# intdash Edge Agent 2
# デベロッパーガイド

intdash Edge Agent 2 v1.2

第 3 版（2024 年 2 月）

aptpod

Rev. 4cc2b7d

# はじめに

**警告:** intdash Edge Agent 2 v1.2.0 では、一部の設定項目に後方互換が失われる変更が含まれており、設定の修正が必要になる場合があります。詳しくは リリースノート (p. 112) をご確認ください。

# 01 intdash Edge Agent 2 とは

intdash Edge Agent 2 は、intdash サーバーとの間で時系列データの送受信を行うエージェントソフトウェアです。エッジデバイス（Linux PC）にインストールして使用します。

intdash Edge Agent 2 を使って時系列データを intdash サーバーに送ることで、intdash サーバーにそのデータを「計測 (measurement)」として保存することができます。

## 1.1 intdash Edge Agent 2 でできること

intdash Edge Agent 2 は以下を行います。

- iSCP (intdash Stream Control Protocol) のストリームメッセージを使って、時系列データをサーバーにストリーミング送信する、およびサーバーからストリーミング受信する（intdash サーバーのリアルタイム API を使った送受信）
- サーバーにデータをストリーミング送信する前に、データをフィルタリングまたはサンプリングする
- ネットワークの切断により送信できなかった時系列データを、自動的に後から送信する（intdash サーバーの REST API を使った遅延アップロード）
- iSCP (intdash Stream Control Protocol) のエンドツーエンド (end to end、E2E) コールを使って、エッジ同士でコミュニケーションする

**重要:** intdash Edge Agent 2 は、従来提供していた旧 intdash Edge Agent を使いやすくリニューアルした新しいプロダクトです。旧 intdash Edge Agent とは操作や設定の方法が異なり、設定ファイルのフォーマットに互換性がない点にご注意ください。
旧 intdash Edge Agent から intdash Edge Agent 2 に移行する場合は、旧 intdash Edge Agent からの移行 (p. 105) を参照してください。

## 1.2 intdash Edge Agent 2 の構成

intdash Edge Agent 2 の本体は、ストリーミング送受信を行うストリーマー（intdash-agent-streamer）と、遅延アップロードやローカルでの時系列データの管理を行うデーモン（intdash-agentd）、エンドツーエンド通信を行う E2E コーラー（intdash-agent-caller）です。

ユーザーは専用コマンド（`intdash-agentctl`）を使ってこれらを操作します。

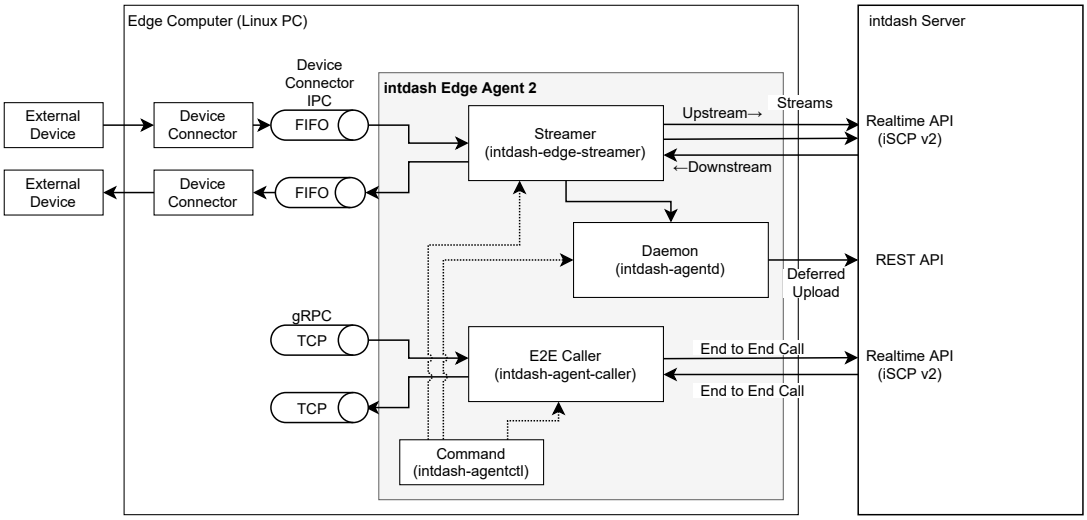



図 1　intdash Edge Agent 2 と外部デバイス、intdash サーバー

ストリーマー、デーモン、E2E コーラー、コマンドの機能はそれぞれ以下のとおりです。

### 1.2.1 ストリーマー（intdash-agent-streamer）

ストリーマー（intdash-agent-streamer）は、intdash サーバーとの間で、リアルタイム API（iSCP）を使って時系列データのストリーミング送信または受信を行うプログラムです。ストリーマーの起動から終了までの時系列データが、intdash の 1 つの計測として扱われます。リアルタイムに送信できなかった時系列データは intdash-agentd に渡され、遅延アップロードされます。

また、ストリーマーでは、時系列データに対するフィルタリング／サンプリング処理も行うことができます。

ストリーマーは `intdash-agentctl run` コマンドによって起動されます。

> **注釈:** intdash において、リアルタイム API を使ってエッジからサーバーに時系列データを送信する経路を「アップストリーム」と呼び、逆に、サーバーからエッジにデータを送信する経路を「ダウンストリーム」と呼びます。単に「ストリーム」と呼んだ場合は、アップストリームまたはダウンストリーム双方のことを指します。

### 1.2.2 デーモン（intdash-agentd）

intdash-agentd は、遅延アップロード、設定の管理、ローカルでの時系列データの管理を行います。

- ネットワークの切断や帯域不足によりストリーマーがサーバーに送信できなかった時系列データは、後から intdash-agentd によりアップロードされます（遅延アップロード）。
- ユーザーが intdash-agentctl コマンドを使って intdash Edge Agent 2 の設定を変更すると、その設定は intdash-agentd によりファイルシステム上に保存されます。
- intdash Edge Agent 2 における時系列データの管理は、intdsah-agentd によって行われます。

インストールされた直後の設定では、エッジデバイスが起動すると intdash-agentd は自動的に起動されます。

### 1.2.3 E2E コーラー（intdash-agent-caller）

intdash での時系列データの送受信には iSCP コネクション上のストリームが使用されますが、これとは別に iSCP コネクション上でエッジからエッジへのエンドツーエンド通信を行うための方法としてエンドツーエンドコール（E2E コール）がありあす。

intdash-agent-caller は、このエンドツーエンド通信を行うためのプログラムです。

E2E コーラーはエッジデバイス上で gRPC サーバーとして振る舞います。エンドツーエンドコールの送信側エッジにおいては、エッジデバイス上で起動しているプロセスが、E2E コーラーに対してリモートプロシージャコールを行うことでエンドツーエンドコールを送信することができます。一方、受信エッジ側では、同様に E2E コーラーに対するリモートプロシージャコールによってコールを受信することができます。

詳細については エンドツーエンド通信 (p. 65) を参照してください。また、E2E コーラーのリモートプロシージャコールで使用する gRPC インターフェイスについては、Agent E2E Call API (p. 70) を参照してください。

E2E コーラーもストリーマーと同様に `intdash-agentctl run` コマンドによって起動されます。

### 1.2.4 コマンド（intdash-agentctl）

intdash-agentctl は、前述のストリーマーやデーモン、E2E コーラーを操作するためのコマンドです。

以下の構文で使用します。

```
intdash-agentctl <command> [<command_options>] [<arguments>...]
```

例えば、`intdash-agentctl config connection --get` を実行すると、intdash-agentd が管理している connection の設定値が返されます。

また、`intdash-agentctl run` を実行すると、現在の設定に従ってストリーマーおよび E2E コーラーが起動されます。

詳細については、intdash-agentctl コマンド (p. 74) を参照してください。

## 1.3 デバイスコネクター

外部デバイス（データを生成するセンサー、または、制御信号を受けて動作するアクチューエーターなど）との間でデータをやり取りするには、外部デバイスと intdash Edge Agent 2 の間を仲介するソフトウェアが必要です。このソフトウェアを「デバイスコネクター」と呼びます。

デバイスコネクターは、intdash Edge Agent 2 と同じエッジデバイスにインストールして使用します。デバイスコネクターと intdash Edge Agent 2 の間のデータの受け渡しには、Agent により作成される FIFO（名前付きパイプ）を使います。

# 02 インストール

## 2.1 システム要件

intdash Edge Agent 2 を正しくインストールするには以下の要件を満たす必要があります。

**最低ハードウェア要件**

- Intel Atom プロセッサー E3815、1.46GHz 相当以上

**推奨ハードウェア要件**

- マルチコア CPU
- 2GB 以上のメモリー
- SSD

**ディストリビューションとアーキテクチャー**

| ディストリビューション | バージョン | アーキテクチャー |
|---|---|---|
| Ubuntu | 22.04(LTS), 20.04(LTS), 18.04(LTS) | x86_64 (or amd64), armhf, arm64 |
| Debian | 11, 10, 9 | x86_64 (or amd64), armhf, arm64 |

## 2.2 インストールする

**警告:** intdash Edge Agent 2 は、旧 intdash Edge Agent と互換性がありません。旧 intdash Edge Agent を使用していたデバイスに intdash Edge Agent 2 をインストールする場合は、旧 intdash Edge Agent をアンインストールしてください。
旧 intdash Edge Agent のアンインストールは以下のコマンドで行います。

```
$ sudo apt-get purge intdash-edge
```

intdash Edge Agent 2 は、アプトポッドの公開リポジトリで「intdash-edge-agent」パッケージとして提供されています。intdash Edge Agent 2 をインストールするには、インストール先エッジデバイスのターミナルで以下のコマンドを実行します。

コマンド内の `${DISTRIBUTION}` には、ご使用の環境に応じて、`ubuntu` または `debian` を指定してください。

```
$ sudo apt-get update
$ sudo apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    lsb-release
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://repository.aptpod.jp/intdash-edge/linux/${DISTRIBUTION}/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/intdash-edge.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/intdash-edge.gpg] \
    https://repository.aptpod.jp/intdash-edge/linux/${DISTRIBUTION} \
    $(lsb_release -cs) \
    stable" \
    | sudo tee /etc/apt/sources.list.d/intdash-edge.list
```

(次のページに続く)

(前のページからの続き)

```
$ sudo apt-get update
$ sudo apt-get install -y intdash-edge-agent2
```

**注釈:** 上記のインストール手順では、推奨される依存パッケージも含めてインストールされます。
必要最低限の機能のみインストールしたい場合は、最後の `apt-get install` コマンドに `--no-install-recommends` オプションを付けて実行してください。

```
$ sudo apt-get install -y --no-install-recommends intdash-edge-agent2
```

**注釈:** 特定のユーザーに intdash Edge Agent 2 を実行するための権限を与えたい場合は、以下のコマンドを実行してください。`<user_name>` に、権限を付与するユーザー名を指定します。

```
$ sudo usermod -aG intdash <user_name>
```

インストール後、以下のコマンドを実行すると、intdash Edge Agent 2 のバージョンが表示されます。

```
$ intdash-agentctl --version
intdash Edge Agent 2 version 1.0.0
```

**注釈:** intdash Edge Agent 2 の詳細なバージョンを確認したい場合は `intdash-agentctl about` コマンドを実行してください。

```
$ intdash-agentctl about
intdash Edge Agent 2 version 1.2.0

Modules:
  intdash-agentctl 1.2.0
  intdash-agent-streamer 1.2.0
  intdash-agent-caller 1.2.0
  intdash-agentd version 1.2.0 (built with go1.20.3)

Environment variables:
  AGENT_LOG                   Log level (t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet])
  AGENT_CLOCK_ID              Clock ID (CLOCK_MONOTONIC|CLOCK_MONOTONIC_RAW) to use for timestamp␣
↪calculation
  AGENT_RUN_DIR               Runtime directory for Agent
  AGENT_LIB_DIR               State directory for Agent
  AGENT_CONF_DIR              Configuration directory for Agent
  AGENT_DAEMON                Executable file name of Agent Daemon
  AGENT_STREAMER              Executable file name of Agent Streamer
  AGENT_CALLER                Executable file name of Agent Caller
```

# 03 Docker による起動

intdash Edge Agent 2 は、Docker イメージを使って起動することも可能です。Amazon ECR Public Gallery で配布されているイメージ を使用してください。

使用方法は以下のとおりです。

## 3.1 準備: 設定ファイルを用意する

intdash Edge Agent 2 のデフォルトの設定を書き出し、ホストマシンにファイルとして保存します。

```
$ docker run --rm public.ecr.aws/aptpod/intdash-edge-agent2 intdash-agentctl config-file show --default >/tmp/
↪agent.yaml
```

設定を編集します。

```
$ vi /tmp/agent.yaml
```

設定の詳細については、intdash Edge Agent 2 設定一覧 (p. 86) を参照してください。

ホストマシンが Linux の場合は、ホストマシン側で以下のように `agent.yaml` のパーミッションを変更します。

```
$ chmod 666 /tmp/agent.yaml
```

## 3.2 intdash Edge Agent 2 を起動する

intdash Edge Agent 2 を実行するため、2 つのコンテナを起動します。以下のいずれかの方法で起動してください：


- docker run で起動する場合 (p. 14)
- docker compose で起動する場合 (p. 16)


### 3.2.1 docker run で起動する場合

intdash Edge Agent 2 のデーモンとストリーマーをそれぞれ起動します。

#### デーモンを起動する

準備: 設定ファイルを用意する (p. 14) で作成した `agent.yaml` をバインドマウントしてデーモンを起動します。環境変数を使って接続先サーバーや認証に関する情報を上書きすることができます。

```
$ docker run \
  --mount type=volume,src=intdash-edge-agent2-run,dst=/var/run/intdash \
  --mount type=volume,src=intdash-edge-agent2-lib,dst=/var/lib/intdash \
  --mount type=bind,src=/tmp/agent.yaml,dst=/var/lib/intdash/agent.yaml \
  --network host \
  --env AGENT_INTDASH_SERVER_URL=https://xxxxx.intdash.jp \
  --env AGENT_INTDASH_PROJECT_UUID=00000000-0000-0000-0000-000000000000 \
```

(次のページに続く)

(前のページからの続き)

```
  --env AGENT_INTDASH_EDGE_UUID=03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx \
  --env AGENT_INTDASH_CLIENT_SECRET=sEh9ZHP.......................iBn5fn_eFM.EXAMPLE \
  public.ecr.aws/aptpod/intdash-edge-agent2 intdash-agentd serve
```

### 設定を変更する（オプション）

デーモン起動後に設定を変更したい場合は、以下のようなコマンドで変更することができます。

アップストリームの設定：

```
$ docker run \
  --mount type=volume,src=intdash-edge-agent2-run,dst=/var/run/intdash \
  --mount type=volume,src=intdash-edge-agent2-lib,dst=/var/lib/intdash \
  --mount type=bind,src=/tmp/agent.yaml,dst=/var/lib/intdash/agent.yaml \
  --network host \
  --rm public.ecr.aws/aptpod/intdash-edge-agent2 intdash-agentctl \
  config upstream --create '
    id: sample-upstream
  '
```

デバイスコネクター IPC の設定：

```
$ docker run \
  --mount type=volume,src=intdash-edge-agent2-run,dst=/var/run/intdash \
  --mount type=volume,src=intdash-edge-agent2-lib,dst=/var/lib/intdash \
  --mount type=bind,src=/tmp/agent.yaml,dst=/var/lib/intdash/agent.yaml \
  --network host \
  --rm public.ecr.aws/aptpod/intdash-edge-agent2 intdash-agentctl \
  config device-connector upstream --create '
    id: up-hello
    data_name_prefix: v1/1/
    dest_ids:
      - sample-upstream
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-hello.fifo
  '
```

### ストリーマーを起動する

同様に、以下のコマンドでストリーマーも起動します。

```
$ docker run \
  --name intdash-agentctl \
  --mount type=volume,src=intdash-edge-agent2-run,dst=/var/run/intdash \
  --mount type=volume,src=intdash-edge-agent2-lib,dst=/var/lib/intdash \
  --mount type=bind,src=/tmp/agent.yaml,dst=/var/lib/intdash/agent.yaml \
  --network host \
  --rm public.ecr.aws/aptpod/intdash-edge-agent2 intdash-agentctl run
```

#### FIFO にデータを書き込む

ストリーマーの実行中に FIFO にデータを書き込むことで、サーバーにデータを送信できます。以下は、FIFO にデータを書き込む例です（FIFO として `/var/run/intdash/up-hello.fifo` を使用する設定になっているものとします）。

```
$ docker exec -it $(docker ps -q --filter "name=intdash-agentctl") /bin/bash
$ echo -en \
    "\xd2\x04\x00\x00\x15\xcd\x5b\x07\x06\x00\x02\x00\x05\x00\x00\x00\x73\x74\x72\x69\x6e\x67\x61\x62\x48\x65\
↪x6c\x6c\x6f" \
    >/var/run/intdash/up-hello.fifo
```

