iocast

Application Programming Interface (API)

Executive Summary

*This document specifies the Iocast **Application Programming Interface** (**API**) an interface between an application server and an Iocast system. Using the Iocast API, an application server (client) may communicate with its nodes and groups, and also perform limited Iocast system management functions.*

Document Number: 19-022
Version: 1.1
Date: 08/01/2019
Author: James M Dabbs III

CriticalResponse
Critical Response Systems, Inc.
1123 Zonolite Road NE Suite 8A
Atlanta, GA 30306-2015

www.criticalresponse.com

Notice
While reasonable efforts have been made to assure the accuracy of this document, Critical Response Systems (CRS) assumes no liability resulting from any inaccuracies or omissions in this document, or from use of the information obtained herein. The information in this document has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies or omissions. CRS reserves the right to make changes to any products and specifications described herein and reserves the right to revise this document and to make changes from time to time in content hereof with no obligation to notify any person of revisions or changes. CRS does not assume any liability arising out of the application or use of any product, software, or circuit described herein; neither does it convey license under its patent rights or the rights of others.

Copyrights
This document and the CRS products described in this document may be, include, or describe copyrighted CRS material, such as computer programs stored in semiconductor memories or other media. Laws in the United States and other countries preserve for CRS and its licensors certain exclusive rights for copyrighted material, including the exclusive right to copy, reproduce in any form, distribute and make derivative works of the copyrighted material. Accordingly, any copyrighted material of CRS and its licensors contained herein or in the CRS products described in this document may not be copied, reproduced, distributed, merged or modified in any manner without the express written permission of CRS. Furthermore, the purchase of CRS products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of CRS, as arises by operation of law in the sale of a product.

Patents
The material in this document is protected by multiple patents. Please see www.criticalresponse.com/patents for more information. Patent pending.

Computer Software Copyrights
The CRS and 3rd party supplied software products described in this document may include copyrighted CRS and other 3rd party supplied computer programs stored in semiconductor memories or other media. Laws in the US and other countries preserve for CRS and other 3rd party supplied software certain exclusive rights for copyrighted computer programs, including the exclusive right to copy or reproduce in any form the copyrighted computer program. Accordingly, any copyrighted CRS or other 3rd party supplied software contained in the CRS products described in this instruction manual may not be copied (reverse engineered) or reproduced in any manner without the express written permission of CRS or the 3rd party supplier. Furthermore, the purchase of CRS products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of CRS or other 3rd party supplied software, except for the normal non-exclusive, royalty free license to use that arises by operation of law in the sale of a product.

License Agreements
The software described in this document is the property of CRS and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

Copyrighted Materials
Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of CRS.

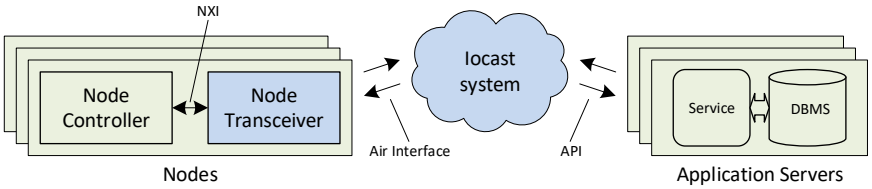
High Risk Materials
Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities). CRS and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

In some cases, CRS components may be promoted specifically to facilitate safety-related applications. With such components, CRS's goal is to help enable customers to design and create solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

Trademarks
Critical Response Systems and *Iocast* are trademarks of *Critical Response Systems, Inc*. All other product or service names are the property of their respective owners.

Document History		
Version	**Date**	**Changes**
0.0	**09/08/2015**	**Initial internal release, branch and redesign from SDP 1.9**
0.1	**10/14/2015**	**Unify language across ETI and backend API spec**
0.2	**10/20/2015**	**Review clean-up**
0.3	**11/01/2015**	**Public clean-up**
0.4	**11/01/2005**	**Table of contents corrected**
1.0	**07/21/2019**	**Re-design, including gRPC, more focused scope, and common language across NXI and air interface 2.X+.**
1.1	**08/01/2019**	**Editing and clean-up.**

1 Iocast Overiew



Iocast is a low-power wide-area network (LPWAN) designed to operate over narrowband radio channels. An Iocast system includes *node transceivers (NXs)*, *base transceivers (BXs)*, and a software-based *control stack*. Iocast establishes a communication network between *nodes* and *application servers*, enabling each to exchange *datagrams* with the other. Nodes (e.g., sensors, controllers, personal devices) connect to an Iocast system using node transceivers, while application servers connect using the *Iocast API*.

Base transceivers transmit data to node transceivers using forward channels, while node transceivers transmit data to base transceivers using reverse channels. Each node transceiver has one unique *primary address* and up to 16 *multicast addresses*, and it integrates into a node using the *Node Transceiver Interface* (NXI). Application Servers send unicast datagrams to a single transceiver using its primary address, and they may also send multicast datagrams to multiple transceivers using multicast addresses.

Reverse channels are time-shared between node transceivers while forward channels are "always on" under control of a base transceiver or set of base transceivers. Forward and reverse channels are universally synchronized together, with the control stack providing coordination, timing, and access arbitration. Channels are organized into *sectors*, the basic Iocast coverage unit. A sector may be as small as a building or campus, or as large as a county, and may include multiple channels and base transceivers. Nodes must *connect* to a sector, authenticating both the node and the home system, before transmitting or receiving datagrams.

