SISO-STD-017-2022

Standard for Web Live, Virtual, Constructive Protocol

Version 1.0

12 December 2022

Prepared by:
Web Live, Virtual, Constructive Product Development Group

7901 4th St N
STE 300-4043
St. Petersburg, FL 33702, USA

SISO Inc. Board of Directors

7901 4th St N
STE 300-4043
St. Petersburg, FL 33702, USA

Revision History

Version	Section	Date (MM/DD/YYY)	Description
Initial Version	All		
0.4		09/02/2015	Added subscription and filtering.
0.5		07/22/2016	Changed subscription and filtering.
0.6		08/22/2017	Updated subscriptions, added object life cycle.
0.7		09/17/2018	Update subscription wildcard, object life cycle.
0.8		01/24/2020	Prepare for balloting.
1.0		06/07/2022	Final draft with BRG adjustments for SAC approval
1.0		12/15/2022	SAC and EXCOM Approval

Participants

At the time this product was submitted to the Standards Activity Committee (SAC) for approval, the WebLVC Product Development Group had the following membership and was assigned the following SAC Technical Area Director:

Product Development Group

Rob Kewley (Chair)
Keith Snively (Secretary)

Keith Snively (SAC Technical Area Director)

Tom van den Berg
Anthony Cramp
Jordan Dauble
Bradford Dillman
Len Granowetter
Jean-Louis Igarza
Stephen Jones
Patrice Le Leydour
Lance Marrou
Regis Mauget
Diana Pineda
Laurent Prignac
David Ronnfeldt
Keith Snively
John Stevens
Michael Tillett
Doug Wood

The Product Development Group would like to especially acknowledge those individuals that significantly contributed to the preparation of this product as follows:

PDG Drafting Group

Rob Kewley (Editor)

Tom van den Berg
Anthony Cramp
Jordan Dauble
Bradford Dillman
Patrice Le Leydour
Keith Snively

The following individuals comprised the ballot group for this product.

Ballot Group

Curtis Blais
Anthony Cramp
Jordan Dauble
Patrice Le Leydour
Robert Kewley
Jonathan Searle
Keith Snively
Jose Ruiz
Simon Skinner
Stéphane Ugolini
Tom van den Berg

When the Standards Activity Committee approved this product on [DD Month YYYY], it had the following membership:

Standards Activity Committee

Grant Bailey (Chair)
Curtis Blais (Vice Chair)
Katherine Ruben (Secretary)

Peggy Gravitz
John Hughes
Patrice Le Leydour
Simon Skinner
Clyde Smithson
Keith Snively
Fuzzy Wells
Michael Woodman

When the Executive Committee approved this product on [DD Month YYYY], it had the following membership:

Executive Committee

Kenneth Konwin (Chair)
Mark McCall (Vice Chair)
Mark McCall (Secretary)

Jeff Abbott
Grant Bailey (SAC Chair)
Damon Curry (CC Chair)
Paul Gustavson
Lana McGlynn
Christopher Metevier
Katherine Morse
Michael O'Connor

Introduction

Web Live, Virtual, Constructive (WebLVC) is an interoperability protocol that enables client applications (to include JavaScript applications running in a web browser) to interoperate as modeling and simulation (M&S) distributed simulations. WebLVC Client applications communicate with other simulators through a WebLVC Server, a web server that communicates with clients using a network protocol, such as Hypertext Transfer Protocol (HTTP) standard. The WebLVC Server may then participate in other distributed simulations on behalf of its clients; this permits interoperation between its WebLVC Clients and external simulators. The WebLVC Protocol defines a standard way of passing simulation data between a client application and a WebLVC Server - independent of the protocols used in other distributed simulations. Thus, a WebLVC Client can participate in a DIS exercise, an HLA federation, a TENA execution, or another distributed simulation environment.

Table of Contents

List of Figures

List of Tables

List of Examples

1 Overview

1.1 Scope

The SISO Standard for Web Live, Virtual, Constructive (WebLVC) Protocol describes an interoperability protocol that allows simulation applications and other external applications to communicate with each other via a WebLVC Server and a standard object model.

1.2 Purpose

The WebLVC Protocol facilitates communication between simulation applications and other external applications through the implementation of common web standards by the WebLVC Server. It enables greater re-use and interoperability because the developers of external applications do not have to learn the intricacies of communicating via simulation standards. They can use the web standards with which they are already familiar, and implementations of these web standards are available in common programming languages.

1.3 Objectives

The WebLVC Protocol specifies a standard way of encoding object update messages, interaction messages, and administrative messages as JSON (JavaScript Object Notation) objects, which are passed between client and server, as well as a basic model of objects and interactions. The WebLVC Standard Object Model describes a basic set of simulation data which can be extended.

1.4 Intended Audience

This document is intended to be used by individuals or groups developing web applications which interact with simulations.

2 References

2.1 SISO Documents

The following SISO documents were used in generating the policies and procedures defined herein. When the following documents are superseded by an approved revision and that causes a conflict with this document, the revision of the below-referenced documents shall supersede this document. These documents are available by through the SISO web site at https://www.sisostds.org/.

Document Number	Title
SISO-REF-010-2021	Reference for Enumerations for Simulation Interoperability, Version 29
IEEE 1278.1™-2012	IEEE Standard for Distributed Interactive Simulation – Application Protocols
IEEE 1516™-2010	IEEE Standard for Modelling and Simulation High Level Architecture
SISO-STD-001.1-2015	Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0
SISO-STD-008-2010	Standard for Military Scenario Definition Language

2.2 Other Documents

Document Number	Title
ECMA-404	The JSON Data Interchange Format
ISO/IEC 21778:2017	Information Technology – The JSON Data Interchange Syntax
http://www.opengeospatial.org/	Open Geospatial Consortium (OGC)
IETF RFC 4648	The Base16, Base32, Base64 Data Encodings
OGC URN Policy	Naming policy for OGC URN's
POXIX.1-2017, Section 9.4	The Open Group Technical Standard Base Specifications, Issue 7, Section 9.4, Extended Regular Expressions
EPSG Dataset	EPSG Geodetic Parameter Dataset
RFC 7946, ISSN: 2070-1721	The GeoJSON Format

3 Definitions, Acronyms, and Abbreviations

English words are used in accordance with their definitions in the most recent edition of Merriam-Webster's Collegiate Dictionary except when special SISO Product-related technical terms are required.

3.1 Definitions

Term	Definition
WebLVC Client	A WebLVC Client is an executing software entity such as a process which initiates a connection to a WebLVC Server.
WebLVC Server	A WebLVC Server informs WebLVC Clients about simulation objects and events.

3.2 Acronyms and Abbreviations

Acronym/Abbr	Definition
ASCII	American Standard Code for Information Exchange
CRS	Coordinate Reference System
dB	Decibel
DIS	Distributed Interactive Simulation
DRM	Dead Reckoning Model
ECEF	Earth-Centered, Earth-Fixed
FOM	Federation Object Model
HLA	High Level Architecture
JSON	JavaScript Object Notation
LROM	Logical Range Object Model
M&S	Modeling and Simulation
MSDL	Military Scenario Definition Language
OGC	Open Geospatial Consortium
OMT	Object Model Template
PDG	Product Development Group
PDU	Protocol Data Unit
REGEX	Regular Expression
RPR FOM	Real-time Platform Reference Federation Object Model
RTI	Runtime Infrastructure
SISO	Simulation Interoperability Standards Organization
TCP	Transmission Control Protocol
TDL	Tactical Data Link
TENA	Test and Training Enabling Architecture
UDP	User Datagram Protocol
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTF	Unicode Transformation Format
WebLVC	Web Live, Virtual, Constructive
WGS	World Geodetic System

4 General Overview

WebLVC is an interoperability protocol that connects web-based applications (typically JavaScript applications running in a web browser) to interoperate as Modeling and Simulation (M&S) distributed simulations. WebLVC Client applications communicate with other simulators through a WebLVC Server. A WebLVC Server may represent a stand-alone distributed simulation or may participate in other distributed simulations on behalf of its clients to permit interoperation between its WebLVC Clients and external simulators. The WebLVC Protocol defines a standard way of passing simulation data between a web-based client application and a WebLVC Server - independent of the protocols used in other distributed simulations. Thus, a WebLVC Client can participate in a Distributed Interactive Simulation (DIS) exercise, a High Level Architecture (HLA) federation, a Test and Training Enabling Architecture (TENA) execution, or other distributed simulation environment.

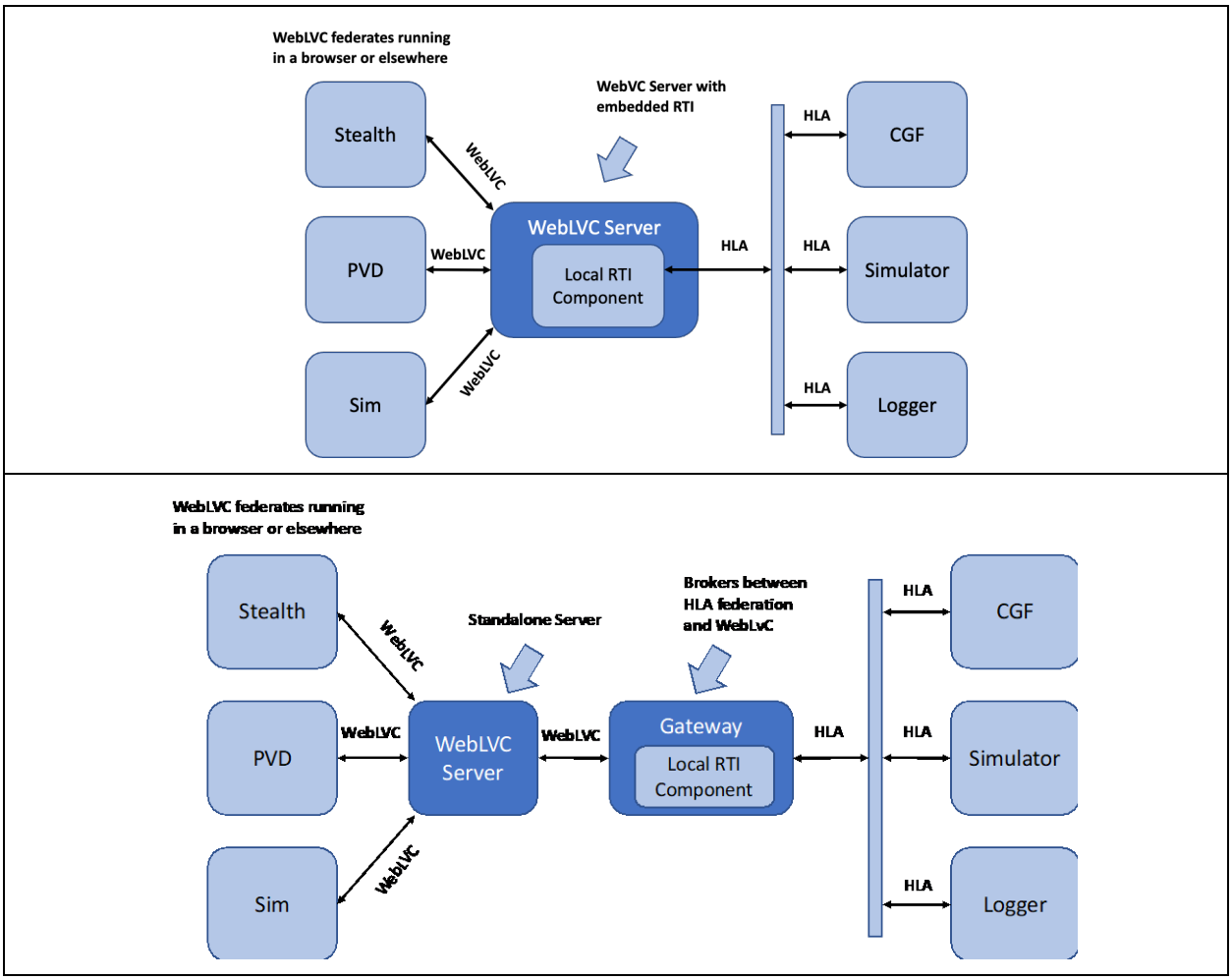


Figure 1 – Example WebLVC configurations for interacting with an HLA federation

The WebLVC Protocol specifies a standard way of encoding object update messages, interaction messages, and administrative messages as JSON (JavaScript Object Notation) objects, which are passed between client and server – typically using WebSockets. WebLVC is flexible enough to support representation of arbitrary types of objects and interactions (i.e., arbitrary Object Models). However, WebLVC does include a "Standard Object Model" definition based on the semantics of the DIS protocol and the HLA Real-Time Platform Reference Federation Object Model (RPR FOM). Users can extend the Standard Object Model by adding new types of objects, attributes, interactions, and parameters; or can choose to represent the semantics of entirely different Object Models (e.g., other HLA Federation Object Models (FOMs), TENA Logical Range Object Models (LROMs), DIS, etc.).

Figure 1 shows two example configurations with a WebLVC Server connected to an HLA federation. In the first case, the WebLVC Server translates directly to HLA via an internal local runtime infrastructure (RTI) connection. In the second example, the WebLVC server is independent of the HLA protocol. Instead, a gateway connects as a WebLVC client and performs the translation. WebLVC is not specific to HLA, and conceptually could be connected to any distributed protocol such as DIS, HLA, TENA, etc.

4.1 WebLVC Use of JSON

A JSON object is a collection of *unordered* name/value pairs, expressed in text, using the syntax of JavaScript. The JSON syntax is described in both ECMA-404 and ISO/IEC 21778:2017. For example, a JSON object representing a can of soda might look like this:

Example 1 – JavaScript Object Notation (JSON)

```
{
    "ContainerType": "can",
    "SizeInLiters": 0.33,
    "Brand": "Coca Cola",
    "Flavor": "Cherry"
}
```

JSON is the natural way of encoding structured data that is intended to be used by web-based JavaScript clients, because a JSON object *is* a string-literal representation of a JavaScript object. The JavaScript language has built-in `JSON.stringify` and `JSON.parse` functions which generate JSON strings from JavaScript objects, and vice versa, without the need for data marshaling or conversion.

Upon receiving the soda can object described above, a JavaScript client can parse it and then directly query the object for its attribute values:

Example 2 – JavaScript use of JSON object

```
var sodaCan = JSON.parse(jsonData);
var numLiters = sodaCan.SizeInLiters;
```

4.2 WebLVC example

The WebLVC specification defines a standard mechanism for representing simulation interoperability messages as JSON objects, by specifying property names, data types, enumeration values and conventions. For example, in order to represent a valid WebLVC attribute object message, a JSON object includes a property called `MessageKind`, a property called `ObjectType`, and a variety of other properties defined by the Object Model being used and the Object Type of the object being updated. In general the Standard Object Model uses attributes names, types, and enumerated values used by DIS, the RPR FOM, and the SISO Enumerations Document.