または、コンテナの外から FIFO にデータを書き込むこともできます。以下の例では、別のコンテナで device-connector-intdash を実行して、そこから FIFO にデータを書き込んでいます。

```
$ docker run \
  --mount type=volume,src=intdash-edge-agent2-run,dst=/var/run/intdash \
  -e DC_REPEAT_PROCESS_SRC_CONF_COMMAND='echo "hello intdash!"' \
  -e DC_REPEAT_PROCESS_SRC_CONF_INTERVAL_MS=1000 \
  -e DC_ISCP_V2_COMPAT_FILTER_CONF_CONVERT_RULE_STRING_NAME=hello \
  -e DC_PRINT_LOG_FILTER_CONF_TAG=hello \
  -e DC_FILE_SINK_CONF_PATH=/var/run/intdash/up-hello.fifo \
  --rm public.ecr.aws/aptpod/device-connector-intdash --config /etc/dc_conf/repeat_process_string.yml
```

### 3.2.2 docker compose で起動する場合

docker compose を使ってデーモンとストリーマーをまとめて起動します。

#### docker-compose.yml を作成する

以下の例のように docker-compose.yml を作成します。ここでは、準備: 設定ファイルを用意する (p. 14) で作成した設定ファイル `agent.yaml` をバインドマウントしてデーモンとストリーマーを起動するよう設定しています。

```
$ cat << EOF > docker-compose.yml
version: '3'

services:
  intdash-agentd:
    image: public.ecr.aws/aptpod/intdash-edge-agent2
    container_name: intdash-agentd
    healthcheck:
      test: intdash-agentd ping
      interval: 1s
      timeout: 5s
      retries: 3
      start_period: 60s
    command: intdash-agentd serve
    environment:
      # You can override the credentials with environment variables
      - AGENT_INTDASH_SERVER_URL=https://xxxxx.intdash.jp
      - AGENT_INTDASH_PROJECT_UUID=00000000-0000-0000-0000-000000000000
      - AGENT_INTDASH_EDGE_UUID=03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
      - AGENT_INTDASH_CLIENT_SECRET=sEh9ZHP.......................iBn5fn_eFM.EXAMPLE
    stop_grace_period: 60s
    volumes:
      - intdash-edge-agent2-run:/var/run/intdash
```

（次のページに続く）

(前のページからの続き)

```
      - intdash-edge-agent2-lib:/var/lib/intdash
      - /tmp/agent.yaml:/var/lib/intdash/agent.yaml
    networks:
      - intdash

  intdash-agentctl:
    image: public.ecr.aws/aptpod/intdash-edge-agent2
    container_name: intdash-agentctl
    depends_on:
      intdash-agentd:
        condition: service_healthy
    healthcheck:
      test: pidof intdash-agent-streamer
      interval: 1s
      timeout: 1s
      retries: 1
      start_period: 10s
    command: intdash-agentctl run --address intdash-agentd:50051
    stop_grace_period: 30s
    volumes:
      - intdash-edge-agent2-run:/var/run/intdash
      - intdash-edge-agent2-lib:/var/lib/intdash
      - /tmp/agent.yaml:/var/lib/intdash/agent.yaml
    networks:
      - intdash

volumes:
  intdash-edge-agent2-run:
    name: intdash-edge-agent2-run
  intdash-edge-agent2-lib:
    name: intdash-edge-agent2-lib

networks:
  intdash:
EOF
```

### コンテナを起動する

以下のコマンドでコンテナを起動します。

```
$ docker compose up
```

### FIFO にデータを書き込む

ストリーマーの実行中に FIFO にデータを書き込むことで、サーバーにデータを送信できます。FIFO にデータを書き込む (p. 16) の例を参照してください。

# 04 ライセンス

### intdash Edge Agent 2

intdash Edge Agent 2 は、Apache License, Version 2.0 によりライセンスされています。

ライセンステキストは、`/usr/share/doc/intdash-edge-agent2/copyright` としてインストールされます。

### device-connector-intdash

付属デバイスコネクター device-connector-intdash は、Apache License, Version 2.0 によりライセンスされています。

ライセンステキストは、`/usr/share/doc/device-connector-intdash/copyright` としてインストールされます。

# 05 チュートリアル 1: アップストリームで送信する

このチュートリアルでは、intdash Edge Agent 2 から intdash サーバーに、「Hello」という文字列を送信します（アップストリーム）。

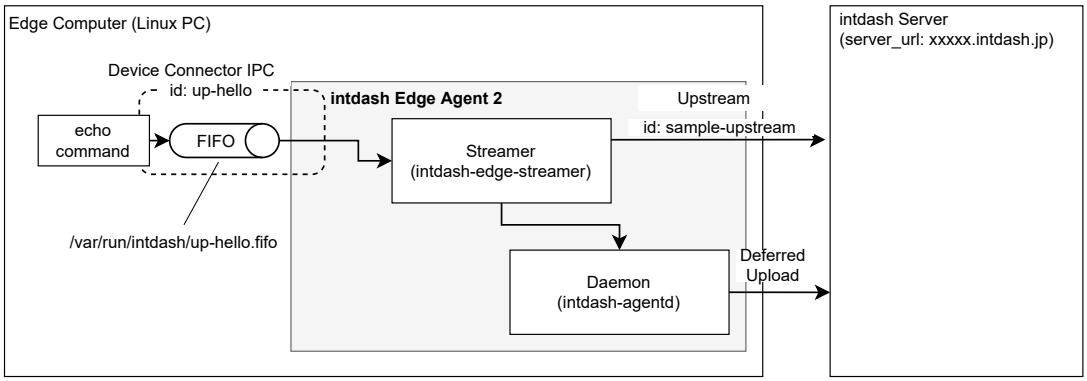



図 2　チュートリアル 1

操作手順は以下のようになります。

1. 接続先サーバーを設定する
2. アップストリームの設定を作成する
3. デバイスコネクター IPC の設定を作成する
4. リアルタイム送信を開始する
5. データを FIFO に書き込む（ここでは手順を簡単にするため、デバイスコネクターによる FIFO への書き込みを echo コマンドで代替します）

## 5.1 準備するもの

このチュートリアルを実行するには以下が必要です。

- intdash Edge Agent 2 をインストール済みのエッジデバイス
- 接続先の intdash サーバーのホスト名（例：xxxxx.intdash.jp）
- プロジェクト UUID
- このエッジデバイスに割り当てる、intdash のエッジ UUID とクライアントシークレット
  - エッジ UUID とクライアントシークレットは、ウェブブラウザーで intdash にログイン後、My Page で発行することができます。エッジは使用するプロジェクトに所属させておいてください。

## 5.2 接続先サーバーを設定する

> **注釈:** デーモン（intdash-agentd）の自動起動が行われない環境（Docker Hub で公開されている Ubuntu など）をご使用の場合は、`sudo /etc/init.d/intdash-agentd start` でデーモンを起動してから以下の操作を行ってください。

intdash-agentctl config コマンドを使って、接続先サーバーと認証情報を設定します。

```
$ intdash-agentctl config connection --modify '
    server_url: https://xxxxxx.intdash.jp
    project_uuid: 00000000-0000-0000-0000-000000000000
    edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
    client_secret: sEh9ZHP.........................iBn5fn_eFM.EXAMPLE
  '
```

`--modify` に続く引数は YAML 形式の文字列です。あらかじめ準備した以下の情報を設定してください。

| キー | 入力する値 |
|---|---|
| server_url | 接続先の intdash サーバーのホスト名 |
| project_uuid | 使用するプロジェクトの UUID |
| edge_uuid | エッジの UUID |
| client_secret | エッジのクライアントシークレット |

**注釈:** YAML 形式ではなく JSON 形式で設定を与えたい場合は、config コマンドの後に `-j` を指定してください。

```
$ intdash-agentctl config -j connection --modify '{
    "server_url": "https://xxxxxx.intdash.jp",
    "project_uuid": "00000000-0000-0000-0000-000000000000",
    "edge_uuid": "03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx",
    "client_secret": "sEh9ZHP.........................iBn5fn_eFM.EXAMPLE"
  }'
```

**注釈:** intdash-agentctl コマンドの使用方法は `-h` オプションにより確認できます。コマンドごとに異なる情報が表示されます。

```
$ intdash-agentctl -h
$ intdash-agentctl config -h
$ intdash-agentctl config connection -h
```

### 5.2.1 アップストリームの設定を作成する

次に、iSCP によるデータ送信の経路（アップストリーム）の設定を作成します。ここでは ID のみを指定していますが、これにより、デフォルト設定のアップストリームが作成されます。アップストリームの ID は任意の文字列を設定可能ですが、ここでは ID を `sample-upstream` とします。

```
$ intdash-agentctl config upstream -c '
    id: sample-upstream
  '
```

### 5.2.2 デバイスコネクター IPC の設定を作成する

デバイスコネクター IPC の設定（使用する FIFO のパスなど）を、以下のコマンドで作成します。

データはデバイスコネクターから intdash Edge Agent 2 へ流れ、その後 intdsah サーバーへ流れるため、ここではアップストリーム方向 (upstream) のデバイスコネクター IPC 設定を作成します。ID には任意の文字列を設定可能です。

```
$ intdash-agentctl config device-connector upstream --create '
    id: up-hello
    data_name_prefix: v1/1/
    dest_ids:
      - sample-upstream
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-hello.fifo
  '
```

このコマンドにより以下の設定が行われます。

| キー | 値 |
|---|---|
| id | このデバイスコネクター IPC 設定に与える任意の ID |
| data_name_prefix | デバイスコネクターで取得したデータポイントに与えるデータ名称の接頭辞 |
| dest_ids | デバイスコネクターで取得したデータポイントを intdash サーバーに送信する際に使用する、行先のアップストリームの ID（ここでは、さきほど作成した `sample-upstream` というアップストリームを指定しています。） |
| format | デバイスコネクターとの通信に使用するデータフォーマット（ここでは、`iscp-v2-compat` を使用します。） |
| ipc.type | intdash Edge Agent 2 とデバイスコネクターの間のデータ通信方法（現在 `fifo` のみに対応しているため、`fifo` を指定してください。） |
| ipc.path | intdash Edge Agent 2 がデバイスコネクターからデータを受け取るための FIFO のパス。この FIFO に書き込まれたデータが intdash Edge Agent 2 に渡されます。 |

**重要:** 受信側で旧バージョンの iSCP を使用する場合は、送信側はルールに沿って `data_name_prefix` を与える必要があります。詳細は iSCP v1 互換のデータ ID (p. 97) を参照してください。

**注釈:** intdash-agentctl のコマンドでは短縮形を使用することもできます。以下のコマンドは上記の例と同等です。

```
$ intdash-agentctl conf dc up -c '
    id: up-hello
    data_name_prefix: v1/1/
    dest_ids:
      - sample-upstream
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-hello.fifo
  '
```

コマンドの短縮形はヘルプで確認することができます。

```
$ intdash-agentctl conf -h
```

## 5.3 リアルタイム送信を開始する

以下のコマンドを実行します。

```
$ intdash-agentctl run
```

これによりストリーマーが起動され、また、デバイスコネクターからデータを受け取るための /var/run/intdash/up-hello.fifo が作成されます。

この時点で intdash サーバーとの接続が確立され、intdash 上は計測が開始されたことになりますが、まだデバイスコネクターからデータポイントが流れてこないため、送信されません。

## 5.4 データを FIFO に書き込む

echo コマンドを使用して FIFO にバイナリデータを書き込みます。それを受け取った intdash Edge Agent 2 がデータをサーバーに送信することを確認します。新しいターミナルを開いて、以下のコマンドを実行してください。

```
$ echo -en \
  "\xd2\x04\x00\x00\x15\xcd\x5b\x07\x06\x00\x02\x00\x05\x00\x00\x00\x73\x74\x72\x69\x6e\x67\x61\x62\x48\x65\
↪x6c\x6c\x6f" \
  >/var/run/intdash/up-hello.fifo
```

上記で echo によって書き込まれるバイナリデータは、「Hello」という文字列に、以下の表のようにデータ型、タイムスタンプ、データ ID などを付与し、専用のフォーマット（FIFO 用データフォーマット）に従ってエンコードしたものです。

**注釈:** echo ではなく Python を使ってバイナリデータを作成する場合は以下のようにします。

```
$ python3 -c "import struct, sys; sys.stdout.buffer.write(struct.pack('<LLHHL6s2s5s', 1234, 123456789, 6,␣
↪2, 5, b'string', b'ab', b'Hello'))" > /var/run/intdash/up-hello.fifo
```

データを FIFO に書き込む際のバイナリデータのフォーマットについては、FIFO 用データフォーマット (p. 100) を参照してください。デバイスコネクター IPC 設定で指定したとおり、ここでは、データフォーマット iscp-v2-compat を使用しています。

| フィールド名 | 値 |
|---|---|
| タイムスタンプ | 1234.123456789 秒（リトルエンディアン:整数秒部 \xd2\x04\x00\x00 、ナノ秒部 \x15\xcd\x5b\x07 ） |
| Data Type Length | 6（リトルエンディアン: \x06\x00 ） |
| Data Name Length | 2（リトルエンディアン: \x02\x00 ） |
| Payload Length | 5（リトルエンディアン: \x05\x00\x00\x00 ） |
| Data Type | string（UTF-8 エンコード: \x73\x74\x72\x69\x6e\x67 ） |
| Data Name | ab（UTF-8 エンコード: \x61\x62 ） |
| Payload | Hello（UTF-8 エンコード: \x48\x65\x6c\x6c\x6f ） |

このように FIFO にデータポイントを書き込むと、intdash Edge Agent 2 がサーバーに送信するデータポイントのデータ ID（ <データ型>:<データ名称> ）は、string:v1/1/ab となります。これは、デバイスコネクター IPC 設定において data_name_prefix: v1/1/ のように設定され、FIFO に書き込んだバイナリデータにおいて、Data Type として string が指定され、ID として ab が指定されているためです。

**注釈:** プレフィックスの v1/1/ は、旧バージョンの iSCP でダウンストリームできるようにするための設定です。このあとの手順で可視化できるようにするために、設定しておく必要があります。

## 5.5 Edge Finder で確認する

ウェブブラウザーで Edge Finder を開き、使用しているエッジのトラフィック画面を表示します。データが送信されていることが確認できます。

**注釈:** Edge Finder のトラフィック画面では、ページを開いたあとに受信したデータしか表示されません。トラフィック画面を開いてから、前の手順の echo コマンドを繰り返してください。


図 3　Edge Finder でトラフィックを確認する

確認が済んだら、`intdash-agentctl run` を実行したターミナルで Ctrl+C を押して、リアルタイム送信を終了します。

# 06 チュートリアル 2: ダウンストリームで受信する

このチュートリアルでは、チュートリアル 1 (p. 19) でサーバーに送信されているデータを、別の intdash Edge Agent 2 で受信します。

intdash サーバーから受信したデータは、通常、FIFO を使ってデバイスコネクターに渡され、デバイスコネクターから外部デバイスに渡されますが、ここでは手順を簡単にするため、デバイスコネクターの代わりに cat コマンドを使ってデータを確認します。

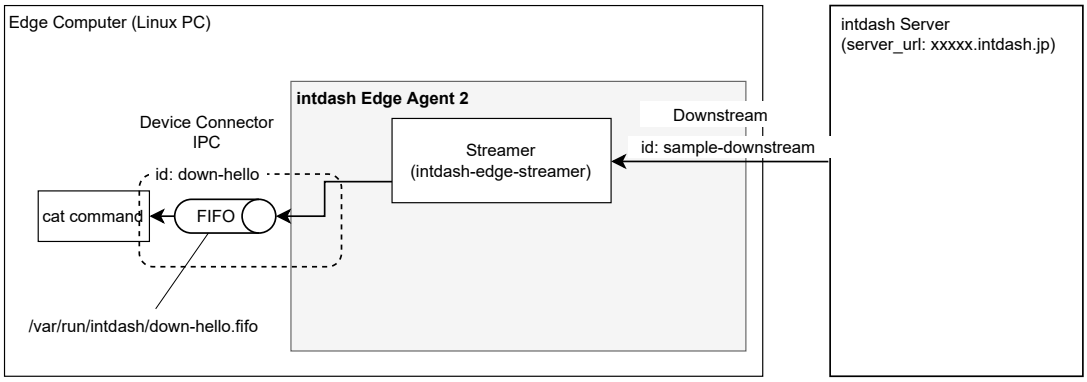



図 4　チュートリアル 2

操作手順は以下のようになります。

1. 接続先サーバーを設定する
2. デバイスコネクター IPC の設定を作成する
3. ダウンストリームの設定を作成する
4. リアルタイム受信を開始する
5. データを確認する

## 6.1 準備するもの

**注釈:** このチュートリアルでは、アップストリーム用エッジとダウンストリーム用エッジの合計 2 つのエッジを使用します。このチュートリアルを実行する前に、先に チュートリアル 1 (p. 19) を行って、アップストリーム用エッジをセットアップしてください。ダウンストリーム用エッジはこのあとの手順でセットアップします。

ダウンストリーム用エッジをセットアップするには、以下が必要です。

- intdash Edge Agent 2 をインストール済みのエッジデバイス
- 接続先の intdash サーバーのホスト名（アップストリーム用エッジに設定したものと同じ）
- プロジェクト UUID（アップストリーム用エッジに設定したものと同じ）
- エッジデバイスに割り当てるエッジ UUID とクライアントシークレット（ダウンストリーム用エッジ）
  - エッジ UUID とクライアントシークレットは、ウェブブラウザーで intdash にログイン後、My Page で発行することができます。エッジは使用するプロジェクトに所属させておいてください。

