iocast

Application Programming Interface (API)

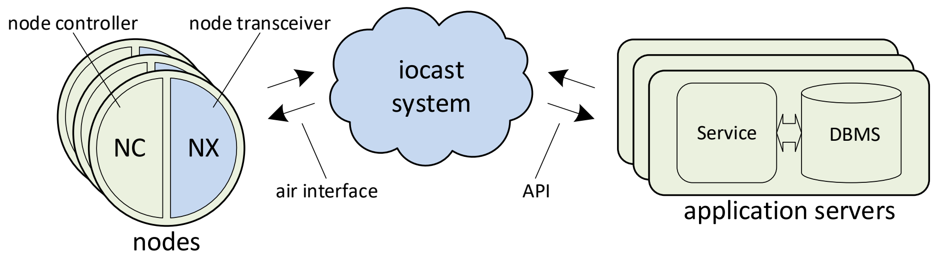
This document specifies the Iocast **Application Programming Interface** (API). Using this API, an application server or other system may connect to an iocast network, communicate with nodes and groups, and perform system management functions.

Document Number: 19-022
Iocast Version 3
Document Version: 3.2
Date: 02/07/2021
Author: James M Dabbs III

CriticalResponse
Critical Response Systems, Inc.
One West Court Square Suite 750
Decatur, GA 30030-2545
www.criticalresponse.com

Document History		
Version	**Date**	**Changes**
3.0	01/01/2021	Streamlined and simplified, harmonized API with encompassing *Iocast version 3.*
3.1	01/04/2021	Editing and clean-up
3.2	02/07/2021	Clarified and detailed control stack description; removed *packed* attributes from proto file references.

1 Iocast Overiew



Iocast is a wireless, wide-area network architecture that operates over narrowband radio channels. An iocast system includes *application servers*, *nodes*, *base transceivers*, and a *control stack*, and it enables nodes and application servers to exchange *datagrams* with each other. Nodes (low-power sensors, controllers, tracking devices, etc.) connect to an iocast network using wireless *node transceivers*, while application servers connect using an *API*. An application server may send datagrams to a single node, or to groups of nodes using multicast addresses. Nodes may respond to datagrams and initiate their own datagrams.

The iocast *air protocol* divides RF channels into *forward channels* and *reverse channels*, and organizes them together into geographical *sectors*. Fixed, high-power base transceivers transmit data to mobile node transceivers using forward channels, while node transceivers transmit data to base transceivers using reverse channels. Reverse channels are time-shared between node transceivers, while forward channels are “always on,” under control of base transceivers. Forward and reverse channels are universally synchronized together using a GPS time base, with control stacks providing coordination, timing, and RF access arbitration for the sectors they operate.

Iocast sectors may be as small as a building or campus, or as large as a state or region, and may include multiple channels and base transceivers. Nodes make a secure connection to a specific sector before transmitting or receiving datagrams. Nodes may be fixed or mobile, low-energy or high performance, and they may roam between interconnected systems.

1.1 Benefits

- Supports *low-energy* as well as *high-performance* nodes.
- Uses narrowband RF channels and a carrier-grade MAC layer.
- Includes bidirectional unicast and multicast datagrams.
- Supports node authentication, mobility, and secure roaming.
- Supports hundreds to billions of nodes per sector.

1.2 Components

1.2.1 System

A system includes a control stack and base transceivers (BXs). A system conceptually owns a set of nodes and sectors and is identified by a unique 32-bit system ID (sysid).

1.2.1.1 Control Stack

A control stack is the “brain” of a network, the real-time software and database required to control sectors and support application server clients. The stack includes two software components, the *home controller* and the *sector controller*. The home controller is roughly analogous to a cellular network’s *Home Location Register* (HLR), while the sector controller is roughly analogous to a *Mobile Switching Center* (MSC).

A control stack may operate as a single instance on a single server or container, or as a distributed system on a server cluster or cloud platform. Furthermore, home controllers may have many-to-many relationships with sector controllers, supporting node mobility between sectors as well as intersystem roaming.

1.2.1.2 Base Transceiver (BX)

A 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 their control stacks using the Base Transceiver Interface (BXI), and they are identified within a sector by their base transceiver ID (bxid).

1.2.2 Node

A node is a wireless object that connects to an iocast system. Nodes conceptually belong to one system and one application server, although they may roam onto other systems. Generally, nodes include a node controller (NC) and a node transceiver (NX), which are connected by the Node Transceiver Interface (NXI); however, these elements may be tightly coupled in more integrated embodiments. In any case, nodes exchange datagrams with their system, and through their system with their application server. Example nodes include weather sensors, water level sensors, intrusion alarms, utility meters, and personal notification devices.

1.2.2.1 Node Transceiver (NX)

Node transceivers (NXs) are digital radio modems that connect to an iocast system. A node transceiver is globally, uniquely identified by its 64-bit node transceiver ID (nxuid). Each node transceiver is bound to one iocast system, called its home system, by a shared private key. Node transceivers may only exchange datagrams with their home system. While nodes may roam onto other systems, these systems simply provide a secure tunnel between the roaming node and its home system. Each node is configured with up to 16 multicast addresses, and it is assigned a unique node address when it connects to a sector. Node transceivers operate with a *node availability* value (*na*), which describes how often the node listens for forward datagrams. This value ranges from 0 to 15, with lower values supporting faster datagram delivery and higher values supporting longer battery life. 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.

1.2.2.2 Node Controller (NC)

Generally, node controllers (NCs) are the embedded intelligence, sensors, and hardware supporting the primary mission of their node. A node controller might measure flow, note movement, or interact with a human user, in addition to communicating with the node transceiver.

1.2.2.3 Node Transceiver Interface (NXI)

Node Controllers and Node Transceivers connect using the Node Transceiver Interface (NXI), a hardware and software interface specification.

1.2.3 Application Server

Application servers are back-end systems which connect to a control stack using the API. Application servers are identified by their 64-bit application ID (appid), which is unique per system. Application servers control a subset of the nodes on the control stack, sending and receive datagrams to and from them to facilitate their application.

1.2.4 Group

Groups are sets of nodes sharing a common multicast address. A group is identified by its address, and it is conceptually owned by one control stack with which it shares a private encryption key. A group is controlled by an application server, which may transmit datagrams to the group and add/remove nodes to/from the group. Group member nodes receive, and may reply to, datagrams sent to the group's multicast address. Each group has a *group availability* value (*ga*), which describes how often nodes in the group must listen for forward datagrams. Group addresses may be *global*, wherein multicast datagrams follow nodes as they roam onto other systems, or they may be *local* and only meaningful on one system.