For example, a web-based flight simulator using the WebLVC attribute update message with the Standard Object Model might convey the current state of the aircraft it is simulating by sending the following WebLVC Message:

Example 3 – WebLVC Message structure

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectName": "F-16 Alpha",
    "ObjectType": "WebLVC:PhysicalEntity",
    "Object": {
        "EntityIdentifier": [1, 2, 1],
        "EntityType": [1, 2, 225, 1, 3, 0, 0],
        "Coordinates": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663],
            "VelocityVector": [57.04, 32.77, 89.263],
            "Orientation": [-1.65, 2.234, -0.771]
        },
        "Marking": "F-16",
        "DamageState": 0
    }
}
```

A `MessageKind` of `AttributeUpdate` indicates that this message is conveying updated object attribute values. "F-16 Alpha" is the name of the WebLVC Object being updated, and the object is of type `"WebLVC:PhysicalEntity"` – one of the types in the Standard Object Model. Note that the various attribute names match those in the RPR FOM, that the DIS/RPR FOM-style `EntityIdentifier` and `EntityType` structures are encoded as arrays of enumerated values, and that the same units and coordinate systems as DIS/RPR FOM may be used for `WorldLocation`, `VelocityVector`, and `Orientation`. Note also that any given WebLVC `AttributeUpdate` message may contain values for only a subset of the Object's properties (e.g., value that have changed since last update).

5 Basic Protocol Specification

This section defines the basic, object-model-independent elements of the WebLVC Protocol. It includes specifications for the following:

- How message errors are processed
- Header information common to all WebLVC Messages
- Administrative messages, such as `Connect` and `LogRequest` messages, which do not depend on the object model being used.
- Common object model properties, such as `ObjectName` and `TimeStamp` may be included in Attribute Update and Interaction messages regardless of what kinds of Objects and Interactions they are describing.

5.1 Conceptual model

The conceptual model for WebLVC communications is shown in below.

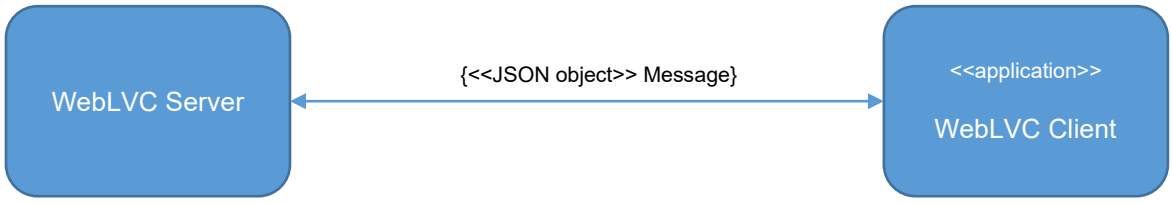


Figure 2 – WebLVC Conceptual Model

Although the WebSocket is a typical means of communication for the above model, the WebLVC specification does not dictate a specific transport mechanism. WebLVC was primarily designed around a use case in which clients are JavaScript applications running in a browser, and for this use case, WebSockets are the obvious choice. However, there may be cases where users plan to use the WebLVC Protocol in native (non-browser-based) applications. In these cases, it may be more natural to pass messages through User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) sockets, or other alternatives.

In WebLVC, each client connection is associated with a single simulation exercise (like a single DIS exercise or HLA federation execution). How WebLVC connections correspond to exercises is not specified by this standard. For example, different message transport endpoints (e.g., different WebSocket Uniform Resource Locators (URLs) or ports) could identify different exercise contexts.

Simulation information is shared either as objects or interactions, which are distributed to all clients connected to a common exercise (depending on client configuration, e.g., subscription). Object and interaction data are represented as named data fields called attributes, conceptually similar to the named properties in a JSON object. While the names of attributes are object-model-dependent, there are several common attributes such as `ObjectName` and `TimeStamp` whose definitions are independent of the type of object or interaction.

An interaction is a simple message, typically representing a simulation event at a point in simulated time.

An object represents simulation data which has a life cycle; objects are created, updated and deleted. Objects have a unique identity and name in each context. Object attribute values are communicated by `AttributeUpdate` messages.

Attribute update messages may omit attributes sent in earlier attribute update messages; omitted attributes are assumed to retain the same value as the last time that attribute was updated. For example, consider a hypothetical `LicensePlate` attribute; if the value hasn't changed since the last attribute update containing `LicensePlate` then that attribute could be omitted. A common pattern for some attributes is to appear only in the initial attribute update, and never to appear in subsequent updates.

Objects have a state for each point in simulated time, defined by applying all attribute updates received by the server up to that simulated time, and other behavior (e.g., server dead reckoning).

5.2 Header information

Header information, such as the type of message a JSON object represents, is encoded just like the rest of the data in a message – through named properties and values.

Required properties:

`MessageKind` – All WebLVC Messages shall include the `MessageKind` property. `MessageKind` is a string that indicates what kind of WebLVC Message an object represents. WebLVC defines the following standard message kinds:

Table 1 – MessageKind descriptions

MessageKind	Description
`Connect`	Request a WebLVC connection.
`ConnectResponse`	Response to Connect.
`Configure`	Configuration request to the server.
`ConfigureResponse`	Response to Configure message.
`AttributeUpdate`	Updates the values of attributes of a WebLVC Object.
`ObjectDeleted`	Notification of an object deletion.
`Interaction`	Notification of interaction with property values.
`SubscribeObject`	Subscribe to `AttributeUpdate` messages.
`UnsubscribeObject`	Cancel object subscription.
`SubscribeInteraction`	Subscribe to Interaction messages.
`UnsubscribeInteraction`	Cancel interaction subscription.
`LogRequest`	Request contents of error log.
`LogResponse`	Response to `LogRequest`.

A client or server may check the value of the `MessageKind` property to determine what kind of message it has received, and therefore determine what additional properties it can expect the message to contain.

5.3 Administrative Messages

WebLVC Clients shall connect to a WebLVC Server by sending a `Connect` message. A successful handshake is required before either the client or server may publish arbitrary message traffic.

A client application shall send a `Connect` message to the server and wait until a `ConnectResponse` message is received before publishing further messages. The `Connect` message describes the requested configuration to the server.

A server shall wait for an acceptable `Connect` message before immediately sending a `ConnectResponse` message, followed by further messages. Upon receipt of a `Connect` message, the server responds with a `ConnectResponse` message, indicating the success or failure of the request.

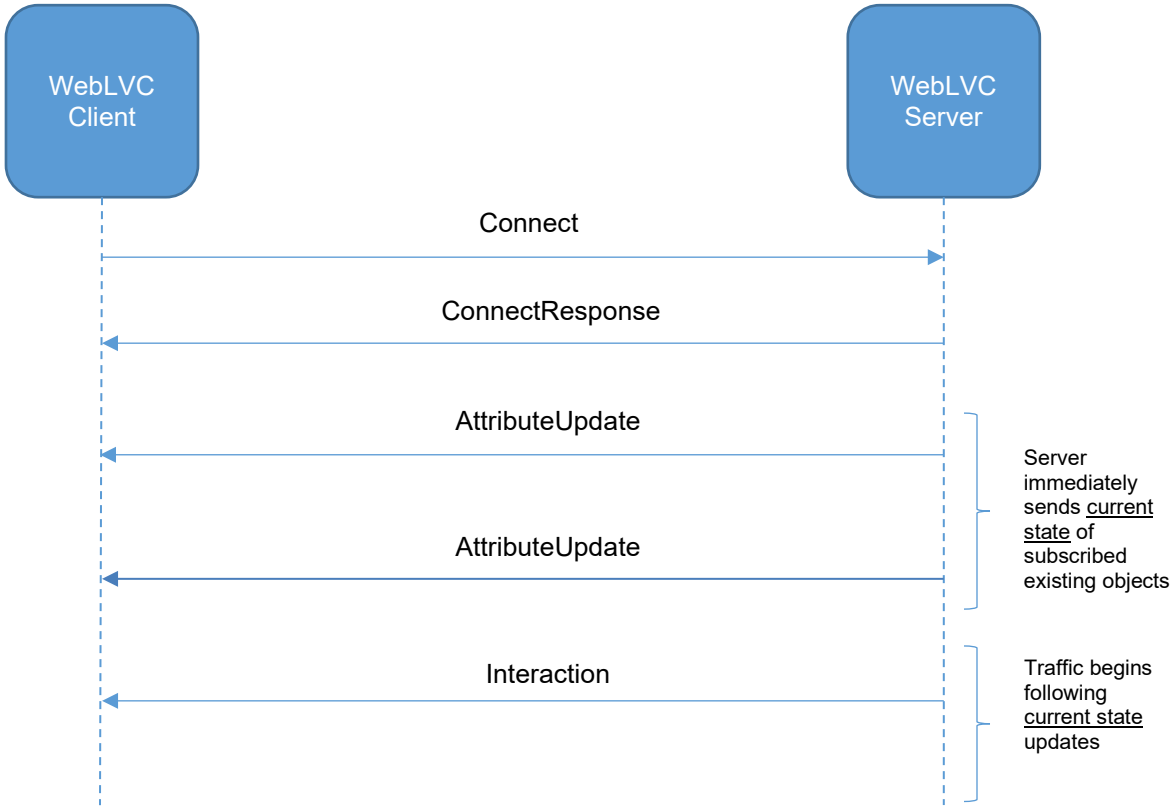


Figure 3 – Successful WebLVC connection handshake

After a `ConnectResponse` indicating success is returned, the server shall immediately send the current state of all objects while respecting subscription, filtering and other configuration supplied by the client embedded in the `Connect` request.

If the server rejects the client connection, the handshake is unsuccessful and a `ConnectResponse` message containing error information is returned, and the configuration of the server remains unchanged.

Figure 4 – Rejected WebLVC connection

5.3.1 Connect Message

A client application shall send a `Connect` message to the server to initiate the connection handshake, which describes the requested configuration for the connection.

Required properties:

`ClientName` – All Connect Messages shall include the `ClientName` property. `ClientName` is a string representing the name of the client.

Optional properties:

`WebLVCVersion` – All `Connect` Messages may include the `WebLVCVersion` property – a number representing the version of the Standard for WebLVC Protocol document to be used for communication between the connecting client and server. If `WebLVCVersion` is omitted, it is assumed that the client is using the default version of the Standard for WebLVC Protocol supported by the server.

`Messages` – an array of messages may be embedded within the `Connect` message. These embedded messages will be processed by the server in order before the `ConnectResponse` is returned. This allows for configuration of the connection (e.g., setting up subscription and filters) *before* any messages are returned by the server. `Connect` and `ConnectResponse` messages are forbidden as embedded messages here, and will cause the containing `Connect` message to be rejected. If any embedded message fails, the `Connect` message is rejected.

The following is an example of a simple `Connect` Message, with no embedded messages. When a WebLVC Server processes this message, it shall assume a subscription to every `ObjectType` and `InteractionType`, and it shall send all associated messages after the `ConnectResponse`. In the second example, Example 5, the server shall only send `AttributeUpdate` and `Interaction` messages of type `Object1` and `Interaction1`.

Example 4 - Connect message

```
{
    "MessageKind": "Connect",
    "ClientName": "MyClient",
    "WebLVCVersion": 1.0
}
```

Example 5 - Connect message with optional embedded messages

```
{
    "MessageKind": "Connect",
    "ClientName": "MyClient",
    "WebLVCVersion": 1.0,
    "Messages": [{
            "MessageKind": "SubscribeObject",
            "ObjectType": "Object1"
        }, {
            "MessageKind": "SubscribeInteraction",
            "InteractionType": "Interaction1"
        }
    ]
}
```

5.3.2 ConnectResponse Message

The server sends this message in response to a `Connect` message to indicate that the client may begin publishing.

Required properties:

`Connected` – status of the connection as a Boolean; true if connected, false if the connection request was rejected.

Optional properties:

`Errors` – an array of `Log` objects, present if connection is rejected. See section 5.7.2 for details.

`WebLVCVersion` – a number indicating the version of the Standard for WebLVC Protocol document which will be used by this connection, present if the connection is successful.

Example 5 - ConnectResponse message

```
{
    "MessageKind": "ConnectResponse",
    "WebLVCVersion": 1.0,
    "Connected": true
}
```

5.3.3 Configure Message

This message is used to set various configuration properties for a connected client. Different clients of the same server may have different configurations. A client may send multiple configure messages to dynamically change the configuration during a single connected session. If the server does not support every configuration specified, the configure operation will fail.

A `Configure` message may be embedded in a `Connect` message so that the configuration takes effect before the server delivers data to the client. This is often desirable, for example, to make sure that the server does not bombard the client with unwanted messages before the client has a chance to indicate its filtering preferences.

Object model definitions such as the Standard Object Model can extend the `Configure` and `ConfigureResponse` messages for further configuration details.

Optional properties:

`TimestampFormat` – specifies the format for the `Timestamp` property of `AttributeUpdate` and `Interaction` messages.

Table 2 - TimestampFormat values

`TimestampFormat`	Timestamp Type	Description	Example
0	string	Default, Hex ASCII string as DIS/RPR FOM.	C0000000
1	number	Seconds since simulation began, double precision.	500.2579
2	number	Milliseconds since January 1, 1970, as integer.	1650748391221

For `TimestampFormat` 0, use a string containing the DIS timestamp field format converted into hexadecimal American Standard Code for Information Exchange (ASCII) character representation as defined by the RPR FOM (See SISO-STD-001.1-2015).

5.3.4 ConfigureResponse message

The `ConfigureResponse` is sent by the server to notify a client whether the requested `Configure` message properties are accepted or rejected. For each property in the `Configure` message, a corresponding property is returned as a Boolean indicating success. Rejected `Configure` properties may be indicated in the Log.

Optional properties:

`TimestampFormat` – a Boolean; true if accepted, false if rejected.

5.4 Object messages

Unlike interactions which are an event at a particular time, WebLVC Objects have a life cycle: they are created, then updated, and eventually they are deleted. There is no message to explicitly create an object, object creation is implied by the first `AttributeUpdate` message for the object, unless suppressed.

5.4.1 Object life cycle

WebLVC Clients create objects when the first `AttributeUpdate` message with a unique `ObjectName` property and a valid `ObjectType` property is sent to the WebLVC Server. Objects are not exclusively owned by clients. Any client may issue an `AttributeUpdate` message for any object at any time. Objects are deleted either explicitly by clients issuing an `ObjectDeleted` message, or by the server.

A WebLVC Server might also create objects which represent external objects in another distributed simulation, such as a DIS exercise. The WebLVC Server does not enforce the HLA concept of object or attribute ownership, so it processes all messages related to an `Object` and its attributes regardless of the source.

5.4.2 AttributeUpdate Message

`AttributeUpdate` messages convey values of the attributes of a WebLVC Object. The rules that dictate when an `AttributeUpdate` shall be sent depend on the object model being used, and the type of object being updated.

Since WebLVC Objects have a current state, properties which haven't changed since the last `AttributeUpdate` message may be omitted and will be assumed to retain the last sent value. However, some properties are required but change rarely or never at all. These will be categorized as "initial properties". Initial properties are typically sent only in the first `AttributeUpdate` message for an object.

Required properties:

`ObjectName` – All `AttributeUpdate` messages shall include an `ObjectName` property. The value of the `ObjectName` property is a string that uniquely identifies the WebLVC Object for which the message is conveying an update. It is the responsibility of the application that is sending updates for an object (which might be either a client or the WebLVC Server) to ensure that it chooses a name that is not already in use by the WebLVC Server. If using the MSDL (Military Scenario Definition Language) standard, the unique designations of units or equipment may be used as object names.

`Object` – a JSON structure for the representation of the object being updated. Note that any JSON structure would be valid with respect to this standard. However, federation participants would have to agree on and document a common set of `ObjectTypes` and corresponding Object structures to ensure compatibility.

Initial properties:

The following property is required for the initial `AttributeUpdate` message for each `Object`. After that, they are optional.

`ObjectType` – a string that indicates the Object Type of the WebLVC Object for which the message is conveying an update (similar to the concept of HLA `ObjectClass` or DIS Protocol Data Unit (PDU) Kind). Based on the value of the `ObjectType` property, an application can determine what other properties it can expect the message to contain. For example, updates for objects of type "`WebLVC:PhysicalEntity`" may contain a `Coordinates` property (an object which contains position, velocity, etc.). The WebLVC Server shall not create any objects without an `ObjectType`. `AttributeUpdate` messages without an `ObjectType` which would otherwise create a new object, shall be rejected by the WebLVC Server and an error recorded in the log. Once `ObjectType` is specified for an object the server shall reject any `AttributeUpdate` messages with different values of `ObjectType` for the same object as an error and record this in the log. Strings are used to represent `ObjectType` (rather than enumerated numbers) because it's easy to understand. Naming conventions may make collisions unlikely among different users' extensions, and avoid the need for a central repository of enumeration-to-string mappings for user extensions. By convention, Object Types defined by the Standard Object Model have an "`WebLVC:`" prefix (e.g. "`WebLVC:PhysicalEntity`"), while user extensions have a prefix that indicates the origin of an object model (e.g., "`MyProject:MyObjectKind`" or "`MyCompany:MyObjectKind`").

Optional Properties:

`Timestamp` – represents the time at which the data is valid. The default format for Timestamp is a hex ASCII string, using the DIS/RPR FOM conventions. Alternate formats can be specified using the `Configure` message. If `Timestamp` is missing, the message is assumed to be valid at the time of receipt.

`Timeout` – the time in seconds after which the object will be automatically deleted by the server. A value of exactly 0 disables the timeout. The default is 0 meaning an object will not timeout and be deleted.

5.4.3 ObjectDeleted Message

An application shall send an `ObjectDeleted` message to inform the recipient that an object it has been updating no longer exists, or no longer matches subscription filters.

Required properties:

`ObjectName` – a string representing the name of the object that has been deleted.

Example 6 - ObjectDeleted message

