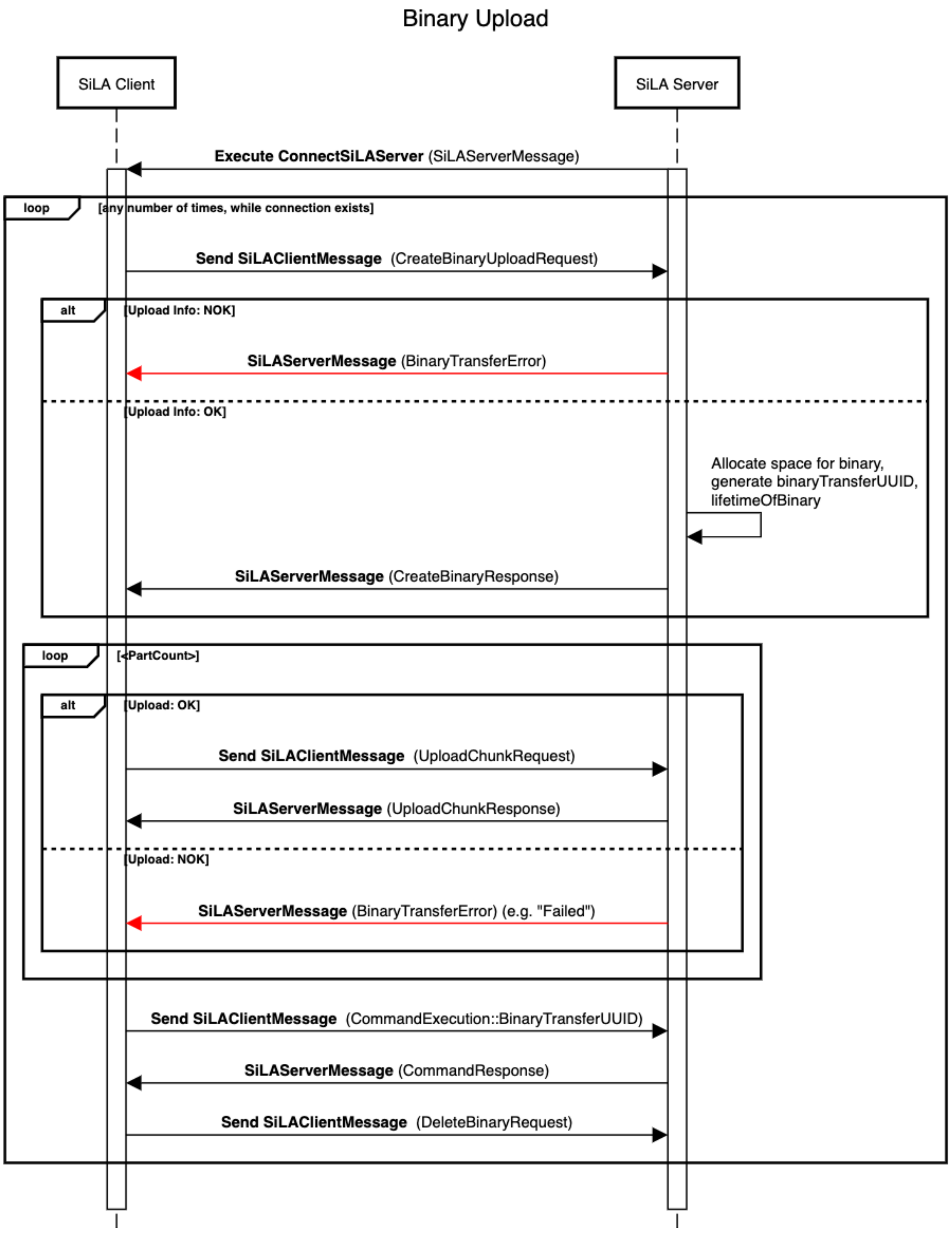
protobuf

```
// from SiLABinaryTransfer.proto
service BinaryUpload {
   rpc CreateBinary (CreateBinaryRequest) returns (CreateBinaryResponse) {}
   rpc UploadChunk (stream UploadChunkRequest) returns (stream UploadChunkResponse)
{}
   rpc DeleteBinary (DeleteBinaryRequest) returns (DeleteBinaryResponse) {}
}

message CreateBinaryRequest {
   uint64 binarySize = 1;
   uint32 chunkCount = 2;
   string parameterIdentifier = 3;  /* fully qualified parameter identifier */
}

message CreateBinaryResponse {
   string binaryTransferUUID = 1;
   Duration lifetimeOfBinary = 2;
}

message UploadChunkRequest {
   string binaryTransferUUID = 1;
   uint32 chunkIndex = 2;
   bytes payload = 3;
}

message UploadChunkResponse {
   string binaryTransferUUID = 1;
   uint32 chunkIndex = 2;
   Duration lifetimeOfBinary = 3;
}

message DeleteBinaryRequest {
   string binaryTransferUUID = 1;
}

message DeleteBinaryResponse {
}
```

Binary Upload Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The ·Binary Upload· over ·Server-Initiated Connection Method· follows the same steps as described in the previous chapter Binary Upload.


Binary Upload
SiLA Client
SiLA Server
Execute ConnectSiLAServer (SiLAServerMessage)
loop [any number of times, while connection exists]
Send SiLAClientMessage (CreateBinaryUploadRequest)
alt [Upload Info: NOK]
SiLAServerMessage (BinaryTransferError)
[Upload Info: OK]
Allocate space for binary, generate binaryTransferUUID, lifetimeOfBinary
SiLAServerMessage (CreateBinaryResponse)
loop [<PartCount>]
alt [Upload: OK]
Send SiLAClientMessage (UploadChunkRequest)
SiLAServerMessage (UploadChunkResponse)
[Upload: NOK]
SiLAServerMessage (BinaryTransferError) (e.g. "Failed")
Send SiLAClientMessage (CommandExecution::BinaryTransferUUID)
SiLAServerMessage (CommandResponse)
Send SiLAClientMessage (DeleteBinaryRequest)

The following ·SiLA Client Messages· and ·SiLA Server Messages· MUST be implemented to offer the ·Binary Upload· over the ·Server-Initiated Connection Method·.

Create a Binary (Step 1)

The ·Binary Upload· is initiated by sending a ·SiLA Client Message· of type CreateBinaryUploadRequest from SiLACloudConnector.proto.

protobuf

```
// from SiLACloudConnector.proto
message CreateBinaryUploadRequest {
    repeated Metadata metadata = 1;
    // from SiLABinaryTransfer.proto
    CreateBinaryRequest createBinaryRequest = 2;
}
```

The response to creating a binary is a ·SiLA Server Message· of type CreateBinaryResponse from SiLABinaryTransfer.proto, see Protocol Buffer Definition for Binary Upload.

Upload Binary Chunk (Step 2)

A ·Binary Chunk· is uploaded by sending a ·SiLA Client Message· of type UploadChunkRequest from SiLABinaryTransfer.proto. The response to Uploading a Binary Chunk is a ·SiLA Server Message· of type UploadChunkResponse.

Issue a Command (Step 3)

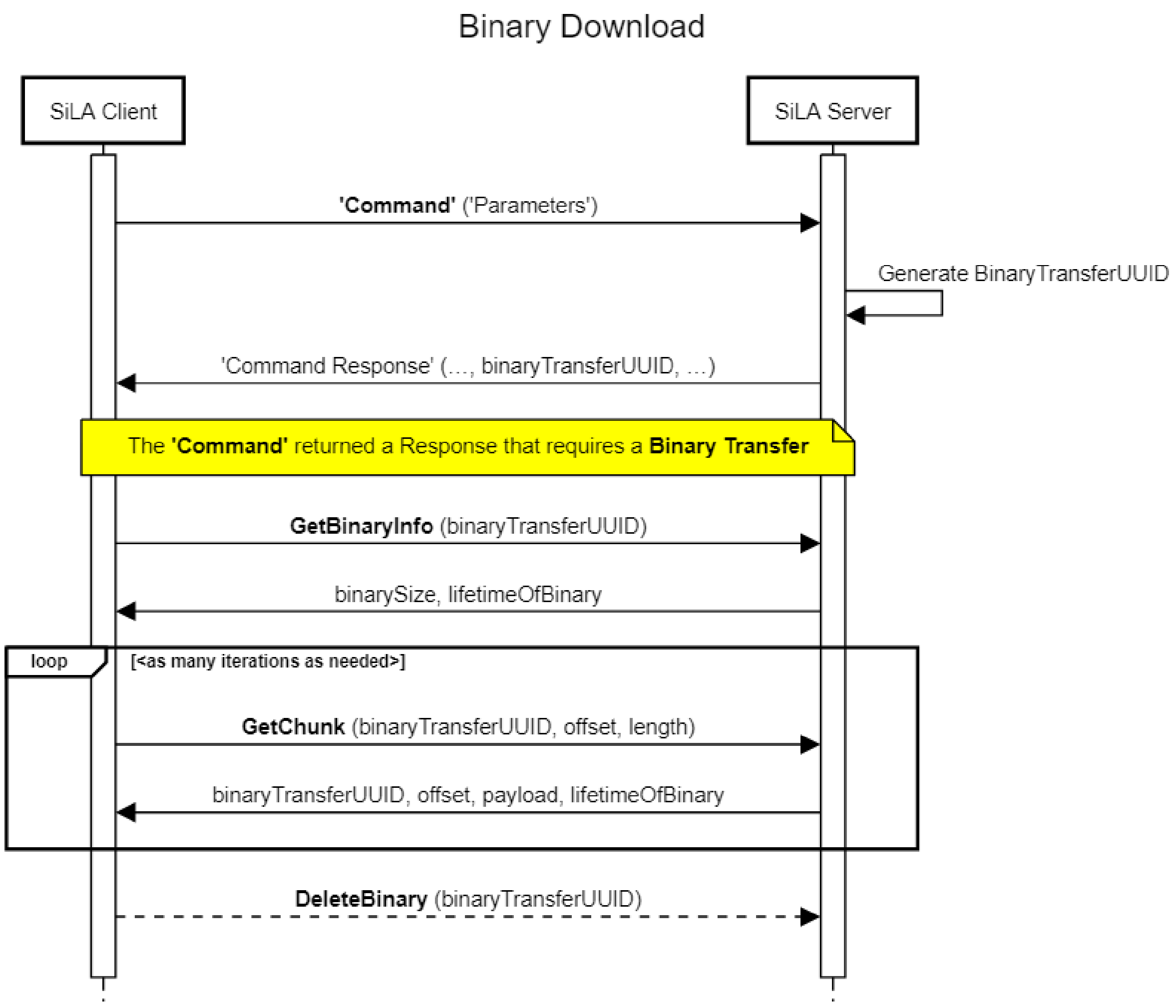
After all promised ·Binary Chunks· have been uploaded, a ·Command· ·SiLA Client Message· can be issued with the associated ·Binary Transfer UUID· as the ·Parameter· to the ·Command·.