1.2.5 Channel

An iocast channel is a narrowband radio channel used to transmit datagrams and control information. Iocast channels include *forward channels* (base to node) and *reverse channels* (node to base).

1.2.6 Sector

A sector is 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. Sectors are identified by their Sector ID (secid), which is unique per system.

1.2.7 Datagram

Nodes and application servers communicate by exchanging *datagrams.* Application servers send *forward datagrams* to single nodes (unicast) or to groups (multicast). Nodes send *reverse datagrams* to application servers. Nodes and application servers know when their unicast datagrams are successfully delivered, and application servers know which recipients successfully receive their multicast datagrams.

1.2.7.1 Requests and Responses

Optionally, datagrams are divided into *request datagrams* and *response datagrams*. Response datagrams allow airtime- and energy-efficient support of low-level request/response protocols, with the main limitation being a relatively narrow response window of the previous 24 datagrams.

Forward Request Datagram	An unsolicited forward datagram sent from an application server to a node or group, not in response to a reverse datagram. Forward request datagrams may be unicast or multicast.
Forward Response Datagram	A forward datagram sent from an application server to a node in response to a reverse datagram. Forward response datagrams must be unicast and may not be multicast.
Reverse Request Datagram	An unsolicited reverse datagram sent from a node to an application server, not in response to a forward datagram.
Reverse Response Datagram	A reverse datagram sent from a node to an application server in response to a forward datagram.

1.2.7.2 Encryption

Datagrams may be natively encrypted, using a shared a private key to provide bidirectional authentication and privacy between a node or group and its control stack. Native encryption adds up to 32 bytes of overhead compared to unencrypted (plain text) datagrams. Applications may use their own encryption schemes instead of or in addition to native iocast encryption.

1.2.7.3 Timestamps

Via the air protocol, nodes have access to a millisecond-resolution timebase. Nodes can use this information to accurately timestamp reverse datagrams as they are queued, efficiently notifying their protocol stack and application server when the datagram was created and how long the transmission process took.

1.2.7.4 Short Datagrams

Normal datagrams range in size from 1 byte to 8,128 bytes; however, iocast also includes *short* datagrams, which are optimized to carry small payloads of 12- to 48-bits. Short datagrams do not support native encryption but in other respects behave identically to normal datagrams at the API and NXI level.

1.2.7.5 Datagram Identifiers

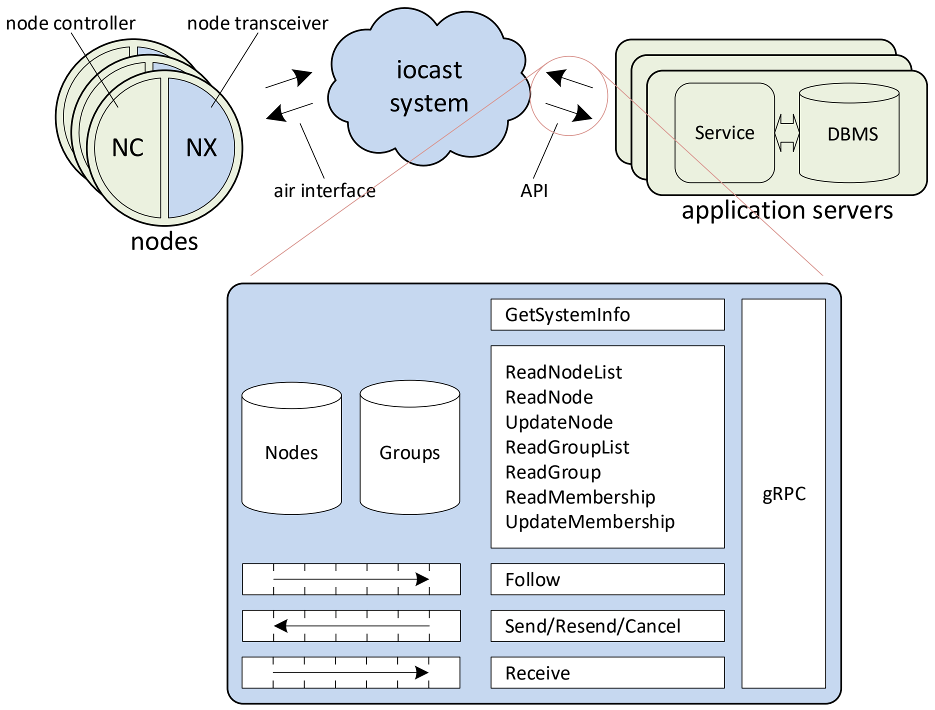
At the system- and API-level, datagrams are identified by a set of system-wide unique 64-bit datagram IDs called the *forward datagram ID* (*forward_datagram_id*) and the *reverse datagram ID* (*reverse_datagram_id*). At the node- and air-protocol level, datagrams are identified by two sets of 5-bit sequence numbers, the *forward datagram sequence number* (*fdsn*), and the *reverse datagram sequence number* (*rdsn*), respectively. The *fdsn* and *rdsn* values cycle from 0 through 31, repeating, and uniquely identify the last 32 datagrams sent to or received from each address.

1.2.8 Number Coordination

Iocast include several numbered codes. Three codes, *nxuid*, *sysid*, and *maddr*, are coordinated globally. Other values are coordinated per system, by system authorities.

Code	**Name**	**Scope**	**Width**
Node transceiver unique identifier	nxuid	global	64
System identifier	sysid	global	32
Sector identifier	secid	per-system	16
Application identifier	appid	per-system	64
Base Transceiver Identifier	bxid	per-sector	16
Multicast address	maddr	global	32
Node address	naddr	per-sector	32

2 API Description



The Iocast API is based on gRPC (see iocast-api-v3.proto) and allows application servers (clients) to connect to control stacks (servers). It includes 12 methods, and does not specify URL, port convention, or authentication; the iocast system operator must provide this information to clients.

2.1 Clients

The API identifies each client (A control stack is the "brain" of a network, the real-time software and database required to control sectors and support application server clients. The stack includes two software components, the *home controller* and the *sector controller*. The home controller is roughly analogous to a cellular network's *Home Location Register* (HLR), while the sector controller is roughly analogous to a *Mobile Switching Center* (MSC).

A control stack may operate as a single instance on a single server or container, or as a distributed system on a server cluster or cloud platform. Furthermore, home controllers may have many-to-many relationships with sector controllers, supporting node mobility between sectors as well as intersystem roaming.

) by its 64-bit application ID (appid). This value is included in method calls as the **appid** field. In cases where colocation and physical or perimeter security is available, the **appid** field itself can be used for lightweight, stand-alone authentication.