**注釈:** 1 つのエッジ（1 つの intdash Edge Agent 2）でアップストリームとダウンストリームを行うことも可能です。

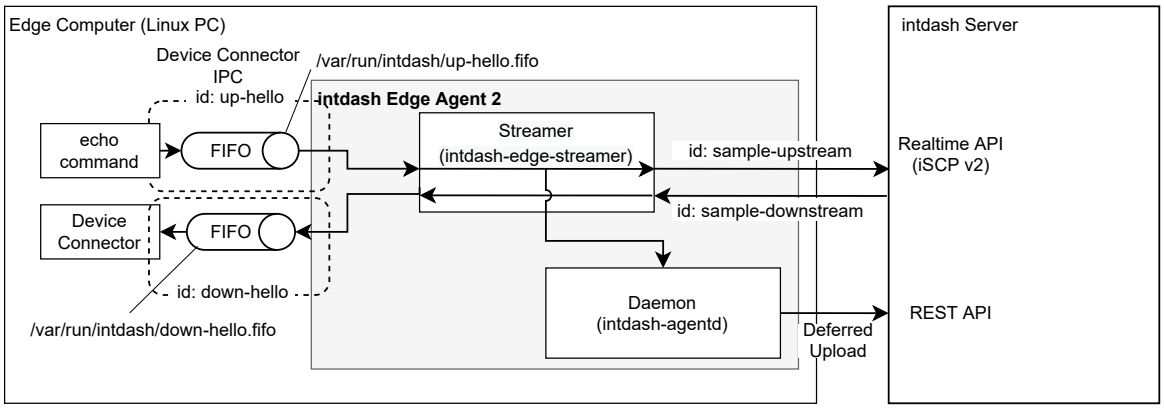



図 5　アップストリームとダウンストリームを 1 つのエッジで行う

## 6.2 接続先サーバーを設定する

### ダウンストリーム用エッジでの操作

**注釈:** デーモン（intdash-agentd）の自動起動が行われない環境（Docker Hub で公開されている Ubuntu など）をご使用の場合は、`sudo /etc/init.d/intdash-agentd start` でデーモンを起動してから以下の操作を行ってください。

intdash-agentctl config コマンドを使って接続先サーバーと認証情報を設定します。

```
$ intdash-agentctl config connection --modify '
    server_url: https://xxxxxx.intdash.jp
    project_uuid: 00000000-0000-0000-0000-000000000000
    edge_uuid: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    client_secret: f1OMTtns.........................JhvNgPJFRk.EXAMPLE
  '
```

`--modify` に続く引数は YAML 形式の文字列です。あらかじめ準備した接続情報を使用してください。

| キー | 入力する値 |
|---|---|
| server_url | intdash サーバーのホスト名 |
| project_uuid | 使用するプロジェクトの UUID |
| edge_uuid | ダウンストリーム用エッジの UUID |
| client_secret | ダウンストリーム用エッジのクライアントシークレット |

### 6.2.1 デバイスコネクター IPC の設定を作成する

**ダウンストリーム用エッジでの操作**

デバイスコネクター IPC の設定（FIFO のパスなど）を、以下のコマンドで新規作成します。

データの流れは「intdash Edge Agent 2 からデバイスコネクターへ」という方向なので、ダウンストリーム方向（downstream）のデバイスコネクター IPC として作成します。

```
$ intdash-agentctl config device-connector downstream --create '
    id: down-hello
    enabled: true
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/down-hello.fifo
  '
```

このコマンドにより以下の設定が行われます。

| キー | 値 |
|---|---|
| id | このデバイスコネクター IPC 設定に与える任意の ID |
| enabled | デバイスコネクター IPC を有効化 (true) |
| format | デバイスコネクターとの通信に使用されるデータフォーマット（ここでは、`iscp-v2-compat` を使用します。） |
| ipc.type | intdash Edge Agent 2 とデバイスコネクターの間のデータ通信方法（現在 `fifo` のみに対応しているため、`fifo` を指定してください。） |
| ipc.path | デバイスコネクターが intdash Edge Agent 2 からデータを受け取るための FIFO のパス。 |

### 6.2.2 ダウンストリームの設定を作成する

**ダウンストリーム用エッジでの操作**

次に、iSCP によるデータ受信の経路（ダウンストリーム）の設定を作成します。ダウンストリームの ID は任意の文字列を設定可能ですが、ここでは `down` という ID とします。

```
$ intdash-agentctl config downstream --create '
    id: sample-downstream
    enabled: true
    dest_ids:
      - down-hello
    filters:
    - src_edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
      data_filters:
      - type: string
        name: v1/1/ab
'
```

| キー | 入力する値 |
|---|---|
| id | このダウンストリームに与える任意の ID |
| enabled | ダウンストリームを有効化 |
| dest_ids | このダウンストリームで受信したデータを渡す先の、デバイスコネクター IPC の ID。さきほど作成した `down-hello` というデバイスコネクター IPC を指定しています。 |
| filters | 受信対象とするデータポイントの指定 |
| filters.[].src_edge_uuid | 受信対象とする送信元のエッジ UUID（チュートリアル 1 で使用したエッジ） |
| filters.[].data_filters | 受信対象とするデータ ID を定義した設定のリスト（チュートリアル 1 の設定に合わせる） |
| filters.[].data_filters.[].type | 受信対象とするデータの型（チュートリアル 1 の設定に合わせる） |
| filters.[].data_filters.[].name | 受信対象とするデータの名前（チュートリアル 1 の設定に合わせる） |

**注釈:** ここで設定している `filters` は、iSCP において「ダウンストリームフィルター」と呼ばれるもので、サーバーから受信する対象のデータポイントを指定するものです。intdash Edge Agent 2 内でのデータの行先を変えるフィルター（フィルタリング／サンプリング (p. 52)）とは別の概念です。

## 6.3 リアルタイム受信を開始する

**ダウンストリーム用エッジでの操作**

以下のコマンドを実行します。

```
$ intdash-agentctl run
```

これによりストリーマーが起動され、`/var/run/intdash/down-hello.fifo` が作成されます。

また、intdash Edge Agent 2 と intdash サーバーとの接続が確立され、サーバーからのデータを待ち受ける状態になります。

## 6.4 データを確認する

**アップストリーム用エッジでの操作**

チュートリアル 1 (p. 19) でセットアップしたアップストリーム用エッジで、`intdash-agentctl run` を実行し、チュートリアル 1 の手順のとおり echo コマンドでデータを書き込みます。データはリアルタイムにサーバーに送信されます。

**ダウンストリーム用エッジでの操作**

サーバーからデータが受信され、デバイスコネクター IPC の設定に従って `/var/run/intdash/down-hello.fifo` に書き込まれます。

確認のため、以下のように `cat` コマンドを使って、FIFO に書き込まれたデータを表示させます。

```
$ cat /var/run/intdash/down-hello.fifo
```

バイナリデータであるためヘッダー部分は認識できる文字にはなりませんが、文字列 `Hello` が表示されており、受信できていることが分かります。

**ダウンストリーム用エッジでの操作**

確認が済んだら、`intdash-agentctl run` を実行したターミナルで Ctrl+C を押して、intdash-agentctl run を

終了します。

**アップストリーム用エッジでの操作**

`intdash-agentctl run` を実行したターミナルで Ctrl+C を押して、intdash-agentctl run を終了します。

# 07 チュートリアル 4: エンドツーエンド通信をする

このチュートリアルでは、エンドツーエンド通信を行います。送信側のエッジデバイスから、intdash Edge Agent 2 と intdash サーバーを経由して、受信側のエッジデバイスまでエンドツーエンドコールを届けます。

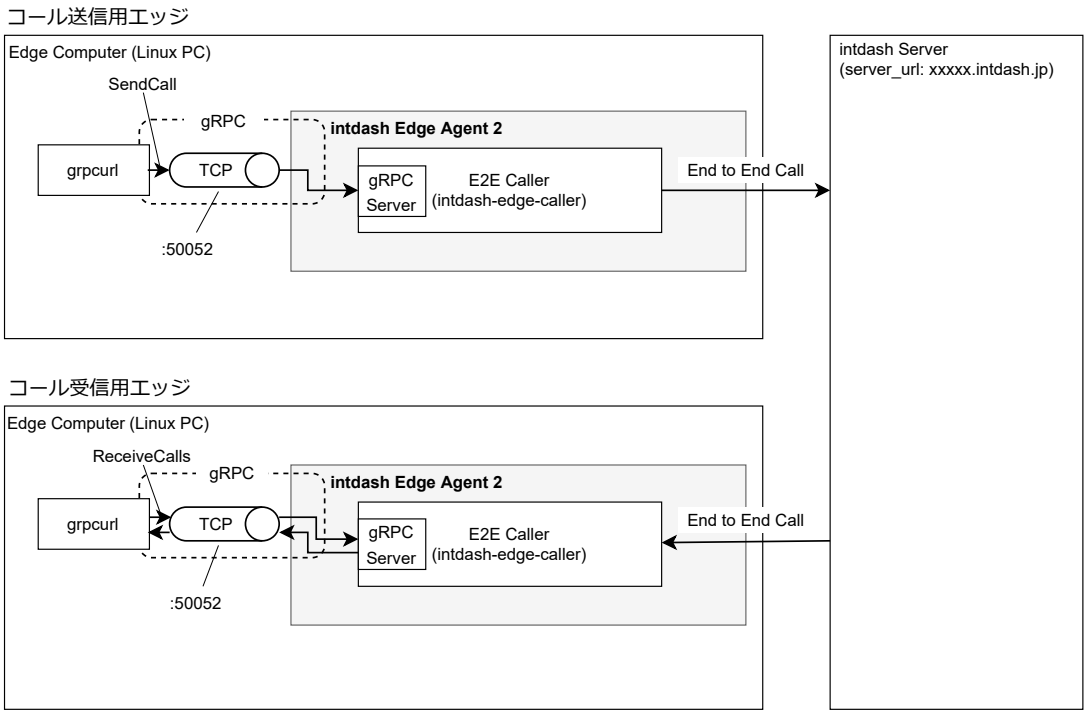



図 6　チュートリアル 4（エンドツーエンド通信をする）

## 7.1 準備するもの

**注釈:** このチュートリアルでは、コール送信用エッジとコール受信用エッジの合計 2 つのエッジを使用します。

このチュートリアルを実行するには以下が必要です。

- intdash Edge Agent 2 をインストール済みのエッジデバイス 2 つ
- 接続先の intdash サーバーのホスト名（例 : xxxxx.intdash.jp）
- プロジェクト UUID
- コール送信用エッジデバイスに割り当てるエッジ UUID とクライアントシークレット
- コール受信用エッジデバイスに割り当てるエッジ UUID とクライアントシークレット
- grpcurl の実行バイナリ（インストールの方法は grpcurl の README.md にある Installation を参照してください。）

エッジ UUID とクライアントシークレットは、ウェブブラウザーで intdash にログイン後、My Page で発行することができます。エッジは使用するプロジェクトに所属させておいてください。

## 7.2 接続先サーバーを設定する

### コール送信用エッジでの操作

**注釈:** デーモン (intdash-agentd) の自動起動が行われない環境 (Docker Hub で公開されている Ubuntu など) をご使用の場合は、`sudo /etc/init.d/intdash-agentd start` でデーモンを起動してから以下の操作を行ってください。

intdash-agentctl config コマンドを使って接続先サーバーと認証情報を設定します。

```
$ intdash-agentctl config connection --modify '
    server_url: https://xxxxxx.intdash.jp
    project_uuid: 00000000-0000-0000-0000-000000000000
    edge_uuid: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    client_secret: f1OMTtns.........................JhvNgPJFRk.EXAMPLE
  '
```

`--modify` に続く引数は YAML 形式の文字列です。あらかじめ準備した接続情報を使用してください。

| キー | 入力する値 |
|---|---|
| server_url | intdash サーバーのホスト名 |
| project_uuid | 使用するプロジェクトの UUID |
| edge_uuid | コール送信用エッジの UUID |
| client_secret | コール送信用エッジのクライアントシークレット |

**注釈:** YAML 形式ではなく JSON 形式で設定を与えたい場合は、config コマンドの後に `-j` を指定してください。

```
$ intdash-agentctl config -j connection --modify '{
    "server_url": "https://xxxxxx.intdash.jp",
    "project_uuid": "00000000-0000-0000-0000-000000000000",
    "edge_uuid": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
    "client_secret": "f1OMTtns.........................JhvNgPJFRk.EXAMPLE"
  }'
```

**注釈:** intdash-agentctl コマンドの使用方法は `-h` オプションにより確認できます。コマンドごとに異なる情報が表示されます。

```
$ intdash-agentctl -h
$ intdash-agentctl config -h
$ intdash-agentctl config connection -h
```

### コール受信用エッジでの操作

intdash-agentctl config コマンドを使って接続先サーバーと認証情報を設定します。

```
$ intdash-agentctl config connection --modify '
    server_url: https://xxxxxx.intdash.jp
    project_uuid: 00000000-0000-0000-0000-000000000000
    edge_uuid: yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy
    client_secret: h7MWjrO.........................XdVdm0Nbs6.EXAMPLE
  '
```

`--modify` に続く引数は YAML 形式の文字列です。あらかじめ準備した接続情報を使用してください。

| キー | 入力する値 |
|---|---|
| server_url | intdash サーバーのホスト名（コール送信用エッジで設定したのと同じ値） |
| project_uuid | 使用するプロジェクトの UUID（コール送信用エッジで設定したのと同じ値） |
| edge_uuid | コール受信用エッジの UUID |
| client_secret | コール受信用エッジのクライアントシークレット |

## 7.3 コールの受信を行う

### 7.3.1 E2E コーラーを起動する

**コール受信用エッジでの操作**

エンドツーエンド通信を行うには、E2E コーラーを起動する必要があります。

以下のコマンドを実行して E2E コーラーを起動します。

```
$ intdash-agentctl run
```

これにより、E2E コーラーが持っている gRPC サーバーがポート 50052 で起動します。

### 7.3.2 コールを受信待ちする

**コール受信用エッジでの操作**

grpcurl を使用して、E2E コーラーの `ReceiveCalls` メソッドをコールします。`ReceiveCalls` はエンドツーエンドコールの受信を指示するメソッドです。

```
$ grpcurl -plaintext 127.0.0.1:50052 agent.e2e.E2ECall/ReceiveCalls
```

> **注釈:** grpcurl のフラグについては `grpcurl -help` で確認できます。

## 7.4 コールの送信を行う

### 7.4.1 E2E コーラーを起動する

**コール送信用エッジでの操作**

以下のコマンドを実行して E2E コーラーを起動します。

```
$ intdash-agentctl run
```

これにより、E2E コーラーが持っている gRPC サーバーがポート 50052 で起動します。

### 7.4.2 コールを送信する

**コール送信用エッジでの操作**

grpcurl を使用して、E2E コーラーの `SendCall` メソッドをコールします。`SendCall` はエンドツーエンドコールの送信を指示するメソッドです。

```
$ grpcurl -plaintext -d '{
  "destination_node_id": "731e9fbd-ec9b-4c52-b959-44f427e03dbf",
  "name": "call",
  "type": "bytes",
  "payload": "aGVsbG8="
}' 127.0.0.1:50052 agent.e2e.E2ECall/SendCall
```

このコマンドにより以下の内容のエンドツーエンドコールが送信されます。`SendCall` メソッドの詳細については、Agent E2E Call API (p. 70) を参照してください。

| キー | 入力する値 |
|---|---|
| destination_node_id | 送信先エッジの UUID |
| name | 名称 |
| type | 型 |
| payload | base64 変換したペイロード |

**重要:** payload に base64 変換が必要な理由
`SendCall` メソッドのリクエストパラメーターとなるメッセージ `SendCallRequest` では、payload は bytes 型のフィールドとして定義されています。
grpcurl のデフォルト設定では入出力は JSON 形式ですが、gRPC メッセージの JSON へのマッピング に従うと、bytes 型は base64 変換を行う必要があります。
そのため、上記のように payload は base64 変換を行ってから JSON に含めています。

## 7.5 受信したコールを確認する

**コール受信用エッジでの操作**

コールの受信待ちを行なっている grpcurl コマンドの標準出力を確認します。

```
$ grpcurl -plaintext -d "{}"  127.0.0.1:50052 agent.e2e.E2ECall/ReceiveCalls
{
  "callId": "f4243196-fc3f-4652-b216-578fbf6acdca",
  "sourceNodeId": "9f9391de-7277-40b9-bd88-78d4065c6869",
  "name": "call",
  "type": "bytes",
  "payload": "aGVsbG8="
}
```

受信したコールの文字列が出力されています。

確認が済んだら、Ctrl+C を押して、grpcurl を終了します。

**コール受信用エッジでの操作**

intdash-agentctl run を実行したターミナルで Ctrl+C を押して、intdash-agentctl run を終了します。

**コール送信用エッジでの操作**

intdash-agentctl run を実行したターミナルで Ctrl+C を押して、intdash-agentctl run を終了します。

# 08 送受信の開始／終了

## 8.1 ストリーマーおよび E2E コーラーを起動する

以下のコマンドを実行すると、ストリーマーおよび E2E コーラーが起動し、データの送受信が開始されます。

```
$ intdash-agentctl run
```