Optionally Delete Binary Data

Binary data is deleted by sending a ·SiLA Client Message· of type DeleteBinaryRequest from SiLABinaryTransfer.proto. The response to deleting a binary is a ·SiLA Server Message· of type DeleteBinaryResponse.

Binary Download Over ·Client-Initiated Connection Method·

[COMPLETE; as of 1.0]


Binary Download
SiLA Client
SiLA Server
'Command' ('Parameters')
Generate BinaryTransferUUID
'Command Response' (..., binaryTransferUUID, ...)
The 'Command' returned a Response that requires a Binary Transfer
GetBinaryInfo (binaryTransferUUID)
binarySize, lifetimeOfBinary
loop
[<as many iterations as needed>]
GetChunk (binaryTransferUUID, offset, length)
binaryTransferUUID, offset, payload, lifetimeOfBinary
DeleteBinary (binaryTransferUUID)

The ·Binary Download· can be started once a ·Binary Transfer UUID· has been received as a ·Command Response· or ·Intermediate Command Response· of a ·Command· execution of after accessing a ·Property·. The ·Binary Download· is divided in two phases: Inspecting the binary data and downloading it.

Inspecting the Binary (Step 1)

[COMPLETE; as of 1.0]

Given the ·SiLA Server· has provided a ·Binary Transfer UUID· associated with the binary data to the ·SiLA Client·. The ·SiLA Server· MUST provide the ·SiLA Client· with the size of the binary data and the ·Lifetime of Binary· indicating when the ·Binary Transfer UUID· will expire.

Download Binary Data (Step 2)

[COMPLETE; as of 1.0]

The ·SiLA Client· can retrieve the binary data by using the associated ·Binary Transfer UUID· and the offset and lengths in bytes of the binary data. The offset and the length both start at the index zero (0). The ·SiLA Server· provides the associated ·Binary Chunk· with the current ·Lifetime of Binary·. The current ·Lifetime of Binary· that MAY be different from the time given initially.

Optionally Delete Binary Data

[COMPLETE; as of 1.0]

Additionally, the ·SiLA Server· MUST provide means to delete the Binary after use. If not deleted, the ·Binary Transfer UUID· MUST be valid until it is expired.

Protocol Buffer Definition for Binary Download

[COMPLETE; as of 1.0]

The following Protocol Buffer definition MUST be implemented to offer the ·Binary Download· service. The deletion of Binary data is replicated from the ·Binary Upload· case as the request is in a different context.

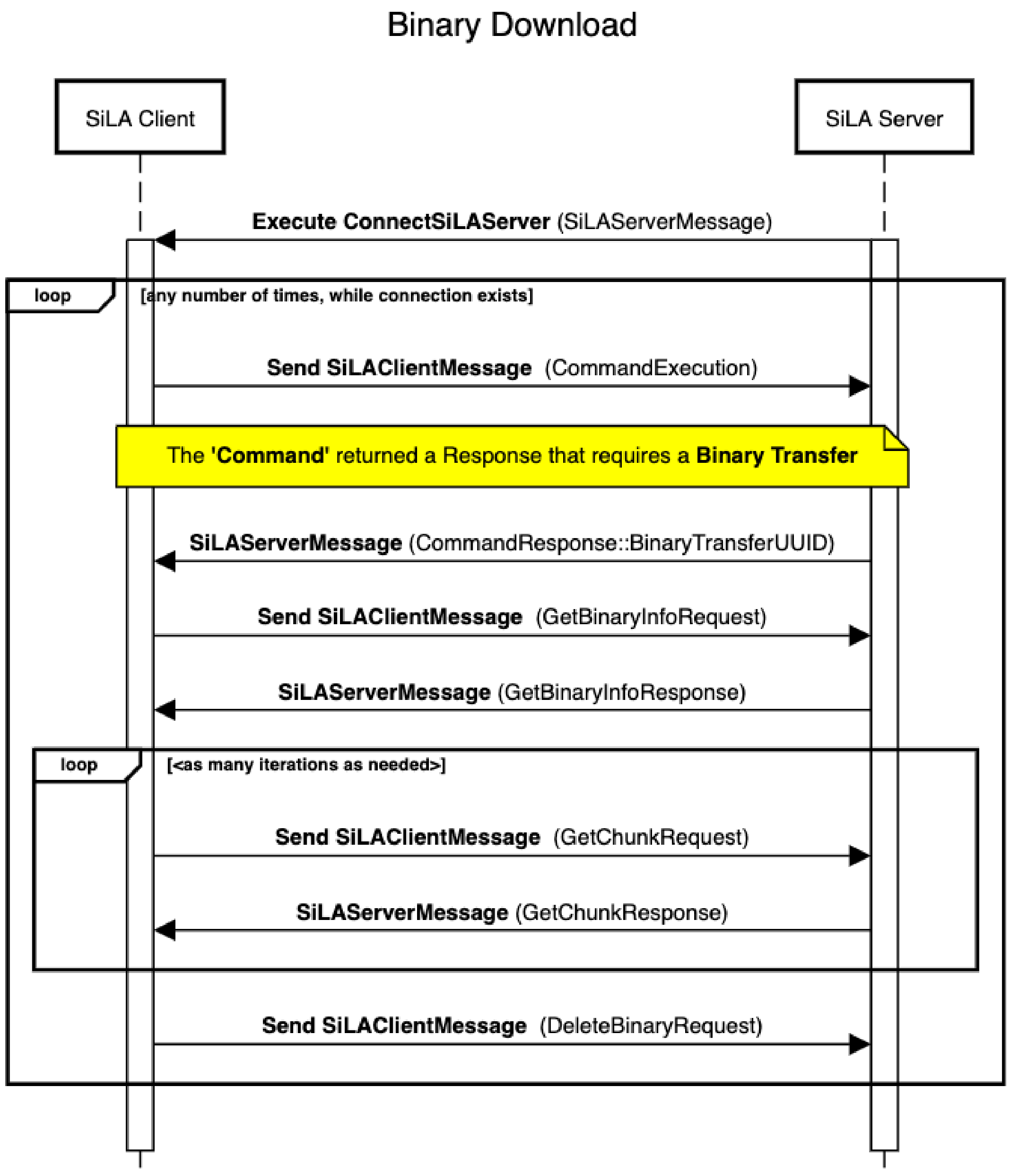
protobuf

```
// from SiLABinaryTransfer.proto
service BinaryDownload {
   rpc GetBinaryInfo (GetBinaryInfoRequest) returns (GetBinaryInfoResponse) {}
   rpc GetChunk (stream GetChunkRequest) returns (stream GetChunkResponse) {}
   rpc DeleteBinary (DeleteBinaryRequest) returns (DeleteBinaryResponse) {}
}

message GetBinaryInfoRequest {
   string binaryTransferUUID = 1;
}

message GetBinaryInfoResponse {
   uint64 binarySize = 1;
   Duration lifetimeOfBinary = 2;
}

message GetChunkRequest {
   string binaryTransferUUID = 1;
   uint64 offset = 2;
   uint32 length = 3;
}

message GetChunkResponse {
   string binaryTransferUUID = 1;
   uint64 offset = 2;
   bytes payload = 3;
   Duration lifetimeOfBinary = 4;
}

message DeleteBinaryRequest {
   string binaryTransferUUID = 1;
}

message DeleteBinaryResponse {
}
```

Binary Download Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

The ·Binary Download· over the ·Server-Initiated Connection Method· follows the same steps as described in the previous chapter Binary Download.


Binary Download
SiLA Client
SiLA Server
Execute ConnectSiLAServer (SiLAServerMessage)
loop
[any number of times, while connection exists]
Send SiLAClientMessage (CommandExecution)
The 'Command' returned a Response that requires a Binary Transfer
SiLAServerMessage (CommandResponse::BinaryTransferUUID)
Send SiLAClientMessage (GetBinaryInfoRequest)
SiLAServerMessage (GetBinaryInfoResponse)
loop
[<as many iterations as needed>]
Send SiLAClientMessage (GetChunkRequest)
SiLAServerMessage (GetChunkResponse)
Send SiLAClientMessage (DeleteBinaryRequest)

The following ·SiLA Client Messages· and ·SiLA Server Messages· MUST be implemented to offer the ·Binary Download· over the ·Server-Initiated Connection Method·.

Inspecting the Binary (Step 1)

[COMPLETE; as of 1.1]

The ·Binary Transfer UUID· associated with the binary data can be sent over a ·SiLA Client Message· of type GetBinaryInfo from SiLABinaryTransfer.proto in order to retrieve info about the binary data. The response is a ·SiLA Server Message· of type GetBinaryInfoResponse from SiLABinaryTransfer.proto.

Download Binary Data (Step 2)

[COMPLETE; as of 1.1]

A ·Binary Chunk· is downloaded by sending a ·SiLA Client Message· of type GetChunk from SiLABinaryTransfer.proto. The response to downloading a ·Binary Chunk· is a ·SiLA Server Message· of type GetChunkResponse.

Optionally Delete Binary Data

[COMPLETE; as of 1.1]

A binary data is deleted by sending a ·SiLA Client Message· of type DeleteBinary from SiLABinaryTransfer.proto.

Binary Transfer Errors

[COMPLETE; as of 1.0]

[Definition: Binary Transfer Error] **Binary Transfer Errors** are errors that can occur during a ·Binary Transfer·. Like the SiLA Error Protocol Buffer Message, the "BinaryTransferError" Protocol Buffer message MUST be serialized and encoded in Base64 (according to RFC 4648) into the gRPC status message. **Binary Transfer Errors** are considered to be ·Undefined Execution Errors·.

Please note, Binary Transfer Errors are not considered ·Framework Errors·. The ·Binary Transfer· is a concept that is defined in Part B only and nonexistent in Part A of the SiLA 2 Specification.

protobuf

```
// from SiLABinaryTransfer.proto
message BinaryTransferError {
    enum ErrorType {
        INVALID_BINARY_TRANSFER_UUID = 0;
        BINARY_UPLOAD_FAILED = 1;
        BINARY_DOWNLOAD_FAILED = 2;
    }
    ErrorType errorType = 1;
    string message = 2;
}
```

Invalid Binary Transfer UUID

[COMPLETE; as of 1.0]

[Definition: Invalid Binary Transfer UUID Error] The **Invalid Binary Transfer UUID Error** MUST be issued when a ·SiLA Server· receives an invalid ·Binary Transfer UUID·.

Binary Upload Failed

[COMPLETE; as of 1.0]

[Definition: Binary Upload Failed Error] The **Binary Upload Failed Error** MUST be issued when a ·SiLA Server· receives binary data that can not be received, e.g. because it does not implement ·Binary Transfer·.

Binary Download Failed

[COMPLETE; as of 1.0]

[Definition: Binary Download Failed Error] The **Binary Download Failed Error** MUST be issued when a ·SiLA Server· receives a request for binary data that can not be sent, e.g. because the requested ·Binary Chunk· does not exist.