```
{
    "MessageKind": "ObjectDeleted",
    "ObjectName": "F-16 Alpha"
}
```

Note that there is no corresponding `ObjectCreation` or `ObjectRegistration` message defined in WebLVC. This is because the first `AttributeUpdate` sent for an object can serve to inform the recipient that there exists an object with the specified name (similar to the DIS protocol, where a recipient is informed that a particular entity exists when it receives its first Entity State PDU from a previously-unknown entity).

Optional Properties:

`Timestamp` – represents the time at which the data is valid, using the format specified by the `Configure` message. If `Timestamp` is missing, the message is assumed to be valid at the time of receipt.

`OutOfScope` – true if the object no longer matches and any subscription filters, or is no longer within the configured `WorldBounds` for this client (See section 6.2.1 for details); otherwise false.

5.5 Interaction Messages

Unlike objects, interactions have no lifecycle. They are single messages which typically provide notification of simulation events.

5.5.1 Interaction Message

Interaction messages are used to convey parameters of a simulation event or interaction. While the names of the properties that carry the interactions parameters are object-model-dependent, there are several common properties, such as `TimeStamp`, whose definitions are independent of the type of interaction being conveyed. The rules that dictate when, or how frequently, an Interaction shall be sent depend on the Object Model being used, and the type of interaction in question.

Required properties:

`InteractionType` is a string that indicates the type of the Interaction conveyed by a WebLVC Message, similar to HLA `InteractionClass` or DIS PDU Kind. All Interaction messages shall include the `InteractionType` property. Based on the value of the `InteractionType` property, an application can determine what other properties it can expect the `Interaction` object to contain. For example, interactions of type "`WebLVC:WeaponFire`" will contain an `Interaction` property with a `MunitionType` property within the `Interaction` object (See Section 6.4.1). Strings are used to represent `InteractionType` (rather than enumerated numbers) because it's easy to understand. Strings allow naming conventions that make collisions unlikely among different users' extensions and avoid the need for a central repository of enumeration-to-string mappings for user extensions. By convention, Interaction Types defined by the Standard Object Model have an "`WebLVC:`" prefix (e.g., "`WebLVC:WeaponFire`"), while user extensions have a prefix that indicates the origin of an object model (e.g., "`MyProject:MyInteractionKind`" or "`MyCompany:MyInteractionKind`").

`Interaction` - a JSON structure for the representation of the interaction. Note that any JSON structure would be valid with respect to this standard. However, federation participants would have to agree on and document a common set of `InteractionTypes` and corresponding `Interaction` structures to ensure compatibility.

Optional Properties:

`Timestamp` – represents the time at which the data is valid. The default format for Timestamp is a hex ASCII string, using the DIS/RPR FOM conventions. Alternate formats can be specified using the `Configure` message. If `Timestamp` is missing, the message is assumed to be valid at the time of receipt.

5.6 Subscription

A WebLVC Server provides subscription and filtering services to each client for objects and interactions. The server will only send object and interaction messages which have been subscribed and pass all supplied filters. Publication is implicit; a client simply sends an object or interaction message to the server and the server immediately matches the published information to each client's subscription.

If no subscription messages have been sent by a client, then full subscription to all objects and interaction messages shall be assumed. A client may provide subscription messages as part of the `Connect` message to the WebLVC Server. See section 5.3.1 for more information.

Each subscription message shall apply to either a single `ObjectType` or a single `InteractionType` and overwrites any previous subscription for that `ObjectType` or `InteractionType`. However, if no `ObjectType` or `InteractionType` is specified, it will result in a subscription to every `ObjectType` or `InteractionType`.

When a subscribed object which had matched previous subscription filters no longer matches any subscription filters, an `ObjectDeleted` message will be sent by the server with property `OutOfScope` as true. If the object later changes state in a way that it now matches a client's subscription criteria again, the server will immediately send the current state of the object and resume sending updates to the client (allowing the client to "rediscover" the object as if it were new).

Note: The main purpose of WebLVC subscription is to reduce the amount of traffic sent from a WebLVC Server to a WebLVC Client. Although it is also possible to use subscription services to reduce traffic in the other direction, this is expected to be a much less common use case. Servers are not required to send subscription messages to clients, and clients are not required to provide subscription services to servers. Clients may ignore subscription messages from the server and sending messages to a server for which that server has not subscribed is not an error.

5.6.1 SubscribeObject Message

A `SubscribeObject` message is a request sent by a client to notify the server of its interest in a subset of objects, based on filter properties supplied in the message. A server will respond by restricting the `AttributeUpdate` messages sent to the client to just those that match the filter supplied in the request. If a message passes filtering criteria, then the entire message is sent to the client, and if a message fails to pass filtering criteria, then the entire message is not sent. In other words, filtering out entire messages based on the values of individual properties is supported, but filtering individual properties out of messages (similar to HLA's attribute-level subscription) is not.

Optional properties:

`ObjectType` – the type of object to which to subscribe, specified as a string. An exact match is required. If `ObjectType` is omitted, then this message is applied to all object types, including any object types currently unknown to the server, e.g., the filter would include new object types introduced later, say by a client `AttributeUpdate`.

Filter Properties:

A subscribe message may optionally specify any properties of a filter (See section 5.6.5). Filters are tested against the current state of the object. If any filter specifies a message property which is not supplied in an `AttributeUpdate` message, the server uses the current value for that attribute.

If no filter properties are specified, then all object messages for the `ObjectType` will be sent.

5.6.2 UnsubscribeObject Message

`UnsubscribeObject` messages remove object subscriptions.

Required properties:

`ObjectType` – the type of object to unsubscribe, as a string. An exact match is required.

5.6.3 SubscribeInteraction Message

`SubscribeInteraction` messages specify which WebLVC `Interaction` messages are sent.

Optional properties:

`InteractionType` – the type of interaction to which to subscribe, specified as a string. An exact match is required. If `InteractionType` is omitted, then this message is applied to all interaction types, including any interaction types currently unknown to the server, e.g., the filter would include new interaction types introduced later, say by a client.

Filter Properties:

A subscribe message may optionally specify any properties of a filter (See section 5.6.5). Unlike object subscriptions, only the supplied attributes in an interaction are matched against subscription filters. Interactions do not have a lifecycle and are independent, so any previous values are irrelevant to current interaction messages.

If no filter properties are specified, then all interaction messages for the `InteractionType` will be sent.

5.6.4 UnsubscribeInteraction Message

`UnsubscribeInteraction` messages remove interaction subscriptions.

Required properties:

`InteractionType` – the type of interaction to unsubscribe, as a string. An exact match is required.

5.6.5 Filters

Filters restrict the transmission of interactions, and of messages associated with WebLVC Objects. Filters only determine whether an entire message is received or not; filters do not alter which properties are included in the message.

A filter may contain a `FilterMatch` or a `FilterList` property, but not both. The `FilterMatch` contains the property matches and is the means by which messages can be filtered based upon their property values (See section 5.6.5.1). The `FilterList` is a list of sub-filters, and it allows for more complex matching logic for advanced use cases. The `FilterType` property controls whether `all`, `any`, or `none` of the contained property matches or sub-filters must succeed. All properties in filters are optional. Appendix A contains additional information on composing and parsing `Filter` messages, which are described in this section.

Optional properties:

`FilterType` – a string describing the application of `FilterMatch` or `FilterList`.

For a `FilterMatch`:

- `all` – All the `FilterMatch` properties must match. This is the default if no `FilterType` is specified.
- `any` – At least one property must match.
- `none` – None of the filter match properties must match.

For a `FilterList`:

- `all` – All of the items in the `FilterList` must match. This is the default if no `FilterType` is specified.
- `any` – At least one item in the `FilterList` must match.
- `none` – None of the `FilterList` items must match.

At most one of the following optional properties may appear:

`FilterMatch` – an object that contains property matches for the given interaction type or object type. A property match has the same name as the interaction or object property being matched and its value is a list or match criteria for that property. (See section 5.6.5.1)

`FilterList` – a list of contained filters. `FilterType` applies to the list of objects, not to the properties in those filter objects. Each filter object in `FilterList` may themselves contain further `FilterType` and `FilterList` or `FilterMatch` properties.

5.6.5.1 FilterMatch

A `FilterMatch` specifies matches for named properties, corresponding to property names in the filtered object or interaction. For each property match, the WebLVC Server compares the value associated with the corresponding named property in each object or interaction being considered, with the value or values provided in the property match. Comparison for objects shall use the current state of the object including the `AttributeUpdate` message to which a match is applied. Properties for which no match was provided shall have no impact on whether a message is sent. In addition, if no value has ever been supplied for a property, then filters which reference that property shall be ignored and the remaining matches are applied; if there are no other matches which reference properties with supplied values then the filter is assumed to pass and the message will be sent.

A property match is defined as an array of one or more criteria. A match shall succeed of any of the criteria successfully matches for the property (Note: `FilterType` has no effect on the array of criteria). For non-nested object values, each criterion shall specify one of:

- an exact value to match
- an object defining a minimum/maximum range
- an object defining a regular expression (REGEX) (in case of a String). WebLVC interprets Extended Regular Expressions as defined in POSIX.1-2017.

For properties which have an object value, each criterion shall be:

- an anonymous `Filter` object for the nested object

Exact value match:

If a criterion is an exact value, then a property shall pass the test by exactly matching the value. Note: more than one criterion may appear in a property match.

For example, the following subscription message indicates the client wishes to receive messages only for objects of type `WebLVC:PhysicalEntity` whose marking text exactly matches any of the strings, `"TankA"`, `"TankB"`, or `"TankC"`.

Example 7 - Exact value filter

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Marking": ["TankA", "TankB", "TankC"]
    }
}
```

Range of values

If a criterion is a min/max structure, an property shall pass the test by either matching or falling between the values.

For example, the following subscription message indicates the client wishes to receive messages only for objects of type `WebLVC:PhysicalEntity` whose marking text falls between the string values `"TankA"` and `"TankZ"`:

Example 8 - Range of values filter

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Marking": [{
                "min": "TankA",
                "max": "TankZ"
            }
        ]
    }
}
```

Regex match:

If a criterion is a regular expression structure, then a property shall pass the test by matching the regular expression value provided in the `regex` property. The regular expression value represents an extended regular expression as defined in POSIX.1-2017, Section 9.4. A regular expression is for matching string values only. If the value type is not a string, then the match shall fail.

For example, the following subscription message indicates the client wishes to receive messages only for objects of type **`WebLVC:PhysicalEntity`** whose marking text begins with the string values "Tank" followed by a capital letter, A-Z.

Example 9 – regex filter

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Marking": [{
                "regex": "^Tank[A-Z]"
            }
        ]
    }
}
```

Nested object match:

If a criterion is a nested object, then a property shall pass the test by matching the anonymous `Filter` object against the nested object.

For example, the following subscription message indicates the client wishes to receive messages only for objects of type `WebLVC:PhysicalEntity` where the Coordinates property is an object which has the property `DeadRockingAlgorthim` which matches either 2 or 4:

Example 10 - Nested object match

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Coordinates": [{
                "FilterMatch": {
                    "DeadReckoningAlgorithm": [2, 4]
                }
            }
        ]
    }
}
```

Comparison Rules

The JSON data type of supplied exact values and range extents shall match the JSON data type of the property being subjected to the filter. The JSON data type for property subjected to a regular expression shall be a String. Any type mismatch shall fail to match. A server shall log some error message to indicate data type mismatch problems.

For numbers, values are compared using simple comparison operators.

Strings shall be compared lexicographically when using a range, using the same character set for both the value and the filter. The result of a lexicographic comparison using different character sets is not defined by this specification. Please note: when using WebLVC over websockets, websockets exchange text information using Unicode Transformation Format (UTF) - 8.

Arrays are compared element-by-element, and for each element comparison rules are again applied (data types must match, arrays compared element-by-element). If the filter array is longer and no corresponding value is available for comparison, the comparison fails. If the end of the filter array is reached and all filter elements have matched successfully, then the comparison succeeds, and further values are ignored.

Objects are compared using `"any"`, `"all"` or `"none"` depending on the top level `"FilterType"` property for the current filter, and each property of the nested object is compared against the corresponding named property match.

Additional Filter Examples

Example 11 – Filter using multiple range matches

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Marking": [{
                "min": "TankA",
                "max": "TankZ"
            }, {
                "min": "PlaneA",
                "max": "PlaneD"
            }
        ]
    }
}
```

In this example, there is a single property match, associated with the Marking attribute,that includes multiple criteria. This example matches any value for `Marking` with a lexicographic value between `TankA` and `TankZ`, or between `PlaneA` and `PlaneD`.