**注釈:** `intdash-agentctl` コマンドで操作を行うためには、intdash-agentd が起動している必要があります。インストールされた直後の設定では intdash-agentd は自動的に起動されますが、手動で起動する場合は、手動によるデーモン起動／終了 (p. 35) を参照してください。

ストリーマーおよび E2E コーラーが起動すると、以下が行われます。

- アップストリームが設定済みの場合は、ストリーマーはデバイスコネクターからデータを受け取り、intdash サーバーのリアルタイム API を使って時系列データの送信を開始します。時系列データの送信開始から終了までが、intdash における 1 つの「計測」になります。
  詳細については、アップストリームによる送信 (p. 38) を参照してください。
- ダウンストリームが設定済みの場合は、ストリーマーは intdash サーバーのリアルタイム API を使って時系列データの受信を開始します。サーバーから受け取ったデータをデバイスコネクターに渡します。
  詳細については、ダウンストリームによる受信 (p. 45) を参照してください。
- E2E コーラーの gRPC サーバーが起動し、他プロセスからのリモートプロシージャコールを待ち受けます。エンドツーエンドコールの送信および受信は、この gRPC サーバーへのコールにより行います。
  詳細については、エンドツーエンド通信 (p. 65) を参照してください。

**注釈:** 遅延アップロードは、ストリーマーの起動や終了とは関係なく intdash-agentd により行われます。

## 8.2 ストリーマーおよび E2E コーラーを終了する

ストリーマーおよび E2E コーラーを終了するには、`intdash-agentctl run` を実行したターミナルで Ctrl+C を押し、intdash-agentctl に SIGINT を送信します。

## 8.3 電源に連動させる

エッジデバイスの電源がオンになったら自動的に `intdash-agentctl run` が実行されてストリーマーおよび E2E コーラーが起動するように設定するには、以下のコマンドを実行します。

```
$ sudo update-rc.d intdash-agentctl-run defaults
```

電源への連動の設定を解除する場合は、以下のコマンドを実行します。

```
$ sudo update-rc.d intdash-agentctl-run remove
```

# 09 手動によるデーモン起動／終了

**重要:** 通常の方法でインストールした場合、エッジデバイスの起動と同時にデーモンが起動される設定になっているため、デーモンの起動や終了を手動で行う必要はありません。デーモンの自動起動が行われない環境（Docker Hub で公開されている Ubuntu など）でデーモンを手動で起動／停止したい場合にこちらの手順を使用します。

デーモン（intdash-agentd）を起動するには、以下のように init スクリプトを実行します。

```
$ /etc/init.d/intdash-agentd start
```

デーモンを終了するには、以下のように init スクリプトを実行します。

```
$ /etc/init.d/intdash-agentd stop
```

# 10 接続先サーバーと認証情報

intdash Edge Agent 2 が intdash サーバーに接続するには、以下の情報が必要です。

- 接続先の intdash サーバーのホスト名（例：xxxxx.intdash.jp）
- プロジェクトの UUID（省略した場合は、Global Project が使用されます）
- このエッジデバイスに割り当てる、intdash のエッジ UUID
- 割り当てられたエッジのクライアントシークレット

**注釈:** プロジェクトの UUID は、Project Console で使用したいプロジェクトを開いた状態で、左上の Copy UUID をクリックすることにより得ることができます。

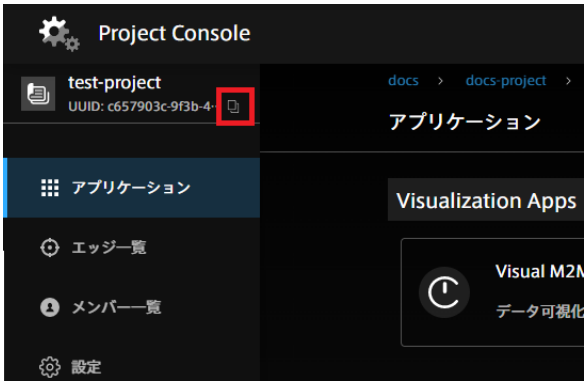



図 7　プロジェクト UUID のコピー

intdash Edge Agent 2 で接続に関する設定を確認するには、以下のコマンドを実行します。

```
$ intdash-agentctl config connection --get
```

設定を変更するには、以下のコマンドを実行します。

```
$ intdash-agentctl config connection --modify <config_in_yaml_format>
```

例: 接続先サーバーと認証情報を設定する。

```
$ intdash-agentctl config connection --modify '
    server_url: https://xxxxxx.intdash.jp
    edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
    client_secret: f1OMTtnsNpE3LAhOarkUqgMyw...............NgPJFRk
  '
```

`config_in_yaml_format` で設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| server_url | string | 接続先 intdash サーバーの URL |
| project_uuid | string | プロジェクト UUID |
| edge_uuid | string | intdash サーバーに接続する際に使用する、このエッジの UUID |
| client_secret | string | intdash サーバーに接続する際に使用するクライアントシークレット |

intdash-agentctl config connection コマンドの使用方法については、connection サブコマンド (p. 75) を参照してください。

# 11 トランスポートの設定

iSCP によるリアルタイム送受信で使用するトランスポートプロトコルを設定します。QUIC または WebSocket のいずれかを選択することができます。（ここで設定するのは、iSCP によるリアルタイム送受信用に使われるトランスポートプロトコルです。遅延アップロードは常に HTTP による REST API で行われます。）

QUIC と WebSocket はいずれも到達保証と順序保証のあるプロトコルです。ただし、QUIC を選択し、ストリームの設定で `qos: unreliable` を指定すると、到達保証と順序保証のないトランスポート（QUIC DATAGRAM）が使用されます。そのため、QUIC と WebSocket はそれぞれ以下のような場合に向いています。

- QUIC: 帯域が十分でない環境で、欠損があってもよいので回線詰まりによる遅延が発生しない方法でデータを確認したい
- WebSocket: 帯域が十分でない環境で、回線詰まりによる遅延があってもよいので欠損なくデータを確認したい

例えば、リアルタイム性を重視して動画を送信したいときには、QUIC を選択した上で、動画内のサブフレームを送信するのに `qos: unreliable` のストリームを使用すると効果的です。サブフレームが欠損した場合、画質は悪化しますが、再生は継続できます。

**重要:** QUIC を使用するためにはサーバー側で設定が必要です。

現在の設定を確認するには、以下のコマンドを実行します。

```
$ intdash-agentctl config transport --get
```

設定を変更するには、以下のコマンドを実行します。

```
$ intdash-agentctl config transport --modify <config_in_yaml_format>
```

例: QUIC を使用するように設定する

```
$ intdash-agentctl config transport --modify 'protocol: quic'
```

`config_in_yaml_format` で設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| protocol | string | • `quic`: QUIC による接続<br>• `websocket`: WebSocket による接続 |

# 12 アップストリームによる送信

intdash サーバーにリアルタイムに時系列データを送信する経路（ストリーム）をアップストリームと呼びます。アップストリームによる送信を行うには以下の準備が必要です。本章ではこれらについて説明します。接続先サーバーと認証情報 (p. 36) は設定済みとします。

- アップストリームの設定を作成する
- デバイスコネクター IPC の設定を作成する
- （必要な場合のみ）フィルターを設定する（フィルターを使用すると、条件に一致したデータポイントの行先のアップストリームを変更することができます）

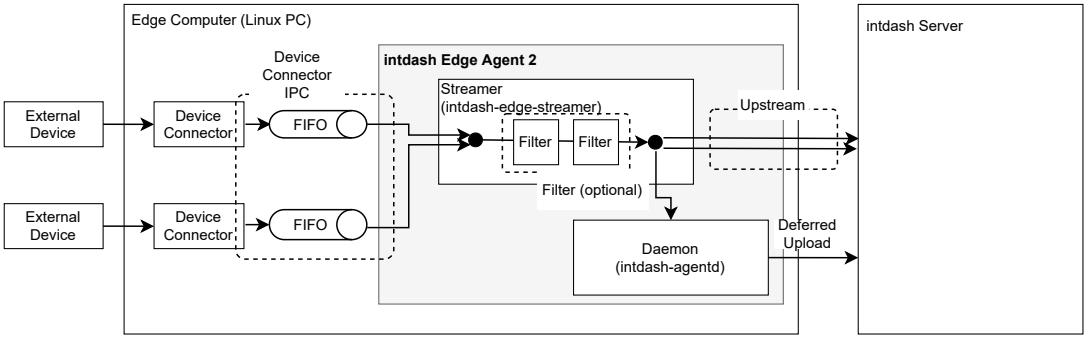



図 8　リアルタイムデータをサーバーに送信する

以上の設定を行ったうえで `intdash-agentctl run` コマンドを実行すると、ストリーマーが起動され、サーバーへのリアルタイム送信が開始されます。ストリーマーの起動から終了までの一連の時系列データが 1 つの計測となります。

送信中に通信が切断された場合や帯域が不十分だった場合、送信できなかったデータはローカルストレージに蓄積され、通信が回復したときに REST API で送信されます。これを遅延アップロードと呼びます。

## 12.1 アップストリーム

intdash Edge Agent 2 では、設定の異なる複数のアップストリームを設定し、同時に使用することができます。センサーデバイスからデバイスコネクターを使って取得されたデータは、指定されたアップストリームを使ってサーバーに送信されます。

各アップストリームについては、通信が切断された場合に遅延アップロードをするか、信頼性のある接続を使用するか、サーバーでのデータ永続化するか、などの設定が可能です。

アップストリームの設定は ID により管理されます。デバイスコネクター IPC 設定やフィルター設定において、行先のストリームを指定する際には、この ID で指定します。

アップストリームを作成するには以下のコマンドを実行します。

```
$ intdash-agentctl config upstream --create <config_in_yaml_format>
```

例:

```
$ intdash-agentctl config upstream --create '
    id: recoverable
    enabled: true
    recover: true
    persist: true
```

（次のページに続く）

(前のページからの続き)

```
    qos: unreliable
    flush_policy: interval
    flush_interval: 5
  '
```

作成済みのアップストリームの設定を変更するには、ID と、変更したい設定を与えます。

```
$ intdash-agentctl config upstream --modify <id> <config_in_yaml_format>
```

設定できるパラメーターについては、以下を参照してください。

| キー | 型 | 説明 |
|---|---|---|
| id | string | アップストリームを識別するための文字列。deferred は予約されているため使用できません。 |
| enabled | bool | アップストリームの有効 (true) ／無効 (false) |
| recover | bool | 送信ができなかった場合の遅延アップロードの有効 (true) ／無効 (false)。true にすると、リアルタイム送信できなかった場合に、遅延アップロードが行われます。ただし、persist が false の場合、遅延アップロードは行われません。 |
| persist | bool | サーバーでのデータ永続化の有効 (true) ／無効 (false) |
| qos | string | partial: iSCP の QoS として PARTIAL を指定します。<br>unreliable: iSCP の QoS として UNRELIABLE を指定します。 |
| flush_policy | string | interval - リアルタイム送信を一定周期でフラッシュします。<br>immediately - リアルタイム送信をデータポイントごとにフラッシュします。 |
| flush_interval | string | リアルタイム送信をフラッシュする間隔（ミリ秒）。flush_policy が interval の場合のみ使用されます。 |

**注釈:** アップストリーム設定の一覧表示、削除などのコマンドについては、intdash-agentctl config upstream/downstream (p. 77) を参照してください。

## 12.2 デバイスコネクター IPC

intdash Edge Agent 2 では、複数のデバイスコネクターを設定し、同時に使用することができます。センサーデバイスからデバイスコネクターを使って取得されたデータは、指定されたアップストリームを使ってサーバーに送信されます。

デバイスコネクターから intdash Edge Agent 2 にデータを渡すための設定は、デバイスコネクター IPC 設定で行います。

デバイスコネクター IPC 設定では、intdash Edge Agent 2 との通信に使用する FIFO のパス、データの行先のアップストリーム、データ名称などを設定します。デバイスコネクター IPC 設定は ID により管理されます。

アップストリーム用のデバイスコネクター IPC 設定を作成するには以下のコマンドを実行します。

```
$ intdash-agentctl config device-connector upstream  --create <config_in_yaml_format>
```

例:

```
$ intdash-agentctl config device-connector upstream --create '
    id: up
    data_name_prefix: v1/1/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/uplink-hello1.fifo
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /home/agent-volume-a/dc-hello-world-1.yaml
  '
```

作成済みのデバイスコネクター IPC 設定を変更するには、ID と、変更したい設定を与えます。

```
$ intdash-agentctl config device-connector upstream --modify <id> <config_in_yaml_format>
```

`config_in_yaml_format` で設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| id | string | デバイスコネクター IPC 設定の識別子 |
| enabled | bool | デバイスコネクター IPC 設定の有効 (`true`) ／無効 (`false`) |
| data_name_prefix | string | iSCP のデータ名称のプレフィックスとして使用する文字列。受信側で iSCP v1 を使用する場合は、決められたルールに沿って `data_name_prefix` を与える必要があります。詳細は データ ID (p. 93) を参照してください。 |
| dest_ids | [string] | データは、ここで指定した ID のアップストリームを使ってサーバーに送られます。複数の ID を指定した場合、データポイントは複製され、それぞれのアップストリームからサーバーに送られます。ただし、フィルターを設定すると、行先のアップストリームを変更することができます。フィルター（必要な場合のみ）(p. 42) を参照してください。 |
| format | string | デバイスコネクターとの通信に使用されるデータフォーマット<br>`iscp-v2-compat`： iSCPv2 と同等の FIFO データフォーマット<br>`logger-msg`： 旧 intdash Edge Agent の FIFO データフォーマット |
| ipc.type | string | intdash Edge Agent 2 とデバイスコネクターのデータ通信方法（現在 `fifo` のみに対応しているため、`fifo` を指定してください。） |
| ipc.path | string | intdash Edge Agent 2 がデバイスコネクターからデータを受信する FIFO のパス |
| launch.cmd | string | intdash Edge Agent 2 が起動したらそれに連動してデバイスコネクターも起動させたい場合は、デバイスコネクターの実行ファイルのパスを指定します。<br>この設定を使ってデバイスコネクターを起動した場合、intdash Edge Agent 2 を終了すると、デバイスコネクターも終了します。 |
| launch.args | [string] | デバイスコネクター起動時の引数。 |
| launch.environment | [string] | デバイスコネクター起動時に追加する環境変数（`VARIABLE=VALUE` の形式で記載します） |

**注釈:** 付属デバイスコネクター device-connector-intdash を使用する場合は、以下のようにします。

```
launch:
  cmd: device-connector-intdash
  args:
  - --config
  - <path-to-pipeline-configuration.yaml>
```

**注釈:** デバイスコネクター IPC 設定の一覧表示、削除などのコマンドについては、intdash-agentctl config device-connector (p. 79) を参照してください。

## 12.3 遅延アップロード用の行先（deferred）

「`persist: true`」かつ「`recover: true`」のアップストリームでは、リアルタイム送信ができなかった場合は遅延アップロードが行われます。しかし、リアルタイム送信を試みることなく最初から遅延アップロードしたい場合は、疑似的な行先ストリームとして `deferred` を指定します。

`deferred` は設定のための名称で、実際には deferred というストリームが intdash Edge Agent 2 とサーバーとの間に作成されるわけではありません。

> **注釈:** `deferred` というストリーム ID は予約されており、ユーザーが作成するストリームにこの名前を付けることはできません。また、`intdash-agentctl config device-connector upstream` で `deferred` の設定を変更したり削除したりすることはできません。
> 遅延アップロードの優先度や、遅延アップロード用のデータに使用するストレージに関しては、遅延アップロード (p. 61) を参照してください。

## 12.4 フィルター（必要な場合のみ）

どのアップストリームを使ってサーバーにデータを送信するか（行先のアップストリーム）はデバイスコネクター IPC 設定で指定しましたが、デバイスコネクター IPC とアップストリームの間にフィルターを挟むことにより、データの行先を他のストリームに変更することができます。

フィルターには条件を設定します。条件に一致するデータだけを別のアップストリームに送ることができます。これにより、一部のデータだけをリアルタイム送信し、その他のデータはリアルタイム送信せずに遅延アップロードするといった設定が可能です。