SiLA Any Type

[COMPLETE; as of 1.0]

The ·SiLA Any Type· is used to send data that can represent any ·SiLA Data Type· not known at the design time of a ·Feature·.

Any receiver of a ·SiLA Any Type· MUST be able to decode the payload without any further context of the ·SiLA Server·. Therefore, a ·SiLA Any Type· MUST NOT be used to send a ·Custom Data Type·, as this would require a receiver to know the ·Feature· the ·SiLA Any Type· was used in.

The Protocol Buffer message of a ·SiLA Any Type· is composed of two fields, a "type" field describing the ·SiLA Data Type· of the data sent or received and a "payload" field that contains the actual data:

protobuf

```
// from SiLAFramework.proto
message Any {
   string type = 1;
   bytes payload = 2;
}
```

Both, a ·SiLA Server· and a ·SiLA Client·, can be the receiver or sender of a ·SiLA Any Type·. The sender MUST create an XML string describing the ·SiLA Data Type· (The "type" Field) and serialize the data according to The "payload" Field. A receiver MUST parse the XML string and deserialize the payload accordingly.

The "type" Field

The "type" field contains a representation of the ·SiLA Data Type· in the form of an XML String that corresponds to the AnyTypeDataType.xsd Schema definition.

The "payload" Field

The actual data to transport MUST be serialized according to the rules defined for Encoding of Protocol Buffers as a message containing one field.

To make this more readable and understandable, the Protocol Buffer messages syntax is used here for explanation. Using this syntax, the data has to be the value of the field "ArbitraryFieldName" of a message of the form as specified by the message "ArbitraryMessageName" below. The "payload" field of the ·SiLA Any Type· MUST contain the serialized form of the message below:

protobuf

```
message ArbitraryMessageName {
   <·SiLA Data Type· as defined by "type" field> ArbitraryFieldName = 1;
}
```

Note, that both the message name (“ArbitraryMessageName”) and the field name (“ArbitraryFieldName ”) are completely arbitrary, as they do not affect the serialized binary form of the Protocol Buffer message.

SiLA Void Type

The SiLA Void Type MUST be implemented by a constrained SiLA String Type with an empty payload as follows:

XML

```
<DataType>
	<Constrained>
		<DataType>
			<Basic>String</Basic>
		</DataType>
		<Constraints>
			<Length>0</Length>
		</Constraints>
	</Constrained>
</DataType>
```

Note: This solution was chosen because of backwards compatibility (refer to meeting minutes #90, #176 and #177).

Example

A ·SiLA Any Type· that contains data of a ·SiLA List Type· of a ·SiLA Structure Type· containing two elements called “X” and “Y” of ·SiLA Real Type· as its ·SiLA Data Type·, results in the following XML string describing the ·SiLA Data Type·:

XML

```
<DataType>
	<List>
		<DataType>
			<Structure>
				<Element>
					<Identifier>X</Identifier>
					<DisplayName>X Coordinate</DisplayName>
					<Description>The X coordinate.</Description>
					<DataType>
						<Basic>Real</Basic>
					</DataType>
				</Element>
				<Element>
					<Identifier>Y</Identifier>
					<DisplayName>Y Coordinate</DisplayName>
					<Description>The Y coordinate.</Description>
					<DataType>
						<Basic>Real</Basic>
					</DataType>
```

```
                        </Element>
                    </Structure>
                </DataType>
            </List>
</DataType>
```

A Protocol Buffer message of the following form must be used to serialize the data contained in the ·SiLA Any Type·, whose ·SiLA Data Type· is of ·SiLA List Type· of a ·SiLA Structure Type· containing two elements called “X” and “Y” of ·SiLA Real Type·:

protobuf

```
message ArbitraryMessageName {
    message ArbitraryFieldName_Struct {
        Real X = 1;
        Real Y = 2;
    }
    repeated ArbitraryFieldName_Struct ArbitraryFieldName = 1
}
```

SiLA Derived Types

[COMPLETE; as of 0.1]

All ·SiLA Derived Types· have different underlying ·SiLA Data Types·. This allows for a recursive mapping with the lowest denominator being the ·SiLA Basic Type·.

SiLA List Type

[COMPLETE; as of 0.1]

A ·SiLA List Type· marks the underlying ·SiLA Data Type· with the Protocol Buffer keyword “**repeated**” that allows the type to be repeated any number of times (including zero).

List of lists SHALL NOT be allowed. Note that there is no repetition possible of an already repeated field, which is why SiLA disallows lists of lists.

protobuf

```
// Depending on the context, the surrounding message MSG,
// the <ID> below becomes:
// For ·Command Parameters·:
// MSG = <·Command Identifier·>_Parameters → <ID> = <·Command Parameter Identifier·>
// For ·Command Responses·:
// MSG = <·Command Identifier·>_Responses → <ID> = <·Command Response Identifier·>
// For ·Intermediate Command Responses·:
// MSG = <·Command Identifier·>_IntermediateResponses →
//           <ID> = <·Intermediate Command Response Identifier· 2>
// For ·Properties·:
// MSG = Get_<·Property Identifier·>_Responses → <ID> = <·Property Identifier·>
// For ·SiLA Client Metadata·:
// MSG = Metadata_<·Metadata Identifier·> → <ID> = <·Metadata Identifier·>
// For ·Custom Data Types·:
```

```
// MSG = DataType_<·Custom Data Type Identifier·> →
//         <ID> = <·Custom Data Type Identifier·>

message MSG {
   ... // Depending on the message MSG, more fields may be here
   repeated <protobuf type> <ID> = X; // X = the gRPC field number
                                      // <protobuf type> = the type mapping for <ID>
   ... // Depending on the message MSG, more fields may be here
}
```

Example

[COMPLETE; as of 1.0]

Given a ·Command· “MoveTrajectory” with a ·Command Parameter· called “Points” that is defined as a ·SiLA List Type· of a ·SiLA Structure Type· containing two elements called “X” and “Y” of ·SiLA Real Type· results in:

protobuf

```
message MoveTrajectory_Parameters {
    message Points_Struct {
        Real X = 1;
        Real Y = 2;
    }
    repeated Points_Struct Points = 1;
}
```

SiLA Structure Type

[COMPLETE; as of 0.1]

The ·SiLA Structure Type· is constructed by appending the keyword “Struct” (as shown in the gRPC Mapping) in a nested message inside the ·Command Parameter·, ·Command Response·, ·Intermediate Command Response·, ·Property·, ·SiLA Client Metadata· or ·Custom Data Type· message. Each element follows the same conventions as for ·SiLA Data Types·.

protobuf

```
// Depending on the context, i.e. the surrounding message MSG,
// the <ID> below becomes:
// For ·Command Parameters·:
// MSG = <·Command Identifier·>_Parameters → <ID> = <·Command Parameter Identifier·>
// For ·Command Responses·:
// MSG = <·Command Identifier·>_Responses → <ID> = <·Command Response Identifier·>
// For ·Intermediate Command Responses·:
// MSG = <·Command Identifier·>_IntermediateResponses →
//         <ID> = <·Intermediate Command Response Identifier· 2>
// For ·Properties·:
// MSG = Get_<·Property Identifier·>_Responses → <ID> = <·Property Identifier·>
// For ·SiLA Client Metadata·:
// MSG = Metadata_<·Metadata Identifier·> → <ID> = <·Metadata Identifier·>
// For ·Custom Data Types·:
// MSG = DataType_<·Custom Data Type Identifier·> →
//         <ID> = <·Custom Data Type Identifier·>

message MSG {
   ... // Depending on the message MSG, more fields may be here
   message <ID>_Struct {
       <·Element Data Type· 1> <·Element Identifier· 1> = 1;
       <·Element Data Type· 2> <·Element Identifier· 2> = 2;
       ...
       <·Element Data Type· N> <·Element Identifier· N> = N;
   }
   <ID>_Struct <ID> = X; // X = the gRPC field number
   ... // Depending on the message MSG, more fields may be here
}
```

Example

[COMPLETE; as of 1.0]

Given a ·Command· “Move” with a ·Command Parameter· called “Coordinates” that is defined as a ·SiLA Structure Type· having four elements called “StartX”, “StartY”, “EndX” and “EndY” of ·SiLA Real Type· results in:

Protobuf

```
message Move_Parameters {
    message Coordinates_Struct {
        Real StartX = 1;
        Real StartY = 2;
        Real EndX = 3;
        Real EndY = 4;
    }
    Coordinates_Struct Coordinates = 1;
}
```

SiLA Constrained Type

[COMPLETE; as of 0.1]

·SiLA Constrained Types· have no mapping into the Protocol Buffer protocol and are ignored. So a ·SiLA Constrained Type· of a ·SiLA Basic Type· looks the same on the Protocol Buffer level as the ·SiLA Basic Type· itself.

However, ·SiLA Servers· MUST apply the ·Constraints· and check the value of the ·SiLA Data Type· to comply with the ·Constraints· and throw appropriate ·Validation Errors· if the ·Constraints· are violated. ·Constraints· are independently validated and the validation result is successively combined using the logical AND operator. This means, if at least one ·Constraint· is violated, an appropriate ·Validation Error· is thrown.

Custom Data Type

[COMPLETE; as of 0.1]

A ·Custom Data Type· is constructed with the same rules as ·SiLA Basic Types· and ·SiLA Derived Types· but defined inside a separate Protocol Buffer message. All fields in other gRPC messages referring to the ·Custom Data Type· MUST refer to the gRPC message as specified here.

protobuf

```
message DataType_<·Custom Data Type Identifier·> {
    <·Custom Data Type Data Type·> <·Custom Data Type Identifier·> = 1;
}
```

Example

[COMPLETE; as of 1.0]

protobuf

```
message DataType_TransferLocations {
    String TransferLocations = 1;
}
```

Error Handling

[COMPLETE; as of 0.1, updates: 1.1]

The SiLA 2 Specification Part A specifies four error categories: ·Validation Errors·, ·Execution Errors·, ·Framework Errors· and ·Connection Errors·. In addition, Part B defines additional types of errors. All situations when these errors can occur are described in the respective chapters.

gRPC Error Mechanism

[COMPLETE; as of 0.1]

The gRPC Error Model introduces a way of handling errors. For this purpose, the mechanism uses a simple concept of a status code combined with an optional string status message. The status codes available in gRPC are limited to the predefined status codes.

The gRPC error model comes with a set of predefined errors which are split into three categories:

1. General errors
2. Network failures
3. Protocol errors

All errors in these three categories are classified as SiLA ·Connection Errors·. ·Connection Errors· usually show up as timeouts in actual implementations. There are many reasons for timeouts (on the network or when a ·SiLA Server· or ·SiLA Client· does not respond due to any reason). Often a SiLA ·Connection Error· is presented to the ·SiLA Server· or ·SiLA Client· by a so-called “deadline exceeded” error in gRPC (gRPC status code DEADLINE_EXCEEDED).