Example 12 – Filter using one range and one exact match

```
{
    "MessageKind": "Subscribe",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterMatch": {
        "Marking": ["TankA", {
                "min": "PlaneA",
                "max": "PlaneD"
            }
        ]
    }
}
```

Again, there is a single match with multiple criteria. This matches any value for `Marking` with a lexicographic value between `PlaneA` and `PlaneD`, or an exact match with the value `TankA`.

Example 13 – Filter using a range for arrays

```
{
    "MessageKind": "Subscribe",
    "ObjectType": "WebLVC:AggregateEntity",
    "FilterMatch": {
        "Dimensions": [{
                "min": [0, 0, 0],
                "max": [10, 10, 10]
            }
        ]
    }
}
```

The filter specifies that `Dimensions` shall be an array data type and the first 3 elements in the array shall be numbers. The `min` and `max` determine the comparison operators applied, so a `Dimensions` value of `[a1, a2, a3]` passes if `0 <= a1 <= 10` and `0 <= a2 <= 10` and `0 <= a3 <= 10`.

Example 14 - Filter with two criteria, using array data types

```
{
    "MessageKind": "Subscribe",
    "ObjectType": "WebLVC:AggregateEntity",
    "FilterMatch": {
        "Dimensions": [{
                "min": [0, 0, 0],
                "max": [10, 10, 10]
            }, {
                "min": [20, 20, 20],
                "max": [50, 50, 50]
            }
        ]
    }
}
```

Because the ranges are tested independently this passes either `Dimensions` between `[0,0,0]` and `[10,10,10]` or between `[20,20,20]` and `[50,50,50]`. A value of `[1,2,25]` passes the first two elements of the first range but fails the third, then it fails the first element of the second range, so it is rejected and not sent.

Example 15 - Filter for two different properties using any

```
{
    "MessageKind": "Subscribe",
    "ObjectType": "WebLVC:AggregateEntity",
    "FilterType": "any",
    "FilterMatch": {
        "Marking": ["TankA", {
                "min": "PlaneA",
                "max": "PlaneD"
            }
        ],
        "Dimensions": [{
                "min": [0, 0, 0],
                "max": [10, 10, 10]
            }
        ]
    }
}
```

This example has two property matches – one for the `Marking` property, and one for the `Dimensions` property. In order for a `WebLVC:AggregateEntity` message to pass, either the `Marking` or `Dimensions` criteria must match (in order to pass the `Marking` property match, the message must match at least one of its criteria).

Example 16 - Disable filters

```
{
    "MessageKind": "Subscribe",
    "ObjectType": "WebLVC:PhysicalEntity"
}
```

This turns off all filters for the `WebLVC:PhysicalEntity ObjectType`. All `WebLVC:PhysicalEntity` objects are received.

5.6.5.2 FilterList

`FilterList` can be used to compose more complex filtering. The `FilterType` property shall apply to the immediately contained `FilterList`. Each entry in the `FilterList` is itself a filter object and may contain its own `FilterType` and `FilterMatch` or `FilterList`.

Example 17 - Filter composition example

```
{
    "MessageKind": "SubscribeObject",
    "ObjectType": "WebLVC:PhysicalEntity",
    "FilterType": "any",
    "FilterList": [{
            "FilterType": "all",
            "FilterMatch": {
                "ForceIdentifier": [1, 4, 7]
            }
        }, {
            "FilterType": "all",
            "FilterMatch": {
                "ForceIdentifier": [2, 5, 8],
                "IsConcealed": [false]
            }
        }
    ]
}
```

Note this example joins the two given filters with `any`, so it matches if either of the contained filter matches. The contained filters have their own `FilterType`, which are set to the default of `all`. This example subscribes to all forces with `ForceIdentifier 1,4,` or `7`. It also subscribes to all forces with `ForceIdentifier 2,5,` or `8` and also have `IsConcealed` set to `false`.

5.7 Log

Each client has access to a log of server-implementation-dependent information related to that client – status information, notifications, errors, etc. After a client makes a successful connection, the WebLVC Server creates a log for each client. Each client has a separate, independent log containing information pertaining only to the corresponding client. Clients may request all or part of its log by sending a `LogRequestMessage`. The server responds by returning an array of `Log` objects in a `LogResponseMessage`.

Any errors during a failed connection attempt are noted in the corresponding `ConnectResponse` message and are not logged to a client log, since the server might not create a client log for an unknown, unconnected client.

5.7.1 LogRequest Message

Request the oldest `Log` objects from the server, ordered in time (oldest first). Returned `Log` objects are not returned again and may be removed from the server.

Optional properties:

`Length` – the maximum number of `Log` objects to return as a number. If omitted the server shall return all `Log` objects it holds when the request is received.

Example 18 - Request to return the 2 oldest log objects

```
{
    "MessageKind": "LogRequest",
    "Length": 2
}
```

5.7.2 LogResponse Message

A server response to a `LogRequest` request message.

Required properties:

`Log` – an array of `Log` JSON objects, ordered by most recent entry first.

The nested properties of the `Log` JSON objects that are communicated within `LogResponse` messages are as follows:

`Timestamp` – a string containing the time and date at which the data was recorded - measured by the server's local time formatted according to ISO 8601.

`Message` – a string containing the text of the log entry. The content and format of status entries, and whether the text is subject to internationalization, localization or other translation, is not specified. Entries are server-implementation-specific.

Example 19 - LogResponse

```
{
    "MessageKind": "LogResponse",
    "Log": [{
            "Timestamp": "2014-12-23T18:25:43.511Z",
            "Message": "Unrecognized MessageKind 'shutdown'"
        }, {
            "Timestamp": "2014-12-23T18:12:23.125Z",
            "Message": "Can't delete object 'unkObj'"
        }
    ]
}
```

6 The Standard Object Model

The WebLVC specification is flexible enough to allow the representation of arbitrary Object Models by defining appropriate `ObjectTypes`, `InteractionTypes`, and appropriate properties of specific kinds of `AttributeUpdate` messages, and `Interaction` messages.

However, WebLVC does include a Standard Object Model, which was developed to serve the needs of web-based clients that require semantics similar to those in the DIS protocol or HLA's RPR FOM. That is, the Standard Object Model defines a set of Object Types, Interaction Types, and corresponding `AttributeUpdate` and Interaction messages based on the DIS/RPR FOM definitions. Wherever possible, the Standard Object Model uses attribute names, parameter names, data types, and enumerations that match the corresponding elements in DIS/RPR FOM.

Although in general the WebLVC Object Model mirrors DIS/RPR FOM very closely, there is not *always* a direct mapping between RPR FOM classes, attributes, and parameters and their WebLVC counterparts. This is because the WebLVC specification attempts to represent data in a way that is consistent with JSON conventions, efficient to convey in JSON messages, and easy to parse and use in JavaScript applications.[1] For example, the `EntityType` property of a `PhysicalEntity` object uses the familiar 7-element SISO enumeration, but it encodes that 7-tuple as a simple JSON array of 7 numbers – e.g., [1,1,225,1,1,0,0] – rather than using the 8-byte binary "struct" used in DIS and the RPR FOM. As another example, WebLVC uses a single `"WebLVC:PhysicalEntity"` type regardless of which kind of physical entity is being simulated, rather than the RPR FOM's fairly deep hierarchy of subclasses of the `PhysicalEntity` class. (Many of the reasons for choosing a deep hierarchy in the RPR FOM don't apply in WebLVC where there is no real concept of classes, inherited attributes, object promotion, etc.)

That being said, the similarities between the WebLVC Object Model and the RPR FOM are far more numerous than the differences.

Many WebLVC Object and interaction attributes are categorized as optional, even if they correspond to DIS/RPR FOM values which are required. This is because WebLVC applications may be used in simulations where there are no DIS simulators or HLA federates, in which case some of the properties required solely for DIS/HLA interoperability can be omitted.

Standard Object Types

Object Types defined by the Standard Object Model are:

`"WebLVC:PhysicalEntity"`

`"WebLVC:AggregateEntity"`

`"WebLVC:EnvironmentalEntity"`

`"WebLVC:EnvironmentObject"`

`"WebLVC:RadioTransmitter"`

[1] Indeed, the fact that some intelligent analysis is needed in order to decide on an appropriate JSON representation for each concept is the reason why the WebLVC specification directly defines a Standard Object Model, rather than defining set of automated rules for generating the Standard Object Model from the HLA RPR FOM.

Standard Interaction Types

Interaction Types defined by Standard Object Model are:

`"WebLVC:WeaponFire"`

`"WebLVC:MunitionDetonation"`

`"WebLVC:StartResume"`

`"WebLVC:StopFreeze"`

`"WebLVC:RadioSignal"`

The following sections define the required and optional properties for `Configure`, `ConfigureResponse`, `AttributeUpdate` messages and Interactions messages associated with each of the standard Object and Interaction types.

6.1 Common Properties and Datatypes

Some properties with identical names and behavior appear in more than one message.

6.1.1 Coordinates

`Coordinates` – An object which describes the position, orientation, and kinematic characteristics of an entity. The makeup of this object depends on the coordinate reference system (CRS) determined during configuration. See section 5.3.3 and section 6.2.1 for more information.

For a geocentric Coordinate Reference System (CRS) (such as Earth-Centered, Earth-Fixed (ECEF) as in DIS), `Coordinates` has the following properties:

> `DeadReckoningAlgorithm` – A number representing the dead-reckoning algorithm to use for the entity, as defined by the SISO Enumerations document, e.g., 4 for "Dead Reckoning Model (DRM) (RVW)".
>
> `WorldLocation, VelocityVector, AccelerationVector` – Arrays of three numbers representing the X, Y, and Z components of world location, velocity and acceleration, as defined by DIS/RPR FOM, e.g., `[4437182.0232, -395338.0731, 873923.4663].`
>
> `Orientation` - An array of three numbers representing the psi, theta, and phi components of the entity's orientation as defined by DIS/RPR FOM, e.g., `[-1.65, 2.234, -0.771].`
>
> `AngularVelocity` – An array of three numbers representing the X, Y, and Z components of angular velocity, as defined by DIS/RPR FOM, e.g., `[0.37, 1.30, -1.43].`
>
> By default (i.e., if a client does not use a Configure message to specify an alternate coordinate system), the Standard Object Model uses the geocentric coordinate reference system that matches DIS/RPR FOM.

For a geodetic CRS (as in EPSG::3857 or EPSG::4362), `Coordinates` has the following properties:

> `WorldLocation` - An array of 3 numbers given as `[latitude, longitude, altitude]`, as determined by the CRS model. `Altitude` is measured with respect to a geodetic reference, for example the World Geodetic System (WGS) 84 ellipsoid. `Altitude` measured in meters.
>
> `VelocityVector` – An array of 3 numbers given as `[course, pitch, speed]` indicating the speed and direction of travel. `Course` is measured in radians with respect to true north. `Pitch` is measured in radians with respect to a vector normal to the Earth's surface. [0, 0, 0] represents an entity on a level surface travelling toward true north at a speed of 0. Speed is measured in meters per second.

`Orientation` - An array of three numbers representing the `[heading, pitch, roll]`. `Heading` is measured in radians with respect to true north. `Pitch` and `roll` are measured in radians with respect to a vector normal to the Earth's surface. [0, 0, 0] represents an entity on a level surface facing true north.

6.1.2 EntityIdentifier

`EntityIdentifier` is DIS-style entity identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3]. This attribute used only for compatibility with DIS/RPR FOM and otherwise is not required. This specification does not prescribe any correspondence between WebLVC Servers, clients and `EntityIdentifier` values, nor how responsibility to handle `EntityIdentifier` is distributed among servers and clients.

6.1.3 Environment appearance

The RPR FOM defines "appearance attributes" as attributes of the `EnvironmentObject` subclasses. The appearance attributes are defined in the RPR FOM either as booleans, or as numbers.

`PercentComplete` - a number used to represent the percent completion of the environment object.

`DamagedAppearance` - a number representing the damaged appearance of the environment object (i.e., no damage, slight damage, moderate damage, and destroyed), as defined in the SISO Enumerations document.

`ObjectPredistributed` - a boolean indicating whether or not the environment object was created before the start of the exercise.

`Deactivated` - a boolean indicating whether or not the environment object has been deactivated (it has ceased to exist in the synthetic environment).

`Smoking` - a boolean indicating whether or not the environment object is smoking (creating a smoke plume).

`Flaming` - a boolean indicating whether or not the environment object is aflame.

6.2 Administrative Messages

The `Configure` and `ConfigureResponse` messages are augmented to provide custom configuration for the Standard Object Model.

6.2.1 Configure Message

The Standard Object Model adds the following properties to the Configure message.

Optional properties:

`CoordinateReferenceSystem` – the coordinate reference system to be used, as a string containing an Open Geospatial Consortium (OGC) coordinate reference system Uniform Resource Name (URN) (See OGC URN Policy) Example: "urn:ogc:def:crs:OGC:1.3:CRS84".

Table 3 - OGC URNs for coordinate reference systems

OGC CRS URN	Description
urn:ogc:def:crs:EPSG::4978	Default, ECEF Cartesian used by DIS/RPR FOM.
urn:ogc:def:crs:EPSG::3857	Web Mercator projection.
urn:ogc:def:crs:EPSG::4326	Geodetic based on WGS 84 ellipsoid.

The `CoordinateReferenceSystem` affects all properties which rely upon a coordinate reference system published by either the connected client or the server.

Example 20 - Configure web Mercator coordinates

```
{
    "MessageKind": "Configure",
    "CoordinateReferenceSystem": "urn:ogc:def:crs:EPSG::3857"
}
```

`ServerDeadReckoning` – an object which describes if and how the server should provide `AttributeUpdate` messages containing `Coordinates` properties. This `ServerDeadReckoning` object may contain one or more optional nested properties:

`Enable` – a Boolean; true if the server should perform dead reckoning and provide `AttributeUpdates` containing the results, false if the server should not perform dead reckoning and provide only normal `AttributeUpdate` messages.

`PositionThreshold` – the maximum absolute change in position when an update is issued, in meters. The default is 1 meter, consistent with DIS/RPR FOM.

`MaximumRate` – the maximum number of updates per second the server should send.

Example 21 - 50m resolution overview map with maximum 5 second update

```
{
    "MessageKind": "Configure",
    "ServerDeadReckoning": {
        "Enable": true,
        "PositionThreshold": 50.0,
        "MaximumRate": 0.2
    }
}
```

`WorldBounds` – a GeoJSON geometry object (See RFC 7946) of type `"Polygon"` or `"MultiPolygon"` that specifies the geographic boundary or boundaries of interest. `AttributeUpdate` and `Interaction` messages outside `WorldBounds` will not be sent from the server. The default area of interest is the entire world.