1.1 Benefits

- Flexible use of narrowband channels
- Synchronous, deterministic protocol with carrier-grade MAC layer
- Variable latency and power consumption configurable per node
- Unicast and multicast datagrams
- 10-mile (nominal) coverage radius per base transceiver
- Node mobility and secure roaming
- Node authentication and security using shared-key AES-128 encryption
- Hundreds to millions of nodes per base transceiver

1.2 Elements

1.2.1 System

An Iocast system includes a control stack plus one or more base transceivers, and it provides Iocast coverage over a defined area or set of areas. A system conceptually owns a set of node transceivers, with which it communicates and controls using a shared private key. An Iocast system is identified by its system ID.

1.2.2 Application Server

An application server is a software or service platform. An application server connects to an Iocast system using the Iocast API, and it communicates with a subset of that system's nodes.

1.2.3 Node

An Iocast node is a wireless object that connects to an Iocast system. Nodes conceptually belong to one system and one application server Nodes use a node transceiver to exchange datagrams with their system, and through that system with their application server. Example nodes include weather sensors, water level sensors, intrusion alarms, utility meters, and personal notification devices.

1.2.4 Node Transceiver (NX)

Node transceivers (NXs) are digital radio modems that connect nodes to an Iocast network. A node transceiver is globally, uniquely identified by its 64-bit node transceiver ID (nxid). Each node transceiver is bound to one *home system* by a shared private key, which facilitates communication to one application server. Nodes transceivers may only communicate with their home system; while nodes may roam onto other host systems, a host system simply provides a tunnel between the roaming node and its home system. Each node is configured with one 32-bit primary address and up to 16 32-bit multicast addresses. Node transceivers operate with a *node availability* value (*na*), which describes how often the node transceiver listens for forward datagrams. This value ranges from 0 to 9, with lower values receiving datagrams more often, and higher values requiring less average power. Node transceivers may be implemented as a physical module, or they may be tightly integrated into a node at a hardware and software level. Node transceivers are configured over-the-air, securely, by their home system.

1.2.5 Group

Groups are sets of nodes sharing a common multicast address. A group is identified by its multicast address and symbolic name, and it is bound to one *home system* by a shared private key. Group member nodes receive, and may reply to, datagrams sent to the group's multicast address.

1.2.6 Datagram

Iocast communications are based on reliable datagrams, binary messages transmitted between nodes and their application servers. Application servers send *forward datagrams* to single nodes (unicast) or to groups of nodes (multicast). Nodes send *reverse datagrams* to their application servers. Datagrams are further divided into *unsolicited datagrams* and *response datagrams*. Response datagrams are sent in response to a prior datagram, while unsolicited datagrams have no such association. Datagrams range in size from 1 byte to 8,160 bytes; however, size may be further restricted on a per-node or per-group basis.

1.2.7 Control Stack

A control stack is the real-time software and database required to support the Iocast air protocol and Iocast API. The control stack includes subscriber data, node connection data, datagram queues, base transceiver control, administrative interfaces, and APIs. The control stack, together with base transceivers, forms the fixed part of an Iocast system.

1.2.8 Base Transceiver (BX)

An Iocast base transceiver is a powerful, fixed radio that transmits data to node transceivers on forward channels and receives data from nodes transceivers on reverse channels. Base transceivers connect to a control stack using the Base Transceiver Interface (BXI), and are identified within a sector by their base transceiver ID.

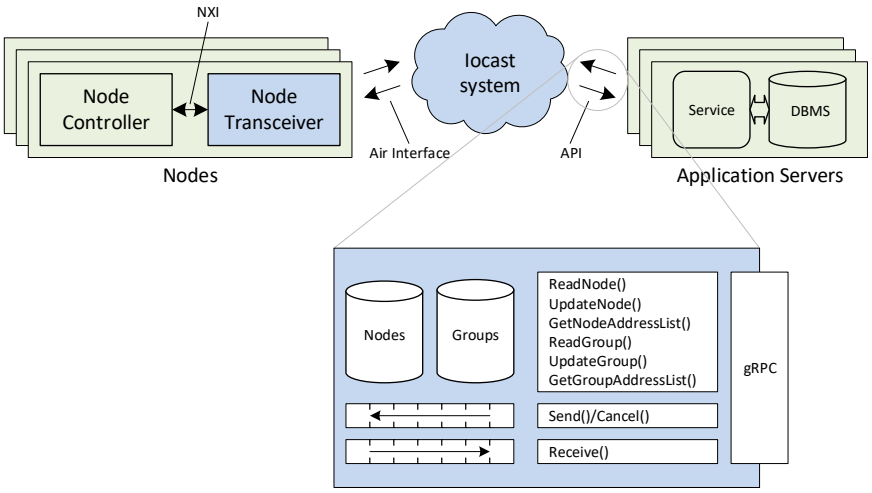
1.2.9 Channel

An Iocast channel is a narrowband RF channels used to transmit datagrams and control information. Iocast channels include *forward channels* and *reverse channels*.

1.2.10 Sector

A sector describes a geographical area of coverage, including one or more base transceivers and two or more channels, under common control of one control stack. A node must connect to a sector before sending and receiving datagrams. Iocast Sectors are identified by their Sector ID.

2 General Description



The Iocast API is based on gRPC and allows application servers (clients) to connect to Iocast systems (servers). Once connected, clients send datagrams, receive datagrams, and manage their nodes and groups.