Raising an Error in gRPC

[COMPLETE; as of 0.1]

The specific implementation of raising an error in gRPC depends on the programming language. The errors defined in the gRPC framework are automatically raised by the gRPC library and therefore, do not need to be explicitly raised in the implementation. Error codes used by the gRPC library MUST NOT be used in the SiLA specific aspects of the implementation.

Handling Error in gRPC

[COMPLETE; as of 0.1, updates: 1.1]

gRPC error handling is programming language specific. However, each implementation provides means to access the status code and status message of each caught error. The status code can be easily handled programmatically to identify the error and take appropriate actions while the message usually contains helpful information to the user why and where the error occurred.

As SiLA implementations that use the ·Client-Initiated Connection Method· are based on HTTP/2 and Protocol Buffers (specified by the wire format of gRPC) all above errors MUST be handled using the gRPC error handling mechanism. The gRPC library already provides a set of errors that occur during the data transmission and type validation. As specified in Part A of this specification, the native gRPC errors are all SiLA specific ·Connection Errors·.

All SiLA ·Connection Errors· are automatically raised by the gRPC library using the gRPC error mechanism. All other errors MUST be raised using the SiLA error mechanism, which is an extension to the gRPC error mechanism to provide additional information about the error.

SiLA Error Mechanism

[COMPLETE; as of 0.1]

Since HTTP/2 and Protocol Buffers (specified by the wire format of gRPC) are used as the base technology for SiLA, the SiLA error mechanism is an extension to the gRPC error mechanism. The main driver for this decision is a simplified implementation by using existing error mechanisms.

The gRPC error model specifies two different fields that are used to communicate information on errors between the gRPC server and the gRPC client:

1. Status code
2. Status message

Error Handling Over ·Client-Initiated Connection Method·

[COMPLETE; as of 0.1, updates: 1.1]

There are different gRPC status codes available (as described in Status codes and their use in gRPC). To avoid ambiguity, all SiLA errors (·Validation Errors·, ·Execution Errors·, ·Framework Errors· but except ·Connection Errors·, which are handled differently, see above) MUST use the gRPC status code ABORTED.

The ·Validation Errors·, ·Execution Errors· and ·Framework Errors· are mapped to the SiLA Error Protocol Buffer Message and serialized into the gRPC status message.

SiLA Error Protocol Buffer Message

[COMPLETE; as of 0.2]

Information about a ·Validation Error·, ·Execution Error· or ·Framework Error· is transferred using a serialized Protocol Buffer message of the form as specified below for the `SiLAError` Protocol Buffer message. The **SiLAError** Protocol Buffer message allows storing information on the error that occurred.

The **SiLAError** Protocol Buffer message MUST be serialized and encoded in Base64 (according to RFC 4648) into the gRPC status message.

protobuf

```
// from SiLAFramework.proto
message SiLAError {
    oneof error {
        ValidationError validationError = 1;
        DefinedExecutionError definedExecutionError = 2;
        UndefinedExecutionError undefinedExecutionError = 3;
        FrameworkError frameworkError = 4;
    }
}

message ValidationError {
    string parameter = 1;  /* fully qualified parameter identifier */
    string message = 2;
}

message DefinedExecutionError {
    string errorIdentifier = 1;  /* fully qualified defined execution error
identifier */
    string message = 2;
}

message UndefinedExecutionError {
    string message = 1;
}

message FrameworkError {
    enum ErrorType {
        COMMAND_EXECUTION_NOT_ACCEPTED = 0;
        INVALID_COMMAND_EXECUTION_UUID = 1;
        COMMAND_EXECUTION_NOT_FINISHED = 2;
        INVALID_METADATA = 3;
        NO_METADATA_ALLOWED = 4;
    }
    ErrorType errorType = 1;
    string message = 2;
}
```

The ·SiLA Client· MAY Base64 decode (according to RFC 4648) and deserialize the **SiLAError** message from the gRPC status message to infer the error which has occurred.

In the case where a ·Defined Execution Error· is raised by the ·SiLA Server·, the **SiLAError** message MUST specify the ·Fully Qualified Defined Execution Error Identifier· as the value of the “errorIdentifier” field.

Error Handling Over ·Server-Initiated Connection Method·

[COMPLETE; as of 1.1]

With the ·Server-Initiated Connection Method·, the `SiLAError` Protocol Buffer message is directly sent inside a ·SiLA Server Message· to the ·SiLA Client·.

Examples

[COMPLETE; as of 0.1]

Validation Error

[COMPLETE; as of 0.1]

A Plate robot is commanded to move a plate from origin to the destination, but the destination is invalid.

Message: “The destination site is out of range for this plate robot. Please give a site within the range. (1...3)”

Defined Execution Error

[COMPLETE; as of 0.2]

A camera-based barcode reader fails to read the code, when calling commands from the “BarcodeController” feature.

·Fully Qualified Defined Execution Error Identifier·:
“org.silastandard/core/BarcodeController/v1/ImageTooDark”

Message: “Barcode reader could not decode the code because the image was too dark. Please increase the lighting.”

Undefined Execution Error

[COMPLETE; as of 0.2]

Plate robot crashes, when calling a command of the “RobotController” ·Feature·.

Message: “The robot crashed to an unknown obstacle and the command could not be finished. Please remove the obstacle and try again.”

Encryption

[COMPLETE; as of 0.2, updates: 1.1]

·SiLA Clients· and ·SiLA Servers· MUST always use TLS to encrypt ·SiLA Client Requests· and ·SiLA Server Responses· within ·Connections·. It is RECOMMENDED for ·SiLA Servers· to use trusted certificates to initiate TLS handshake. A ·SiLA Client· SHALL only accept trusted certificates implicitly (i.e.silently) for TLS handshake. A ·SiLA Client· MUST NOT accept untrusted certificates for TLS handshake implicitly. A ·SiLA Client· MAY accept untrusted certificates for TLS handshake explicitly (e.g. by notifying a user or ensuring the trust in another appropriate way).

If a ·SiLA Server· uses an untrusted certificate, the ·SiLA Server· MUST use a certificate with the Common Name (CN) set to “SiLA2”.

It is RECOMMENDED that the ·SiLA Server· uses a certificate with an extension with OID “1.3.6.1.4.1.58583” set to the string representation of the ·SiLA Server UUID· (e.g. “c80f3a3f-de77-...”, see also ·UUID·). The number “58583” is the PEN of the SiLA Organization, see also IANA Private Enterprise Number (PEN).

The client MAY reject servers that do not have the ·SiLA Server UUID· stored in the certificate.

Untrusted certificates SHALL be accepted for setups using private IP addresses, according to RFC1918 in IPv4 networks (24-bit block 10.0.0.0 – 10.255.255.255, 20-bit block 172.16.0.0 – 172.31.255.255 and 16-bit block 192.168.0.0 – 192.168.255.255) and according to RFC4193, Unique Local Addresses (ULA), in IPv6 Networks.

SiLA Server Discovery

[COMPLETE; as of 0.1, updates: 1.1]

·SiLA Server Discovery· MUST be implemented based on zero-configuration networking, which enables automatic discovery of ·SiLA Server· on Internet Protocol (IP) networks. ·SiLA Server Discovery· uses industry standard IP protocols to allow ·SiLA Servers· to automatically discover each other without the need to enter IP addresses or to configure DNS servers.

Note, ·SiLA Server Discovery· is only useful in the context of the ·Client-Initiated Connection Method·.

SiLA Server Discovery Support

[COMPLETE; as of 0.1, updates: 1.1]

A ·SiLA Server· MUST implement multicast DNS (mDNS) [RFC6762], and DNS-based Service Discovery (DNS-SD) [RFC6763].

In order to provide a true zero-configuration experience, the ·SiLA Server· MUST have ·SiLA Server Discovery· enabled by default. It MUST NOT be possible to disable any part of ·SiLA Server Discovery·. This safeguards the users from accidentally configuring the ·SiLA Server· in such a way that they can no longer communicate with it.

The address of the advertised service MUST represent the socket [RFC793] on which the HTTP server is exposing all ·Features· of the ·SiLA Server·.

For local discovery on the same physical network, the ·SiLA Server· SHOULD expect DNS requests via mDNS, for wide-area discovery centralized DNS Servers, that are queried via unicast, might be installed, e.g. for discovering remote services. Beyond a certain size, every service-discovery protocol has to transition from using peer-to-peer multicast to some kind of centralized repository using a wide-area protocol.

If a ·SiLA Server· uses an untrusted certificate, the ·SiLA Server· MUST send a textual representation of the PEM-encoded Certificate Authority used as a TXT (see [RFC1035]) record. The TXT record MUST contain the certificate where each line is stored separately (given that strings in a TXT record have a limitation to 255 characters) in a string prepended with “ca<l>=”

where “I” MUST be the 0-indexed line number. This allows clients to check the certificate before attempting to connect.

Service Naming

[COMPLETE; as of 0.1]

Terminology

[COMPLETE; as of 0.1]

DNS Service Discovery “Service Instance Names” are of the form:

Service Instance Name =

```
<instance>.<service>.<domain>
```

For SiLA discovery the following rules apply:

- The instance MUST be the ·SiLA Server UUID·, represented in the string representation (e.g. “f81d4fae-7dec-11d0-a765-00a0c91e6bf6”), as specified by RFC 4122.
- The service MUST be “_sila._tcp“.
- The domain MUST be “local.”.

An example Service Instance Name is:

```
25597b36-e9bf-11e8-aeb5-f2801f1b9fd1._sila._tcp.local.
```

The Service Instance Name can be up to 255 bytes of UTF-8 [RFC6762 - Appendix C]. The bytes are used by:

- 36 bytes for the ·SiLA Server UUID·
- 14 bytes for the service and domain
- 5 bytes for the delimiter dots

Note that on the discovery part there is no additional information of the ·SiLA Server·'s functionality. Any info about the functionality SHOULD be retrieved by using the ·SiLA Service Feature·.

Handling Name Conflicts

[COMPLETE; as of 0.1]

The multicast DNS (mDNS) specification [RFC6762] describes how to handle name conflicts for (device) hostnames. However, SiLA avoids name conflicts completely, by using a unique identifier for the “Service Instance Name”.

SiLA Server Properties

[COMPLETE; as of 1.1]

In order for ·SiLA Clients· to be able to provide rich information about ·SiLA Servers· without having to call the ·SiLA Service Feature·, the DNS-SD TXT records [RFC6762] properties field SHOULD contain the ·SiLA Server· properties ·SiLA 2 Version·, ·SiLA Server Name· and ·SiLA Server Description·, encoded as key/value pairs as specified below.

The key/value pairs MUST be separated by “=” (equal sign, ASCII code 61_{dec}). If no value is specified for a key, the value is ignored as if it was not present in the record. TXT record entries are separated by a leading byte that indicates the total length of the key/value string. See also [RFC1035]. Further formatting rules are specified for zeroconf. The encoding of the TXT record must be UTF-8.