Example 22 - Box from Lon 10, Lat 10 to Lon 30, Lat 40

```
{
    "MessageKind": "Configure",
    "WorldBounds": {
        "type": "Polygon",
        "coordinates": [
            [[10.0, 10.0], [30.0, 10.0], [30.0, 40.0],
            [10.0, 40.0], [10.0, 10.0]]
        ]
    }
}
```

(Note the use of non-capitalized property names (e.g. "type"). Property values are capitalized. This is in accordance with the GeoJSON specification. Also, GeoJSON specifies coordinates as Longitude, Latitude, Altitude (optional) with Longitude first.)

`ObjectBounds` – an object that specifies the name of a WebLVC Object, and a range around the object within which the client is interested in updates. `AttributeUpdate` and `Interaction` messages that are outside this range will not be sent from the server. The `ObjectBounds` object has the following properties.

`ObjectName` – the name of the WebLVC Object upon which the area of interested is centered.

`Range` – the range in meters from the object beyond which messages will not be sent.

Example 23 - Watch only things within 30 km of ownship

```
{
    "MessageKind": "Configure",
    "ObjectBounds": {
        "ObjectName": "ownship",
        "Range": 30000
    }
}
```

6.2.2 ConfigureResponse

The Standard Object Model adds the following properties to the Configure message.

Optional properties:

`CoordinateReferenceSystem` – a Boolean; true if accepted, false if rejected.

`ServerDeadReckoning` – a Boolean; true if accepted, false if rejected.

`WorldBounds` – a Boolean; true if accepted, false if rejected.

`ObjectBounds` – a Boolean; true if accepted, false if rejected.

6.3 Standard AttributeUpdate Messages

6.3.1 WebLVC:PhysicalEntity Attribute Update Message

Initial Properties:

`EntityType` - Entity type expressed as an array of seven numbers representing `EntityKind`, `Domain`, `CountryCode`, `Category`, `Subcategory`, `Specific`, and `Extra`, as defined by the SISO Enumerations document, e.g., [1,2,225,1,6,0,0] (See SISO-REF-010-2021 References for Enumerations in Simulation Interoperability). Required only in the first update sent for a particular entity, or upon change.

Optional Properties:

`EntityIdentifier` – DIS-style entity identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3].

`Coordinates` – An object which describes the position, orientation, and kinematic characteristics of an entity. See section 6.1.1 Coordinates.

`ForceIdentifier` – A number representing the `ForceIdentifier` of the entity, as defined by DIS/RPR FOM.

`Marking` – A string, maximum length 11 characters, representing the marking text of the entity, as defined by DIS/RPR FOM. It is the responsibility of the WebLVC Server to perform character set conversion as required to interoperate with external distributed systems. For example, a WebLVC implementation using websockets uses UTF-8 character encoding, but DIS might use ASCII encoding.

Appearance Properties:

The RPR FOM defines a large number of "appearance attributes" as attributes of the `PhysicalEntity` class or its subclasses. The appearance attributes are defined in the RPR FOM either as Booleans, or as having various enumeration types. For example, `DamageState` is defined as an enumeration, and `IsConcealed` is defined as a Boolean. Rather than repeat them all here, this specification includes them by reference: Any of these appearance attributes may be included in a `WebLVC:PhysicalEntity AttributeUpdate` message by including a property with the same name as the corresponding RPR FOM attribute. Use numbers to represent the desired values of attributes that are defined as enumerations, as defined by the SISO Enumerations documents. Use the Boolean values `true` or `false` to represent the desired values of attributes that are defined in the RPR FOM as Booleans.

Here is an example of a WebLVC Message defining an attribute update for an object named "F-16 Alpha", which is of type "`WebLVC:PhysicalEntity`".

Example 24 - WebLVC:PhysicalEntity

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectName": "F-16 Alpha",
    "ObjectType": "WebLVC:PhysicalEntity",
    "Object": {
        "EntityIdentifier": [1, 2, 1],
        "EntityType": [1, 2, 225, 1, 3, 0, 0],
        "Coordinates": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663],
            "VelocityVector": [57.04, 32.77, 89.263],
            "Orientation": [-1.65, 2.234, -0.771]
        },
        "Marking": "F-16",
        "DamageState": 1,
        "EngineSmokeOn": true,
        "IsConcealed": false
    }
}
```

Notice that this message contains only a subset of the possible properties of physical entities, and that the order of the properties does not matter.

6.3.2 WebLVC:AggregateEntity Attribute Update Message

The `WebLVC:AggregateEntity` Attribute Update message is used to represent the DIS/RPR FOM concept of an aggregate entity.

Initial properties:

`EntityType` - DIS-style entity type expressed as an array of seven numbers representing `EntityKind`, `Domain`, `CountryCode`, `Category`, `Subcategory`, `Specific`, and `Extra`, as defined by the SISO Enumerations document, e.g., [1,2,225,1,6,0,0]. Required only in the first update sent for a particular aggregate, or upon change.

Optional properties:

`EntityIdentifier` – DIS-style entity identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3].

`Coordinates` – An object which describes the position, orientation, and kinematic characteristics of an entity. See section 6.1.1 Coordinates.

`ForceIdentifier` – A number representing the `ForceIdentifier` of the entity, as defined by DIS/RPR FOM.

`Marking` – A string representing the marking text of the entity, as defined by DIS/RPR FOM.

`Aggregate State` – A number representing the aggregation/disaggregation state of the aggregate, as defined by DIS/RPR FOM.

`Formation` – A number representing the formation of the aggregate, as defined by DIS/RPR FOM.

`Dimensions` – An array of three numbers representing the bounding box that the aggregate occupies, as defined by DIS/RPR FOM. The values are measured from the center of mass of the aggregate to the edge of the X, Y, and Z axes of the body coordinate system of the aggregate, where the axes are aligned with the aggregate's orientation, e.g., `[2.0, 2.5, 1.75]`.

`Subordinates` – An array of `ObjectName` values, for those constituent entities that are also represented by individual object instances.

`AggregateSubordinates` – List of `ObjectName` values, for those constituent sub-aggregates that are also represented by individual object instances.

`SilentEntities` – An array of objects of the form `{ ObjectType, count }`, that represent the type and number of subordinate entities that are not represented by individual object instances.

`SilentAggregates` – An array of objects of the form `{ ObjectType, count }`, that represent the type and number of subordinate aggregates that are not represented by individual object instances.

`SilentEntitiesDamageState` – An array of objects of form `{ObjectType, DamageState, count}`, representing the damage status of unpublished subordinate entity types. `DamageState` is the RPR/DIS enumerated value for damage state. Only entries with a damage state other than the default `DamageStateNone` need to be specified.

Here is an example of a WebLVC Message defining an attribute update for an aggregate object named "Platoon 1", which is of type "`WebLVC:AggregateEntity`".

Example 25 - WebLVC:AggregateEntity

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectName": "Platoon 1",
    "ObjectType": "WebLVC:AggregateEntity",
    "Object": {
        "EntityIdentifier": [1, 2, 3],
        "EntityType": [1, 1, 225, 3, 2, 0, 0],
        "Coordinates": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663],
            "Orientation": [-1.65, 2.234, -0.771]
        },
        "Marking": "Platoon 1",
        "Dimensions": [10.0, 20.0, 1.0],
        "Subordinates": ["Tank1", "Tank2", "Tank3"],
        "Formation": 3
    }
}
```

6.3.3 WebLVC:EnvironmentalEntity Attribute Update Message

The `WebLVC:EnvironmentalEntity` Attribute message is used to represent the DIS/RPR FOM concept of an environmental process object.

Initial properties:

`Type` – DIS-style entity type expressed as an array of seven numbers representing `EntityKind`, `Domain`, `CountryCode`, `Category`, `Subcategory`, `Specific`, and `Extra`, as defined by the SISO Enumerations document, e.g., [1,4,0,0,0,0,0]. Required only in the first update sent for a particular environmental entity, or upon change.

Optional properties:

`ProcessIdentifier` – DIS-style identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3].

`ModelType` – An unsigned integer identifier for the model used for generating this environmental process. Value is exercise-specific.

`EnvironmentProcessActive` – Boolean status of the environment process, indicating whether or not is active.

`SequenceNumber` – An integer value used in tagging a series of attribute update messages with unique values. If used, value should start with zero and increment by one for each update sent.

`GeometryRecords` – An array of one or more `GeometryRecords`, used to describe the physical extent of the object. A `GeometryRecord` is a GeoJSON geometry object of one of the following types: `"LineString"`, `"Polygon"`, `"Point"`, or `"Line"`. **Note: GeoJSON specifies coordinates as Longitude, Latitude, Altitude (optional) with Longitude first.**

Example 26 - WebLVC:EnvironmentalEntity

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectType": "WebLVC:EnvironmentalEntity",
    "ObjectName": "Environmental Process 1",
    "Object": {
        "ProcessIdentifier": [1, 2, 3],
        "Type": [1, 4, 0, 0, 0, 0, 0],
        "ModelType": 1,
        "EnvironmentProcessActive": true,
        "SequenceNumber": 5,
        "GeometryRecords": [
            "type": "LineString",
            "coordinates": [
                [-74.70372, 39.137688],
                [-74.70272, 39.137588]
            ]
        ]
    }
}
```

6.3.4 WebLVC:RadioTransmitter Attribute Update Message

The `WebLVC:RadioTransmitter` Attribute Update message is used to represent a device that transmits electromagnetic energy in the radio frequency range.

Optional Properties:

`HostObjectName` - String representing the object that this radio is attached to. This identifier is unique across the simulation

`EntityIdentifier` - DIS-style entity identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3]. Identifies the id of the radio. This is the same as the host object name when the radio is coming from DIS.

`RadioIndex` - A number used to identify the radio transmitter within the scope of the entity.

`RadioEntityType` - DIS-style radio type expressed as an array of 6 numbers representing entity kind domain, country, category, nomenclature version, and nomenclature. See Radio in the SISO Enumerations document.

`TransmitState` - A number indicating the state of the transmitter (whether it is off, on but not transmitting, or on and transmitting), as defined by the SISO enumerations document (See Transmit State).

`InputSource` - A number representing the source of the radio transmission (e.g., pilot, copilot, driver, etc.), as defined in the SISO Enumerations document (See Input Source).

`WorldAntennaLocation` - An array of three numbers representing the world location of the radiating portion of the transmitter, as defined by the coordinate reference system, e.g., [4437182.0232, -395338.0731, 873923.4663] for ECEF as in DIS/RPR FOM.

`RelativeAntennaLocation` - Arrays of three numbers representing the X, Y, and Z components of relative location of the radiating portion of the transmitter, as defined by DIS/RPR FOM, e.g., [1.0, 0.0, -3.0].

`AntennaPatternType` – A number representing the radiation pattern of the antenna, as defined in the SISO Enumerations document (See Antenna Pattern Type). Its value determines the content of the `AntennaPatternParameters` property.

`Frequency` – A number representing the center frequency used by the radio in transmission. Expressed as a positive integer number in units of hertz.

`TransmitBandwidth` – A number representing the bandpass of the radio, expressed as a position floating point number in units of hertz.

`Power` – A number representing the average power being transmitted in units of decibel-milliwatts.

`ModulationType` – A JSON object representing the type of modulation used for radio transmission. The properties of this object are:

> `SpreadSpectrum` – A number representing the spread spectrum technique in use, as defined in the SISO Enumerations document.
>
> `Major` – A number representing the major classification of the modulation type, as defined in the SISO Enumerations document.
>
> `Detail` – A number representing certain detailed information depending on the major modulation type, as defined in the SISO Enumerations document.
>
> `System` – A number used to specify the interpretation of the Modulation Parameter field(s) in the `RadioTransmitterUpdate` message, as defined in the SISO Enumerations document.

`CryptoMode` – A number representing the crypto mode (baseband or diphase), as defined in the SISO Enumerations document.

`CryptoSystem` – A number representing the use of crypto or secure voice equipment, as defined in the SISO Enumerations document.

`CryptoKey` – A number representing the encryption key (if encryption is in use), zero otherwise.

`AntennaPatternParameters` – A JSON object that may be present, depending on the value of the `AntennaPatternType` property (DIS/RPR FOM Beam Antenna Pattern record):

> `BeamDirection` – An array of three numbers representing the psi, theta, and phi components of the beam direction as defined by DIS/RPR FOM, e.g., [-1.65, 2.234, -0.771].
>
> `AzimuthBeamwidth` - A number representing the full width of the beam to the -3 Decibel (dB) power density points in the x-y plane of the beam coordinate system, in radians.
>
> `ElevationBeamwidth` - A number representing the full width of the beam to the -3 dB power density points in the x-z plane of the beam coordinate system, in radians.
>
> `ReferenceSystem` - A number specifying the reference coordinate system with respect to which the beam direction is specified, as defined in the SISO Enumerations document.
>
> `Ez` A number representing the DIS/RPR FOM concept of the magnitude of the Z-component (in beam coordinates) of the Electrical field.

`Ex` A number representing the DIS/RPR FOM concept of the magnitude of the X-component (in beam coordinates) of the Electrical field.

`Phase` - A number representing the phase angle between `Ez` and `Ex` in radians.

`FrequencyHopInUse` - A boolean indicating whether or not frequency hopping is in use.

`PseudoNoiseInUse` - A boolean indicating whether or not pseudo noise is in use.

`TimeHopInUse` - A boolean indicating whether or not time hopping is in use.

Example 27 - WebLVC:RadioTransmitter

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectType": "WebLVC:RadioTransmitter",
    "ObjectName": "Radio 1",
    "Object": {
        "EntityIdentifier": [1, 1, 3001],
        "RadioIndex": 1,
        "RadioEntityType": [225, 2, 3, 4],
        "TransmitState": 1,
        "RadioEntityType": [1, 1, 225, 3, 4, 5],
        "InputSource": 2,
        "WorldAntennaLocation": [4437182.0232, -395338.0731, 873923.4663],
        "RelativeAntennaLocation": [1.0, 0.0, -3.0],
        "AntennaPatternType": 1,
        "Frequency": 44056,
        "Power": 50.5,
        "ModulationType": {
            "SpreadSpectrum": 1,
            "Major": 2,
            "Detail": 3,
            "System": 4
        },
        "CryptoMode": 1,
        "CryptoSystem": 4,
        "CryptoKey": 2348238752,
        "AntennaPatternParameters": {
            "BeamDirection": [-1.65, 2.234, -0.771],
            "AzimuthBeamwidth": 0.25,
            "ElevationBeamwidth": 0.78,
            "ReferenceSystem": 2,
            "Ez": 2.533,
            "Ex": 1.29,
            "Phase": 0.707
        },
        "FrequencyHopInUse": true,
        "PseudoNoiseInUse": false,
        "TimeHopInUse": true
    }
}
```

6.3.5 WebLVC:EnvironmentObject Attribute Update Message

The `WebLVC:EnvironmentObject` Attribute Update message is used to represent the DIS/RPR FOM concept of an environment object.