The Iocast API includes 11 gRPC methods. The *iocast-api.proto* file can be found here.

2.1 API Scope

The Iocast API is designed enable communication between an application server and its nodes, including limited support of node and group configuration toward that end. The API does not provide the ability to manage an Iocast system or provision nodes; these functions require other product-specific interfaces and processes.

2.2 Communication Methods

The API includes three communication methods, which enable the client to send datagrams to nodes and groups, and to receive datagrams from nodes.

- **Send**
- **Cancel**
- **Receive**

2.3 Management Methods

The API includes six management methods, which enable the client to read and update node and group records, and to correspondingly reconfigure notes over-the-air.

- **ReadNode**
- **UpdateNode**
- **GetNodeAddressList**
- **ReadGroup**
- **UpdateGroup**
- **GetGroupAddressList**
- **Monitor**

2.4 Channel Information Method

The API includes one additional method, which enables a client to determine its roles and the behavior of the channel.

- **GetChannelInfo**

2.5 Application Servers

Application servers are back-end systems which communicate with nodes through the Iocast API. Application servers are *clients* in the API model, and Iocast systems are *servers*. Each application server conceptually "owns" a set of nodes and groups with which it may exchange datagrams. Each application server is assigned one or more API Keys, potentially with different roles. An application server may use one channel, or it may be partitioned into multiple processes with multiple channels, but each channel requires a unique API Key.

2.6 API Key

Each gRPC channel requires an API Key, which identifies the application server making the connection as well as certain characteristics of the channel itself. Each API Key has a *communication role* and a *management role*, which determine the API methods available to the client, as well as how the client communicates with its nodes and groups. An API Key may be used by only one client for one gRPC channel at any given time, although a client may have multiple active API keys. For each client, exactly one key must be assigned with the COMMUNICATION_ROOT role.

2.6.1 Management Roles

- **MANAGEMENT_NONE**
  The client may not use any management methods.

- **MANAGMENET_READ_ONLY**
  The client may not use the **UpdateNode** or **UpdateGroup** methods, but may use other management methods.

- **MANAGEMENT_READ_WRITE**
  The client may use any management method.

2.6.2 Communication Roles

- **COMMUNICATION_NONE**
  The client may not use any communications methods.

- **COMMUNICATION_SECONDARY**
  The client may use any communications method, but unsolicited reverse datagrams are not returned through the Receive method.

- **COMMUNICATION_ROOT**
  The client may use any communications method, and unsolicited reverse datagrams are returned through the **Receive** method. Only one API Key per client may be configured with this role.

2.7 Quota

Iocast allows for optional *quotas*, which describe how many datagrams and bytes per day may be sent to and from a node or sent to a group. If quotas are used, a **Quota** message type is included in several method responses, describing the total and remaining available datagram and byte counts.

```
message QuotaInfo {
    message Quota {
        uint32 datagram_count = 1;
        uint32 byte_count = 2;
    }

    Quota total = 1;
    Quota remaining = 2;
    uint32 max_datagram_bytes = 3;
}
```

2.8 Datagram ID

Datagrams are identified by their Datagram ID, a concatenation of their destination (forward message) or source (reverse message) address, and their datagram number. Several messages include a nested **DatagramID** message, which includes an **address** and **number** fields to identify these two fields. The **address** field contains the MAC address of a node or group, and the **number** field contains a 32-bit datagram number, assigned by the server in the case of reverse messages, and assigned by the client in the case of forward messages. Forward and reverse datagrams use different namespaces for **number**, which may overlap.

```
message DatagramID {
    uint32 address = 1;
    uint32 number = 2;
}
```

2.9 Timestamp

Several API messages include a **timestamp** field. These fields contain a timestamp, expressed as the number of whole seconds since midnight, January 1, 2019, in GPS time.

2.10 gRPC Status Codes

In gRPC implementations, each method call returns a status code (see, Error Handling) indicating top level success or failure of the call. Several of these codes have specific meaning within the Iocast API, as follows:

gRPC Status Code	Iocast API Meaning
GRPC_STATUS_OK	The method completed successfully.
GRPC_STATUS_NOT_FOUND	The specified node, group, or datagram record does not exist.
GRPC_STATUS_INVALID_ARGUMENT	A request message is invalid.
GRPC_STATUS_ALREADY_EXISTS	An active, forward datagram already exists with the same ID.
GRPC_STATUS_PERMISSION_DENIED	The management or communication role does not permit the method call.
GRPC_STATUS_ABORTED	Optimistic locking failed during an update method because of a stale row.
GRPC_STATUS_RESOURCE_EXHAUSTED	The Send method failed due to a group or node quota violation.
GRPC_STATUS_UNAVAILABLE	An update method failed because the node or group is currently busy.

3 API Methods

3.1 Send

The **Send** method sends a forward datagram to a node or group. The client passes a **SendRequest** message to the method, and the method returns a **SendResponse** message to the client. Iocast is primarily a store and forward technology, and the **Send** method queues the datagram at the server for future delivery. After a successful **Send** method call, the client can use Receive to track the datagram's progress, or the **Cancel** method to delete it from the queue.

3.1.1 SendRequest

```
message SendRequest {
    DatagramID forward_datagram_id = 1;
    bool encrypt = 2;
    uint32 priority = 3;
    bytes data = 4;
}
```

The **SendRequest** message describes a forward datagram and the destination to which it should be sent, including the following fields:

- **forward_datagram_id**
  This field specifies the destination address (node or group MAC address) and datagram number of the forward datagram ID, which will be used by future **Cancel** or **Receive** method responses to identify the datagram.