Note that ·SiLA Server Name· and ·SiLA Server Description· might be too long to be encoded as a key/value pair as indicated above. Key/value pairs exceeding the 255 bytes MUST be truncated. Note, that the UTF-8 encoding uses up to 4 bytes per character, depending on the Unicode of the respective character and also consider the bytes used for the key and the equal sign.

```
version=<·SiLA 2 Version·>
server_name=<·SiLA Server Name·>
description=<·SiLA Server Description·>
```

Example

```
| 0x0B | version=1.1 | 0x1C | server_name=HelloSiLA_server | 0x17 | description=Hello
SiLA! |
```

Definition Index

A list of all Terms defined in Part A:

[Definition: Term] a Term is something used with a special meaning. The definition is labeled as such and the Term it defines is displayed in boldface. The end of the definition is not specially marked in the displayed or printed text. Uses of defined Terms are links to their definitions, set off with middle dots, for instance ·Term·.
[Definition: SiLA Server] A SiLA Server is a system (a software system, a laboratory instrument, or device) that offers ·Features· to a ·SiLA Client·. Every SiLA Server MUST implement the ·SiLA Service Feature·.
[Definition: Lifetime of a SiLA Server] The Lifetime of a SiLA Server is the time span between the state of power-up and being ready to accept ·Connections· and the shutdown process of a ·SiLA Server·, after which no new ·Connections· will be accepted.
[Definition: SiLA Server Name] The SiLA Server Name is a human readable name of the ·SiLA Server·. By default this name SHOULD be equal to the ·SiLA Server Type·. This property MUST be configurable via the ·SiLA Service Feature·’s “Set Server Name” Command. This property has no uniqueness guarantee. A SiLA Server Name is the ·Display Name· of a ·SiLA Server· (i.e. MUST comply with the rules for any ·Display Name·, hence be a string of UNICODE characters of maximum 255 characters in length).
[Definition: SiLA Server Type] The SiLA Server Type is a human readable ·Identifier· of the ·SiLA Server· used to describe the entity that the ·SiLA Server· represents. For example, the make and model for a hardware device (·SiLA Device·). A SiLA Server Type MUST comply with the rules for any ·Identifier· and start with an upper-case letter (A-Z) and MAY be continued by lower and

upper-case letters (A-Z and a-z) and digits (0-9) up to a maximum of 255 characters in length.
[Definition: SiLA Server UUID] The SiLA Server UUID is a ·UUID· of a ·SiLA Server·. Each ·SiLA Server· MUST generate a ·UUID· once, to uniquely identify itself. It needs to remain the same even after the ·Lifetime of a SiLA Server· has ended.
[Definition: SiLA Server Version] The SiLA Server Version is the version of the SiLA Server. A "Major" and a "Minor" version number (e.g. 1.0) MUST be provided, a Patch version number MAY be provided. Optionally, an arbitrary text, separated by an underscore MAY be appended, e.g. “3.19.373_mighty_lab_devices”.
[Definition: SiLA Server Vendor URL] The SiLA Server Vendor URL is the URL to the website of the vendor or the website of the product of this SiLA Server. This URL SHOULD be accessible at all times. The URL is a Uniform Resource Locator as defined in RFC 1738.
[Definition: SiLA Server Description] The SiLA Server Description is the description of the SiLA Server. It SHOULD include the use and purpose of this SiLA Server.
[Definition: SiLA Device] A SiLA Device is a ·SiLA Server· that is a physical thing made or adapted for a particular purpose, especially a piece of mechanical or electronic equipment. A SiLA Device is a specialization of a ·SiLA Server· with additional requirements regarding joining a communication network.
[Definition: SiLA Client] A SiLA Client is a system (a software system, a laboratory instrument, or device), that is using ·Features· offered by a ·SiLA Server·.
[Definition: Feature] Each Feature describes a specific behavior of a ·SiLA Server· (e.g. the ability to measure a spectrum, to register a sample in a LIMS, control heating, etc.). Features are implemented by a ·SiLA Server· and used by a ·SiLA Client·. The ·SiLA Service Feature· MUST be implemented by each ·SiLA Server·. The ·SiLA Service Feature· offers basic information about the ·SiLA Server· and about all other Features the ·SiLA Server· implements.
[Definition: Connection] A Connection is the communication channel between a ·SiLA Client· and a ·SiLA Server·, established over a communication network. All information exchange between a ·SiLA Client· and a ·SiLA Server· MUST be exchanged through the Connection.
[Definition: Address] An Address is an identifier that uniquely identifies a ·SiLA Client· or a ·SiLA Server· in a communication network.
[Definition: Connection Method] The Connection Method specifies which of the parties is establishing the ·Connection·.
[Definition: Client-Initiated Connection Method] With the Client-Initiated Connection Method, the ·SiLA Client· is establishing the ·Connection· to the ·SiLA Server·. This ·Connection Method· is available as of ·SiLA 2 Version· “0.1”.
[Definition: Server-Initiated Connection Method] With the Server-Initiated Connection Method, (a.k.a. “cloud connectivity” or “reverse connection”), the ·SiLA Server· is establishing the ·Connection· to the ·SiLA Client·. This ·Connection Method· is available as of ·SiLA 2 Version· “1.1”.
[Definition: SiLA Client Request] A SiLA Client Request is a piece of information sent from a ·SiLA Client· to a ·SiLA Server· within a ·Connection·.
[Definition: SiLA Server Response] A SiLA Server Response is a piece of information sent from a ·SiLA Server· to a ·SiLA Client· within a ·Connection·. A SiLA Server Response is always returned in reply to a ·SiLA Client Request·. For subscriptions (to ·Observable Properties· or ·Observable Commands·), a streamed SiLA Server Response is sent as long as the subscription is active.
[Definition: Header] A Header is the first part of a ·SiLA Client Request· or ·SiLA Server Response·. A Header contains for example ·SiLA Client Metadata· of the ·Payload· and error information.
[Definition: Trailer] A Trailer is the last part of a streamed ·SiLA Server Response·. A Trailer

contains for example error information.
[Definition: Payload] The Payload is the actual data exchanged between a ·SiLA Client· and a ·SiLA Server·.
[Definition: Feature Framework] Feature Framework is an overarching ·Term· describing the intention of ·Features· in SiLA 2, how to design, store and maintain them.
[Definition: Feature Designer] A Feature Designer is a person designing a ·Feature·. Usually this SHALL be a subject matter expert for the domain that the ·Feature· addresses.
[Definition: SiLA Standard Feature] A SiLA Standard Feature is a ·Feature· that has been standardized by the SiLA organization according to the standardization procedure. An officially standardized and released ·Feature· MUST have the ·Originator· set to "org.silastandard" and the ·Maturity Level· to "Normative". There is a formal process for becoming a SiLA Standard Feature according to the SiLA by-laws.
[Definition: Online Feature Repository] The Online Feature Repository is a place for storing ·Feature Definitions·. It can be found on GitLab.
[Definition: Feature Definition] The Feature Definition describes a certain behavior of a ·Feature· in an exact and very specific way.
[Definition: Feature Identifier] A Feature Identifier is the ·Identifier· of a ·Feature·. Each ·Feature· MUST have a Feature Identifier. All ·Features· sharing the scope of the same ·Originator· and ·Category· MUST have unique Feature Identifiers. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.
[Definition: Feature Display Name] A Feature Display Name is the ·Display Name· of a ·Feature·.
[Definition: Feature Description] A Feature Description is the ·Description· of a ·Feature·. A Feature Description MUST describe the behaviors / capabilities the ·Feature· models in human readable form and with as many details as possible. The Feature Description SHOULD contain all details about the ·Feature· as described under best practice below. The Feature Description MUST be human readable text in American English (see also Internationalization).
[Definition: Command] A Command models an action that will be performed on a ·SiLA Server·. A Command MAY except ·Command Parameters· and MAY return ·Command Responses· after Command execution or MAY provide ·Intermediate Command Responses· during execution. There are ·Unobservable Commands· and ·Observable Commands·.
[Definition: Command Identifier] A Command Identifier is the ·Identifier· of a ·Command·. A Command Identifier MUST be unique within the scope of a ·Feature·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.
[Definition: Command Display Name] A Command Display Name is the human readable ·Display Name· of a ·Command·.
[Definition: Command Description] A Command Description is the ·Description· of a ·Command·.
[Definition: Unobservable Command] Any ·Command· for which observing the progress or status of the ·Command· execution on the ·SiLA Server· is not possible or does not make sense, SHOULD be defined as an Unobservable Command, i.e. a ·Command· with item "Observable" set to "No".
[Definition: Observable Command] Any ·Command· for which observing the progress or status of the ·Command· execution on the ·SiLA Server· is possible and makes sense, e.g. measuring a spectrum, SHOULD be an Observable Command, i.e. a ·Command· with item "Observable" set to "Yes".
[Definition: Command Observable Setting] The Command Observable Setting specifies whether a ·Command· is an ·Observable Command· or an ·Unobservable Command·.
[Definition: Command Parameter] A Command Parameter is a ·Parameter· of a ·Command·. It MUST be submitted with the ·Command· execution request. A ·Command· MAY have one or more

Command Parameters.
[Definition: Command Parameter Identifier] The Command Parameter Identifier is the ·Identifier· of a ·Command Parameter·. A Command Parameter Identifier MUST be unique within the scope of all ·Parameters· of a ·Command·.
[Definition: Command Parameter Display Name] A Command Parameter Display Name is the ·Display Name· of a ·Command Parameter·.
[Definition: Command Parameter Description] A Command Parameter Description is the ·Description· of a ·Command Parameter·.
[Definition: Command Parameter Data Type] A Command Parameter Data Type is The ·SiLA Data Type· of a ·Command Parameter·.
[Definition: Command Response] A Command Response contains a result of a ·Command· execution. A ·Command· MAY have one or more Command Responses.
[Definition: Command Response Identifier] A Command Response Identifier is the ·Identifier· of a ·Command Response·. A Command Response Identifier MUST be unique within the scope of al ·Command Responses· of a ·Command·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.
[Definition: Command Response Display Name] A Command Response Display Name is the ·Display Name· of a ·Command Response·.
[Definition: Command Response Description] A Command Response Description is a ·Description· of a ·Command Response·.
[Definition: Command Response Data Type] A Command Response Data Type is the ·SiLA Data Type· of a ·Command Response·.
[Definition: Intermediate Command Response] An Intermediate Command Response is a partial ·SiLA Server Response· of an ·Observable Command·. An ·Observable Command· MAY have one or more Intermediate Command Responses.
[Definition: Intermediate Command Response Identifier] An Intermediate Command Response Identifier is the ·Identifier· of an ·Intermediate Command Response·. An Intermediate Command Response Identifier MUST be unique within the scope of all ·Intermediate Command Responses· of a ·Command·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.
[Definition: Intermediate Command Response Display Name] An Intermediate Command Response Display Name is the ·Display Name·of an ·Intermediate Command Response·.
[Definition: Intermediate Command Response Description] An Intermediate Command Response Description is the ·Description· of an ·Intermediate Command Response·.
[Definition: Intermediate Command Response Data Type] An Intermediate Command Response Data Type is the ·SiLA Data Type· of an ·Intermediate Command Response·.
[Definition: Command Execution UUID] A Command Execution UUID is the ·UUID· of a ·Command· execution. It is unique within one instance of a ·SiLA Server· and its lifetime (·Lifetime of a SiLA Server·).
[Definition: Lifetime of Execution] The Lifetime of Execution is the duration during which a ·Command Execution UUID· is valid. The Lifetime of Execution is always a relative duration with respect to the point in time the ·SiLA Server· initiated the response to the ·SiLA Client· (the point in time when the SiLA Server returns the ·Command Execution UUID· to the ·SiLA Client·). It is the responsibility of the ·SiLA Client· to account for potential transmission delays between ·SiLA Server· and ·SiLA Client·.
[Definition: Command Execution Info] The Command Execution Info provides information about the current status of a ·Command· being executed. It consists of the ·Command Execution Status·, and optionally the ·Progress Info· and an ·Estimated Remaining Time·. In addition, an updated

·Lifetime of Execution· MUST be provided, if a ·Lifetime of Execution· has been provided at ·Command· initiation.