Initial Properties:

`EnvironmentObjectType` - DIS-style entity type expressed as an array of four numbers representing environment object Domain, Kind, Category, and Subcategory, as defined by the SISO Enumerations document, e.g., [1,1,3,1]. Required only in the first update sent for a particular object, or upon change.

Optional Properties:

`EnvironmentObjectIdentifier` - DIS-style entity identifier expressed as an array of three numbers representing `SiteID`, `ApplicationID`, and `EntityNumber`, e.g., [1,2,3]

`type` - a string representing the kind of the environment object. Either `"Area"`, `"Line"`, or `"Point"`.

`ForceIdentifier` - A number representing the `ForceIdentifier` of the environment object, as defined by DIS/RPR FOM.

`Area` – present if `"EnvironmentObjectGeometryType"` is "Area", this property is an array of `WorldLocation` that describes polygon vertices in world coordinates. Each `WorldLocation` is expressed as an array of three numbers expressing position in the selected coordinate reference system.

`Line` – present if `"EnvironmentObjectGeometryType"` is `"Line"`, this property is an array of objects that describes a multi-segmented line, where each segment object has the following properties:

Required properties:

`SegmentNumber` – a number which uniquely identifies this segment object within this Line.

`WorldLocation` - an array of three numbers expressing position in the selected coordinate reference system.

`Orientation` - an array of three numbers expressing orientation in the selected coordinate reference system.

`Length` - the length of the segment, in meters, extending in the positive X direction.

`Width` - the total width of the segment, in meters; one-half of the width shall extend in the positive Y direction, and one-half of the width shall extend in the negative Y direction.

`Height` - the height of the segment, in meters, above ground.

`Depth` - the depth of the segment, in meters, below ground level.

Optional properties:

Appearance properties listed below may be included in the object.

`Point` – present if `"EnvironmentObjectGeometryType"` is `"Point"`, this property is an object with the following properties:

Required properties:

`WorldLocation` - an array of three numbers expressing position in the selected coordinate reference system.

`Orientation` - an array of three numbers expressing orientation in the selected coordinate reference system.

Appearance Properties:

Here is an example of a WebLVC Message defining an attribute update for an object named "xxx", which is of type `"WebLVC:EnvironmentObject"`.

Example 28 - WebLVC:EnvironmentObject

```
{
    "MessageKind": "AttributeUpdate",
    "ObjectName": "Small Crater",
    "ObjectType": "WebLVC:EnvironmentObject"
    "Object": {
        "EnvironmentObjectIdentifier": [1, 2, 3],
        "EnvironmentObjectType": [1, 1, 3, 1],
        "ForceIdentifier": 1,
        "EnvironmentObjectGeometryType": "Point",
        "Point": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663],
            "Orientation": [-1.65, 2.234, -0.771]
        },
        "PercentComplete": 100,
        "DamagedAppearance": 0,
        "ObjectPredistributed": false,
        "Deactivated": false,
        "Smoking": false,
        "Flaming": false
    }
}
```

Notice that this message contains only a subset of the possible properties of environment objects.

6.4 Standard Interaction Messages

6.4.1 WebLVC:WeaponFire Interaction Message

Optional properties:

`AttackerId` - The `ObjectName` of the object that fired the weapon.

`TargetId` - The `ObjectName` of the object that was targeted by the Attacker, if applicable.

`MunitionType` – The type of the munition object. A DIS-style entity type expressed as an array of seven numbers representing `EntityKind`, `Domain`, `CountryCode`, `Category`, `Subcategory`, `Specific`, and `Extra`, as defined by the SISO Enumerations document, e.g., [1,2,225,1,6,0,0].

`MunitionId` – The object name of the munition, if it is simulated as an entity object. If not, then this field may be omitted.

`EventId` – A string identifier generated by the issuing client, used to associate related fire and detonation events.

`WarheadType` – A number representing the type of warhead fitted to the munition being fired, as defined by DIS/RPR FOM.

`FireMissionIndex` - A unique number to identify the fire mission. Used to associate weapon fire interactions in a single fire mission.

`FuseType` – A number representing the type of fuse on the munition, as defined by DIS/RPR FOM. This field may be omitted if undefined.

`Quantity` - The number of rounds fired in the fire event. A value of zero indicates a single round.

`Rate` – A number representing the rate at which munitions are discharged in rounds per minute.

`Range` – A number, representing the range in meters assumed by firing entity in computing the fire control solution.

`Coordinates` – An object which describes the position and velocity of the launched munition in the coordinate reference system. See section 6.1.1 Coordinates.

Here is an example of a WebLVC Message defining an interaction of type `"WebLVC:WeaponFire"`.

Example 29 - WebLVC:WeaponFire

```
{
    "MessageKind": "Interaction",
    "InteractionType": "WebLVC:WeaponFire",
    "Interaction": {
        "AttackerId": "Tank1",
        "TargetId": "Tank2",
        "MunitionType": [2, 2, 225, 2, 3, 0, 0],
        "Coordinates": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663],
            "VelocityVector": [57.04, 32.77, 89.263]
        }
    }
}
```

6.4.2 WebLVC:MunitionDetonation Interaction Message

Optional properties:

`AttackerId` – The `ObjectName` of the object that fired the weapon.

`TargetId` – The `ObjectName` of the object that was targeted by the Attacker. For indirect fire this field may be omitted.

`MunitionId` – The object name of the munition, if it is simulated as an entity object. If not, then this field may be omitted.

`MunitionType` – The type of the munition object. A DIS-style entity type expressed as an array of seven numbers representing `EntityKind`, `Domain`, `CountryCode`, `Category`, `Subcategory`, `Specific`, and `Extra`, as defined by the SISO Enumerations document, e.g., [1,2,225,1,6,0,0].

`EventId` – A string identifier generated by the issuing client, used to associate related fire and detonation events.

`Coordinates` – An object which describes the position, orientation and kinematic characteristics of an entity. See section 6.1.1 Coordinates.

`EntityLocation` – Location of detonation with respect to the target entity's coordinate system, expressed as an array of three numbers, e.g., `[-1.4, 2.7, 0.6]`.

`Result` – A number representing the Result of the detonation as specified by DIS/RPR FOM.

`FuseType` – A number representing the type of fuse on the munition, as defined by DIS/RPR FOM.

`Quantity` – The number of rounds fired in the fire event. A value of zero indicates a single round.

`Rate` – A number representing the rate at which munitions are discharged in rounds per minute.

`WarheadType` – A number representing the type of warhead fitted to the munition being fired, as defined by DIS/RPR FOM.

Here is an example of a WebLVC Message defining an interaction of type `"WebLVC:MunitionDetonation"`.

Example 30 - WebLVC:MunitionDetonation

```
{
    "MessageKind": "Interaction",
    "InteractionType": "WebLVC:MunitionDetonation",
    "Interaction": {
        "AttackerId": "Tank1",
        "TargetId": "Tank2",
        "MunitionType": [2, 2, 225, 2, 3, 0, 0],
        "Coordinates": {
            "WorldLocation": [4437182.0232, -395338.0731, 873923.4663]
        },
        "Result": 1
    }
}
```

6.4.3 WebLVC:StartResume Interaction Message

The `WebLVC:StartResume` message directs one or more simulators to start or resume simulating or start or resume simulation of a specific entity.

Required Properties:

`ReceivingEntity` – The DIS/RPR FOM simulation ID of the receiving simulation(s), or the entity ID of a specific entity, expressed as an array of 3 numbers. The allowable values for the ID follow the DIS/RPR FOM rules.

`RequestIdentifier` – An integer used to identify a particular request. The value should be incremented for each request.

`RealWorldTime` – A number representing the real-world time of the request, as defined in by DIS/RPR FOM.

`SimulationTime` – A number representing the time of day in simulation time of the request, as defined by DIS/RPR FOM.

Optional Properties:

`OriginatingEntity` – The DIS/RPR FOM simulation ID (Site ID, Application ID) of the simulator requesting the start/resume action, expressed as an array of two numbers.

6.4.4 WebLVC:StopFreeze Interaction Message

The `WebLVC:StopFreeze` message directs one or more simulators to stop simulating or stop simulation of a specific entity.

Required Properties:

`ReceivingEntity` – The DIS/RPR FOM simulation ID of the receiving simulation(s), or the entity ID of a specific entity, expressed as an array of 3 numbers. The allowable values for the ID follow the DIS/RPR FOM rules.

Optional Properties:

`OriginatingEntity` – The DIS/RPR FOM simulation ID (Site ID, Application ID) of the simulator requesting the stop/freeze action, expressed as an array of two numbers.

`RealWorldTime` – A number representing the real world time of the request, as defined in by DIS/RPR FOM.

`Reason` – A number indicating the reason for stopping the simulation. Allowed values are an enumeration defined by DIS/RPR FOM.

`ReflectValues` – A Boolean indicating whether entity or entities should continue to reflect attributes while frozen.

`RunInternalSimulationClock` – A Boolean indicating whether the entity or entities should continue to run their internal simulation clocks while frozen.

`UpdateAttributes` – A Boolean indicating whether entity or entities should continue to update attributes while frozen.

6.4.5 WebLVC:RadioSignal Interaction Message

The `WebLVC:RadioSignal` interaction message represents the wireless transmission and reception of audio or digital data via electromagnetic waves.

Optional Properties:

`RadioIdentifier` – A string representing the particular transmitter that is transmitting. This identifier is unique across the simulation.

`EncodingClass` – A number representing the encoding scheme being utilized, as defined by the SISO Enumerations document (See Radio signal encoding class).

`EncodingType` – A number representing the type of Tactical Data Link (TDL) message. If set to zero, this is not a TDL message. When this is a positive number, the number represents the type of TDL message, as defined by the SISO Enumerations document scheme being utilized, as defined by the SISO Enumerations document as (See Radio signal encoding type).

`TDLType` – A number representing the type of TDL message included in the signal message, as defined by the SISO Enumerations document (See TDL Type).

`SampleRate` – A number representing sample rate in samples per second for audio data. The data rate in bits per second for digital data.

`SampleCount` – A number representing the number of samples in the audio data.

`SampleData` – A base64-encoded string representing the data content of the message. The encoded string includes padding. See IETF RFC 4648.

`DatabaseIndex` – A number representing the index of a pre-recorded audio file that is to be looked up in a known database for playing.

`UserProtocolID` – The protocol id of the data stream.

Example 31 - WebLVC:RadioSignalInteraction

```
{
    "MessageKind": "Interaction",
    "InteractionType": "WebLVC:RadioSignalInteraction",
    "Interaction": {
        "RadioIdentifier": "Radio 1",
        "EncodingClass": "Tank2",
        "EncodingType": 0,
        "SampleRate": 44056,
        "SampleData": "Zg=="
    }
}
```

7 Extending the WebLVC Protocol

As described above, the Standard for WebLVC Protocol includes a Standard Object Model based on a subset of the DIS/RPR FOM. However, the WebLVC standard can be easily extended by its users in many ways in order to facilitate the exchange of data that is outside of the Standard Object Model.

- Users can define new `ObjectTypes` and `InteractionTypes`, and define various properties associated with each new message type.
- Users can add new properties to existing `StandardObjectModel` message types.

JSON (and JavaScript) is naturally extensible in a backwards compatible way. When a client receives a message, it can check the message type against types it knows about and ignore messages of unknown types. Similarly, when a client receives a message of a type is does know about, it can query the message for values for properties it expects and ignore properties whose names are unexpected.

7.1 Extending the WebLVC Protocol, based on DIS

Imagine that a DIS application has extended the DIS protocol (or is using some of the built-in extension capabilities of DIS 7) to add new attributes to the Entity State PDU. A user who wants to pass that attribute to a WebLVC Client can define a WebLVC (JSON) representation of the attribute and define a new property of `WebLVC:PhysicalEntity` to be used to convey it. There is no formal Meta-Model in WebLVC to represent object model extensions – users can document their extensions in a way similar to the way this document defines the Standard Object Model. What is required to implement the use case described above, is for someone to extend their WebLVC Server to support the mapping of the new DIS attribute to the new WebLVC property. Depending on how the WebLVC Server is built and distributed, this extension might be accomplished through editing source code, using a plug-in API, or through a run-time configuration mechanism. However it is accomplished, the point is that user intervention is required in order to "teach" the WebLVC Server which DIS field or new PDU type should be mapped to which WebLVC property or message type.

7.2 Extending the WebLVC Protocol Automatically, based on HLA

If a user has already defined an object model in the form of an HLA FOM, it is possible for a WebLVC Server to *automatically* extend the WebLVC Protocol to include WebLVC elements that directly correspond to custom HLA Object Classes, Interaction Classes, Attributes and Parameters. This is because unlike typical DIS extensions, HLA FOM elements are machine readable in a standard way. A FOM provides information about class names, as well as attribute and parameter names and datatypes – everything that an application needs to know how to build corresponding JSON elements within a WebLVC Object model.

Not only is it possible for a WebLVC Server to extend the WebLVC Protocol, it is also possible for a WebLVC Server to *automatically* know how to translate between the new FOM elements and the corresponding new WebLVC elements – without requiring user extension of server capabilities. In order to achieve this, section 7.3 documents a set of rules for mapping between HLA data types and structures and their corresponding WebLVC representations. If all WebLVC Servers map a specific HLA data type to the same representation on the WebLVC side, client developers will know exactly what to expect, and can write client code accordingly.

For example, the JSON format identifies the following data types:

Table 4 - JSON type examples

JSON type	Examples
Number	3.14
String	"my name"
Boolean	true, false
Array	[1, 2, "3"]
Object	{ a: 1, b: "a string" }
Null	Null

Because JSON has only a few types, there might be many valid mappings from simulation data models with a greater set of more specific types. Should a WebLVC number be mapped as an integer or some floating point type? Or, should a JSON string map as an array of characters, or does it correspond to some enumerated value? Starting from the more specific data model should simplify the conversion.

The next section describes standard rules for generating WebLVC extensions from datatypes of an HLA FOM.

7.3 Standard Mapping Rules for HLA and RPR FOM

The High Level Architecture uses a meta-model called an object model template (OMT) to encode simulation data into a Federation Object Model (FOM). The OMT organizes data type definitions into tables. Rules for mapping between standard HLA and RPR FOM datatypes, and WebLVC datatypes are detailed below.

RPR FOM includes datatypes intended only as padding, to ensure correct alignment of data. These padding datatypes do not map to corresponding WebLVC datatypes and are omitted from WebLVC datatypes.

7.3.1 Basic data representation table

The OMT describes representations of data separately from their types. Unless a more specific mapping applies, OMT simple datatypes should use the following mapping for their representation.