- **encrypt**
  This field specifies whether the Iocast system should encrypt the message. Note that the client may implement its own encrypt method the in addition to, or instead of, the Iocast encryption system.

- **priority**
  This field specifies the relative scheduling priority of the datagram, with 0 being the lowest priority and 15 being the highest priority.

- **data**
  This field contains the forward datagram payload data.

3.1.2 SendResponse

```
message SendResponse {
    uint32 estimated_delivery = 1;
    bool quota_violation = 2;
    QuotaInfo quota = 3;
}
```

Upon successful completion, the **Send** method responds with a **SendResponse** message, which includes the following fields:

- **estimated_delivery_timestamp**
  This field specifies when the node is expected to receive the datagram, using the timestamp format described in section 2.9.

- **quota_violation**
  The method call has violated the quota of the destination node or group. The datagram has been queued, but may be delayed or sent at a lower priority.

- **quota**
  If present, this field describes the quota state of the destination node or group after accounting for the present datagram.

3.2 Cancel

The **Cancel** method cancels a forward datagram sent previously using the **Send** method. The client passes a **CancelRequest** message to the method, and the method returns a **CancelResponse** message to the client.

3.2.1 CancelRequest

```
message CancelRequest {
    DatagramID forward_datagram_id = 1;
}
```

The **CancelRequest** message includes a **forward_datagram_id** of the datagram to cancel, which must match the **forward_datagram_id** fields of the **SendRequest** used to originally queue the datagram.

3.2.2 CancelResponse

```
message CancelResponse {
}
```

Upon successful completion, the **Cancel** method responds with a **CancelResponse** message, which contains no additional information.

3.3 Receive

The **Receive** method waits for reverse datagrams and forward datagram progress updates. The client passes a **ReceiveRequest** message to the method, and the method returns a stream of **ReceiveResponse** messages to the client.

3.3.1 ReceiveRequest

```
message ReceiveRequest {
    uint64 last_sequence_number = 1;
}
```

The **ReceiveRequest** message contains one field, **last_sequence_number**, which specifies the sequence number of the last event received by the client in earlier calls. If unknown, this value should be set to zero. The server will release resources tied to events with this and earlier serial numbers, and events in the **ReceiveResponse** message will start with the earliest available sequence number after this value.

3.3.2 ReceiveResponse

```
message ReceiveResponse {
    message Event {
        message ReverseDatagram {
            DatagramID reverse_datagram_id = 1;
            DatagramID forward_datagram_id = 2;
            bool encrypted = 3;
            bytes data = 4;
        }

        message ForwardProgress {
            enum Code {
                QUEUED = 0;
                CANCELLED = 1;
                TRANSMITTING = 2;
                TRANSMITTED = 3;
                DELIVERED = 4;
                DELIVERY_TIMEOUT = 5;
                DELIVERY_ERROR = 6;
            };

            DatagramID forward_datagram_id = 1;
            bool final = 2;
            Code code = 3;
        }

        uint32 timestamp = 1;

        oneof event {
            ReverseDatagram reverse_datagram = 2;
            ForwardProgress forward_progress = 3;
        }
    }

    uint64 first_sequence_number = 1;
    repeated Event events = 2;
}
```

The **Receive** method returns a stream of **ReceiveResponse** messages. Each message includes two fields, **first_sequence_number** and **events**. The **first_sequence_number** field contains the sequence number of the first item in the **events** field, with subsequent items numbered in ascending order. The events field contains a list of **Event** messages, each containing a **timestamp** field (2.8), plus one of two possible event fields: **reverse_datagram** or **forward_progress**. The method will return after a limited period of time or a limited number of events, requiring the client to call the method repeatedly to receive a continuous flow of events.

3.3.2.1 reverse_datagram

If present, the **reverse_data** field informs the client that a reverse datagram has been received from a node. This field contains a nested **ReverseDatagram** message, which includes the following fields:

- **reverse_datagram_id**
  This field contains the source address and the server-assigned datagram number (2.8).

- **forward_datagram_id**
  If present, the reverse datagram is a response to a forward datagram identified by the field. If this field is absent, the reverse datagram is unsolicited.

- **encrypted**
  This field indicates the datagram was encrypted by the node transceiver and decrypted by the system.

- **data**
  This field contains the reverse datagram payload data,

3.3.2.2 forward_progress

If present, the **forward_progress** field informs the client that progress has been made transmitting a forward datagram to a node or group. This field contains a nested **ForwardProgress** message, which includes the following fields:

- **forward_datagram_id**
  This field identifies the forward datagram, including the destination address datagram number (2.8).

- **final**
  This field is set to true to reflect the final progress event on this message.

- **code**
  This field contains the progress event code.

3.4 ReadNode

The **ReadNode** method reads information from one or more node records. The client passes a **ReadNodeRequest** message to the method, and the method returns a stream of **ReadNodeResponse** messages to the client.

3.4.1 ReadNodeRequest

```
message ReadNodeRequest {
    repeated uint32 node_addresses = 1;
}
```

The client passes a **ReadNodeRequest** message to the method, which includes one or more addresses in the **node_addresses** field, each of which indicates the primary address of a node being read.