2.2 Nodes

The API identifies each node by its 64-bit *nxuid* value (1.2.2.1). In most cases this value is passed in the **nxuid** field. Clients may read their nodes from the control stack database, exchange datagrams with their nodes, and make changes to their nodes' groups as well as their *na*, *ne*, and *me* values.

2.3 Groups

The API identifies each group by its 32-bit multicast address (*maddr*). Groups have a visibility setting, which determine which clients may access them and how they may be used. Group visibility types include Private, Public, and External as follows:

- **Private**
  The client may send datagrams to the group and access the group with *Management Methods*. Other clients cannot see or access the group.
- **Public**
  The client may send datagrams to the group and access the group with *Management Methods*. Other clients may access the group with *Management Methods*.
- **External**
  The group belongs to another client that has set visibility to public. The client may access the group with *Management Methods*, but not send datagrams to the group.

Clients may read their groups from the control stack database, send datagrams to their groups, and add their nodes to group membership.

2.4 Datagrams

The main purpose of the iocast API is to allow clients to exchange datagrams with nodes and groups. Datagrams are uniquely identified by a 64-bit ID, assigned by the server. Datagram ID values appear throughout the API in fields **forward_datagram_id** and **reverse_datagram_id** for forward and reverse datagrams, respectively. Datagram ID values use a different numbering space than the **fdsn** and **rdsn** datagram sequence numbers in the NXI specification and the air protocol; however, the sequence numbers are made available to API clients through the **Receive** and **Follow** methods.

2.5 Datagram Payload

```
message DatagramPayload
{
    message Short
    {
        uint64 data = 1;
        uint32 bit_count = 2;
    }

    message Normal
    {
        bytes data = 1;
        bool encrypted = 2;
    }

    oneof data
    {
        Short short = 1;
        Normal normal = 2;
    }
}
```

Datagram payloads are described using the **DatagramPayload** message. It contains one of two possible data fields, either **short** or **normal**. The **short** field contains a **data** word and a **bit_count** value, describing the payload as the least-significant **bit_count** bits of **data**. The normal field contains a **data** field of payload bytes and an **encrypted** field, which describes whether iocast encryption is to be used.

2.6 Datagram Progress

```
enum ForwardDatagramProgress {
    QUEUED = 0;
    CANCELLED = 1;
    FDSN_ASSIGNED = 2;
    TRANSMITTING = 3;
    TRANSMITTED = 4;
    DELIVERED = 5;
    DELIVERY_TIMEOUT = 6;
    DELIVERY_ERROR = 7;
    RESENT = 8;
}
```

The system notifies the client as it processes forward datagrams, sending an event upon each of major state changes. These event codes are enumerated as follows:

- **QUEUED**
  The system has accepted the datagram into its queue.
- **CANCELLED**
  The client has successfully cancelled the datagram.
- **FDSN_ASSIGNED**
  The system has assigned a forward datagram ID to the datagram.
- **TRANSMITTING**
  The system has started transmission of the datagram.
- **TRANSMITTED**
  The system has finished transmission of the datagram.
- **DELIVERED**
  The node has acknowledged successful receipt of the message.
- **DELIVERY_TIMEOUT**
  The datagram was not delivered because of a timeout.
- **DELIVERY_ERROR**
  The datagram was not delivered due to a fatal error.
- **RESENT**
  The datagram has been resent.

2.7 Timestamps

Several API messages include a timestamp field, expressed as the number of milliseconds since midnight, January 1, 1970. Depending on the specific use, this value may be expressed in UTC or GPS time.

2.8 Capacity

```
message Capacity {
    message Quota {
        uint32 forward_queue_length = 1;
        uint32 forward_datagram_count = 2;
        uint32 forward_bit_count = 3;
        uint32 reverse_queue_length = 4;
        uint32 reverse_datagram_count = 5;
        uint32 reverse_bit_count = 6;
    }

    Quota max_quota = 1;
    Quota current_quota = 2;
    uint32 forward_datagram_max_bits = 3;
    uint32 reverse_datagram_max_bits = 4;
}
```

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

2.9 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 node, group, or datagram does not exist.
GRPC_STATUS_INVALID_ARGUMENT	A request message is invalid.
GRPC_STATUS_PERMISSION_DENIED	The client may not send a datagram to an EXTERNAL group.
GRPC_STATUS_FAILED_PRECONDITION	The client may not only resend a datagram previously sent to a group. The client may not resend a datagram currently being sent.
GRPC_STATUS_ABORTED	Optimistic locking failed because of a stale row_version.
GRPC_STATUS_RESOURCE_EXHAUSTED	The group or node has exceeded its quota.
GRPC_STATUS_UNAVAILABLE	The node or group is busy, or the datagram can no longer be cancelled, or the datagram has fallen out of the response window.

2.10 API Method Summary

The API includes 13 methods, roughly divided into *communication methods*, *management methods*, and one *system information* method. Application that only send and receive datagrams may only require the communication methods, while more complex systems may make use of management methods to dynamically configure and manage groups and nodes.

2.10.1 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**
- **Resend**
- **Cancel**
- **Receive**

2.10.2 Management Methods

The API includes seven management methods. Six of these enable the client to read and update node rows, read group rows, and update group membership. The sixth method (Follow) enables the client to synchronize its own database with real-time database updates from the server.

- **ReadNodeList**
- **ReadNode**
- **UpdateNode**
- **ReadGroupList**
- **ReadGroup**
- **ReadMembership**
- **UpdateMembership**
- **Follow**

2.10.3 System Information Method

The API includes one additional method, which enables a client to determine its roles and the behavior of the channel.This event signals that the event queue has been purged and that a discontinuity may appear in the event sequence numbers. The value contains the number of events that were purged.

- **GetSystemInfo**

3 API Methods

3.1 Send

```
    rpc Send (SendRequest) returns (SendResponse) {}
```

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. After a successful **Send** method call, the client can use the Receive method to track the datagram's progress, or the **Resend**

```
    rpc Resend (ResendRequest) returns (ResendResponse) {}
```

The **Resend** method resends a forward datagram to a group. The client passes a **ResendRequest** message to the method, and the method returns a **ResendResponse** message to the client. After a successful **Resend** method call, the client can use Receive to track the datagram's progress, or the Error! Not a valid bookmark self-reference. method to delete it from the queue.

3.1.1 ResendRequest

```
message ResendRequest {
    uint64 appid = 1;
    uint64 forward_datagram_id = 2;
    repeated uint64 nxuid_list = 3;
}
```

The client passes a **ResendRequest** message when invoking the **Resend** method. The **ResendRequest** message includes the following fields:

- **appid**
  This field contains the client's appid (2.1).
- **forward_datagram_id**
  This field identifies the datagram to be resent, which must be a datagram previously sent to a group address.
- **nxuid_list**
  This field identifies nodes which should be covered by the retransmission of the datagram. If this list is empty, then the datagram is retransmitted in all sectors connected to member nodes. If this list contains entries, then the datagram is only retransmitted in sectors connected to nodes in the list.