Table 5 - HLA basic data representation in WebLVC

OMT name	WebLVC type
HLAinteger16BE	Number
HLAinteger32BE	Number
HLAinteger64BE	Number
HLAfloat32BE	Number
HLAfloat64BE	Number
HLAoctetPairBE	Number (unsigned)
HLAinteger16LE	Number
HLAinteger32LE	Number
HLAinteger64LE	Number
HLAfloat32LE	Number
HLAfloat64LE	Number
HLAoctetPairLE	Number (unsigned)
HLAoctet	Number (unsigned)
RPRunsignedInteger8BE	Number (unsigned)
RPRunsignedInteger16BE	Number (unsigned)
RPRunsignedInteger32BE	Number (unsigned)
RPRunsignedInteger64BE	Number (unsigned)
Unsignedinteger16BE	Number (unsigned)
Unsignedinteger32BE	Number (unsigned)
Unsignedinteger64BE	Number (unsigned)
OMT13any	Number (unsigned)

Basic data representations shall map as either Number, String or Boolean data types.

7.3.2 Simple datatype table

Scalar data types are described in the OMT simple datatype table.

Table 6 - HLA simple datatype table in WebLVC

OMT name	WebLVC type
HLAASCIIchar	String (single character)
HLAunicodeChar	String (single character)
HLAbyte	Number (unsigned)
HLAinteger64Time	Number
HLAfloat64Time	Number
HLAfederateHandle	Number
HLAmsec	Number
HLAseconds	Number
HLAcString	String
HLAstring	String
HLAcount	Number
HLAhandle	Base64-encoded string
timeType	String
String	String
Int	Number
Long	Number
long long	Number
Octet	Number (unsigned)
short short	Number
unsigned int	Number (unsigned)
unsigned long	Number (unsigned)
unsigned long long	Number (unsigned)
unsigned short	Number (unsigned)
Double	Number
Float	Number

OMT character datatypes shallmap as single character strings.

7.3.3 Enumerated datatype table

Enumerations are described in the OMT enumerated datatype table.

Table 7 - HLA enumerated datatype table in WebLVC

OMT name	OMT Enumerator	WebLVC type	WebLVC value
HLAboolean	HLAfalse	Boolean	false
	HLAtrue	Boolean	true
RPRboolean	HLAfalse	Boolean	false
	HLAtrue	Boolean	true
OMT13boolean	HLAfalse	Boolean	false
	HLAtrue	Boolean	true

Except for the Boolean enumerations explicitly listed above, map the values of enumerators shall be numbers.

> *Design reasoning: using the enumerator name as a string would require updating intermediating software (e.g., gateways) as the strings change. Numbers can be passed without translation, so new enumerations names are ignored.*

Example:

Table 8 - example HLA enumerated datatype table in WebLVC

OMT name	OMT Enumerator	OMT value	WebLVC type	WebLVC value
HLAboolean	HLAfalse	0	Boolean	false
	HLAtrue	1	Boolean	true
AntennaPatternTypeEnum32	OmniDirectional	0	Number	0
	Beam	1	Number	1
	SphericalHarmonic	2	Number	2

7.3.4 Array datatype table

Standard HLA arrays are converted to either strings or arrays as shown in the table below.

Table 9 - HLA array datatype table in WebLVC.

OMT name	WebLVC type
HLAASCIIstring	String
HLAunicodeString	String
HLAopaqueData	Base64-encoded string
HLAtoken	n/a
MarkingArray11	String
MarkingArray31	String
OctetArray {all variations}	Base64-encoded string

HLAtoken is a zero-length array and is not converted to a WebLVC representation because JSON already has termination rules for strings and arrays.

WebLVC does not distinguish between fixed and variable length arrays.

OMT arrays shall convert to WebLVC arrays, except character arrays (where array element representations map as characters, or octets intended as characters), and except byte arrays. Character arrays shall convert to WebLVC strings, and octet arrays shall convert to WebLVC base64-encoded strings.

7.3.5 Fixed record datatype table

Fixed records shall be mapped as JSON objects. Each field in the OMT is mapped to a JSON object name/value pair, where the name is the OMT field name and the value depends on the OMT field type.

Example:

Table 10 - example HLA fixed record datatype table in WebLVC

Record name	Field			Encoding	Semantics
	Name	Type	Semantics		
MessageLocation	X	HLAfloat64BE	X coord of message org.	HLAfixedRecord	Coordinates where the message originates.
	Y	HLAfloat64BE	Y coord of message org.		

Should map as:

Example 32 - Map fixed record

```
{
   "X" : 2.78,
   "Y" : 3.14
}
```

7.3.6 Variant record datatype table

Variant records shall be mapped as an array of length 2. The first element is the enumerator in integral number representation. The second element is the value of the corresponding alternative. Example:

Table 11 - example HLA variant record datatype table in WebLVC

Record Name	Discriminant				Alternative			Encoding	Semantics
	Name	Type	Enumerator	Value	Name	Type	Semantics		
Waiter Value	Val Index	Experience Level	Trainee	0	Course Passed	HLAboolean	Ratings scale for employees under training.	HLAvariantRecord	Datatype for waiter performance rating value
			[Apprentice .. Senior], Master	1	Rating	RateScale	Ratings scale for permanent employees.		
			HLAother	2	NA	NA	All others.		

Possible mappings:

`[ 0, true ]` – a trainee which has passed the course

`[ 1, 8.0 ]` – depends on the definition of `RateScale`.

For a more comprehensive example, consider the RPR FOM fixed record SpatialStruct (semantics omitted):

Table 12 – SpatialStruct definition

Record name	Field		Encoding
	Name	Type	
SpatialStruct	DeadReckoningAlgorithm-A-Alternatives	SpatialStruct-DeadReckoningAlgorithm	HLAfixedRecord

It contains a single `field DeadReckoningAlgorithm-A-Alternatives` which is the variant record `SpatialStruct-DeadReckoningAlgorithm` (semantics omitted):

Table 13 – SpatialStruct-DeadReckoningAlgorithm definition

Record Name	Discriminant				Alternative		Encoding
	Name	Type	Enumerator	Value	Name	Type	
SpatialStruct-DeadReckoningAlgorithm	DeadReckoningAlgorithm	DeadReckoningAlgorithmEnum8	Other	0			HLAvariantRecord
			Static	1	SpatialStatic	SpatialStaticStruct	
			DRM_FPW	2	SpatialFPW	SpatialFPStruct	
			DRM_RPW	3	SpatialRPW	SpatialRPStruct	
			DRM_RVW	4	SpatialRVW	SpatialRVStruct	
			DRM_FVW	5	SpatialFVW	SpatialFVStruct	
			DRM_FPB	6	SpatialFPB	SpatialFPStruct	
			DRM_RPB	7	SpatialRPB	SpatialRPStruct	
			DRM_RVB	8	SpatialRVB	SpatialRVStruct	
			DRM_FVB	9	SpatialFVB	SpatialFVStruct	

The fixed record SpatialStruct would be mapped to a JSON object containing a single property named DeadReckoningAlgorithm-A-Alternatives. That property would be an array of length 2, corresponding to the variant record SpatialStruct-DeadReckoningAlgorithm.

Assuming the discriminant enumerator was DRM_FVB for further example, then the alternative would be of type SpatialFVStruct. The required definitions are shown in the table below.

Table 14 – SpatialFVStruct and dependencies definitions

Record name	Field		Encoding
	Name	Type	
SpatialFVStruct	WorldLocation	WorldLocationStruct	HLAfixedRecord
	IsFrozen	OMT13boolean	
	Orientation	OrientationStruct	
	VelocityVector	VelocityVectorStruct	
	AccelerationVector	AccelerationVectorStruct	
WorldLocationStruct	X	Double	HLAfixedRecord
	Y	Double	
	X	Double	
OrientationStruct	Psi	Float	HLAfixedRecord
	Theta	Float	
	Phi	Float	
VelocityVectorStruct	Xvelocity	Float	HLAfixedRecord
	Yvelocity	Float	
	Zvelocity	Float	
AccelerationVectorStruct	Xacceleration	Float	HLAfixedRecord
	Yacceleration	Float	
	Zacceleration	Float	

Note that OMT13boolean and RPRboolean would be mapped as WebLVC boolean because the enumerators are part of the HLA specification (they begin with the HLA prefix). Otherwise they would be mapped as their numeric values.

Table 15 – Boolean definitions

Name	Enumerator	Value	WebLVC type	WebLVC value
OMT13boolean/ RPRboolean	HLAfalse	0	Boolean	false
	HLAtrue	1	Boolean	true

The full example is shown below.

Example 33 - SpatialFVStruct object mapped from HLA RPR 2 FOM

```
{
    "DeadReckoningAlgorithm-A-Alternatives": [9, {
            "WorldLocation": {
                "X": 4437182.0232,
                "Y": -395338.0731,
                "Z": 873923.4663
            },
            "IsFrozen": false,
            "Orientation": {
                "Psi": -1.65,
                "Theta": 2.234,
                "Phi": -0.771
            },
            "VelocityVector": {
                "XVelocity": 57.04,
                "YVelocity": 32.77,
                "ZVelocity": 89.263
            },
            "AccelerationVector": {
                "XAcceleration": 0.0,
                "YAcceleration": 0.0,
                "ZAcceleration": 0.0
            }
        }
    ]
}
```

Assume `spatialObj` contains the above `SpatialFVStruct` object. Access to the X position would look like:

Example 34 - WorldLocation from mapped SpatialStruct object

```
spatialObj.DeadReckoningAlgorithm-A-Alternatives[1].WorldLocation.X
```

For comparison, consider access to the X position of a WebLVC Coordinates object `coordObj`:

Example 35 - WorldLocation from mapped WebLVC coordinates object

```
coordObj.WorldLocation[0]
```

Appendix A - Filters

This appendix serves as a reference for composing and parsing a WebLVC filter, along with additional examples. A filter may be included in a WebLVC subscription message to determine if an `AttributeUpdate` or `Interaction` message is passed to a WebLVC Client or not.

a) Notation

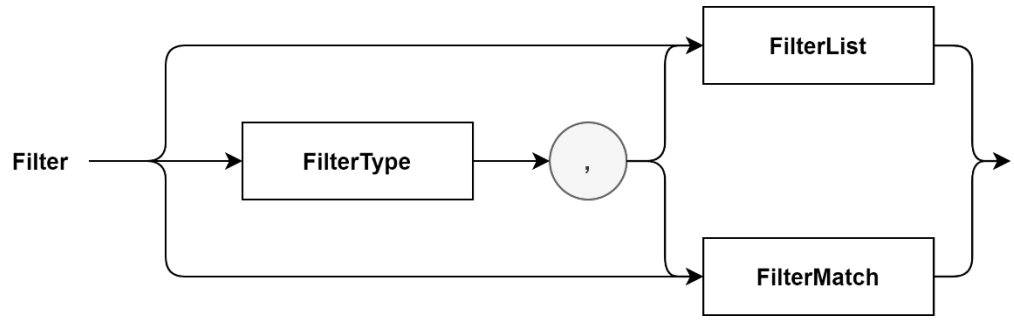
The following sections make use of diagrams to specify the structure of a filter. The elements used in the diagrams are:

- Grey-colored element: literal value as shown in the element;
- White-colored element: refers to another element that expands the initial element;
- Arrow: the direction for expanding elements.

Between elements there is optional whitespace (See ECMA-404/ISO/IEC 21778:2017). The use of whitespace is not included in the diagrams.

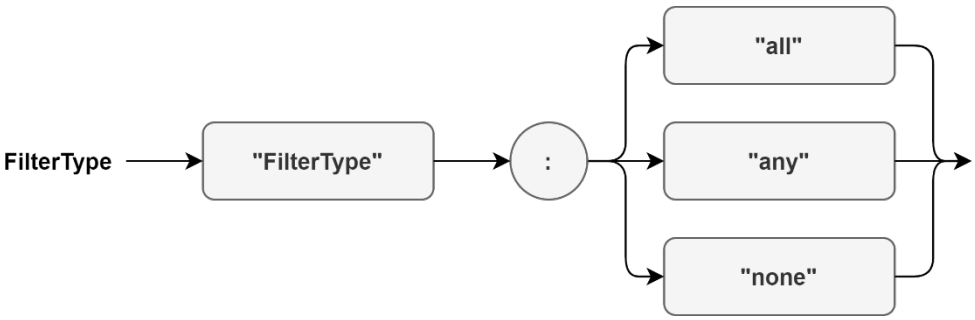
b) Filter

`Filter` is the main element. A `Filter` is either a `FilterType` with a `FilterList`, or `FilterType` with a `FilterMatch`. The `SubscribeObject` and `SubeInteraction` messages are themselves `Filter` messages with additional properties. The `FilterType` is optional and if not specified the `FilterType` shall assumed to be `"all"`. In this example, note that JSON allows any ordering of the `FilterType`, `FilterList`, and `FilterMatch` properties.



c) FilterType

The `FilterType` specifies the application of `Filter` objects in the `FilterList` or the application of the `FilterMatch` object. The `FilterType` is defined by the filter property named `"FilterType"`, followed by a colon and three possible values.



For `FilterList` the `FilterType` shall be applied as follows:

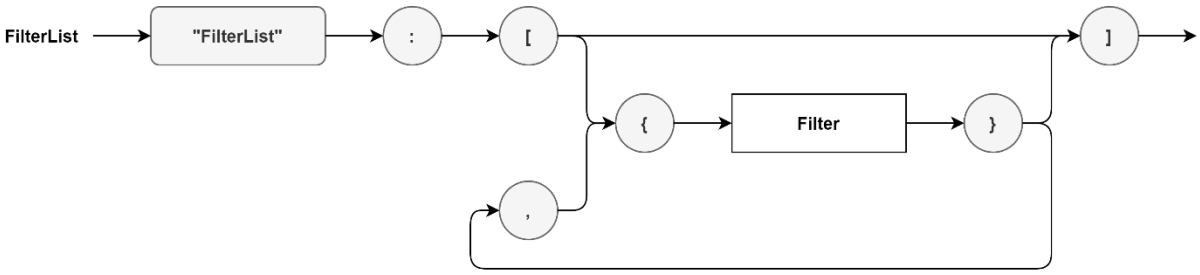
- `all`: each `Filter` in the `FilterList` must match. This is the default if no `FilterType` is specified.
- `any`: at least one `Filter` in the `FilterList` must match.
- `none`: no `Filter` in the `FilterList` must match.