フィルターの設定方法については、フィルタリング／サンプリング (p. 52) を参照してください。

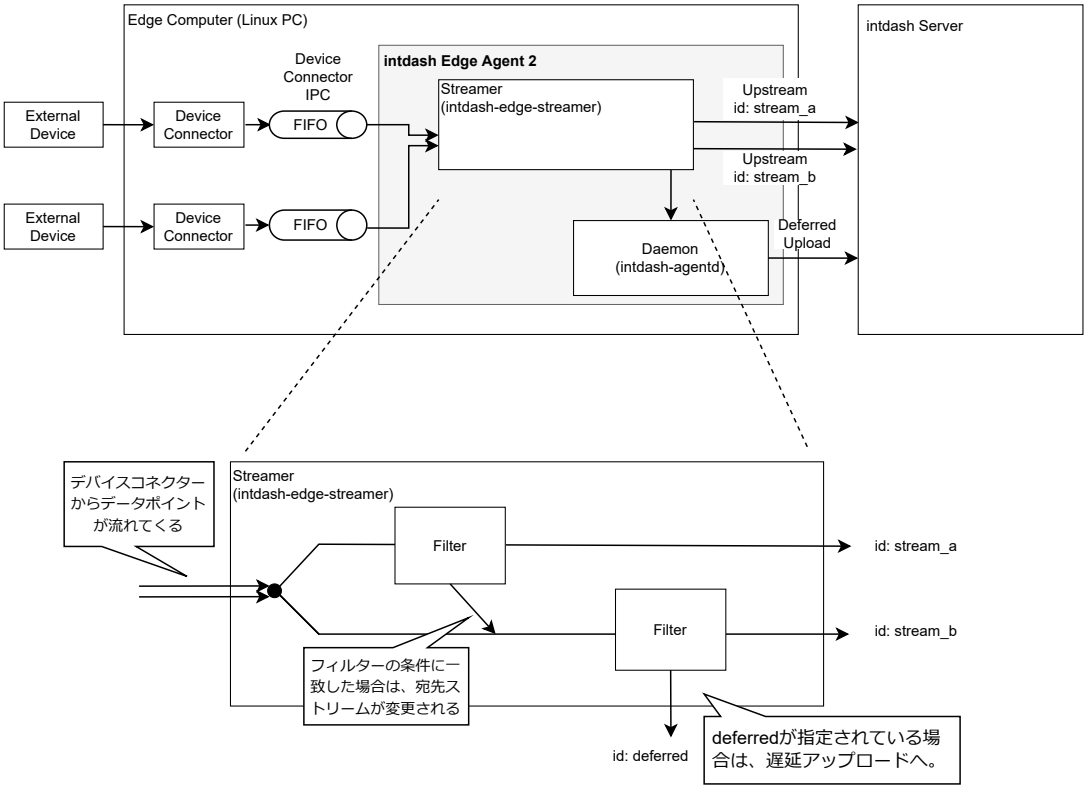



図 9　フィルター（アップストリームの場合）

**注釈:** デバイスコネクター IPC 設定やフィルター設定では、存在しないストリーム ID を指定することができます。
最終的に、存在しない ID を行先としているデータポイントは破棄されるため、存在しないストリーム ID を一時的な行先として使用すると便利な場合があります。以下の例では、デバイスコネクター IPC 設定の `dest_ids` において、存在しないアップストリーム `stream_x` を行先として指定しています。そのうえで、フィルタリングの段階で、フィルター条件に一致したデータポイントの行先を変更しています。フィルタリングの段階を完了してもなお `stream_x` を行先としているデータポイントは、破棄されます。

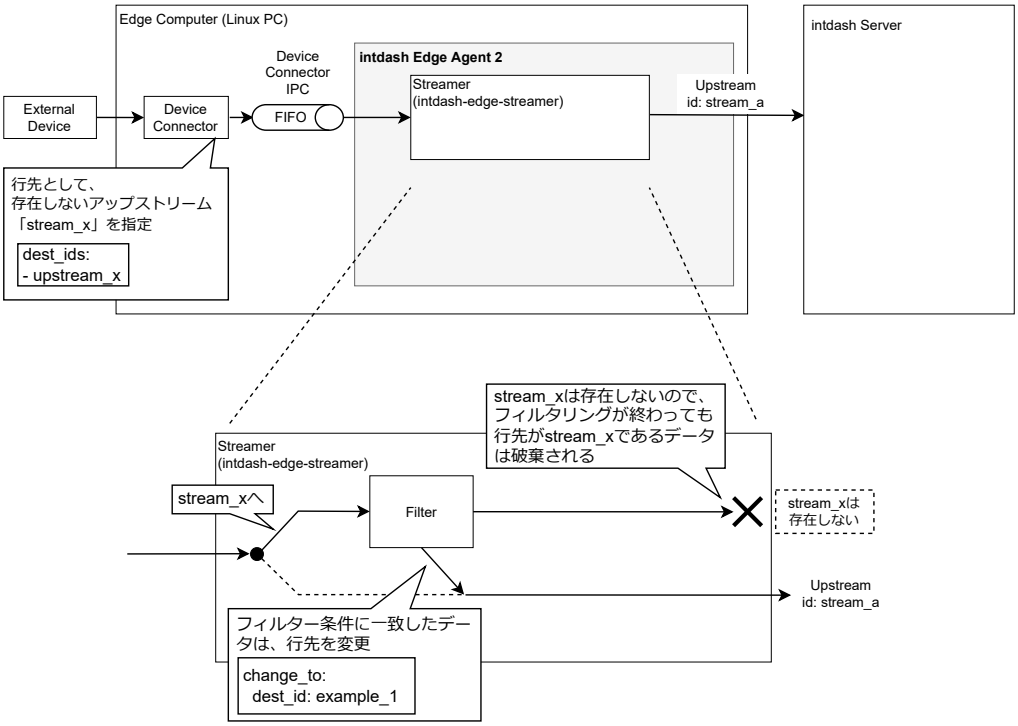



図 10　存在しないアップストリーム ID の使用

## 12.5 送信の開始／終了

アップストリームの設定とデバイスコネクター IPC の設定を行ったうえで、`intdash-agentctl run` を実行すると、intdash サーバーへの接続が行われます。

この状態で、デバイスコネクターから FIFO にデータポイントを書き込むことで、そのデータポイントは、intdash Edge Agent 2 から intdash サーバーに送信されます。

リアルタイムデータの送受信を終了する場合は、`intdash-agentctl run` を実行したターミナルで Ctrl+C を押してください。

「`recover: true`」のアップストリームにおいてデータの欠損が発生した場合や、行先ストリームとして `deferred` が指定されていた場合は、遅延アップロードが行われます。リアルタイムデータの送受信を終了しても、intdash-agentd が起動している間は、遅延アップロードが継続されます。

## 12.6 基準時刻について

intdash において、基準時刻とは、計測を開始した時刻を指します。intdash Edge Agent 2 が送信する基準時刻には以下の 2 つがあります。

- EdgeRTC による基準時刻
  - intdash Edge Agent 2 を実行しているシステムの時計による基準時刻です。計測開始時は必ず EdgeRTC で基準時刻が作成されます。その後、基準時刻は一定間隔でサーバーに送信されます。
- NTP による基準時刻
  - NTP サーバーから取得した時計による基準時刻です。NTP サーバーとの通信ができたら NTP 基準時刻が作成されます。その後、基準時刻は一定間隔でサーバーに送信されます。

# 13 ダウンストリームによる受信

アップストリームと反対に、intdash サーバーから時系列データをリアルタイム受信する経路をダウンストリームと呼びます。ダウンストリームによる受信を行うには以下の準備が必要です。(接続先サーバーと認証情報 (p. 36) は設定済みとします。)

- デバイスコネクター IPC の設定を作成する
- ダウントリームの設定を作成する
- (必要な場合のみ) フィルターを設定する (フィルターを使用すると、条件に一致したデータポイントの行先のデバイスコネクターを変更することができます)

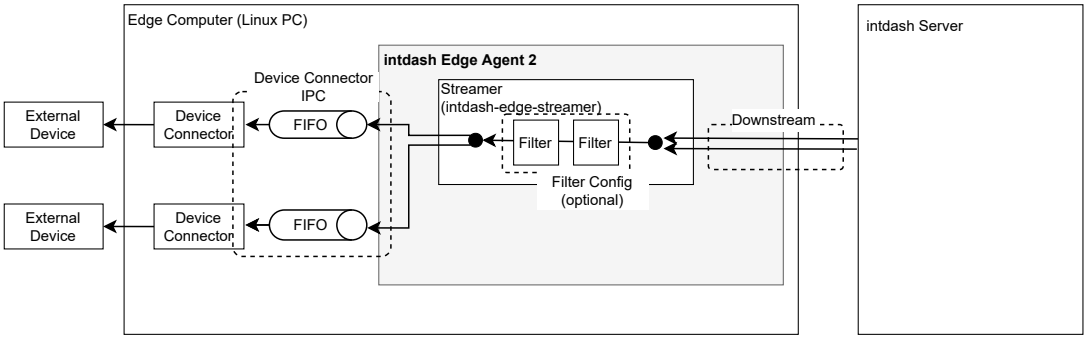



図 11　リアルタイムデータをサーバーから受信する

以上の設定を行ったうえで、`intdash-agentctl run` コマンドを実行することによりサーバーからのリアルタイム受信が開始されます。

> **注釈:** ダウンストリームの場合、データは、ストリームからデバイスコネクター IPC (さらにデバイスコネクターへ) という方向で流れますが、設定しやすさの観点からデバイスコネクター IPC を先に説明します。

## 13.1 デバイスコネクター IPC

intdash Edge Agent 2 では、複数のデバイスコネクターを設定し、同時に使用することができます。ダウンストリームにより intdash Edge Agent 2 が受信したデータは、デバイスコネクターを使って外部デバイス (アクチュエーターなど) に送られます。

デバイスコネクターにデータを渡すための設定は、デバイスコネクター IPC 設定で行います。デバイスコネクター IPC 設定では、intdash Edge Agent 2 との通信に使用する FIFO のパスなどを設定します。

デバイスコネクター IPC 設定の作成時には ID を付与します。後ほど、どのデバイスコネクターにデータを流すかを指定する際にこの ID を使用します。

ダウンストリーム用のデバイスコネクター IPC 設定を作成するには以下のコマンドを実行します。

```
$ intdash-agentctl config device-connector downstream --create <config_in_yaml_format>
```

例:

```
$ intdash-agentctl config device-connector downstream --create '
    id: cat
    data_name_prefix: v1/1/
    enabled: true
    format: iscp-v2-compat
```

(次のページに続く)

(前のページからの続き)

```
    ipc:
      type: fifo
      path: /var/run/intdash/downlink-hello.fifo
'
```

作成済みのデバイスコネクター IPC 設定を変更するには、ID と、変更したい設定を与えます。

```
$ intdash-agentctl config device-connector downstream --modify <id> <config_in_yaml_format>
```

`config_in_yaml_format` で設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| id | string | デバイスコネクター IPC 設定の識別子 |
| data_name_prefix | string | iSCP のデータ名称のプレフィックスを指定する文字列。ここで空文字以外のプレフィックスが指定されている場合、以下の処理が行われます。<br>• データ名称の先頭がここで指定されたプレフィックスと一致するデータポイントのみをデバイスコネクターに送信する。<br>• データポイントのデータ名称からプレフィックスを削除する。 |
| enabled | bool | デバイスコネクター IPC 設定の有効 (`true`) ／無効 (`false`) |
| format | string | デバイスコネクターが受信するデータのフォーマット<br>`iscp-v2-compat`： iSCPv2 と同等の FIFO データフォーマット<br>`logger-msg`： 旧 intdash Edge Agent の FIFO データフォーマット |
| ipc.type | string | intdash Edge Agent 2 とデバイスコネクターのデータ通信方法（現在 `fifo` のみに対応しているため、`fifo` を指定してください。） |
| ipc.path | string | デバイスコネクターが intdash Edge Agent 2 からデータを受信する FIFO のパス |
| launch.cmd | string | intdash Edge Agent 2 が起動したらそれに連動してデバイスコネクターも起動させたい場合は、デバイスコネクターの実行ファイルのパスを指定します。付属デバイスコネクターを使用する場合は、`device-connector-intdash` とします。 |
| launch.args | [string] | デバイスコネクター起動時の引数。付属デバイスコネクターを使用するために `launch.cmd` を `device-connector-intdash` とした場合は、第 1 引数を `--config` 、第 2 引数をパイプライン設定ファイルのパスとします。<br>`- --config`<br>`- <path-to-pipeline-configuration.yaml>` |
| launch.environment | [string] | デバイスコネクター起動時に追加する環境変数（`VARIABLE=VALUE` の形式で記載します） |

**重要:** 付属デバイスコネクターを使用する場合は、intdash Edge Agent 2 からデータを取得し、外部デバイスに渡すまでの流れを定義したパイプライン設定ファイルが必要です。
パイプライン設定ファイルについては、デバイスコネクター (p. 51) を参照してください。

**注釈:** デバイスコネクター IPC 設定の一覧表示、削除などのコマンドについては、device-connector コマンド (p. 79) を参照してください。

## 13.2 ダウンストリーム

intdash Edge Agent 2 では、複数のダウンストリームを設定し、同時に使用することができます。

ダウンストリームの設定では、どのエッジからのどのデータ ID を持つデータを受信するかを設定します。ストリームの設定は ID により管理されます。

ダウンストリームを作成するには以下のコマンドを実行します。

```
$ intdash-agentctl config downstream --create <config_in_yaml_format>
```

例:

```
intdash-agentctl config downstream --create '
    id: down
    enabled: true
    dest_ids:
      - cat
    qos: unreliable
    filters:
    - src_edge_uuid: f5e7a9d2-099b-432f-86e4-9b496af27d3b
      data_filters:
      - type: string
        name: v1/1/a
'
```

作成済みのダウンストリームの設定を変更するには、ID と、変更したい設定を与えます。

```
$ intdash-agentctl config downstream --modify <id> <config_in_yaml_format>
```

`config_in_yaml_format` で設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| id | string | ストリーム設定を識別するための文字列。`deferred` は予約されているため使用できません。 |
| enabled | bool | iSCP でのダウンストリームの有効 (`true`) ／無効 (`false`) |
| dest_ids | [string] | データは、ここで指定された ID のデバイスコネクター IPC を使って外部デバイスに送られます。複数の ID を指定した場合、データポイントは複製され、それぞれのデバイスコネクター IPC に送られます。ただし、フィルターを設定すると、行先のデバイスコネクター IPC を変更することができます。フィルター（必要な場合のみ）(p. 48) を参照してください。 |
| qos | string | `partial`: iSCP の QoS として PARTIAL を指定します。<br>`unreliable`: iSCP の QoS として UNRELIABLE を指定します。 |
| filters | [object] | ダウンストリーム対象とするデータを定義した設定のリスト |
| filters.[].src_edge_uuid | string | ダウンストリーム対象とする送信元のエッジ UUID。この UUID のエッジから送信されたデータを受信します。 |
| filters.[].data_filters | [object] | ダウンストリーム対象とするデータ ID を定義した設定のリスト。ここで指定したデータ ID（データ型とデータ名称）に一致したデータを受信します。 |
| filters.[].data_filters.[].type | [object] | ダウンストリーム対象とするデータ型 |
| filters.[].data_filters.[].name | [object] | ダウンストリーム対象とするデータ名称（`#`、`+` を利用したワイルドカード指定が可能。ワイルドカードについては iSCP の仕様を参照してください。） |

**注釈:** ダウンストリーム設定の一覧表示、削除などのコマンドについては、upstream、downstream サブコマンド (p. 77) を参照してください。

**注釈:** 上記の `filters.[].data_filters` は、どのデータをサーバーからダウンストリームするかを設定するもの（ダウンストリームフィルター）です。intdash Edge Agent 2 が受信したデータについて、その行先を変更する フィルター (p. 48) とは別の機能です。

## 13.3 フィルター（必要な場合のみ）

どのデバイスコネクター IPC にデータを送信するかはストリームの設定で指定しましたが、ストリームとデバイスコネクター IPC の間にフィルターを挟むことにより、データの行先を他のデバイスコネクターに変更することができます。

フィルターには条件を設定し、その条件に一致するデータだけを別のデバイスコネクター IPC に送ることができます。これにより、一部のデータだけをデバイスコネクター A に送信し、その他のデータはデバイスコネクター B に送信するといった設定が可能です。

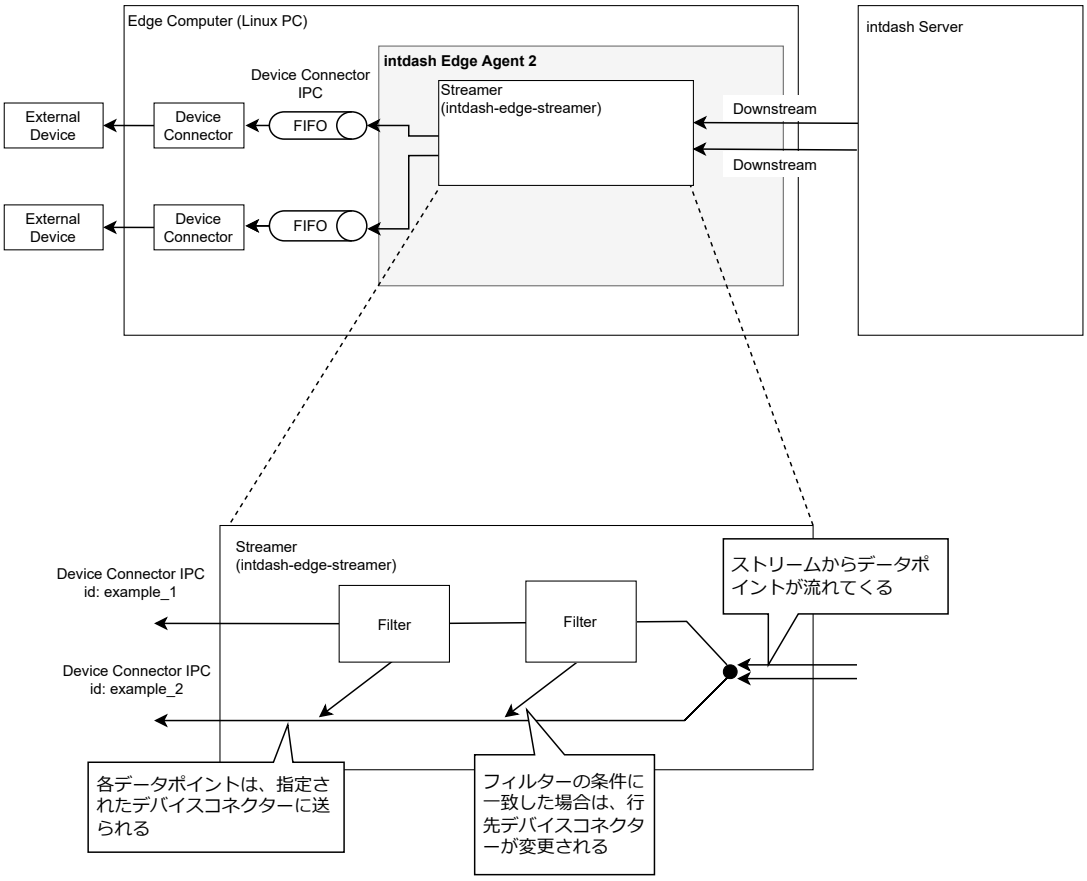



図 12　フィルター（ダウンストリームの場合）

フィルターの設定方法については、フィルタリング／サンプリング (p. 52) を参照してください。

**注釈:** ダウンストリームの設定やフィルターの設定では、存在しないデバイスコネクター IPC の ID を指定することができます。最終的に、存在しない ID を行先としているデータポイントは破棄されるため、存在しないデバイスコネクター IPC の ID を一時的な行先として使用すると便利な場合があります。
以下の例では、ダウンストリーム設定の `dest_ids` において、存在しないデバイスコネクター IPC `dc_x` を行先として指定しています。そのうえで、フィルタリングの段階で、フィルター条件に一致したデータポイントの行先を変更しています。フィルタリングの段階を完了してもなお `dc_x` を行先としているデータポイントは、破棄されます。