3.1.2 ResendResponse

```
message ResendResponse {
    uint64 estimated_delivery_timestamp = 1;
    Capacity capacity = 2;
}
```

Upon successful completion, the **Resend** method responds with a **ResendResponse** 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 2.7.
- **capacity**
  If present, this field describes the current capacity of the destination node or group after accounting for the present datagram.

Cancel method to delete it from the queue.

3.1.3 SendRequest

```
message SendRequest {
    uint64 appid = 1;

    oneof destination
    {
        uint32 maddr = 2;
        uint64 nxuid = 3;
    }

    uint32 priority = 4;
    uint64 reverse_datagram_id = 5;
    DatagramPayload payload = 6;
}
```

The client passes a **SendRequest** message when invoking the **Send** method. The **SendRequest** message describes a forward datagram and includes the following fields:

- **appid**
  This field contains the client's appid (2.1).
- **destination**
  This field specifies the destination, a node (**nxuid**) or group (**maddr**) of the datagram. Note that the client may not send a datagram to an EXTERNAL group.
- **priority**
  This field specifies the client-side scheduling priority of the datagram, with 0 being the lowest priority and 15 being the highest priority.
- **reverse_datagram_id**
  If non-zero, this field indicates that the datagram is a forward response datagram, identifying the reverse datagram to which it is responding.
- **payload**
  This field contains the datagram payload. The **normal** field contains **data**, a sequence of bytes, along with an **encrypted** flag specifying native encryption (true) or plaintext (false). The short field specifies a bit field.

3.1.4 SendResponse

```
message SendResponse {
    uint64 forward_datagram_id = 1;
    uint64 estimated_delivery_timestamp = 2;
    Capacity capacity = 3;
}
```

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

- **forward_datagram_id**
  This field specifies the system-assigned ID of the datagram.
- **estimated_delivery_timestamp**
  This field specifies when the node is expected to receive the datagram, using the timestamp format described in section 2.7.
- **capacity**
  If present, this field describes the current capacity of the destination node or group after accounting for the present datagram.

3.2 Resend

```
    rpc Resend (ResendRequest) returns (ResendResponse) {}
```

The **Resend** method resends a forward datagram to a group. The client passes a **ResendRequest** message to the method, and the method returns a **ResendResponse** message to the client. After a successful **Resend** method call, the client can use Receive to track the datagram's progress, or the Error! Not a valid bookmark self-reference. method to delete it from the queue.

3.2.1 ResendRequest

```
message ResendRequest {
    uint64 appid = 1;
    uint64 forward_datagram_id = 2;
    repeated uint64 nxuid_list = 3;
}
```

The client passes a **ResendRequest** message when invoking the **Resend** method. The **ResendRequest** message includes the following fields:

- **appid**
  This field contains the client's appid (2.1).
- **forward_datagram_id**
  This field identifies the datagram to be resent, which must be a datagram previously sent to a group address.
- **nxuid_list**
  This field identifies nodes which should be covered by the retransmission of the datagram. If this list is empty, then the datagram is retransmitted in all sectors connected to member nodes. If this list contains entries, then the datagram is only retransmitted in sectors connected to nodes in the list.

3.2.2 ResendResponse

```
message ResendResponse {
    uint64 estimated_delivery_timestamp = 1;
    Capacity capacity = 2;
}
```

Upon successful completion, the **Resend** method responds with a **ResendResponse** 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 2.7.
- **capacity**
  If present, this field describes the current capacity of the destination node or group after accounting for the present datagram.

3.3 Cancel

```
    rpc Cancel (CancelRequest) returns (CancelResponse) {}
```

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.

3.3.1 CancelRequest

```
message CancelRequest {
    uint64 appid = 1;
    uint64 forward_datagram_id = 2;
}
```

The client passes a **CancelRequest** message when invoking the **Cancel** method. The **CancelRequest** message identifies the client (**appid**) as well as the forward datagram to cancel (**forward_datagram_id**).

3.3.2 CancelResponse

```
message CancelResponse {
}
```

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

3.4 Receive

```
    rpc Receive (ReceiveRequest) returns (stream ReceiveResponse) {}
```

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

3.4.1 ReceiveRequest

```
message ReceiveRequest {
    uint64 appid = 1;
    uint64 last_sequence_number = 2;
    uint32 duration_ms = 3;
}
```

The client passes a **ReceiveRequest** message when invoking the **Receive** method. The **ReceiveRequest** message includes the following fields:

- **appid**
  This field contains the application ID (2.1) of the client.
- **last_sequence_number**
  This field specifies the sequence number of the last event received by the client in earlier calls. This field servers as a reverse acknowledgement for the response stream. The server will release resources tied to events with serial numbers up to and including **last_sequence_number**, and the **ReceiveResponse events** field will begin with the first available sequence number after this value.
- **duration_ms**
  This field specifies the amount of time to keep the return stream active. Once this number of milliseconds has elapsed, the server ends the return stream and the client must issue another method call.

3.4.2 ReceiveResponse

```
message ReceiveResponse {
    message Event {
        message ReverseDatagram {
            uint64 reverse_datagram_id = 1;
            uint64 forward_datagram_id = 2;
            uint64 nxuid = 3;
            uint32 rdsn = 4;
            uint64 node_timestamp = 5;
            bool encrypted = 6;
            DatagramPayload payload = 7;
        }

        message ForwardDatagramProgress {
            uint64 forward_datagram_id = 1;
            uint32 node_timestamp = 2;
            uint32 fdsn = 3;
            bool final = 4;
            ForwardDatagramProgress progress = 5;
        }

        uint64 timestamp = 1;

        oneof event {
            ReverseDatagram reverse_datagram = 2;
            ForwardDatagramProgress forward_datagram_progress = 3;
        }
    }

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

Upon successful completion, the **Receive** method responds with a **ReceiveResponse** message, which contains two fields, **first_sequence_number** and **event_list**. The **first_sequence_number** field contains the sequence number of the first and oldest item in the **event_list** field, with subsequent items numbered in ascending order. The **eventList** field contains a series of events.

Each event in the **event_list** field contains a **timestamp** field (2.1), a GPS timestamp of the event at the server, plus one of two possible event fields: **reverse_datagram** or **forward_datagram_progress**. The method will return after a limited period of time (**duration_ms**, 3.4.1), requiring the client to call the method at regular intervals to receive a continuous flow of events.

3.4.2.1 reverse_datagram

If present, the **reverse_datagram** 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 server-assigned ID of the reverse datagram.
- **forward_datagram_id**
  If non-zero, this field indicates that the datagram is a reverse response datagram, and it identifies the forward datagram to which it is responding.
- **nxuid**
  This field identifies the node transmitting the datagram.
- **rdsn**
  This field specifies the reverse datagram number assigned to the datagram.
- **node_timestamp**
  If non-zero, this field specifies when the datagram was created at the node (2.7).
- **payload**
  This field contains the datagram payload.

3.4.2.2 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 contains the ID of the forward datagram, assigned by the **Send** method and returned by the **SendRequest** message.
- **node_timestamp**
  If non-zero, this field contains the timestamp of the event (2.7) as measured at the node.
- **fdsn**
  If set to 0x40, this value indicates the system has not yet assigned an fdsn value;. otherwise, this field contains the fdsn of the datagram.
- **final**
  This field indicates whether the event is the final progress event for the datagram. If true, then the system will not send additional progress events regarding the datagram.
- **progress**
  This field indicates the type of progress (2.6).

3.5 ReadNodeList