3.4.2 ReadNodeResponse

```
message ReadNodeResponse {
    message Node {
        message Record {
            enum ConnectionState {
                DISCONNECTED = 0;
                CONNECTED = 1;
                LOST = 2;
            }

            enum ConfigurationState {
                CURRENT = 0;
                RECONFIGURING = 1;
                PROBLEM = 2;
            }

            message GroupMembership {
                uint32 group_addresses = 1;
                bool inhibit_response = 2;
            }

            uint64 version = 1;
            string esn = 2;
            repeated GroupMembership groups = 3;
            bool disable = 4;
            uint32 availability = 5;
            ConnectionState connection = 6;
            ConfigurationState configuration = 7;
            QuotaInfo quota = 8;
            uint32 system_id = 9;
            uint32 sector_id = 10;
            uint32 estimated_delivery_timestamp = 11;
            repeated uint32 datagram_queue = 12;
        }

        uint32 node_address = 1;
        Record node_record = 2;
    }

    repeated Node nodes = 1;
}
```

The **ReadNode** method responds with a **ReadNodeResponse** message, which includes one repeated field **nodes**, containing one **Node** message for each address value in the **node_addresses** field of the **ReadNodeRequest** message. Each **Node** message includes two fields, **node_address** and **node_record**. The **node_address** field specifies the MAC address of the node. The **node_record** field, if present, contains a nested **Record** message describing the node record. If the **node_record** field is not present, the node does not exist on the server. The nested **Record** message includes the following fields:

- **version**
  This field contains the version of the node's underlying database row. Update methods employ optimistic locking; when calling the method, the client should include the same **version** value as the most recent **ReadNode** call.

- **nxid**
  This field contains the node transceiver unique identifier, a 64-bit code-coordinated number that uniquely identifies each transceiver.

- **groups**
  This field contains the list of groups programmed into the node transceiver. Each item in this list includes a group address field **group_address**, and an **inhibit_response** field. If the **inhibit_response** field is set to true, the node may not reply to datagrams sent to the group's multicast address.

- **disable**
  If set to true, this field indicates the node is disabled, and its transceiver cannot send or receive datagrams.

- **availability**

This field describes the node's availability, or how often it wakes up to listen for forward datagrams. This value ranges from 0 to 9.

- **connection**
  This field describes the network-side state of the node, either *CONNECTED*, *DISCONNECTED*, or *LOST*.

- **configuration**
  This field describes the configuration state of the node, either CURRENT, RECONFIGURING, or PROBLEM.

- **quota**
  This field describes the current state of the node's quota.

- **system_id**
  If **connection** is *CONNECTED* or *LOST*, then this field describes the system to which the node is connected. If **connection** is *DISCONNECTED*, then this value is omitted.

- **sector_id**
  If **connection** is *CONNECT* or *LOST*, then this field identifies the sector to which the node is connected. If **connection** is *DISCONNECTED*, then this value is omitted.

- **estimated_delivery_timestamp**
- This field specifies when the node is expected to receive the datagram, using the timestamp format described in section 2.9.

- **datagram_queue**
  This field lists the forward datagrams in queue for the node's MAC address. The list includes the datagram number portion of the forward datagram ID (2.8) of each datagram, starting with the first datagram in queue and ending with the last datagram in queue.

3.5 UpdateNode

The **UpdateNode** method updates a system node record. The client passes an **UpdateNodeRequest** message to the method, and the method returns an **UpdateNodeResponse** message to the client. Note that this method requires the client have a *MANAGEMENT_READ_WRITE* role (2.6.1), and its use may result in over-the-air programming activity to synchronize configuration of the node transceiver.

3.5.1 UpdateNodeRequest

```
message UpdateNodeRequest {
    message UpdateRecord {
        message Group {
            uint32 group_addresses = 1;
            bool inhibit_response = 2;
        }

        uint64 version = 2;
        repeated Group groups = 3;
        bool disable = 4;
        uint32 availability = 5;
    }

    message CancelAllDatagrams {
    }

    message Reconfigure {
    }

    uint32 node_address = 1;

    oneof node_action {
        UpdateRecord update_record = 2;
        CancelAllDatagrams cancel_all_datagrams = 3;
        Reconfigure reconfigure = 4;
    }
}
```

The client passes an **UpdateNodeRequest** message to the **UpdateNode** method, which includes two fields: **node_address,** followed by either **update_record**, **cancel_all_datagrams**, or **reconfigure** indicating how to update the node record.

3.5.1.1 update_record

If present, the **update_record** field instructs the server to update the underlying node record. This field contains a nested **UpdateRecord** message, which includes the following fields:

- **version**
  The **UpdateNode** method employs optimistic row locking, and the **version** field must match the current version of the node row for the method to complete successfully. This field should match the **version** returned by the most recent **ReadNode** method call. If **version** does not match the current version of the node's underlying row, the method will fail with a result code of **GRPC_STATUS_ABORTED**. If the method is successful, the record's new version field will be incremented by 1.

- **groups**
  This field contains the list of groups (multicast addresses) programmed into the node transceiver. Each item in this list includes a group address value, and an **inhibit_response** field. If the **inhibit_response** field is set to true, the node may not reply to datagrams sent to the group's multicast address.

- **disable**
  If set to true, this field disables the node transceiver's ability to send or receive datagrams.

- **availability**
  This field describes the node's availability, or how often it wakes up to listen for forward datagrams. This value ranges from 0 to 9.

3.5.1.2 cancel_all_datagrams

If present, the **cancel_all_datagrams** field instructs the server to cancel all datagrams in the node's datagram queue. Only datagrams still in queue will be reliably cancelled; the server may not be able to cancel datagrams already in the transmission process.