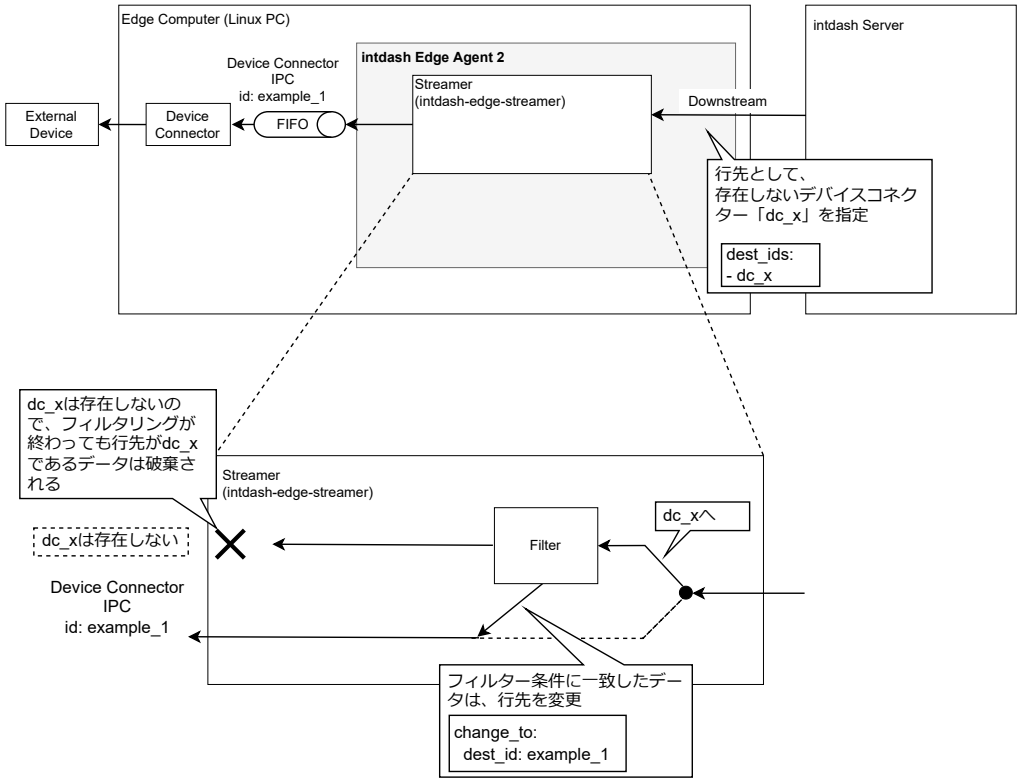



図 13　存在しないデバイスコネクター IPC の ID を使用

## 13.4 受信を開始／終了

ダウンストリームの設定とデバイスコネクター IPC の設定を行ったうえで、`intdash-agentctl run` を実行すると、intdash サーバーへの接続が行われます。

別のエッジからサーバーにデータがリアルタイム送信されているとき、そのデータがこのエッジにとってのダウンストリーム対象であればこのエッジで受信されます。

リアルタイムデータの送受信を終了する場合は、`intdash-agentctl run` を実行したターミナルで Ctrl+C を押してください。

# 14 デバイスコネクター

センサーやアクチューエーターなどの外部デバイスと intdash Edge Agent 2 の間でデータを仲介するソフトウェアをデバイスコネクターと呼びます。

デバイスコネクターと intdash Edge Agent 2 の間のデータの受け渡しには、Agent により作成される FIFO（名前付きパイプ）を使います。

- アップストリームの場合、デバイスコネクターが FIFO にデータを書き込むと、Agent はそれを読み取り、サーバーに送信します。
- ダウンストリームの場合、Agent がサーバーから受信したデータを FIFO に書き込むと、デバイスコネクターがそれを読み取ります。

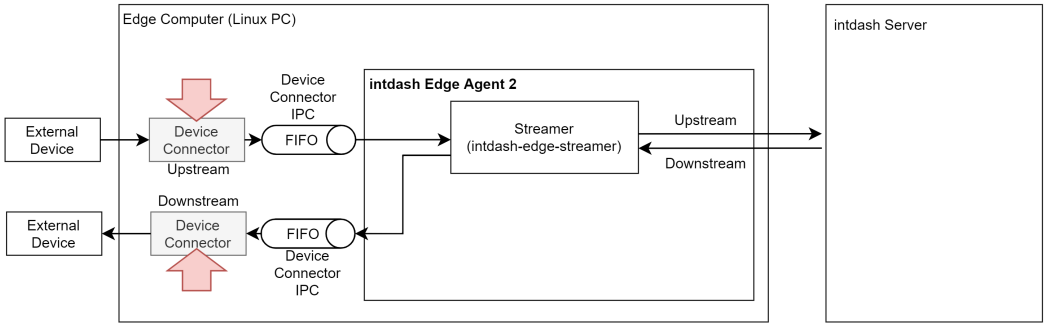



図 14　デバイスコネクター

所定のフォーマットのデータを FIFO に書き込む（または FIFO から読み込む）ことができれば、どのようなプログラムでもデバイスコネクターとして使用することができます。

FIFO で読み書きするデータは iscp-v2-compat フォーマット (p. 100) にしてください。

> **注釈:** FIFO で読み書きするデータのフォーマットとして、旧 intdash Edge Agent 用の logger-msg フォーマットを使用することも可能です（非推奨）。logger-msg フォーマットを使用するには、デバイスコネクター IPC に関する設定（アップストリーム用 (p. 91)、ダウンストリーム用 (p. 91)）の `format` を変更してください。

デバイスコネクターの起動や使用する FIFO のパスについては、デバイスコネクター IPC に関する設定（アップストリーム用 (p. 91)、ダウンストリーム用 (p. 91)）で設定できます。

## 14.1 付属デバイスコネクター

intdash Edge Agent 2 には、device-connector-intdash というデバイスコネクターが付属しています。

このデバイスコネクターは、intdash Edge Agent 2 をインストールすると依存パッケージとしてインストールされます。使用方法については、device-connector-intdash の概要 (p. 140) を参照してください。

# 15 フィルタリング／サンプリング

## 15.1 フィルターとは

intdash Edge Agent 2 では、intdash Edge Agent 2 内を流れるデータポイントに対して、フィルターを適用することができます。フィルターを使用すると、以下のことができます。

- アップストリーム方向の場合: デバイスコネクターから取得したデータポイントのうち、指定された条件に一致するデータポイントを選び出し、指定されたアップストリーム、または、遅延アップロードの処理に渡す。
- ダウンストリーム方向の場合: サーバーから取得したデータポイントのうち、指定された条件に一致するデータポイントを選び出し、指定されたデバイスコネクターに渡す。

さまざまな条件を指定するために、条件の指定方法が異なるいくつかのフィルタータイプが用意されています。例えば、name タイプのフィルターでは、条件としてデータ名称を指定します。この場合、流れてきたデータポイントのデータ名称が指定されたものと一致すると、フィルターの定義に従って処理されることになります。

使用できるフィルターのタイプは以下のとおりです。条件の設定方法については各フィルタータイプの説明を参照してください。:

- always (p. 55)
- type (p. 55)
- name (p. 56)
- src-id (p. 57)
- rename (p. 57)
- sampling (p. 58)
- h264-essential-nal-units (p. 60)

**注釈:** フィルターは、上記のタイプのみを使用できます。ユーザーが独自のフィルタータイプを開発することはできません。既存のフィルタータイプでは不足がある場合は、同様の仕組みをデバイスコネクターにおいて開発することで対応できます。

複数のフィルターを設定して多段階の条件を設定することもできます。複数のフィルターを設定した場合、設定ファイルに書かれた順にフィルターが適用されます。

また、一定時間ごとに最初の 1 つのデータポイントを選択するフィルター（sampling タイプのフィルター）を設定することで、サンプリングを実行することができます。

**注釈:** ここで説明するフィルターは、intdash Edge Agent 2 内を流れるデータに対して適用されるものです。サーバーからダウンストリームする対象を指定するための「ダウンストリームフィルター」とは別の機能です。ダウンストリームフィルターについては、ダウンストリーム (p. 47) を参照してください。

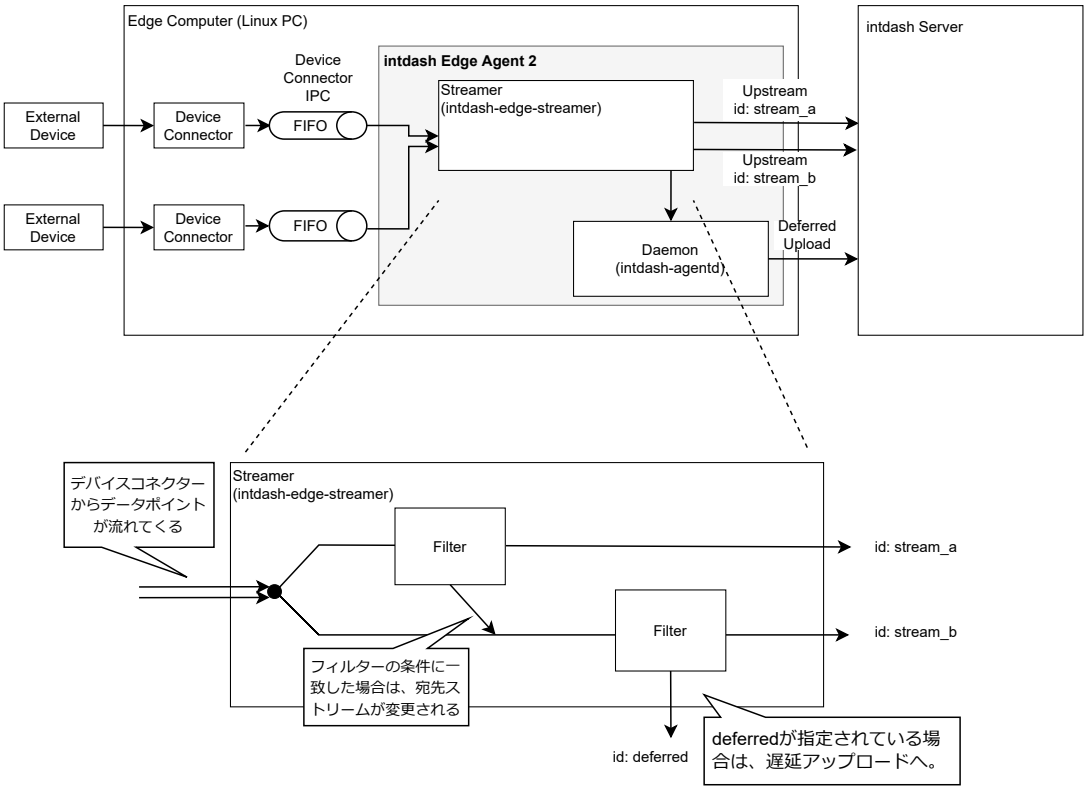



図 15　フィルター（アップストリームの場合）

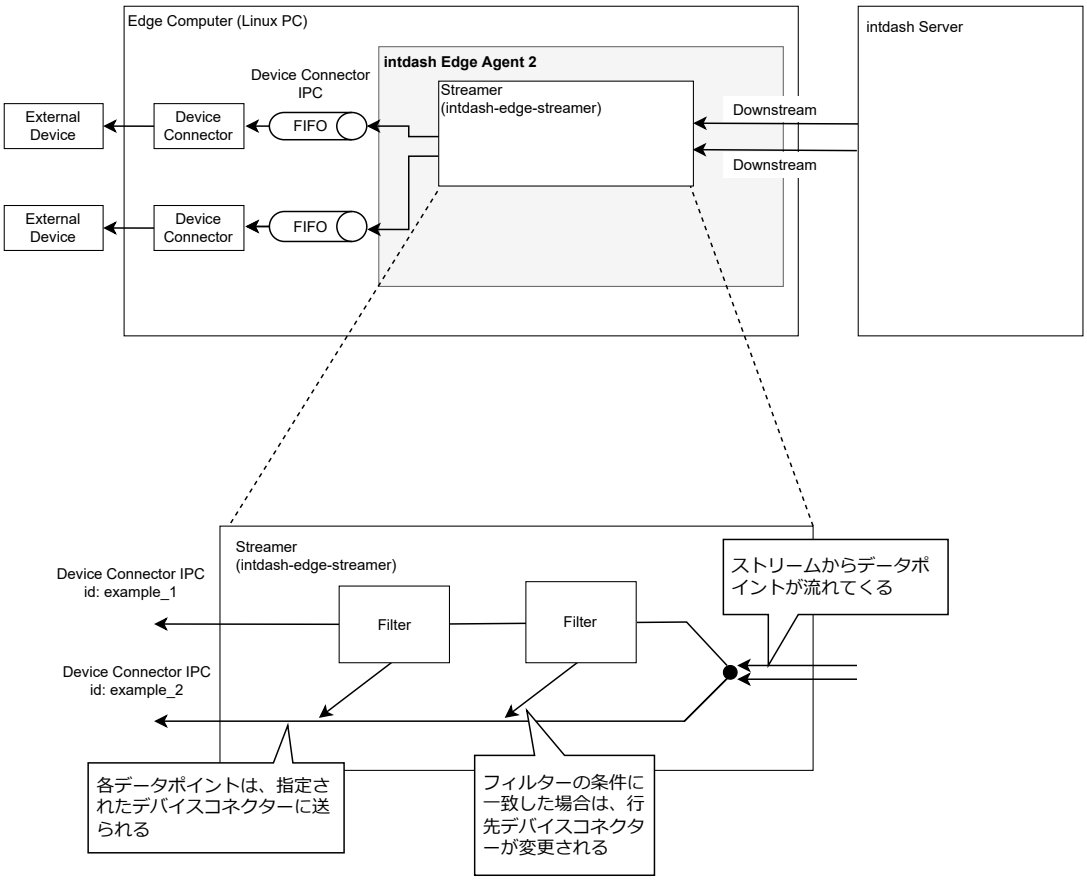



図 16　フィルター（ダウンストリームの場合）

## 15.2 フィルターを設定する

フィルターを作成するには以下のコマンドを実行します。

アップストリーム方向:

```
$ intdash-agentctl config filter upstream --create <config_in_yaml_format>
```

ダウンストリーム方向:

```
$ intdash-agentctl config filter downstream --create <config_in_yaml_format>
```

例：

```
$ intdash-agentctl config filter upstream --create '
    id: string_to_deferred
    enabled: true
    type: type
    target:
      dest_ids:
      - recoverable
      type: string
    change_to:
      dest_id: deferred
  '
```

作成済みのフィルターの設定を変更するには、ID と、変更したい設定を与えます。

```
$ intdash-agentctl config filter upstream --modify <id> <config_in_yaml_format>
$ intdash-agentctl config filter downstream --modify <id> <config_in_yaml_format>
```

| キー | 型 | 説明 |
|---|---|---|
| id | string | フィルター設定を識別する任意の ID |
| enabled | bool | フィルターの有効 (true) ／無効 (false) |
| type | string | フィルターのタイプ (p. 55) |
| target | object | フィルターのタイプに応じた対象データポイントの条件指定。条件の指定方法については、フィルターの種類別の説明を参照してください。データポイントがここで記載された条件に一致した場合、データポイントに change_to で指定された変更が行われます。 |
| change_to | object | フィルターの条件に合致したデータポイントは、ここに指定された変更が行われます。 |