```
    rpc ReadNodeList (ReadNodeListRequest) returns (stream ReadNodeListResponse) {}
```

The **ReadNodeList** method returns a list of node UIDs. The client passes a **ReadNodeListRequest** message to the method, and the method returns a stream of **ReadNodeListResponse** messages to the client.

3.5.1 ReadNodeListRequest

```
message ReadNodeListRequest {
    uint64 appid = 1;
    uint64 nxuid_low = 2;
    uint64 nxuid_high = 3;
    uint32 maddr = 4;
}
```

The client passes a **ReadNodeListRequest** message when invoking the **ReadNodeList** method. The **ReadNodeListRequest** message includes an **appid** value identifying the client, as well as three other fields specifying the range of nodes to return, **nxuid_low**, **nxuid_high**, and **maddr**. The **ReadNodeList** method will return a list of nodes that meets the following criteria:

- nxuid ≥ **nxuid_low**, and
- nxuid ≤ **nxuid_high**, and
- if **maddr** ≠ 0, then the node is a member of the group

3.5.2 ReadNodeListResponse

```
message ReadNodeListResponse {
    repeated uint64 nxuid_list = 1;
}
```

Upon successful completion, the **ReceiveNodeList** method responds with a stream of **ReceiveNodeListResponse** messages, each containing one or more node nxuids in the field **nxuid_list**. Upon successful completion, the server returns a complete set of nxuids prior to ending the stream.

3.6 ReadNode

```
    rpc ReadNode (ReadNodeRequest) returns (stream ReadNodeResponse) {}
```

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.

3.6.1 ReadNodeRequest

```
message ReadNodeRequest {
    uint64 appid = 1;
    repeated uint64 nxuid_list = 2;
}
```

The client passes a **ReadNodeRequest** message when invoking the **ReadNode** method. The **ReadNodeRequest** message includes an **appid** value identifying the client, and a **nxuid_list** field containing a list of nxuids specifying node rows to read.

3.6.2 ReadNodeResponse

```
message ReadNodeResponse {
    message Row {
        message Data {
            message Connection {
                enum State {
                    DISCONNECTED = 0;
                    CONNECTED = 1;
                    LOST = 2;
                }

                State state = 1;
                uint32 sysid = 2;
                uint32 secid = 3;
                double latitude = 4;
                double longitude = 5;
                int32 timezone = 6;
                uint32 naddr = 7;
                uint32 na = 8;
                bool ne = 9;
                bool me = 10;
            }

            message Group {
                uint32 maddr = 1;
                MembershipState state = 2;
            }

            enum ConfigurationState {
                CURRENT = 0;
                SENDING = 1;
                WAITING = 2;
            }

            uint64 nxuid = 1;
            uint64 row_version = 2;
            repeated Group groups = 3;
            uint32 na = 4;
            bool ne = 5;
            bool me = 6;
            Connection connection = 7;
            ConfigurationState configuration_state = 8;
            Capacity capacity = 9;
            uint32 estimated_delivery_timestamp = 10;
            repeated uint64 datagram_queue = 11;
        }

        message Null {
            uint64 nxuid = 1;
        }

        oneof contents {
            Data data = 2;
            Null null = 3;
        }
    }

    repeated Row row_list = 1;
}
```

Upon successful completion, the **ReceiveNode** method responds with a stream of **ReceiveNodeLResponse** messages, each of which includes one repeated field **row_list**. Each element of **rows_list** corresponds to a nxuid value in the **ReadNodeRequest** message, and includes either a **data** message, containing row data, or a **null** message, including only the **nxuid** value and indicates the node was not found. Where present, the **data** field includes the following fields:

- **nxuid**
  This field identifies the node.
- **row_version**
  This field contains the current version number of the node's underlying database row.
- **groups**
  This field contains the list of groups programmed into the node transceiver. Each item in this list includes the field **maddr**, identifying the group, and **state**, describing the synchronization state.
- **na**
  This field specifies the current availability of the node.
- **ne**
  This fields indicates whether the node's address is enabled. If true, the node will decode forward datagrams and may send reverse datagrams; if false, the node may not send or receive datagrams.
- **me**
  This fields indicates whether the node's multicast addresses are enabled. If true, and the node is configured with multicast addresses, the node will decode multicast frames as well as datagrams addressed to its multicast addresses. If this value is false, the node will sleep through multicast frames and ignore multicast datagrams.
- **connection**
  This field describes the network-side connection state of the node. If the state is **DISCONNECTED**, then the node is disconnected. If the state is **CONNECTED** or **LOST** then the connection state is described by the remaining fields of the **Connection** message as follows:
  - **sysid, secid**
    These fields specify the node's current system and sector.
  - **latitude, longitude**
    These fields specify the latitude and longitude of the centroid of the currently connected sector (in radians).
  - **timezone**
    This field specifies the timezone of the currently connected sector (in minutes of offset from UTC).
  - **naddr**
    This field specifies the currently assigned node address.
  - **na**
    This field specifies the current availability of the node. This field may occasionally vary from the same field in the node row because of over-the-air synchronization lag.
  - **ne**

This fields indicates whether the node’s address is enabled. If true, the node will decode forward datagrams and may send reverse datagrams; if false, the node may not send or receive datagrams. This field may occasionally vary from the same field in the node row because of over-the-air synchronization lag.

  - **me**
    This fields indicates whether the node’s multicast addresses are enabled. If true, and the node is configured with multicast addresses, the node will decode multicast frames as well as datagrams addressed to its multicast addresses. If this value is false, the node will sleep through multicast frames and ignore multicast datagrams. This field may occasionally vary from the same field in the node row because of over-the-air synchronization lag.

- **configuration_state**
  This field describes the configuration state of the node, either **CURRENT**, **SENDING**, or **WAITING**.
- **capacity**
  This field describes the current state of the node’s capacity.
- **estimated_delivery_timestamp**
  This field specifies when the node would be expected to receive the next datagram, using the timestamp format described in section 2.7.
- **datagram_queue**
  This field lists the forward datagrams in queue for the node. The list includes the datagram number portion of the forward datagram ID (2.1) of each datagram, starting with the first datagram in queue and ending with the last datagram in queue.

3.7 UpdateNode