3.5.1.3 reconfigure

This field's presence schedules an over-the-air resynchronization of the node's transceiver.

3.5.2 UpdateNodeResponse

```
message UpdateNodeResponse {
    message DatagramResult {
        enum Result {
            CANCELLED = 0;
            NOT_CANCELLED = 1;
        }

        uint32 datagram_id = 1;
        Result result = 2;
    }

    repeated DatagramResult datagram_results = 1;
}
```

Upon successful completion, the **UpdateNode** method responds with an **UpdateNodeResponse** message, which may contain a **datagram_results** field showing the disposition of a **cancel_all_datagrams** field in the **UpdataNodeRequest** message. If present, the **datagram_results** field contains a list datagrams, each described using the of the datagram number portion of the datagram ID (2.8), and whether the datagram was successfully cancelled.

3.6 GetNodeAddressList

The **GetNodeAddressList** method returns a list of node addresses, describing either all nodes belonging to a particular group, or all nodes belonging to the client. The client passes **a GetNodeAddressListRequest** message to the method, and the method returns a stream of **GetNodeAddressListResponse** messages to the client.

3.6.1 GetNodeAddressListRequest

```
message GetNodeAddressListRequest {
    uint32 group_address = 1;
}
```

The client passes a **GetNodeAddressListRequest** message to the **GetNodeAddressList** method, which includes a **group_address** field. The **group_address** field specifies the group to with the returned nodes must belong, or a 0 to indicate all nodes should be returned.

3.6.2 GetNodeAddressListResponse

```
message GetNodeAddressListResponse {
    repeated uint32 node_addresses = 1;
}
```

The **GetNodeAddressList** method returns a stream of **GetNodeAddressListResponse** messages. Each **GetNodeAddressListResponse** message contains one field, **node_addresses**, containing a list of node addresses. Upon successful completion, the method will return complete set of node addresses prior to the end of the stream.

3.7 ReadGroup

The **ReadGroup** method returns information from a system group record. The client passes a **ReadGroupRequest** message to the method, and the method returns a stream of **ReadGroupRes** messages to the client.

3.7.1 ReadGroupRequest

```
message ReadGroupRequest {
    uint32 group_address = 1;
    bool include_membership = 2;
}
```

The client passes a **ReadGroupRequest** message to the **ReadGroup** method, which includes a **group_address** field to identify the desired group record, and an **include_membership** field to specify whether the method should include membership information.

3.7.2 ReadGroupResponse

```
message ReadGroupResponse {
    message Record {
        uint64 version = 1;
        string name = 2;
        QuotaInfo quota = 3;
        repeated uint32 datagram_queue = 4;
    }

    message Member {
        uint32 node_addresses = 1;
        bool node_configuration_pending = 2;
        bool inhibit_response = 3;
    }

    uint32 group_address = 1;
    Record group_record = 2;
    repeated Member group_members = 3;
}
```

The **ReadGroup** method returns a stream of **ReadGroupResponse** messages describing one group. Each **ReadGroupResponse** message contains the following fields

- **group_address**
  This field contains the group address, which matches the **group_address** field of the **ReadGroupRequest** message passed to the method.

- **group_record**
  This field contains a nested **Record** message, describing the core group record. If more than one **ReadGroupResult** message is required to complete the method, this field only appears in the first message of the stream.

- **group_members**
  This field contains a sequence of membership records, each containing the node address, a flag indicating whether the node is permitted to respond to group messages, and a flag indicating whether the node's configuration is presently synchronized with the system. For large groups, multiple **ReadGroupResponse** messages may be returned in the stream, each including a **group_membership** field.

3.7.2.1 group_record

If present, the **group_record** field contains a nested **Record** message with the following fields describing the group record:

- **version**
  The **UpdateGroup** method employs optimistic row locking, and the **version** field must match the current version of the group row for the method to complete successfully. This field should match the **version** returned by the most recent **ReadGroup** method call. If **version** does not match the current version of the group's underlying record, the method will fail with a result code of **GRPC_STATUS_ABORTED**. If the method is successful, the record's new version field will be incremented by 1.

- **name**
  This field contains a symbolic name of the group.

- **quota**
  If present, this field describes the current state of the group's quota.

- **datagram_queue**
  This field lists the forward datagrams in queue for the node's MAC address. The list includes the datagram number portion of the forward datagram ID (2.8) of each datagram, starting with the first datagram in queue and ending with the last datagram in queue.

3.7.2.2 group_members

If present, the **group_members** field contains a list of nested **Member** messages, each describing a member of the group. The Member message includes the following three fields:

- **node_addresses**
  This field contains the member node primary address.

- **node_configuration_pending**
  If set to true, this field indicates the node is awaiting over-the-air configuration programming concerning the group.

- **inhibit_response**
  If set to true, this field indicates that the member node may not respond to datagrams sent to the group.

3.8 UpdateGroup

The **UpdateGroups** method updates a system group record. The client passes an **UpdateGroupRequest** message to the method, and the method returns an **UpdateGroupResponse** message to the client. Note that this method requires the client have a *MANAGEMENT_READ_WRITE* role (2.6.1), and its use may result in significant over-the-air programming activity to synchronize configuration of multiple node transceivers.