[Definition: Command Execution Status] The Command Execution Status provides details about the execution status of a ·Command·. It is either, and in this sequence, first “Command Waiting”, second “Command Running” and third “Command Finished Successfully” or “Command Finished With Error”. The Command Execution Status cannot be reverted back to a previous state that has already been left. That is, once the ·Command· is running, the state cannot go back to waiting etc.

[Definition: Progress Info] Progress Info is the estimated progress of a ·Command· execution, in percent (0...100%).

[Definition: Estimated Remaining Time] Estimated Remaining Time is the estimated remaining execution time of a ·Command·.

[Definition: Property] A Property describes certain aspects of a ·SiLA Server· that do not require an action on the ·SiLA Server·. Properties may be changed by actions of the SiLA server. Properties are always read-only.

[Definition: Unobservable Property] An Unobservable Property is a ·Property· that can be read at any time but no subscription mechanism is provided to observe its changes.

[Definition: Observable Property] An Observable Property is a ·Property· that can be read at any time and that offers a subscription mechanism to observe any change of its value.

[Definition: Property Identifier] A Property Identifier is the ·Identifier· of a ·Property·. A Property Identifier MUST be unique within the scope of a ·Feature·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.

[Definition: Property Display Name] A Property Display Name is the ·Display Name· of a ·Property·.

[Definition: Property Description] A Property Description is the ·Description· of a ·Property·.

[Definition: Property Observable Setting] The Property Observable Setting specifies whether a ·Property· is an ·Observable Property· or an ·Unobservable Property·. The value MUST be either “Yes” (·Observable Property·) or “No” (·Unobservable Property·).

[Definition: Property Data Type] A Property Data Type is the ·SiLA Data Type· of a ·Property·.

[Definition: Property Subscription] A Property Subscription is the subscription of an ·Observable Property· by a ·SiLA Client·.

[Definition: SiLA Client Metadata] SiLA Client Metadata is information that a ·SiLA Server· expects to receive from a ·SiLA Client· when executing a ·Command· or reading or subscribing to a ·Property·. If expected SiLA Client Metadata is not received, a Invalid Metadata Framework Error must be issued. This must be checked before parameter validation. Each SiLA Client Metadata has a specific ·Metadata Identifier· and a ·SiLA Data Type·. Metadata is intended for small pieces of data, and transmission might fail for values larger than 1 KB.

[Definition: Metadata Identifier] A Metadata Identifier is the ·Identifier· of a ·SiLA Client Metadata·. A Metadata Identifier MUST be unique within the scope of a ·Feature·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.

[Definition: Metadata Display Name] A Metadata Display Name is the ·Display Name· of a ·SiLA Client Metadata·.

[Definition: Metadata Description] The Metadata Description is the ·Description· of a ·SiLA Client Metadata·.

[Definition: Metadata Data Type] A Metadata Data Type is the ·SiLA Data Type· of a ·SiLA Client Metadata·.

[Definition: Defined Execution Error Identifier] The Defined Execution Error Identifier is the ·Identifier· of a ·Defined Execution Error·. A Defined Execution Error Identifier MUST be unique within the scope of a ·Feature·. Uniqueness MUST be checked without taking lower and upper

case into account, see Uniqueness of Identifiers.
[Definition: Defined Execution Error Display Name] The Defined Execution Error Display Name is the ·Display Name· of a ·Defined Execution Error·.
[Definition: Defined Execution Error Description] The Defined Execution Error Description is the ·Description· of a ·Defined Execution Error·.
[Definition: Custom Data Type] A Custom Data Type allows to assign a custom ·Fully Qualified Custom Data Type Identifier·, ·Custom Data Type Display Name· and ·Custom Data Type Description· to a ·SiLA Data Type·. ·SiLA Data Types· that have been defined in this way can be used as ·SiLA Data Types· of ·Parameters·, ·Command Responses·, ·Intermediate Command Responses·, ·Properties· or ·SiLA Client Metadata· like any other ·SiLA Data Type·.
[Definition: Custom Data Type Identifier] A Custom Data Type Identifier is the ·Identifier· of a ·SiLA Data Type·. A Custom Data Type Identifier MUST be unique within the scope of a ·Feature·. The ·Identifiers· of the ·SiLA Basic Types· are reserved and MUST NOT be used.
[Definition: Custom Data Type Display Name] A Custom Data Type Display Name is the ·Display Name· of a ·SiLA Data Type·.
[Definition: Custom Data Type Description] A Custom Data Type Description Is the ·Description· of a ·SiLA Data Type·.
[Definition: Custom Data Type Data Type] A Custom Data Type Data Type is the ·SiLA Data Type· of a ·Custom Data Type·.
[Definition: SiLA Data Type] A SiLA Data Type describes the data type of any information exchanged between ·SiLA Client· and ·SiLA Server·. A SiLA Data Type MUST either be a ·SiLA Basic Type· or a ·SiLA Derived Type·. A SiLA Data Type is used to describe the content communicated in:
[Definition: SiLA Basic Type] ·SiLA Data Types· are separated into SiLA Basic Types and ·SiLA Derived Types·. The following SiLA Basic Types are predefined by SiLA.
[Definition: SiLA Numeric Type] The ·SiLA Integer Type· and the ·SiLA Real Type· are SiLA Numeric Types.
[Definition: SiLA String Type] The SiLA String Type represents a plain text composed of maximum 2 x 220 UNICODE characters. Use the ·SiLA Binary Type· for larger data.
[Definition: SiLA Integer Type] The SiLA Integer Type represents an integer number within a range from the minimum value of -263 up to the maximum value of 263-1.
[Definition: SiLA Real Type] The SiLA Real Type represents a real number as defined per IEEE 754 double-precision floating-point number. This is a ·SiLA Numeric Type·.
[Definition: SiLA Boolean Type] The SiLA Boolean Type represents a Boolean value. This is a ·SiLA Data Type· representing one of two possible values, usually denoted as true and false.
[Definition: SiLA Binary Type] The SiLA Binary Type represents arbitrary binary data of any size such as images, office files, etc.
[Definition: SiLA Date Type] The SiLA Date Type represents a ISO 8601 date (year [1–9999]3), month [1–12], day [1–31]) in the Gregorian calendar, with an additional timezone (as an offset from UTC). A SiLA Date Type consists of the top-open interval of exactly one day in length, beginning on the beginning moment of each day (in each timezone), i.e. '00:00:00', up to but not including '24:00:00' (which is identical with '00:00:00' of the next day).
[Definition: SiLA Time Type] The SiLA Time Type represents a ISO 8601 time (hours [0–23], minutes [0–59], seconds [0–59], milliseconds [0–999], with an additional timezone [as an offset from UTC]).
[Definition: SiLA Timestamp Type] The SiLA Timestamp Type represents both, ISO 8601 date and time in one (year [1–9999]3), month, day, hours [0–23], minutes [0–59], seconds [0–59], milliseconds [0–999], with an additional timezone [as an offset from UTC]).

[Definition: SiLA Any Type] The SiLA Any Type represents information that can be of any ·SiLA Data Type·, except for a ·Custom Data Type· (i.e. the SiLA Any Type MUST NOT represent information of a ·Custom Data Type·). The value of a SiLA Any Type MUST contain both the information itself and the ·SiLA Data Type·.
[Definition: SiLA Void Type] The SiLA Void Type represents no data. It MUST only be used as a value of the ·SiLA Any Type·.
[Definition: SiLA Derived Type] There are three different SiLA Derived Types defined:
[Definition: SiLA List Type] The SiLA List Type is an ordered list with entries of the same ·SiLA Data Type·.
[Definition: SiLA Structure Type] The SiLA Structure Type is a structure composed of one or more named elements with the same or different ·SiLA Data Types·.
[Definition: Element Identifier] The Element Identifier is the ·Identifier· of this element of the structure. The ·Identifier· MUST be unique within the scope of a given ·SiLA Structure Type·. Uniqueness MUST be checked without taking lower and upper case into account, see Uniqueness of Identifiers.
[Definition: Element Display Name] The Element Display Name is the ·Display Name· of an element of the structure.
[Definition: Element Description] The Element Description is the ·Description· of an element of the structure.
[Definition: Element Data Type] The Element Data Type is the ·SiLA Data Type· of an element of the structure.
[Definition: SiLA Constrained Type] The SiLA Constrained Type is a ·SiLA Data Type· with one or more ·Constraints· that act as a logical AND. The SiLA Constrained Type MUST be based on either a ·SiLA Basic Type· or a ·SiLA List Type·, or a ·SiLA Constrained Type·. The ·Constraints· in the type itself and the type it is based on are to act together as a logical conjunction (AND).
[Definition: Constraint] A Constraint limits the allowed value, size, range, etc. that a ·SiLA Data Type· can assume. A ·SiLA Server· MUST check the all the Constraints and issue a ·Validation Error· if ·Constraints· are violated.
[Definition: Constraint Identifier] A Constraint Identifier is the ·Identifier· of the ·Constraint·. The Constraint Identifier MUST be one of the identifiers defined in the two tables below: Constraints to SiLA Basic Types and Constraints to SiLA List Type.
[Definition: Constraint Value] The Constraint Value is the actual parameterization of the ·Constraint·.
[Definition: Length Constraint] A Length Constraint specifies the exact number of characters allowed. The ·Constraint Value· MUST be an integer number equal or greater than zero (0) up to the maximum value of 263-1.
[Definition: MinimalLength Constraint] A Minimal Length Constraint specifies the minimum number of characters (for a ·SiLA String Type·) or bytes (for a ·SiLA Binary Type·) allowed. The ·Constraint Value· MUST be an integer number equal or greater than zero (0) up to the maximum value of 263-1.
[Definition: Maximal Length Constraint] A Maximal Length Constraint specifies the maximum number of characters (for a ·SiLA String Type·) or bytes (for a ·SiLA Binary Type·) allowed. The ·Constraint Value· MUST be an integer number greater than zero (0) up to the maximum value of 263-1.
[Definition: Set Constraint] A Set Constraint defines a set of acceptable values for a given ·SiLA Basic Type·. The list of acceptable ·Constraint Values· must have the same ·SiLA Data Type· as the ·SiLA Basic Type· that this ·Constraint· applies to.
[Definition: Pattern Constraint] A Pattern Constraint defines the exact sequence of characters that

are acceptable, as specified by a so-called regular expression. The ·Constraint Value· MUST be a XML Schema Regular Expression (Regular Expressions Quick Start).