```
    rpc UpdateNode (UpdateNodeRequest) returns (UpdateNodeResponse) {}
```

The **UpdateNode** method updates a node row at the system. The client passes an **UpdateNodeRequest** message to the method, and the method returns an **UpdateNodeResponse** message to the client. Note that this method may result in over-the-air programming activity to synchronize the node.

3.7.1 UpdateNodeRequest

```
message UpdateNodeRequest {
    uint64 appid = 1;
    uint64 nxuid = 2;
    uint64 row_version = 3;
    repeated uint32 maddr_list = 4;
    uint32 na = 5;
    bool me = 6;
    bool ne = 7;
}
```

The client passes a **UpdateNodeRequest** message when invoking the **UpdateNode** method. The **UpdateNodeRequest** message includes the following fields:

- **appid**
  This field contains the application ID (2.1) of the client.

- **nxuid**
  This field contains the node UID of the row to be updated.

- **row_version**
  The **UpdateNode** method employs optimistic row locking, and the **row_version** field must match the current version of the node row for the method to complete successfully. If **node_version** does not match the current row version, 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.

- **maddr_list**
  This field contains the list of multicast addresses programmed into the node transceiver, each representing one group.

- **na**
  This field specifies the current availability of the node.

- **ne**
  This fields indicates whether the node's address is enabled. If true, the node will decode forward datagrams and may send reverse datagrams; if false, the node may not send or receive datagrams.

- **me**
  This fields indicates whether the node's multicast addresses are enabled. If true, and the node is configured with multicast addresses, the node will decode multicast frames as well as datagrams addressed to its multicast addresses. If this value is false, the node will sleep through multicast frames and ignore multicast datagrams.

3.7.2 UpdateNodeResponse

```
message UpdateNodeResponse {
    uint64 row_version = 1;
}
```

Upon successful completion, the **UpdateNode** method responds with an **UpdateNodeResponse** message, which contains the new **row_version** value after the update is complete.

3.8 ReadGroupList

The **ReadGroupList** method returns a list of groups ID's. The client passes a **ReadGroupListRequest** message to the method, and the method returns a stream of **ReadGroupListResponse** messages to the client.

3.8.1 ReadGroupListRequest

```
message ReadGroupListRequest {
    uint64 appid = 1;
    uint32 maddr_low = 2;
    uint32 maddr_high = 3;
    uint64 nxuid = 4;
}
```

The client passes a **ReadGroupListRequest** message when invoking the **ReadGroupList** method. The **ReadGroupListRequest** message includes an **appid** value identifying the client, as well as three other fields specifying the range of groups to return, **maddr_low**, **maddr_high**, and **nxuid**. The **ReadGroupList** method will return a list of groups that meet the following criteria:

- maddr ≥ **maddr_low**, and
- maddr ≤ **maddr_high**, and
- if **nxuid** ≠ 0, then the group includes **nxuid** in its membership.

3.8.2 ReadGroupListResponse

```
message ReadGroupListResponse {
    repeated uint32 maddr_list = 1;
}
```

The **ReadGroupList** method returns a stream of **ReadGroupListResponse** messages. Each message includes one field, **maddr_list**, containing a list of multicast group addresses. Upon successful completion, the server returns a complete set of such group addresses prior to ending the stream.

3.9 ReadGroup

```
    rpc ReadGroup (ReadGroupRequest) returns (stream ReadGroupResponse) {}
```

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 **ReadGroupResponse** messages to the client.

3.9.1 ReadGroupRequest

```
message ReadGroupRequest {
    uint64 appid = 1;
    repeated uint32 maddr_list = 2;
}
```

The client passes a **ReadGroupRequest** message when invoking the **ReadGroup** method. The **ReadGroupRequest** message includes an **appid** value identifying the client, and a **maddr_list** field containing a list of multicast addresses, specifying the group rows to read.

3.9.2 ReadGroupResponse

```
message ReadGroupResponse {
    message Row {
        message Data {
            enum Visibility {
                PRIVATE = 0;
                PUBLIC = 1;
                EXTERNAL = 2;
            }

            uint32 maddr = 1;
            uint64 row_version = 2;
            string label = 3;
            uint32 ga = 4;
            Visibility visibility = 5;
            uint32 sysid = 6;
            Capacity capacity = 7;
            repeated uint64 datagram_queue = 8;
        }

        message Null
        {
            uint32 maddr = 1;
        }

        oneof contents {
            Data data = 2;
            Null null = 3;
        }
    }

    repeated Row row_list = 1;
}
```

Upon successful completion, the **ReceiveGroup** method responds with a stream of **ReceiveGroupResponse** messages, each of which includes one repeated field **row_list**. Each element of **row_list** corresponds to a **maddr_list** value in the **ReadNodeRequest** message, and includes either a **data** message, containing row data, or a **null** message, including only the **maddr** value and indicates the node was not found. Where present, the **data** field includes the following fields:

- **maddr**
  This field contains the multicast address of the group.
- **row_version**
  This field contains the current version number of the group's underlying database row.

- **label**
  This field contains a symbolic label of the group.
- **ga**
  This field contains the group available value, describing how quickly group messages are received by nodes in the group.
- **visibility**
  This field identifies the visibility of the group from an API perspective. **PRIVATE** means that the group belongs to the calling client and is not visible to other clients. **PUBLIC** means that the group belongs to the calling client but is visible to other clients, who may add their own nodes to it. **EXTERNAL** means that the group belongs to another client but is visible to the calling client.
- **sysid**
  This field identifies the system scope of the group. A value of zero means that the group address is globally coordinated may receive messages on any system. A non-zero value identifies the system ID where the group may receive messages.
- **capacity**
  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 group. The list includes the datagram number portion of the forward datagram ID (2.1) of each datagram, starting with the first datagram in queue and ending with the last. For groups with **visibility** set to **EXTERNAL**, this value is always empty.

3.10 ReadMembership