And for `FilterMatch` the `FilterType` shall be applied as follows:

- `all`: all properties provided in the `FilterMatch` must match. This is the default if no `FilterType` is specified.
- `any`: at least one of the properties in the `FilterMatch` must match.
- `none`: none of the properties in the `FilterMatch` must match.

d) FilterList

A `FilterList` is a list of zero or more `Filters`. How the `Filters` are combined in the matching is specified by the `FilterType`. The `FilterList` is defined by the property named `"FilterList"`, followed by a colon and an array with zero or more `Filters`.



i) Example 1

This example defines a filter with a `FilterList` that may be used in the subscription on `WebLVC:PhysicalEntity`. It filters on red force identifiers and non-concealed blue force identifiers.

```
{
    "FilterType": "any",
    "FilterList": [{
            "FilterMatch": {
                "ForceIdentifier": [1, 4, 7, 10, 13, 16, 19, 22, 25, 28]
            }
        }, {
            "FilterMatch": {
                "ForceIdentifier": [2, 5, 8, 11, 14, 17, 20, 23, 26, 29],
                "IsConcealed": [false]
            }
        }
    ]
}
```

Pass:

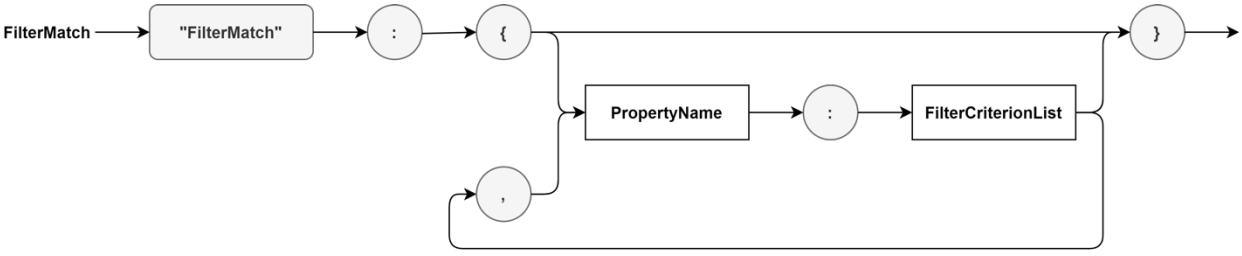
```
{
    "ForceIdentifier": 2,
    "IsConcealed": false
}
```

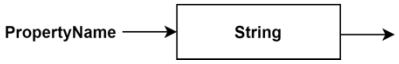
Fail:

```
{
    "ForceIdentifier": 3,
    "IsConcealed": false
}
```

e) FilterMatch

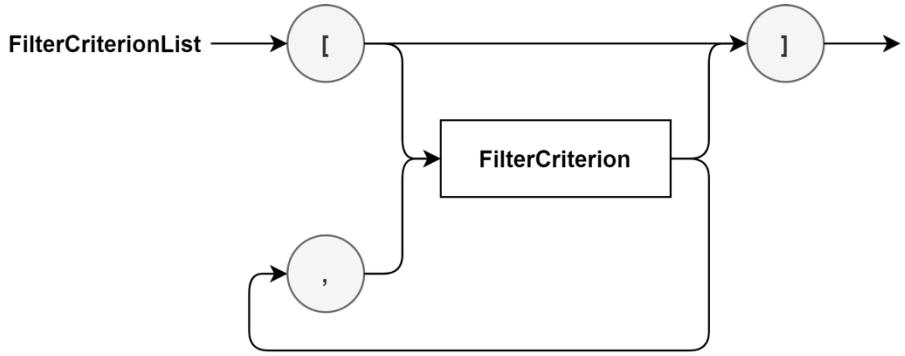
A `FilterMatch` specifies properties named for corresponding properties in the filtered object or interaction. For each `FilterMatch` property, the value associated with the corresponding named property in the object being considered is matched with the `FilterCriterion` values in the `FilterCriterionList`. Properties in the object being considered for which no `FilterMatch` property is provided, have no impact on the matching. All, any, or none of the `FilterMatch` properties must match to pass the filter, depending on the `FilterType`. A `PropertyName` is a String value, where String is defined in [ref: https://www.json.org].





f) FilterCriterionList

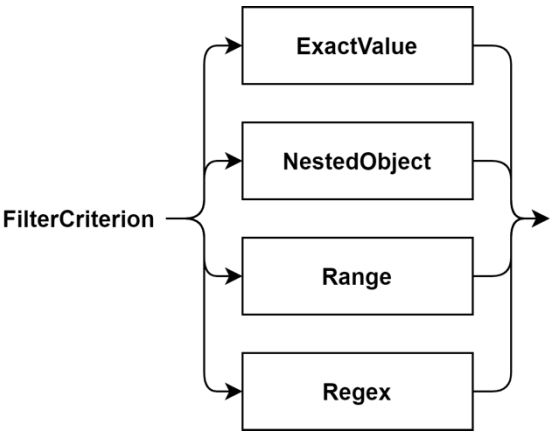
A `FilterCriterionList` is defined as an array of zero or more `FilterCriterion`. At least one `FilerCriterion` must match in order for the `FilterMatch` property to match. So, an empty list is, by definition, a no-match.



g) FilterCriterion

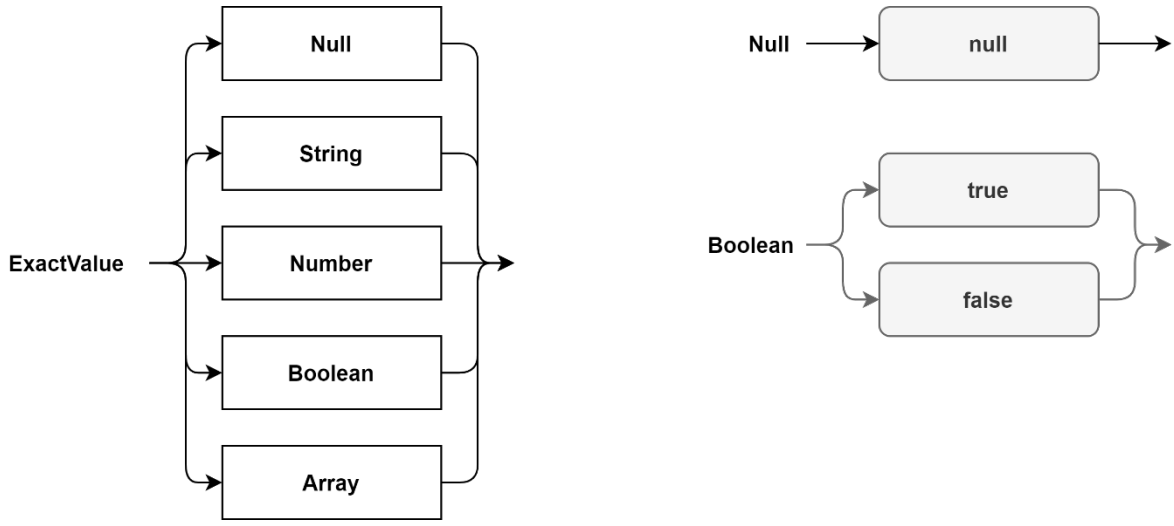
A `FilterCriterion` is used to match a property value of a filtered object. A `FilterCriterion` is either:

- an exact value to match,
- a filter to match if the property is an object,
- a range of values to match, or
- an extended regular expression to match.



h) ExactValue

An `ExactValue` in a `FilterCriterion` is either a Null value, a String value, Number value, a Boolean value, or an array. An object value must be specified as a `NestedObject` in the `FilterCriterion`.



A Null value matches if the corresponding property value of the filtered object is Null. A property value that is unknown (i.e. the value is not known to the WebLVC Server) is assumed to have a Null value. The Null value is defined in [ref: https://www.json.org].

String, Number and Boolean values are defined in [ref: https://www.json.org]. These values match if the corresponding property value of the filtered object is exactly the same.

An array value matches if the corresponding property value of the filtered object is an array with an equal or larger size, and where all array items match with the corresponding array items in the property value.

i) Example 1

This example provides a filter for an object with exact values. Note the use of the null value for `propertyB`: the property value must be either unknown or have the value “B” in order to pass the filter.

```
{
    "FilterMatch": {
        "propertyA": ["A"],
        "propertyB": [null, "B"],
        "propertyC": ["C"]
    }
}
```

Pass:

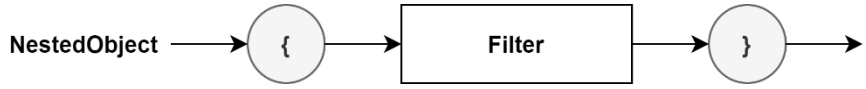
```
{
    "propertyA": "A",
    "propertyC": "C",
    "propertyD": "D"
}
```

Fail:

```
{
    "propertyA": "A",
    "propertyB": "X",
    "propertyC": "C"
}
```

i) NestedObject

An `NestedObject` in a `FilterCriterion` is used to match a property value that is an object. The `NestedObject` is recursively defined as a Filter.



i) Example 1

This example provides a filter with a nested object to filter on a sub-property named “Details”.

```
{
    "FilterMatch": {
        "MessageHeader": ["HelloWorld"],
        "MessageBody": [{
                "FilterMatch": {
                    "Details": [{
                            "regex": "good day|hello|howdy"
                        }
                    ]
                }
            }
        ]
    }
}
```

Pass:

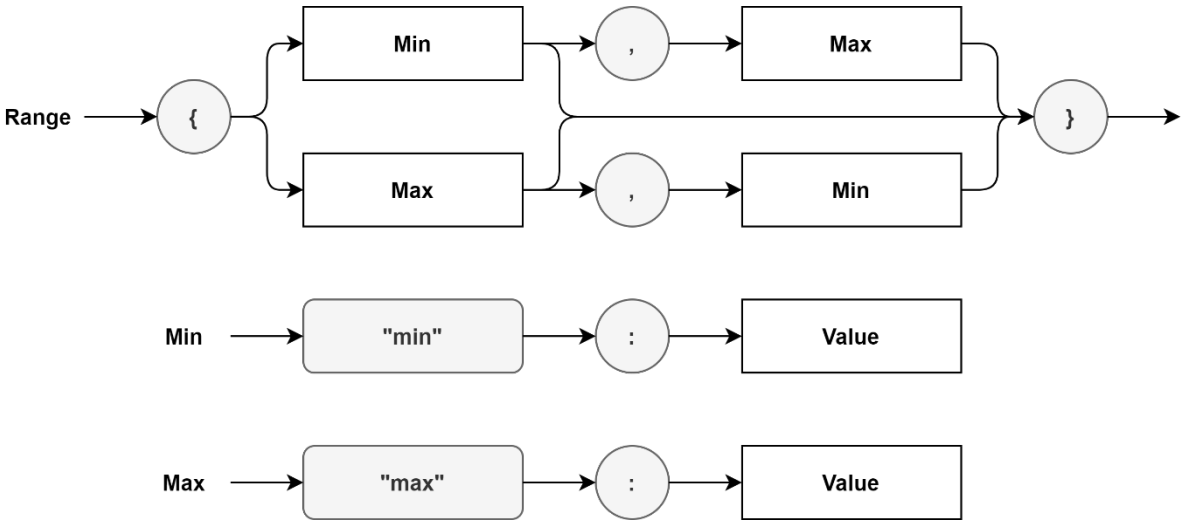
```
{
    "MessageHeader": "HelloWorld",
    "MessageBody": {
        "Details": "hello"
    }
}
```

Fail:

```
{
    "MessageHeader": "HelloWorld",
    "MessageBody": {
        "Details": "good morning"
    }
}
```

j) Range

A `Range` consists of an optional `Min` and an optional `Max` value, where a value is either a string, a number, a boolean, a null, an object, or an array (See ECMA-404/ISO/IEC 21778:2017).



The value type of the min and max value of the range must be the same. For instance, both types must be a number, a string or an object. If the value types are different then the range shall not match, regardless of the property value that is matched against the range.

The following rules apply in matching a property value against a range:

1. The property value type must be the same as the value type of the range.
2. A number value is in range if the value falls between the min and max value (inclusive).
3. A boolean value is in range if the value falls between the min and max value (inclusive), where false is considered a zero number and true as a one number.
4. A string value is in range if falls lexicographically between the min and max value (inclusive).
5. A null value is in range.
6. An object value is recursively matched property by property with the min and max value properties, and falls in range if all its properties are in range. The following applies:
    - If for a given object value property there is no min or max value property, then no min or max value applies to the matching of the object value property;
    - If for a given min or max value property there is no object value property, then a null object value property shall be assumed in the matching.
7. An array value is also recursively matched item by item with the min and max value (which must be arrays too). The following applies:
    - If for a given object value item there is no min or max value item, then no min or max value item applies to the matching of the object value item;
    - If for a given min value item or max value item there is no object value item, then a null object value item shall be assumed.
8. If only a min or only a max value is provided for the range, then matching is only against the minimum or the maximum value.

i) Example 1

This example provides a range filter with a min and max array of equal size:

```
{
    "FilterMatch": {
        "someProperty": [{
                "min": [-10, -10, -10],
                "max": [10, 10, 10]
            }
        ]
    }
}
```

Pass:

```
{
    "someProperty": [-5.5, 8.7, 2]
}
```

Pass:

```
{
    "someProperty": [-5.5, 8.7, 2, 3, 4, 5]
}
```

Fail:

```
{
    "someProperty": [11.1, 0.7, 0]
}
```

Fail:

```
{
    "someProperty": [1, 2]
}
```

ii) Example 2

This example provides a range filter with a min and max array of different size:

```
{
    "FilterMatch": {
        "someProperty": [{
                "min": [-10, -10, -10],
                "max": [10, 10, 10, 10, 10]
            }
        ]
    }
}
```

Pass:

```
{
    "someProperty": [0.5, 0.7, 0, 0.6, -11]
}
```

Fail:

```
{
    "someProperty": [1, 2, 3]
}
```

iii) Example 3

This example provides a range filter where the min-max values are objects, but with different properties:

```
{
    "FilterMatch": {
        "SomeProperty": [{
                "min": {
                    "name": "A",
                    "weight": 5,
                    "length": 11
                },
                "max": {
                    "name": "Z",
                    "weight": 9,
                    "width": 12
                }
            }
        ]
    }
}
```

Pass:

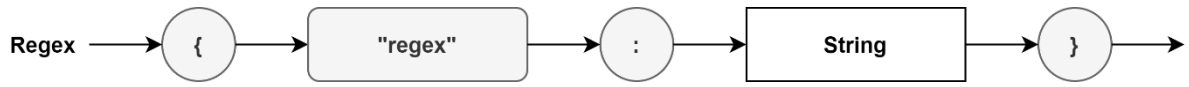
```
{
    "SomeProperty": {
        "name": "K",
        "weight": 6,
        "length": 110,
        "width": 11,
        "depth": -1
    }
}
```

Fail:

```
{
    "SomeProperty": {
        "name": "K",
        "weight": 6,
        "length": 110,
        "width": 13
    }
}
```

k) Regex

The `Regex` value represents an extended regular expression as defined in POSIX.1-2017, Section 9.4. A regex is for matching string values only. If the value type is not a string then the match fails.



i) Example 1

This example provides a filter with a simple extended regular expression to filter on marking names.

```
{
    "FilterMatch": {
        "Marking": [{
                "regex": "Tank[A-Z]"
            }
        ]
    }
}
```

Pass:

```
{
    "Marking": "TankA"
}
```

Pass:

```
{
    "Marking": "abcTankAdef"
}
```

Fail:

```
{
    "Marking": "Tank0"
}
```

ii) Example 2

This example provides an extended regular expression filter with a precise match of a name.

```
{
    "FilterMatch": {
        "Marking": [{
                "regex": "^Tank[A-Z]$"
            }
        ]
    }
}
```

Pass:

```
{
    "Marking": "TankA"
}
```

Fail:

```
{
    "Marking": "abcTankAdef"
}
```