[Definition: Maximal Exclusive Constraint] A Maximal Exclusive Constraint specifies the upper bounds for ·SiLA Numeric Types· (the value which is constrained MUST be less than this ·Constraint·) and ·SiLA Date Type·, ·SiLA Time Type· and ·SiLA Timestamp Type· (the value which is constrained MUST be before this ·Constraint·). The ·Constraint Value· must be of the same ·SiLA Data Type· as the ·SiLA Basic Type· that this ·Constraint· applies to.

[Definition: Maximal Inclusive Constraint] A Maximal Inclusive Constraint specifies the upper bounds for ·SiLA Numeric Types· (the value which is constrained MUST be less than or equal to this ·Constraint·) and ·SiLA Date Type·, ·SiLA Time Type· and ·SiLA Timestamp Type· (the value which is constrained MUST be before or at this ·Constraint·). The ·Constraint Value· must be of the same ·SiLA Data Type· as the ·SiLA Basic Type· that this ·Constraint· applies to.

[Definition: Minimal Exclusive Constraint] A Minimal Exclusive Constraint specifies the lower bounds for ·SiLA Numeric” Types· (the value which is constrained MUST be greater than this ·Constraint·) and ·SiLA Date Type·, ·SiLA Time Type· and ·SiLA Timestamp Type· (the value which is constrained MUST be after this ·Constraint·). The ·Constraint Value· must be of the same ·SiLA Data Type· as the ·SiLA Basic Type· that this ·Constraint· applies to.

[Definition: Minimal Inclusive Constraint] A Minimal Inclusive Constraint specifies the lower bounds for ·SiLA Numeric Types· (the value which is constrained MUST be greater than or equal to this ·Constraint·) and ·SiLA Date Type·, ·SiLA Time Type· and ·SiLA Timestamp Type· (the value which is constrained MUST be at or after this ·Constraint·). The ·Constraint Value· must be of the same ·SiLA Data Type· as the ·SiLA Basic Type· that this ·Constraint· applies to.

[Definition: Unit Constraint] A Unit Constraint specifies the unit of a physical quantity, see Unit Constraint for a definition of the allowed ·Constraint Values·.

[Definition: Content Type Constraint] A Content Type Constraint specifies the type of content of a binary or textual ·SiLA Data Type· based on a RFC 2045 ContentType, see Content Type Constraint for a definition of the allowed ·Constraint Values·.

[Definition: Fully Qualified Identifier Constraint] A Fully Qualified Identifier Constraint specifies the content of the ·SiLA String Type· to be a ·Fully Qualified Identifier· and indicates the type of the identifier. Note that this is comparable to a ·Pattern Constraint·; that is, the content is not required to actually identify something, it just has to be a semantically correct ·Fully Qualified Identifier·. The ·Constraint Value· MUST be exactly one of this list:

[Definition: Schema Constraint] A Schema Constraint specifies the type of content of a binary or textual ·SiLA Data Type· based on a schema, see Schema Constraint for a definition of the allowed ·Constraint Values·..

[Definition: Allowed Types Constraint] An Allowed Types Constraint defines a list of ·SiLA Data Types· that the ·SiLA Any Type· is allowed to represent. The ·Constraint Value· MUST be a list of ·SiLA Data Types·, but MUST NOT be a ·Custom Data Type· or a ·SiLA Derived Type· containing a ·Custom Data Type·.

[Definition: Unit Label] The Unit Label is the arbitrary label denoting the physical unit that the ·Unit Constraint· defines. The Unit Label MUST be a string of UNICODE characters up to a maximum of 255 characters in length.

[Definition: Conversion Factor] The Conversion Factor specifies the conversion from the unit with a given ·Unit Label· into SI units, according to the definition in chapter Unit Conversion. The Conversion Factor MUST be a real number as defined per IEEE 754 double-precision floating-point number.

[Definition: Conversion Offset] The Conversion Offset specifies the conversion from the unit with a given ·Unit Label· into SI units, according to the definition in chapter Unit Conversion. The

Conversion Offset MUST be a real number as defined per IEEE 754 double-precision floating-point number.
[Definition: SI Base Unit] The SI Base Unit is the combination of SI units that specifies the same base quantity or dimension as indicated by the ·Unit Label·. The following items MUST be provided for each SI unit N, that makes up the SI Base Unit (N is an integer number in the range of 1 – 8).
[Definition: SI Unit Name] The SI Unit Name is a name referring to an SI unit and MUST be either:
[Definition: SI Unit Exponent] The SI Unit Exponent is the exponent to apply to the SI unit corresponding to ·SI Unit Name· and must be an integer number within a range from the minimum value of -263 up to the maximum value of 263-1.
[Definition: Element Count Constraint] An Element Count Constraint specifies the exact number of elements that a list MUST have. The ·Constraint Value· MUST be an integer number equal or greater than zero (0) up to the maximum value of 263-1.
[Definition: Minimal Element Count Constraint] An Minimal Element Count Constraint specifies the exact number of elements that a list MUST have in minimum. The ·Constraint Value· MUST be an integer number equal or greater than zero (0) up to the maximum value of 263-1.
[Definition: Maximal Element Count Constraint] An Maximal Element Count Constraint specifies the exact number of elements that a list MUST have in maximum. The ·Constraint Value· MUST be an integer number equal or greater than zero (0) up to the maximum value of 263-1.
[Definition: Attribute] Attributes are a part of a ·Feature Definition·. Attributes are not necessary for the implementation of the ·Features· themselves, but help to maintain them. The following Attributes exist and are mandatory: ·SiLA 2 Version·, ·Feature Version·, ·Maturity Level·, ·Originator· and ·Category·.
[Definition: SiLA 2 Version] The version of the SiLA 2 Specification that this ·Feature· was developed against. Any ·SiLA Server· or ·SiLA Client· that was developed against a SiLA 2 Version with a ·Major SiLA 2 Version· of “1” or larger MUST be able to interoperate with each other (backwards and forwards compatibility). The SiLA 2 Version is a combination of the ·Major SiLA 2 Version· and the ·Minor SiLA 2 Version·, separated by a dot (.).
[Definition: Major SiLA 2 Version] The Major SiLA 2 Version. MUST be an integer greater or equal than zero (0).
[Definition: Minor SiLA 2 Version] The Minor SiLA 2 Version. MUST be an integer greater or equal than zero (0).
[Definition: Feature Version] Any ·Feature· MUST specify a Feature Version so that different versions of a ·Feature· can be distinguished during its life cycle. A Feature Version consists of a ·Major Feature Version· and a ·Minor Feature Version·. A ·Feature· MUST specify both a ·Major Feature Version· and a ·Minor Feature Version·.
[Definition: Major Feature Version] The Major Feature Version of a ·Feature·. MUST be an integer greater or equal than zero (0).
[Definition: Minor Feature Version] The Minor Feature Version of a ·Feature·. MUST be an integer greater or equal than zero (0).
[Definition: Maturity Level] SiLA 2 defines the following Maturity Levels, in order of increasing maturity:
[Definition: Draft] The ·Maturity Level· Draft means that the ·Feature· is in a development state.
[Definition: Verified] The ·Maturity Level· Verified means that the ·Feature· has been verified as meeting the normative part of the SiLA 2 specification, including the best practices.
[Definition: Normative] The ·Maturity Level· Normative means that the ·Feature· is now considered stable and has been subject to a round of formal balloting. This ·Maturity Level· MUST only be applied to ·Features· with ·Originator· “org.silastandard”. The requirements of a ·Verified· ·Feature· also apply to Normative ·Features·.

[Definition: Originator] The Originator is a text identifying the organization who created and owns a ·Feature·.
[Definition: Category] The Category is mandatory, but can be set to “none”. It MAY be used to group ·Features· and assign them with a logical or semantic category or category and sub-categorie(s). The main purpose for Categories is to group ·Features· into domains of application.
[Definition: Feature Definition Language] The Feature Definition Language is a way to store ·Feature Definitions· programmatically in an XML (text) based, human and machine readable way. The XML schema can be found on GitLab as FeatureDefinition.xsd.
[Definition: SiLA Service Feature] The SiLA Service Feature is the ·Feature· each ·SiLA Server· MUST implement. Each ·SiLA Server· MUST at least implement the SiLA Service Feature with ·Major Feature Version· equals one (1). It is the entry point to a ·SiLA Server· and helps to discover the ·Features· it implements. The SiLA Service Feature specifies ·Commands· and ·Properties· to discover the ·Features· a ·SiLA Server· implements as well as details about the ·SiLA Server·.
[Definition: Connection Configuration Service Feature] The Connection Configuration Service Feature is a ·Feature· that all ·SiLA Servers· conforming to ·SiLA 2 Version· equal to or greater than “1.1” SHALL implement. The Connection Configuration Service Feature specifies ·Commands· and ·Properties· to configure the ·Connection Method· of a ·SiLA Server·.
[Definition: Validation Error] A Validation Error is an error that occurs during the validation of ·Parameters· before executing a ·Command·.
[Definition: Execution Error] An Execution Error is an error which occurs during a ·Command· execution, a ·Property· access or an error that is related to the use of ·SiLA Client Metadata·.
[Definition: Defined Execution Error] A Defined Execution Error is an ·Execution Error· that has been defined by the ·Feature Designer· as part of the ·Feature·. Defined Execution Errors enable the ·SiLA Client· to react more specifically to an ·Execution Error·, as the nature of the error as well as possible recovery procedures are known in better detail.
[Definition: Undefined Execution Error] Any ·Execution Error· which is not a ·Defined Execution Error· is an Undefined Execution Error.
[Definition: Framework Error] A Framework Error is an error which occurs when a ·SiLA Client· accesses a ·SiLA Server· in a way that violates the SiLA 2 specification. The Framework Error MUST include human readable information in the American English language (see Internationalization) about the error and SHOULD provide proposals for how to resolve the error.
[Definition: Command Execution Not Accepted Error] The Command Execution Not Accepted Error is a ·Framework Error· and MUST be issued in case the ·SiLA Server· does not allow the ·Command· execution.
[Definition: Invalid Command Execution UUID Error] The Invalid Command Execution UUID Error is a ·Framework Error· which MUST be issued when a ·SiLA Client· is trying to get or subscribe to ·Command Execution Info·, ·Intermediate Command Response· or ·Command Response· of an ·Observable Command· with an Invalid ·Command Execution UUID·.
[Definition: Command Execution Not Finished Error] The Command Execution Not Finished Error is a ·Framework Error· and MUST be issued when a ·SiLA Client· is trying to get the ·Command Response· of an ·Observable Command· when the ·Command· has not been finished yet.
[Definition: Invalid Metadata Error] The Invalid Metadata Error is a ·Framework Error· and MUST be issued by a ·SiLA Server· if a required ·SiLA Client Metadata· has not been sent to the ·SiLA Server· or if the sent ·SiLA Client Metadata· has the wrong ·SiLA Data Type· (e.g. not the one that was specified in the ·Feature Definition· for the respective ·SiLA Client Metadata·).
[Definition: No Metadata Allowed Error] The No Metadata Allowed Error is a ·Framework Error· and