```
rpc ReadMembership (ReadMembershipRequest) returns (stream ReadMembershipResponse) {}
```

The **ReadMembership** method reads a group's membership, or the list of nodes programmed with the group specified group address. The client passes an **ReadMembershipRequest** message to the method, and the method returns a **ReadMembershipResponse** message to the client.

3.10.1 ReadMembershipRequest

```
message ReadMembershipRequest
{
    uint64 appid = 1;
    uint32 maddr = 2;
}
```

The client passes a **ReadMembershipRequest** message when invoking the **ReadMembership** method. The **ReadMembershipRequest** includes an **appid** value identifying the client, and **maddr** field specifying the multicast address of the group to read.

3.10.2 ReadMembershipResponse

```
message ReadMembershipResponse
{
    message Member {
        uint64 nxuid = 1;
        MembershipState state = 2;
    }

    repeated Member member_list = 1;
}
```

Upon successful completion, the **ReadMembership** method responds with an **ReadMembershipResponse** message, which contains the field **membership**, which contains the list of nodes belonging to the group.

3.11 UpdateMembership

```
    rpc UpdateMembership (UpdateMembershipRequest) returns (UpdateMembershipResponse) {}
```

The **UpdateMembership** method adds nodes to a group, and/or removes nodes from a group. The client passes an **UpdateMembershipRequest** to the method, and the method returns an **UpdateMembershipResponse** message to the client. Note that this method may result in over-the-air programming activity to synchronize the affected nodes.

3.11.1 UpdateMembershipRequest

```
message UpdateMembershipRequest
{
    message Operation1 {
        bool remove_all_nodes = 2;
        repeated uint64 add_nodes = 3;
    }

    message Operation2 {
        repeated uint64 remove_nodes = 3;
        repeated uint64 add_nodes = 4;
    }

    uint64 appid = 1;

    uint32 maddr = 2;

    oneof operation {
        Operation1 operation1 = 3;
        Operation2 operation2 = 4;
    }
}
```

The client passes an **UpdateMembershipRequest** message when invoking the **UpdateMembership** method. The **UpdateMembershipRequest** method includes a **maddr**, specifying the multicast address of the group to modify, followed by an operation to remove and add nodes. Two operations are possible. The first operation (**operation1**) optionally removes all nodes from the group (**remove_all_nodes**) and then adds a list of nodes (**add_nodes**). The second operation (**operation2**) removes a list of nodes (**remove_nodes**) and then adds a list of nodes (**add_nodes**).

When nodes are added to a group using this method, the group address is added to the node row as the last value in the **groups** field (3.6.1 and 3.6.2). Changes happen instantaneously to the system database but require time to replicate over the air to the nodes themselves.

3.11.2 UpdateMembershipResponse

```
message UpdateMembershipResponse
{
    message Exception {
        enum Code
        {
            NotFound = 0;
            Overflow = 1;
        }

        uint64 nxuid = 1;
        Code code = 2;
    }

    repeated Exception exceptions = 1;
}
```

Upon successful completion, the **UpdateMembership** method responds with an **UpdateMembershipResponse** message. The **UpdateMembershipResponse message** includes a list of exceptions, identifying nodes which could not be added to the group. Each **Excption** message includes a **nxuid** as well as **code** enumeration. The **NotFound** code means that the node does not exist, and the **Overflow** code means that the node is already a member of 16 groups. Under normal circumstances, where each node is successfully added to the group, the **exceptions** field is empty.

3.12 Follow

```
    rpc GetSystemInfo (GetSystemInfoRequest) returns (GetSystemInfoResponse) {}
```

The **Follow** method returns a stream of events from the server. These events include datagram traffic activity as well as changes to node, group, and membership rows. In architectures where the application server (client) maintains its own database, this method enables the client to synchronize the relevant tables of its database to those of the server. The client passes a **FollowRequest** message to the method, and the method returns a stream of **FollowResponse** messages to the client.

3.12.1 FollowRequest

```
message FollowRequest {
    uint64 appid = 1;
    uint64 last_sequence_number = 2;
    uint32 duration_ms = 3;
    bool purge = 4;
}
```

The client passes an **FollowRequest** message when invoking the **Follow** method. The **FollowRequest** message includes the following fields:

- **appid**
  This field contains the application ID (2.1) of the client.
- **last_sequence_number**
  This field specifies the sequence number of the last event received by the client in earlier calls. This field servers as a reverse acknowledgement for the response stream. The server will release resources tied to events with serial numbers up to and including **last_sequence_number**, and the **FollowResponse events** field will begin with the first available sequence number after this value.
- **duration_ms**
  This field specifies the amount of time to keep the return stream active. Once this number of milliseconds has elapsed, the server ends the return stream and the client must issue another method call.
- **purge**
  This field tells the server to purge the monitor event cache prior to starting the response stream. The first event returned in the response stream will be the purge event.

3.12.2 FollowResponse

```
message FollowResponse {
    message Event {
        message ForwardDatagram {
            uint64 forward_datagram_id = 1;

            oneof destination
            {
                uint32 maddr = 2;
                uint64 nxuid = 3;
            }

            uint64 reverse_datagram_id = 4;
            uint64 expected_delivery = 5;
            uint32 priority = 6;
            DatagramPayload payload = 7;
        }

        message ForwardDatagramProgress {
            uint64 forward_datagram_id = 1;
            uint64 forward_datagram_client_id = 2;
            uint32 node_timestamp = 3;
            uint32 fdsn = 4;
            bool final = 5;
            ForwardDatagramProgress progress = 6;
        }

        message ReverseDatagram {
            uint64 reverse_datagram_id = 1;
            uint64 forward_datagram_id = 2;
            uint64 nxuid = 3;
            uint32 rdsn = 4;
            uint64 node_timestamp = 5;
            bool encrypted = 6;
            DatagramPayload payload = 7;
        }

        message MembershipChange {
            uint32 maddr = 1;
            uint64 nxuid = 2;
            MembershipState state = 3;
        }

        uint64 timestamp = 1;

        oneof event {
            ForwardDatagram forward_datagram = 2;
            ForwardDatagramProgress forward_datagram_progress = 3;
            ReverseDatagram reverse_datagram = 4;
            uint64 node_update = 5;
            uint64 node_add = 6;
            uint64 node_delete = 7;
            uint32 group_update = 8;
            uint32 group_add = 9;
            uint32 group_delete = 10;
            MembershipChange membership_change = 11;
            uint32 overrun = 12;
            uint32 purge = 13;
        }
    }

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

Upon successful completion, the **Receive** method responds with a **ReceiveResponse** message, which contains two fields, **first_sequence_number** and **event_list**. The **first_sequence_number** field contains the sequence number of the first and oldest item in the **event_list** field, with subsequent items numbered in ascending order. The **event_list** field contains an series of events.