## 15.3 フィルターのタイプ

フィルターのタイプと、それぞれで指定できる条件は以下のとおりです。

### 15.3.1 always

always タイプのフィルターでは、データポイントの行先を条件にしてフィルタリングが行われます。データポイントの行先が target.dest_ids で列挙された行先のいずれかである場合、その行先は change_to で指定されたものに変更されます。

条件指定（ target ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |

change_to では、以下のように新しい行先を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

例：

```yaml
id: example
enabled: true
type: always
target:
  dest_ids:
    - recoverable
change_to:
  dest_id: deferred
```

この例では、行先が recoverable であるデータポイントすべてについて、行先が deferred に変更されます。

### 15.3.2 type

type タイプのフィルターでは、データポイントの行先とデータ型を条件にしてフィルタリングが行われます。データポイントの行先が target.dest_ids で列挙された行先のいずれかであり、データ型が target.type で指定されたものである場合、その行先は change_to で指定されたものに変更されます。

条件指定（ target ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |
| type | string | iSCP v2 のデータ型名によるフィルター条件（正規表現） |

change_to では、以下のように新しい行先を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

例：

```
id: example
enabled: true
type: type
target:
  dest_ids:
    - recoverable
  type: h26.+
change_to:
  dest_id: deferred
```

この例では、行先が recoverable であるデータポイントのうち、データ型が h26.+ の正規表現にマッチするものについて、行先が deferred に変更されます。

### 15.3.3 name

name タイプのフィルターでは、データポイントの行先とデータ名称を条件にしてフィルタリングが行われます。データポイントの行先が target.dest_ids で列挙された行先のいずれかであり、データ名称が target.name で指定されたものである場合、その行先は change_to で指定されたものに変更されます。

条件指定（ target ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |
| name | string | データ名称によるフィルター条件（正規表現） |

change_to で行先を変更できます。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

例：

```
id: example
enabled: true
type: name
target:
  dest_ids:
    - recoverable
  name: v1/.+
change_to:
  dest_id: deferred
```

この例では、行先が recoverable であるデータポイントのうち、データ名称が v1/.+ の正規表現にマッチするものについて、行先が deferred に変更されます。

### 15.3.4 src-id

src-id タイプのフィルターでは、データポイントの行先と生成元（アップストリームではデバイスコネクター IPC、ダウンストリームではストリーム）の ID を条件にしてフィルタリングが行われます。データポイントの行先が target.dest_ids で列挙された行先のいずれかであり、生成元が target.src_id で指定されたものである場合、その行先は change_to で指定されたものに変更されます。

条件指定（ target ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |
| src_id | string | アップストリームの場合、生成元デバイスコネクター IPC の ID（正規表現）。<br>ダウンストリームの場合、生成元ストリームの ID（正規表現）。 |

change_to では、以下のように新しい行先を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

例：

```yaml
id: example
enabled: true
type: src-id
target:
  dest_ids:
    - recoverable
  src_id: dc?
change_to:
  dest_id: deferred
```

この例では、行先が recoverable であるデータポイントのうち、生成元の ID が dc? の正規表現にマッチするものについて、行先が deferred に変更されます。

### 15.3.5 rename

データポイントの行先が target.dest_ids で列挙された行先のいずれかであり、データ型、データ名称、生成元の ID（アップストリームではデバイスコネクター IPC の ID、ダウンストリームではストリームの ID）が条件に一致した場合、そのデータポイントのデータ名称がリネームされます。

正規表現を空文字にした項目は、条件に一致するかどうかの評価対象にはなりません。例えば、target.type と target.name を指定し、target.src_id（生成元 ID）を空文字にした場合、target.type と target.name がマッチすれば、生成元 ID に関係なくリネームが行われます。

条件指定（ target ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | （オプショナル）アップストリームの場合、行先ストリームの ID のリスト。ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |
| type | string | iSCP v2 のデータ型名によるフィルター条件（正規表現） |
| name | string | データ名称によるフィルター条件（正規表現） |
| src_id | string | 生成元 ID によるフィルター条件（正規表現） |

`change_to` では、以下のように新しい行先とデータ名称を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |
| name | string | 新しいデータ名称 |

例：

```yaml
id: example
enabled: true
type: rename
target:
  dest_ids:
    - recoverable
    - deferred
  name: "v1/2/.+"
  type: ""
  src_id: ""
change_to:
  name: "v1/10/aaa"
```

この例では、行先が `recoverable` または `deferred` であるデータポイントのうち、データ名称が `v1/2/.+` の正規表現にマッチするものについて、データ名称が `v1/10/aaa` に変更されます。行先は変更されません。

### 15.3.6 sampling

sampling タイプのフィルターでは、データポイントのサンプリングが行われます。

sampling タイプのフィルターでは、行先、データ型、データ名称、生成元（アップストリームではデバイスコネクター IPC、ダウンストリームではストリーム）の ID を条件として設定します。

指定された行先（ `target.dest_ids` のうちいずれか）を持つデータポイントのうち、指定された正規表現すべてにマッチしたデータポイントがサンプリングの対象となります。サンプリング対象のうち、サンプリング間隔 `interval_ms` 内で最初の 1 つのデータポイントのみ、行先が `change_to.dest_id` の値に変更されます。サンプリング間隔内で 2 つ目以降のデータポイントの行先は変更されません。

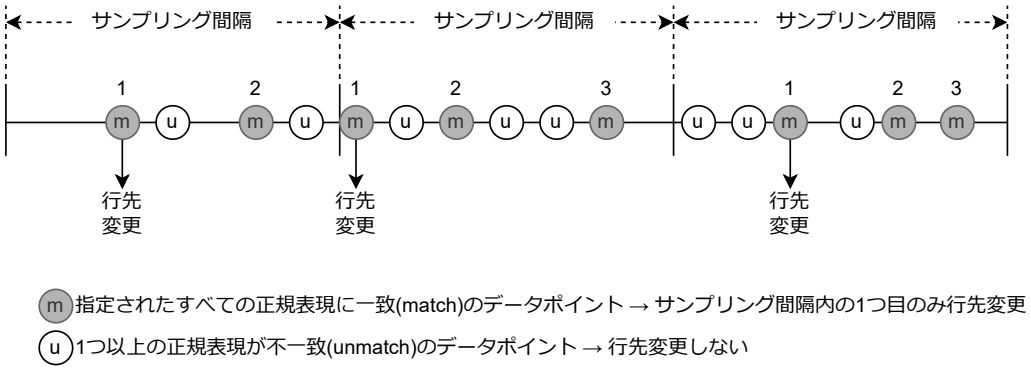



図 17　サンプリング

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |
| interval_ms | integer | サンプリング間隔（ミリ秒） |
| type | string | iSCP v2 のデータ型名によるフィルター条件（正規表現） |
| name | string | データ名称によるフィルター条件（正規表現） |
| src_id | string | 生成元 ID によるフィルター条件（正規表現） |

`change_to` では、以下のように新しい行先を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

例：

```
id: example
enabled: true
type: sampling
target:
  dest_ids:
    - recoverable
  name: "v1/1/abc"
  type: ""
  src_id: ""
  interval_ms: 300
change_to:
  dest_id: deferred
```

この例では、行先が `recoverable` であり、かつデータ名称が正規表現 `v1/1/abc` にマッチするデータポイントのうち、300 ミリ秒に 1 つだけ、行先が `deferred` に変更されます。それ以外のデータポイントの行先は変更されません。

### 15.3.7 h264-essential-nal-units

h264-essential-nal-units タイプのフィルターは、h264_nal_unit 型専用です。

h264-essential-nal-units タイプのフィルターでは、データポイントの行先と、ペイロードの内容によってフィルタリングが行われます。データポイントの行先が `target.dest_ids` で列挙された行先のいずれかであり、ペイロードが H.264 動画の再生において重要な NAL ユニット（SPS、PPS または IDR ピクチャー）である場合、行先が変更されます。

条件指定（ `target` ）では以下を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_ids | [string] | アップストリームの場合、行先ストリームの ID のリスト。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID のリスト。 |

`change_to` では、以下のように新しい行先を指定します。

| キー | 型 | 説明 |
|---|---|---|
| dest_id | string | アップストリームの場合、行先ストリームの ID。<br>ダウンストリームの場合、行先デバイスコネクター IPC の ID。 |

**注釈:** H.264 動画を送信する際に、適切に設定されたストリームとこのフィルターを組み合わせることで、Visual M2M Data Visualizer においてより少ない遅延で H.264 動画を再生できます。H.264 動画を NALU ごとに送信する (p. 118) の使用例を参照してください。

例：

```yaml
id: example
enabled: true
type: h264-essential-nal-units
target:
  dest_ids:
    - h264_nal_unit_extra_units
change_to:
  dest_id: h264_nal_unit_key_units
```

この例では、行先が `h264_nal_unit_extra_units` であるデータポイントのうち、再生表示に必須の NAL ユニットの行先が `h264_nal_unit_key_units` に変更されます。

# 16 遅延アップロード

`recover: true` の設定になっているアップストリームにおいてデータが送信できなかった場合、また、アップストリームとして deferred (p. 42) が指定された場合は、そのデータは intdash Edge Agent 2 内に保存され、順次遅延アップロードされます。

遅延アップロードは intdash-agentd により行われます。そのため、ネットワークに接続され、intdash-agentd が起動していれば常に遅延アップロードが行われます。

遅延アップロードによりサーバーに保存されたデータは、intdash Edge Agent 2 からは自動的に削除されます。

> **注釈:** 遅延アップロード用のデータは、データベースファイル `/var/lib/intdash/agent.db` に保存されます。ただし環境変数 `AGENT_LIB_DIR` が設定されている場合は `${AGENT_LIB_DIR}/agent.db` が使用されます。

遅延アップロードの優先度や、遅延アップロード用のデータに使用するストレージの容量は変更することができます。

## 16.1 優先度を設定する

以下のようなコマンドで設定します。

```
$ intdash-agentctl config deferred -m 'priority: same_as_realtime'
```

設定できるパラメーターは以下のとおりです。

| キー | 型 | 説明 |
|---|---|---|
| priority | string | 遅延アップロードのネットワーク通信優先度<br>• `higher_than_realtime` -リアルタイム送受信よりも高い優先度で遅延アップロードを行います。遅延アップロードが帯域を占有した場合はリアルタイム送信は行われません。<br>• `same_as_realtime` -リアルタイム送受信と同じ優先度で遅延アップロードを行います。遅延アップロードが帯域を占有することはありませんが、遅延アップロードによりリアルタイム送受信が影響を受ける場合があります。<br>• `lower_than_realtime` -リアルタイム送受信よりも低い優先度で遅延アップロードを行います。リアルタイム送受信が帯域を占有した場合は遅延アップロードは行われません。 |

## 16.2 データ蓄積の上限を設定する

遅延アップロード用のデータの蓄積に上限を設定することができます。

以下のようなコマンドで設定します。

```
$ intdash-agentctl config deferred -m '
    limit_data_storage: true
    data_storage_capacity: 102400
  '
```

| キー | 型 | 説明 |
|---|---|---|
| limit_data_storage | bool | 古い計測の自動削除の有効（ `true` ）／無効（ `false` ） |
| data_storage_capacity | integer | 遅延アップロード用のデータの保存のために使用するディスク容量（ `MiB` ）。データの量がこの容量を超える場合は、古いデータから順に削除されます。 |

**警告:**

- `limit_data_storage` を `true` にすると、遅延アップロード用のデータが `data_storage_capacity` を超えた場合に、古いデータから順に削除されます。削除されたデータを復元することはできません。
- `limit_data_storage` を `false` にすると、遅延アップロード用のデータが増加し、ディスクの使用量が 90% 以上になった場合、ストリーマーが自動的に終了し、計測が終了します。

# 17 ローカルストレージの管理

intdash Edge Agent 2 で計測を行うと、ローカルストレージ内のデータベースに、計測に関する情報と未送信データが保存されます。

**注釈:** ストリーマーを起動し終了するまでの一連の時系列データが 1 つの計測です。

`intdash-agentctl measurement` コマンドにより、この intdash Edge Agent 2 で行った計測についての情報を確認することができます。

```
$ intdash-agentctl measurement
  UUID                                    BASE_TIME                       PENDING_DATA_SIZE
  2f968138-c8d1-4277-998c-6c1d0cb15a11    2022-09-12T04:43:39.662313221Z  123
  4529ef77-911b-4256-af8f-6e592ead6d86    2022-09-12T04:53:59.074454299Z  0
  3f6e2c18-8275-486a-abbd-53ddadc3af6e    2022-09-12T05:21:21.819922212Z  10
```

計測は古い順に表示されます。一番下が最新の計測です。表示される情報は以下のとおりです。

| キー | 説明 |
|---|---|
| UUID | 計測の UUID。intdash サーバーで管理している計測 UUID と同じです。 |
| BASE_TIME | 計測の基準時刻（取得できた基準時刻のうち最も優先度の高い基準時刻）。 |
| PENDING_DATA_SIZE | 遅延アップロード用の送信待ちデータのサイズ（Byte）。0 の場合、遅延アップロード用のデータはありません。 |

## 17.1 計測を削除する

intdash Edge Agent 2 内に保存されている、計測に関する情報を削除するには、`--delete` オプションの引数として計測の UUID を 指定します。

**重要:**
- 計測を削除すると、その計測に関するすべてのデータが intdash Edge Agent 2 から削除されます。その計測にサーバーに未送信のデータがあった場合、そのデータも削除されます。
- intdash Edge Agent 2 内で計測を削除しても、サーバー側のデータは影響を受けません。

```
$ intdash-agentctl measurement --delete 4529ef77-911b-4256-af8f-6e592ead6d86
```

**注釈:** 実行中の計測を削除することはできません。

# 18 ステータスの確認

アップストリーム、ダウンストリーム、デバイスコネクター IPC、遅延アップロードの現在の状態を確認するには、`intdash-agentctl status` コマンドを実行します。

1 つのストリーム、 1 つのデバイスコネクター IPC についての情報が 1 行で表示されます。また、遅延アップロードについての情報（TYPE が `deferred` となっている行）も表示されます。

```
$ intdash-agentctl status
  ID                  TYPE                DIRECTION   ENABLED STATUS.CODE     STATUS.ERROR
  up                  stream              upstream    true    connected
  down                stream              downstream  true    connected
  up-hello            device-connector    upstream    true    disconnected    uninitialized
  deferred            deferred            upload      true    connected
```

| 列 | 説明 |
|---|---|
| ID | ストリームまたはデバイスコネクター IPC の ID。遅延アップロードの場合は `deferred` と表示されます。 |
| TYPE | • `stream`: ストリーム<br>• `device-connector`: デバイスコネクター IPC<br>• `deferred`: 遅延アップロード |
| DIRECTION | 送信方向。ストリームまたはデバイスコネクター IPC については方向が表示されます（`upstream` ／ `downstream`）。遅延アップロードの場合は `upload` と表示されます。 |
| ENABLED | 設定の有効 (`true`) ／無効 (`false`) |
| STATUS.CODE | 接続の状態が表示されます。<br>• `connected`: 接続<br>• `disconnected`: 接続していない<br>• `quiet`: 接続しているがデータが流れていない<br>• <空白>: 状態の情報が未取得 |
| STATUS.ERROR | エラー発生時にエラーの概要が表示されます。 |

# 19 エンドツーエンド通信

intdash Edge Agent 2 は、時系列データの送受信で使用するストリームとは別に、エッジからエッジへのエンドツーエンド通信をデバイスコネクターに提供します。

エンドツーエンド通信のためのプログラム「E2E コーラー」は gRPC によるインターフェイスを提供しているため (Agent E2E Call API (p. 70))、エンドツーエンドコールの受信・送信は、デバイスコネクターから gRPC によるリモートプロシージャコールを行うことで実行します。

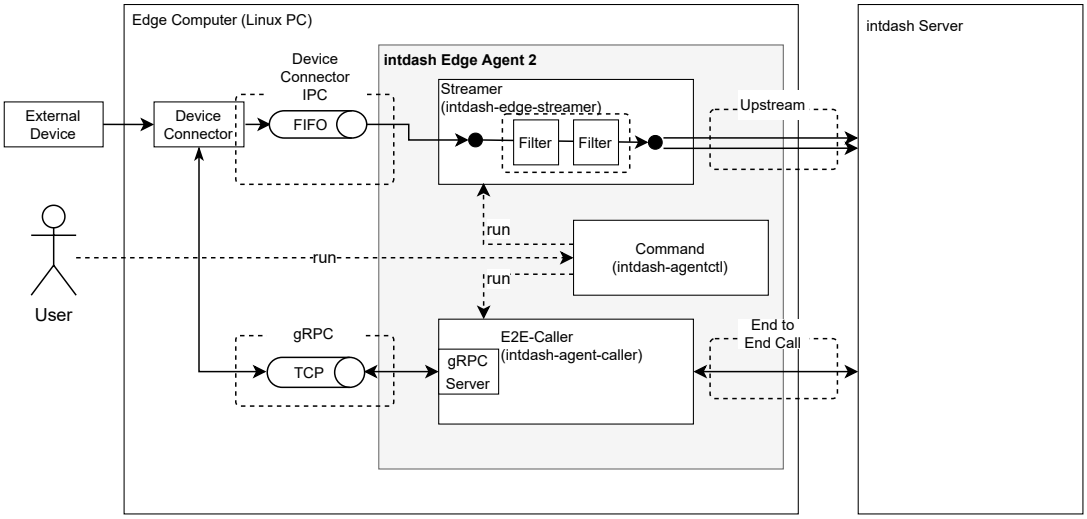



図 18　Streamer による時系列データ送信と、E2E Caller によるエンドツーエンド通信

E2E コーラーを起動するには、接続先サーバーと認証情報 (p. 36) の設定を行ったうえで、intdash-agentctl run (p. 74) コマンドを実行します。intdash-agentctl run (p. 74) の実行中は E2E コーラーの gRPC サーバーがポート 50052 で起動します。