MUST be issued when a ·SiLA Server· receives a call of the ·SiLA Service Feature· that contains ·SiLA Client Metadata·.
[Definition: Connection Error] SiLA 2 treats any error with the ·Connection· between a ·SiLA Client· and a ·SiLA Server· as a Connection Error. In contrast to the other error types, Connection Errors are not issued by the ·SiLA Server· nor the ·SiLA Client·, but by the underlying infrastructure (such as the communication network, the operating system, etc.).
[Definition: SiLA Server Discovery] SiLA Server Discovery is a set of mechanisms that can be used by a ·SiLA Server· to advertise itself (i.e. its ·Address·) in the communication network to a ·SiLA Client·. Using the SiLA Server Discovery mechanism a ·SiLA Client· is able to discover the ·Address· of a ·SiLA Server· in the communication network. The goal of ·SiLA Server Discovery· is to enable small, ad-hoc automation setups in labs.
[Definition: SiLA Feature Discovery] SiLA Feature Discovery allows a ·SiLA Client· to discover the ·Features· of a ·SiLA Server·. A ·SiLA Server· MUST enable any ·SiLA Client· to discover available ·Features· of a ·SiLA Server· through the ·SiLA Service Feature·, therefore a ·SiLA Server· MUST always implement the ·SiLA Service Feature·.
[Definition: Fully Qualified Identifier] Each ·Feature· and its components (·Commands·, ·Properties·, ·SiLA Data Types·, etc) SHALL be identifiable by a Fully Qualified Identifier. A Fully Qualified Identifier is guaranteed to be a universally unique identifier, see also Uniqueness of Fully Qualified Identifiers. A Fully Qualified Identifier MUST be a string of UNICODE characters up to a maximum of 2048 characters in length.
[Definition: Fully Qualified Feature Identifier] A Fully Qualified Feature Identifier is a name that uniquely identifies a ·Feature· among all potentially existing ·Features·. It is a combination of the ·Originator·, ·Category·, ·Feature Version· and ·Feature Identifier· in the form of
Fully Qualified Feature Identifier != ·Originator· + “/” + ·Category·+ “/” + ·Feature Identifier· + “/” + “v” + ·Major Feature Version·.
[Definition: Fully Qualified Command Identifier] A Fully Qualified Command Identifier is a name that uniquely identifies a ·Command· among all potentially existing ·Commands·.
Fully Qualified Command Identifier != ·Fully Qualified Feature Identifier· + “/” + ”Command” + “/” ·Command Identifier·.
[Definition: Fully Qualified Command Parameter Identifier] A Fully Qualified Command Parameter Identifier is a name that uniquely identifies a ·Command Parameter· among all potentially existing ·Command Parameters·.
Fully Qualified Command Parameter Identifier != ·Fully Qualified Command Identifier· + “/” + ”Parameter” + “/” ·Command Parameter Identifier·.
[Definition: Fully Qualified Command Response Identifier] A Fully Qualified Command Response Identifier is a name that uniquely identifies a ·Command Response· among all potentially existing ·Command Responses·.
Fully Qualified Command Response Identifier != ·Fully Qualified Command Identifier· + “/” + ”Response” + “/” ·Command Response Identifier·.
[Definition: Fully Qualified Intermediate Command Response Identifier] A Fully Qualified Intermediate Command Response Identifier is a name that uniquely identifies a ·Intermediate Command Response· among all potentially existing ·Intermediate Command Responses·.
Fully Qualified Intermediate Command Response Identifier != ·Fully Qualified Command Identifier· + “/” + ”IntermediateResponse” + “/” ·Intermediate Command Response Identifier·.
[Definition: Fully Qualified Defined Execution Error Identifier] A Fully Qualified Defined Execution Error Identifier is a name that uniquely identifies a ·Defined Execution Error· among all potentially existing ·Defined Execution Errors·.
Fully Qualified Defined Execution Error Identifier != ·Fully Qualified Feature Identifier· + “/” +

”DefinedExecutionError” + “/” ·Defined Execution Error Identifier·.

[Definition: Fully Qualified Property Identifier] A Fully Qualified Property Identifier is a name that uniquely identifies a ·Property· among all potentially existing ·Properties·.

Fully Qualified Property Identifier != ·Fully Qualified Feature Identifier· + “/” + ”Property” + “/” ·Property Identifier·.

[Definition: Fully Qualified Custom Data Type Identifier] A Fully Qualified Custom Data Type Identifier is a name that uniquely identifies a ·Custom Data Type· among all potentially existing ·Custom Data Types·.

Fully Qualified Custom Data Type Identifier != ·Fully Qualified Feature Identifier· + “/” + ”DataType” + “/” ·Custom Data Type Identifier·.

[Definition: Fully Qualified Metadata Identifier] A Fully Qualified Metadata Identifier is a name that uniquely identifies a ·SiLA Client Metadata· among all potentially existing ·SiLA Client Metadata·.

Fully Qualified Metadata Identifier != ·Fully Qualified Feature Identifier· + “/” + ”Metadata” + “/” ·Metadata Identifier·.

Example: org.silastandard/core/AuthorizationService/v1/Metadata/AccessToken

[Definition: Identifier] An Identifier is a name that serves as explicit identifier for different components in SiLA 2. For example, each ·Feature· and its components (e.g. ·Commands·, ·Command Parameters·, etc.) MUST be identifiable by an Identifier. An Identifier MUST be a string of UNICODE characters, start with an upper-case letter (A-Z) and MAY be continued by lower and upper-case letters (A-Z and a-z) and digits (0-9) up to a maximum of 255 characters in length.

[Definition: Display Name] Each ·Feature· and many of its components (e.g. ·Commands·, ·Command Parameters·, etc.) MUST have a human readable Display Name. This is the name that will be visible to the user. A Display Name MUST be a string of UNICODE characters of maximum 255 characters in length. The Display Name MUST be human readable text in American English (see also Internationalization).

[Definition: Description] A Description is a human readable text that describes the behavior and provides additional information about the described element (e.g. ·Feature·, ·Command·, etc.) with details. A Description MUST be a string of UNICODE characters of any number of characters. The Description MUST be human readable text in American English (see also Internationalization).

[Definition: Parameter] Parameters are used to parameterize specific SiLA components (e.g. ·Commands·). Each Parameter MUST have a ·SiLA Data Type· assigned.

[Definition: UUID] A UUID is a Universally Unique IDentifier according to RFC 4122. SiLA always uses the UUID in its string representation (e.g. “f81d4fae-7dec-11d0-a765-00a0c91e6bf6”), as specified by the formal definition of the UUID string representation in RFC 4122. It is RECOMMENDED to always use lower case letters (a-f). In any case, comparisons of UUIDs in their string representation must always be performed ignoring lower and upper case, i.e. “a” = “A”, “b” = “B”, … , “f” = “F”.

[Definition: Defined Execution Error Parameter] A Defined Execution Error Parameter is a ·Parameter· of a ·Defined Execution Error·. It MAY be submitted with the ·Defined Execution Error·.

[Definition: Defined Execution Error Parameter Identifier] A Defined Execution Error Parameter Identifier is the ·Identifier· of a ·Defined Execution Error Parameter·. A Defined Execution Error Parameter Identifier MUST be unique within the scope of a ·Defined Execution Error·.

[Definition: Defined Execution Error Parameter Display Name] A Defined Execution Error Parameter Display Name is the ·Display Name· of a ·Defined Execution Error Parameter·.

[Definition: Defined Execution Error Parameter Description] A Defined Execution Error Parameter Description is the ·Description· of a ·Defined Execution Error Parameter·.

[Definition: Observable Property Filter] A ·Feature Designer· SHALL be able to define Observable

Property Filters for ·Property Subscriptions·.
[Definition: Observable Property Filter Identifier] An Observable Property Filter Identifier is the ·Identifier· of an ·Observable Property Filter·. An Observable Property Filter Identifier MUST be unique for a specific ·Property·.
[Definition: Observable Property Filter Display Name] An Observable Property Filter Display Name is the ·Display Name· of an ·Observable Property Filter·.
[Definition: Observable Property Filter Description] A Filter Description is the ·Description· of an ·Observable Property Filter·
[Definition: Optional Response] A ·Feature Designer· SHALL be able to define Command Responses as optional, indicating that there MAY be a response.
[Definition: SiLA Quantity Type] This is the same as ·SiLA Real Type·, however, a ·Unit Label· and the conversion information that is required to properly convert from the specified ·Unit Label· to SI units MUST be added when sending SiLA Quantity Types between ·SiLA Server· and ·SiLA Client·.
[Definition: Dimension Label] Every “Dimension” ·Constraint· MUST be comprised of a Dimension Label based on its different dimension components. Each Dimension Label contains “Conversion Information” describing how the dimension is derived from base dimensions by providing the exponent for each base dimension involved according to the following formula:
[Definition: SiLA Client UUID] A SiLA Client UUID is a ·UUID· of a ·SiLA Client·. The SiLA Client UUID MUST be generated once and always remain the same.

Normative References

[COMPLETE; as of 0.1, updates: 1.1]

01. [RFC3927] Dynamic Configuration of IPv4 Link-Local Addresses
02. [RFC2131] Dynamic Host Configuration Protocol
03. [RFC6762] Multicast DNS
04. [RFC6763] DNS-Based Service Discovery
05. [RFC2136] Dynamic Updates in the Domain Name System (DNS UPDATE)
06. [RFC2132] DHCP Options and BOOTP Vendor Extensions
07. [RFC1157] A Simple Network Management Protocol (SNMP)
08. [RFC5785] Defining Well-Known Uniform Resource Identifiers (URIs)
09. [IETF BCP 47 language tag] Tags for Identifying Languages
10. [RFC4122] A Universally Unique Identifier (UUID) URN Namespace
11. [RFC3986] Uniform Resource Identifier (URI): Generic Syntax
12. [RFC1035] DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
13. http://www.zeroconf.org/Rendezvous/txtrecords.html DNS-SD (Rendezvous) TXT record format
14. https://oidref.com/2.25 Global OID reference database

= = = END OF NORMATIVE PART = = =