Each **Event** message in the **event_list** field includes a **timestamp** field (2.1), marking the time when the event was observed at the server, plus one of twelve possible event types in the **event** field: **forward_datagram**, **forward_datagram_progress**, **reverse_datagram**, **node_update**, **node_add**, **node_delete**, **group_update**, **group_add**, **group_delete**, **membership_change**, **overrun**, or **purge**.

3.12.2.1 forward_datagram

This event signals that a forward datagram has been received by an API client. The field contains a **ForwardDatagram** message, which includes the following fields:

- **forward_datagram_id**
  This field contains the ID of the forward datagram, assigned by the **Send** method and returned by the **SendRequest** message.
- **destination**
  This field specifies the destination, a node (**nxuid**) or group (**maddr**) of the datagram. Note that the client may not send a datagram to an EXTERNAL group.
- **reverse_datagram_id**
  If present, this field indicates the datagram is a forward response datagram, and contains the ID of the reverse datagram being replied to. If absent, the forward datagram is an unsolicited request.
- **priority**
  This field contains the server assigned absolute priority of the datagram, which is formed from the priority specified in the Send method call, as well as other priority factors managed by the server.
- **payload**
  This field contains the datagram payload.

3.12.2.2 forward_datagram_progress

This event signals that a forward datagram has progressed toward deliver to its destination node or group. This field contains a **ForwardDatagramProgress** message, which includes the following fields:

- **forward_datagram_id**
  This field contains the ID of the forward datagram, assigned by the **Send** method and returned by the **SendRequest** message.
- **node_timestamp**
  If non-zero, this field contains the timestamp of the event (2.7) as measured at the node.
- **fdsn**
  If set to 0x40, this value indicates the system has not yet assigned an fdsn; otherwise, this field contains the fdsn of the datagram.
- **final**
  This field indicates whether the event is the final progress event for the datagram. If true, then the system will not send additional progress events regarding the datagram unless the client successfully invokes a **Resend** method on the datagram.
- **progress**
  This field indicates the type of progress (2.6).

3.12.2.3 reverse_datagram

This event signals that the system received a reverse datagram. The field contains a **ReverseDatagram** message, which includes the following fields.

- **reverse_datagram_id**
  This field contains the server-assigned ID of the reverse datagram.
- **forward_datagram_id**
  If non-zero, this field indicates that the datagram is a reverse response datagram, and it identifies the forward datagram to which it is responding.

- **nxuid**
  This field identifies the node transmitting the datagram.
- **rdsn**
  This field specifies the reverse datagram number assigned to the datagram.
- **node_timestamp**
  If non-zero, this field specifies when the datagram was created at the node (2.7).
- **payload**
  This field contains the datagram payload.

3.12.2.4 node_update

This event signals that a node row has been updated. The value contains the node's nxuid,

3.12.2.5 node_add

This type of event signals that a node row has been added. The value contains the node's nxuid,

3.12.2.6 node_delete

This type of event signals that a node row has been deleted The value contains the node's nxuid,

3.12.2.7 group_update

This event signals that a group row has been updated. The value contains the group's multicast address (*maddr*).

3.12.2.8 group_add

This event signals that a group row has been added. The value contains the group's multicast address (*maddr*).

3.12.2.9 group_delete

This event signals that a group row has been deleted. The value contains the group's multicast address (*maddr*).

3.12.2.10 membership_change

This event signals that the group membership has changed. The field contains a **MembershipChange** message, which includes the following fields.

- **maddr**
  This field identifies the group's multicast address.
- **nxuid**
  This field identifies the member node.
- **state**
  The state of the membership.

3.12.2.11 Overrun

This event signals that the event queue has been overrun and that a discontinuity may appear in the event sequence numbers. The value contains the number of events that were lost.

3.12.2.12 Purge

This event signals that the event queue has been purged and that a discontinuity may appear in the event sequence numbers. The value contains the number of events that were purged.

3.13 GetSystemInfo

```
    rpc GetSystemInfo (GetSystemInfoRequest) returns (GetSystemInfoResponse) {}
```

The **GetSystemInfo** method returns information regarding the system and the client's connection to the system. The client passes a **GetSystemInfoRequest** message to the method, and the method returns a **GetSystemInfoResponse** message to the client.

3.13.1 GetSystemInfoRequest

```
message GetSystemInfoRequest {
    uint64 appid = 1;
}
```

The client passes an **FollowRequest** message when invoking the **Follow** method. The **FollowRequest** message includes an **appid** field, identifying the client.

3.13.2 GetSystemInfoResponse

```
message GetSystemInfoResponse {
    message FifoState {
        uint32 depth = 1;
        uint64 newest_sequence_number = 2;
    }

    message SystemInfo {
        enum State {
            NORMAL_OPERATIONS = 0;
            MINOR_ALARM = 1;
            MAJOR_ALARM = 2;
            OFF_LINE = 3;
        }

        uint32 sysid = 1;
        State state = 2;
        string name = 3;
        string location = 4;
        string contact_info = 5;
    }

    SystemInfo system_info = 2;
    FifoState receive_fifo_state = 3;
    FifoState follow_fifo_state = 4;
    uint64 current_time_gps = 5;
    uint64 current_time_utc = 6;
}
```

Upon successful completion, the **GetSystemInfo** method responds with a **GetSystemInfoResponse** message, which contains the following fields:

- **system_info**
  This field contains the current information about the system, including the following subfields:
  - **sysid**
    This field contains the server's iocast system identifier
  - **state**
    This field reflects the overall condition of the system. The **NORMAL_OPERATIONS** state indicates that the system is operating normally. The **MINOR_ALARM** state indicates that trouble exists, but the system is operating and the client's API operations are unaffected. The **MAJOR_ALARM** state indicates that system trouble may delay or interfere with client's API operations, and **OFF_LINE** indicates that the system is not presently operational.

  - **name**
    This field contains the descriptive name of the system.
  - **location**
    This field contains a description of the system's physical location.
  - **contact_info**
    This field contains contact information (phone or email) for the system administrator.
- **receive_fifo_state**
  This field specifies the number of events currently in the client's receive queue, along with the newest sequence number.
- **follow_fifo_state**
  This field specifies the number of events currently in the client's follow queue, along with the newest sequence number.3339
- **current_time_gps**
  This field contains the current GPS time, measured by the server at the start of the method call, reported as per section 2.7.
- **current_time_utc**
  This field contains the current UTC time, measured by the server at the start of the method call, reported as per section 2.7.