3.8.1 UpdateGroupRequest

```
message UpdateGroupRequest {
    message UpdateRecord {
        uint64 version = 1;
        string name = 2;
    }

    message CancelAllDatagrams {
    }

    message AddMembers {
        message Member {
            uint32 node_addresses = 1;
            bool inhibit_response = 2;
        }

        repeated Member members = 1;
    }

    message RemoveMembers {
        repeated uint32 members = 1;
    }

    message RemoveAllMembers {
    }

    uint32 group_address = 1;

    oneof group_action {
        UpdateRecord update_record = 2;
        CancelAllDatagrams cancel_all_datagrams = 3;
        AddMembers add_members = 4;
        RemoveMembers remove_members = 5;
        RemoveAllMembers remove_all_members = 6;
    }
}
```

The client passes an **UpdateGroupRequest** message to the **UpdateGroup** method, which includes two fields: **group_address,** followed by either **update_record**, **cancel_all_datagrams**, **add_members**, **remove_members**, or **remove_all_members** indicating how to update the node record.

3.8.1.1 update_record

If present, the **update_record** field instructs the server to update the underlying group record. This field contains a nested **UpdateRecord** message, which includes the following fields:

- **version**
  The **UpdateGroup** method employs optimistic row locking, and the **version** field must match the current version of the group row for the method to complete successfully. This field should match the **version** returned by the most recent **ReadGroup** method call. If **version** does not match the current version of the groups's underlying record, the method will fail with a result code of **GRPC_STATUS_ABORTED**. If the method is successful, the record's new version field will be incremented by 1.

- **name**
  This field contains the symbolic name of the group.

3.8.1.2 cancel_all_datagrams

If present, the **cancel_all_datagrams** field instructs the server to cancel all datagrams in the group's datagram queue. Only datagrams in queue will be reliably cancelled; the server may not be able to cancel datagrams already in the transmission process.

3.8.1.3 add_members

If present, the **add_members** field instructs the server to add nodes in the **members** field to the group. Each element in the list includes a **node_address** and **inhibit_response** field identifying nodes to add to the group, or to update if they are already members of the group. Nodes added to the group in this way have their record modified, with the group added to the end of their **groups** field.

3.8.1.4 remove_members

If present, the **remove_members** field instructs the server to remove nodes in the **members** field from the group. Nodes removed to the group in this way have their record modified.

3.8.1.5 remove_all_members

If present, the **remove_all_members** field instructs the server to remove all nodes from the group. Nodes removed from the group in this way have their record modified.

3.8.2 UpdateGroupResponse

```
message UpdateGroupResponse {
    message MembershipResult {
        enum Result {
            SUCCESS = 0;
            NODE_NOT_FOUND = 1;
            NODE_GROUP_OVERFLOW = 2;
            NODE_BUSY = 3;
        }

        uint32 node_address = 1;
        Result result = 2;
    }

    message DatagramResult {
        enum Result {
            SUCCESS = 0;
            FAILED = 1;
        }

        uint32 datagram_id = 1;
        Result result = 2;
    }

    repeated MembershipResult membership_results = 1;
    repeated DatagramResult datagram_results=2;
}
```

The **UpdateGroup** method responds with an **UpdateGroupResponse** message, which may contain two fields, **membership_results** and **datagram_results**. If present, the **membership_results** field contains a list of nodes and whether they were successfully added or removed from the group. If present, the **datagram_results** field contains a list datagrams, each described using the of the datagram number portion of the datagram ID (2.8), and whether the datagram was successfully cancelled.

3.9 GetGroupAddressList

The **GetGroupAddressList** method returns a list of group addresses, describing all nodes belonging to the client. The client passes a **GetGroupAddressListRequest** message to the method, and the method returns a stream of **GetGroupAddressListResponse** messages to the client.

3.9.1 GetGroupAddressListRequest

```
message GetGroupAddressListRequest {
}
```

The client passes a **GetGroupAddressListRequest** message to the **GetGroupAddressList** method, which contains no fields.

3.9.2 GetGroupAddressListResponse

```
message GetGroupAddressListResponse {
    repeated uint32 group_addresses = 1;
}
```

The **GetGroupAddressList** method returns a stream of **GetGroupAddressListResponse** messages. Each message includes one field, **group_addresses**, which contains a list of 32-bit group MAC addresses Upon successful completion, a complete set of group addresses will be returned prior to the end of the stream.

3.10 Monitor

The **Monitor** method waits for server-driven events concerning changes to node and group records. In architectures where the application server (client) maintains its own database, the Monitor method is intended to help the client synchronize the relevant tables of its database to those of the server. The client passes a **MonitorRequest** message to the method, and the method returns a stream of **MonitorResponse** messages to the client.

3.10.1 MonitorRequest

```
message MonitorRequest {
    uint64 last_sequence_number = 1;
    bool purge = 2;
}
```

The **MonitorRequest** message describes which management events the server should return to the client and how long the server should wait for them. It includes including the following fields:

- **last_sequence_number**
  This field specifies the sequence number of the last event received by the client in earlier calls. If unknown, this value should be set to zero. The server will release resources tied to events with this and earlier serial numbers, and events in the **MonitorResponse** message will start with the earliest available sequence number after this value.

- **purge**
  This field tells the server to purge the monitor event cache. The first event returned in the response stream will be the purge event.

3.10.2 MonitorResponse

```
message MonitorResponse {
    message Event {
        message NodeUpdate {
            uint32 node_address = 1;
            uint64 version = 2;
            bool membership = 3;
            bool queue = 4;
            bool quota = 5;
            bool connection = 6;
            bool configuration = 7;
            bool availability = 8;
        }

        message NodeAdd {
            uint32 node_address = 1;
        }

        message NodeDelete {
            uint32 node_address = 1;
        }

        message GroupUpdate {
            uint32 group_address = 1;
            uint64 version = 2;
            bool membership = 3;
            bool queue = 4;
            bool quota = 5;
        }

        message GroupAdd {
            uint32 group_address = 1;
        }

        message GroupDelete {
            uint32 group_address = 1;
        }

        message Marker {
        }

        uint32 timestamp = 1;

        oneof event {
            NodeUpdate node_update = 2;
            NodeAdd node_add = 3;
            NodeDelete node_delete = 4;
            GroupUpdate group_update = 5;
            GroupAdd group_add = 6;
            GroupDelete group_delete = 7;
            Marker overrun = 8;
            Marker purge = 9;
        }
    }

    uint64 first_sequence_number = 1;
    repeated Event events = 2;
}
```

The **Monitor** method returns a stream of **MonitorResponse** messages. Each message includes two fields, **first_sequence_number** and **events**. The **first_sequence_number** field contains the sequence number of the first item in the **events** field, with subsequent items numbered in ascending order. The **events** field contains a list of nested **Event** messages, each containing a **timestamp** field (2.8), plus one of eight possible event fields: **node_update**, **node_add**, **node_delete**, **group_update**, **group_add**, **group_delete**, **overrun**, or **purge**. The method will return after a limited period of time or a limited number of events, requiring the client to call the method repeatedly to receive a continuous flow of events.

3.10.2.1 node_update

A **node_update** field indicates a node record has been updated. This field contains a nested **NodeUpdate** message, which includes an **address** field (node primary address), and a **version** field (record version immediately after the change). Additionally, the **Node** message contains several additional fields describing how the record changed:

- **membership**
  A value of true means the node's **groups** field has changed.

- **queue**
  A value of true means the node's **datagram_queue** has changed.

- **quota**
  A value of true means the node's **quota** field has changed.

- **connection**
  A true value means the node's **connection**, **system_id**, or **sector_id** fields have changed.

- **configuration**
  A true value means the node's **configuration** field has changed.

- **availability**
  A true value means the node's **availability** field has changed.

3.10.2.2 node_add

A **node_add** field indicates that the system has added a node belonging to the client.

3.10.2.3 node_delete

A **node_delete** field indicates the server has deleted a node belonging to the client.

3.10.2.4 group_update

A **group_update** field indicates a group record has been updated. This field contains a nested **Group** message, which includes an **address** field (group MAC address), and a **version** field (record value immediately after the change). Additionally, the **Group** message contains several additional fields describing how the record changed:

- **membership**
  A value of true means the group's membership has changed.

- **queue**
  A value of true means the group's **datagram_queue** has changed.

- **quota**
  A value of true means the group's **quota** field has changed.

3.10.2.5 group_add

A **group_add** field indicates that the system has added a node belonging to the client.

3.10.2.6 group_delete

A **group_delete** field indicates the server has deleted a node belonging to the client.

3.10.2.7 overrun

An **overrun** field indicates the monitor event internal FIFO has overrun. The presence of this event means some number of events have been lost, and a discontinuity may appear in the event sequence numbers.

3.10.2.8 purge

A **purge** field indicates that the monitor event internal FIFO has been purged. The presence of this event means some number of events have been lost, and a discontinuity may appear in the event sequence numbers.

3.11 GetChannelInfo

The **GetChannelInfo** method returns information regarding the channel. The client passes a **MonitorRequest** message to the method, and the method returns a **MonitorResponse** message to the client.

3.11.1 GetChannelInfoRequest

```
message GetChannelInfoRequest {
}
```

The client passes a **GetChannelInfoRequest** message to the **GetChannelInfo** method, which contains no fields.

3.11.2 GetChannelInfoResponse

```
message GetChannelInfoResponse {
    enum CommunicationRole {
        COMMUNICATION_NONE = 0;
        COMMUNICATION_SECONDARY = 1;
        COMMUNICATION_ROOT = 2;
    }

    enum ManagementRole {
        MANAGEMENT_NONE = 0;
        MANAGEMENT_READ_ONLY = 1;
        MANAGEMENT_READ_WRITE = 2;
    }

    message FifoState {
        uint32 length = 1;
        uint64 oldest_sequence_number = 2;
        uint64 newest_sequence_number = 3;
    }

    CommunicationRole communication_role = 1;
    ManagementRole management_role = 2;
    FifoState monitor_fifo_state = 3;
    FifoState receive_fifo_state = 4;
    uint32 timestamp = 5;
    int32 utc_correction = 6;
}
```

The **GetChannelInfo** method returns a **GetChannelInfoResponse** message, which includes four fields describing the channel:

- **communication_role**
  This field contains the communication role assigned to the client (2.6.2).

- **management_role**
  This field contains the management role assigned to the client (2.6.1).

- **monitor_fifo_state**
  This field specifies the number of events currently in the channel's monitor queue, along with the oldest and newest sequence number.

- **receive_fifo_state**
  This field specifies the number of events currently in the channel's receive queue, along with the oldest and newest sequence number.

- **timestamp**
  This field contains the current GPS time, measured by the server at the start of the method call, reported as per section 2.9.

- **utc_correction**
  This field contains the current correction between GPS time and UTC time, measured by the server at the start of the method call.