この状態で、デバイスコネクターなどの別プロセスから E2E コーラー対して Agent E2E Call API (p. 70) の任意のメソッドを呼び出すことで、エンドツーエンド通信を実行することができます。

> **注釈:** gRPC サーバーに割り当てるポートは intdash-agentctl run (p. 74) の `--e2e-caller-port` オプションで変更できます。

gRPC のクライアント作成方法については gRPC の Documentation を参照してください。

Agent E2E Call API の説明、および、.proto ファイルについては Agent E2E Call API (p. 70) を参照してください。

# 20 ログの確認

intdash Edge Agent 2 はログを出力します。intdash-agentd のログは、`/var/log/intdash-agentd.log` に出力されます。コマンドのログ、およびコマンドにより起動されるストリーマー、デバイスコネクターのログは、intdash-agentctl の標準出力に出力されます。

環境変数 AGENT_LOG にログレベルを設定すると、そのレベル以上のログが出力されます。設定できるログレベルは、trace、debug、information、warning、error、quiet です。quiet を選択すると、ログは出力されません。

**注釈:**

- ストリーマー、デーモン、E2E コーラー、コマンドとも、環境変数 AGENT_LOG を参照します。
- 環境変数はプロセス起動時に読み込まれるので、AGENT_LOG の設定を intdash-agentd に反映するには、デーモンを再起動する必要があります。

# 21 設定の書き出し／読み込み

intdash Edge Agent 2 の設定は、コマンドを使って書き出すことができます。設定をファイルに書き出すことで、設定を俯瞰したり、履歴管理したり、直接修正したりすることが可能になります。

また、書き出された設定を読み込んで適用することも可能です。

> **注釈:** intdash Edge Agent 2 の設定は、設定ファイル `/var/lib/intdash/agent.yaml`（ただし環境変数 `AGENT_LIB_DIR` が設定されている場合は `${AGENT_LIB_DIR}/agent.yaml`）として保存されていますが、このファイルを直接編集することは避けてください。間違って正しくない形式にしてしまうと、最悪の場合 intdash Edge Agent 2 が正常に動作しなくなる場合があります。
> 設定変更は、`intdash-agentctl config` コマンドを使って行うか、以下で説明する `intdash-agentctl config-file apply` コマンドを使ってファイルを読み込むことで行ってください。これらのコマンドでは設定内容のバリデーションが行われるため、安全に設定を変更することができます。

## 21.1 設定を出力する

以下のコマンドを実行すると、現在の設定を標準出力に書き出すことができます。

```
$ intdash-agentctl config-file show
```

ファイルに保存したい場合は、出力をリダイレクトしてください。

設定情報の見方については、intdash Edge Agent 2 設定一覧 (p. 86) を参照してください。

## 21.2 ファイルから設定を読み込む

上記の手順 (p. 67) で出力した設定は、以下の手順で読み込むことができます。

1. `intdash-agentctl run` を実行している場合（計測を実行中の場合）は Ctrl+C を押して停止します。
2. 以下のコマンドを実行します。

```
$ intdash-agentctl config-file apply -c <path_to_conf_file>
```

# 22 intdash Edge Agent 2 REST API

intdash Edge Agent 2 は REST API を持っており、API へのリクエストにより設定の確認や変更を行うことができます。

REST API の仕様については以下を参照してください。

- Agent Command API

REST API を使ってアクセスが可能な設定は以下のとおりです。

## 22.1 接続先サーバーと認証情報

接続先サーバーと認証情報に関する設定には `/connection` エンドポイントを使用します。

API リファレンス：Connection to Server

## 22.2 トランスポート

トランスポートの設定には `/transport` エンドポイントを使用します。

API リファレンス：Connection to Server

## 22.3 ストリーム

ストリームに関する設定には `/upstreams` と `/downstreams` エンドポイントを使用します。

API リファレンス：Stream

## 22.4 フィルター

ストリーマーを流れるデータに対するフィルターの設定には `/filters_upstream` と `/filters_downstream` エンドポイントを使用します。

API リファレンス：Filter

## 22.5 デバイスコネクター IPC

デバイスコネクター IPC に関する設定には `/device_connectors_upstream` と `/device_connectors_downstream` エンドポイントを使用します。

API リファレンス：Device Connector IPC

## 22.6 遅延アップロード

未送信データの遅延アップロードに関する設定には `/deferred_upload` エンドポイントを使用します。

API リファレンス：Deferred Upload

## 22.7 ローカル計測データ

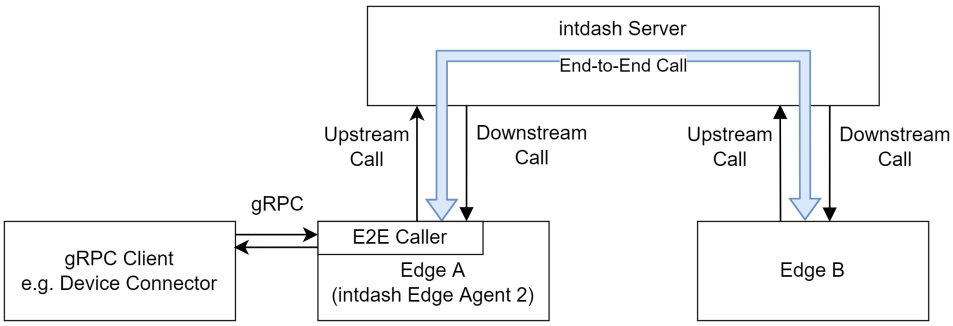
ローカルストレージ内に保存されいてる計測データの管理には `/measurements` エンドポイントを使用します。

API リファレンス：Measurement

# 23 Agent E2E Call API

intdash の E2E コールは、intdash サーバーに接続されているエッジとエッジがサーバーを介してメッセージをやり取りする仕組みです。エッジからサーバーへのメッセージ送信では iSCP 2.0 の Upstream Call が、サーバーからエッジへの送信では Downstream Call が使用されます。

intdash Edge Agent 2 において、この E2E コールの送受信は E2E コーラーが担当します。E2E コーラーは gRPC インターフェイス（Agent E2E Call API）を提供しており、クライアント（デバイスコネクターなど）からこれを呼び出すことで E2E コールの送受信を行うことができます。




Agent E2E Call API を使用するには、gRPC で通信するために必要なサービスとデータ構造を定義した .proto ファイル をダウンロードして、クライアントを作成してください。

proto ファイルに記載のとおり、Agent E2E Call API には以下のメソッドが用意されています。



## 23.1 SendCall

`SendCall` を実行すると、intdash Edge Agent 2 は、別のエッジに向けて E2E コールを送信します。

iSCP 2.0 のレベルでは、サーバーに向けて Upstream Call が送信されることになります。Upstream Call が持つフィールドの詳細については、iSCP 2.0 のプロトコル仕様書を参照してください。

Upstream Call のフィールドのうち、リクエスト用の構造体 `SendCallRequest` に存在しないものは、intdash Edge Agent 2 によって自動的に付与されます。

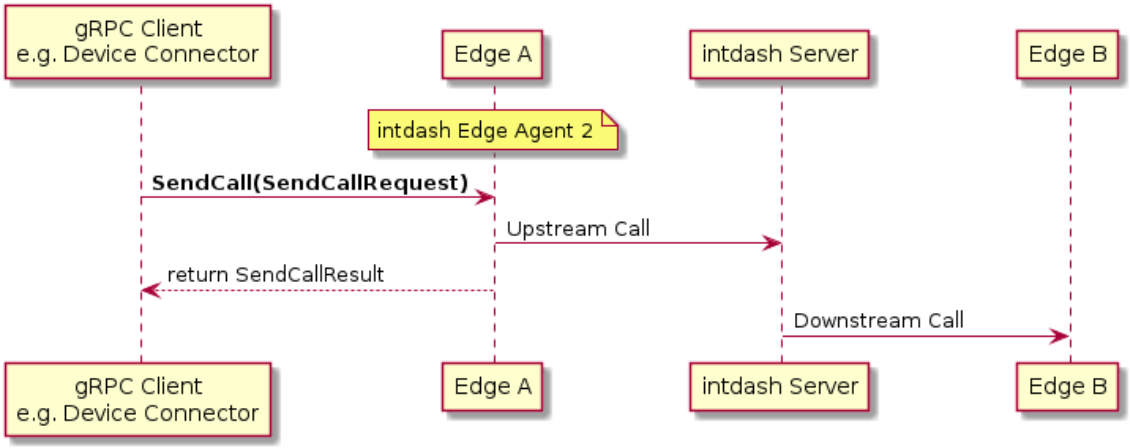



図 19　SendCall

## 23.2 SendReplyCall

`SendReplyCall` を実行すると、intdash Edge Agent 2 は、別のエッジからの呼びかけの E2E コールに対して応答の E2E コールを送信します。

iSCP 2.0 のレベルでは、サーバーに向けて Upstream Call が送信されることになります。Upstream Call が持つフィールドの詳細については、iSCP 2.0 のプロトコル仕様書を参照してください。

Upstream Call のフィールドのうち、リクエスト用の構造体 `SendReplyCallRequest` に存在しないものは、intdash Edge Agent 2 によって自動的に付与されます。また、`SendReplyCallRequest` のフィールド `request_call_id` には、元の呼びかけのコール ID を指定してください。

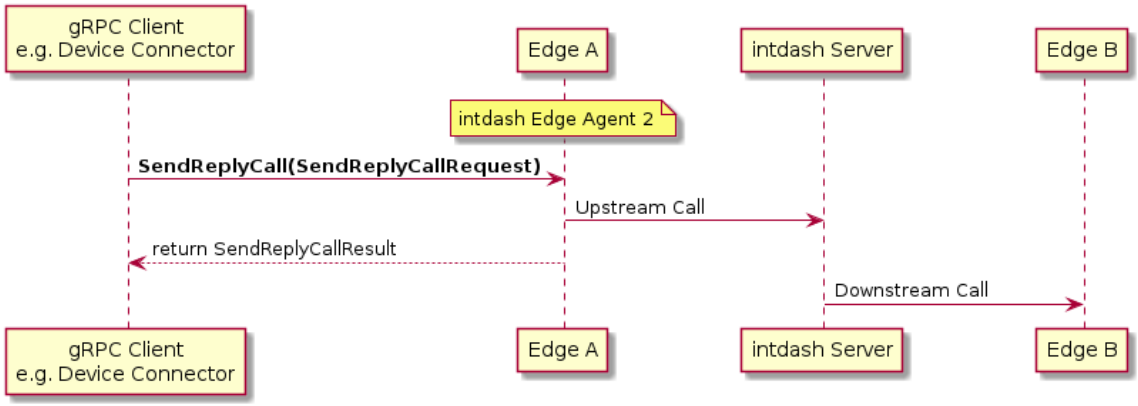



図 20　SendReplyCall

## 23.3 SendCallAndWaitReplyCall

`SendCallAndWaitReplyCall` を実行すると、intdash Edge Agent 2 は、別のエッジに向けて E2E コールを送信し、送信先のエッジから応答が返されるまで処理をブロックします。

iSCP 2.0 のレベルでは、サーバーに向けて Upstream Call が送信されます。また、送信先エッジからの応答を、サーバーから Downstream Call として受信できるまで待機することになります。

Upstream Call および Downstream Call が持つフィールドの詳細については、iSCP 2.0 のプロトコル仕様書を参照してください。

Upstream Call のフィールドのうち、リクエスト用の構造体 `SendCallAndWaitReplyCallRequest` に存在しないフィールドは、intdash Edge Agent 2 によって自動的に付与されます。

受信した Downstream Call は、構造体 `SendCallAndWaitReplyCallResult` の形式で返されます。

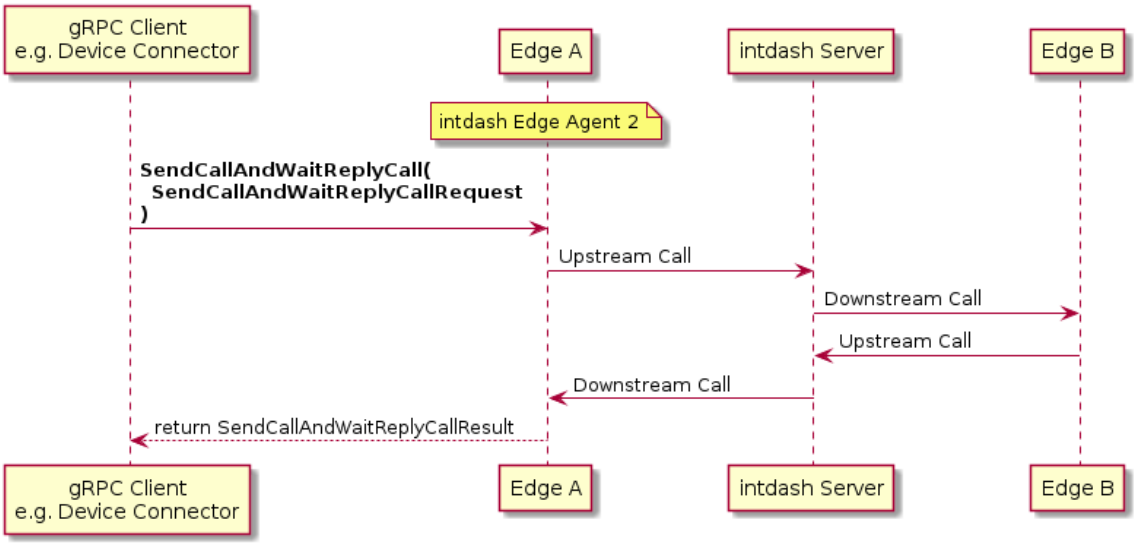



図 21　SendCallAndWaitReplyCall

## 23.4 ReceiveCalls

`ReceiveCalls` を実行すると、intdash Edge Agent 2 は、別のエッジからのコール（ただしリプライコール以外）を受信するためのストリームを返します。このメソッドは server streaming RPC です。

iSCP 2.0 のレベルでは、サーバーから複数の Downstream Call を受信することになります。

ストリームから取り出すことができる構造体 `ReceiveCallsResult` が、受信した Downstream Call を表します。

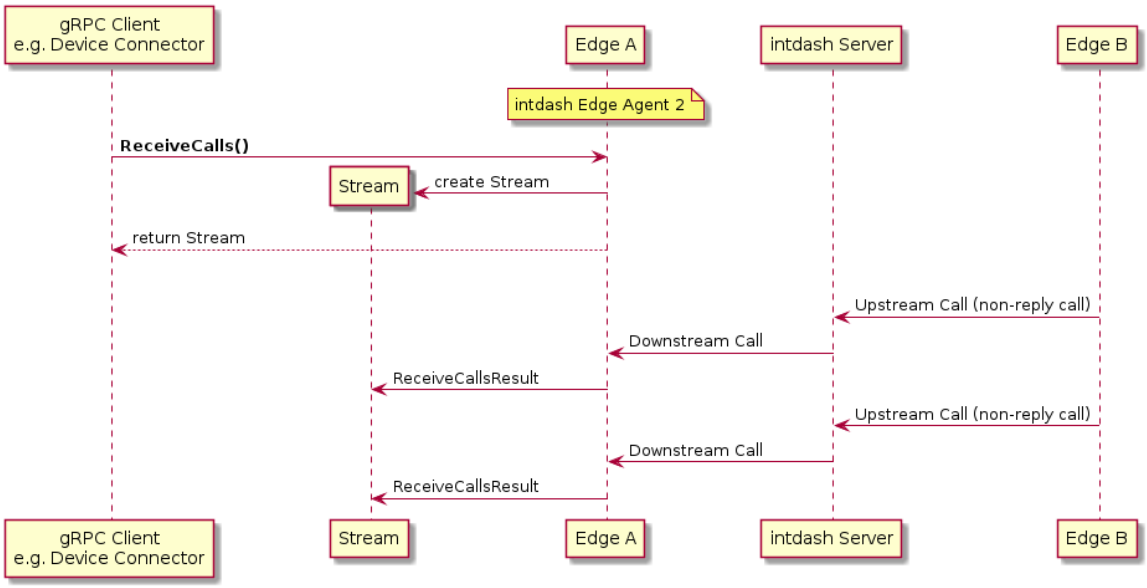



図 22　ReceiveCalls

## 23.5 ReceiveReplyCalls

`ReceiveReplyCalls` を実行すると、intdash Edge Agent 2 は、別のエッジからのリプライコールを受信するストリームを返します。このメソッドは server streaming RPC です。

iSCP 2.0 のレベルでは、サーバーから複数の Downstream Call を受信することになります。

ストリームから取り出すことができる構造体 `ReceiveReplyCallsResult` が、受信した Downstream Call を表します。

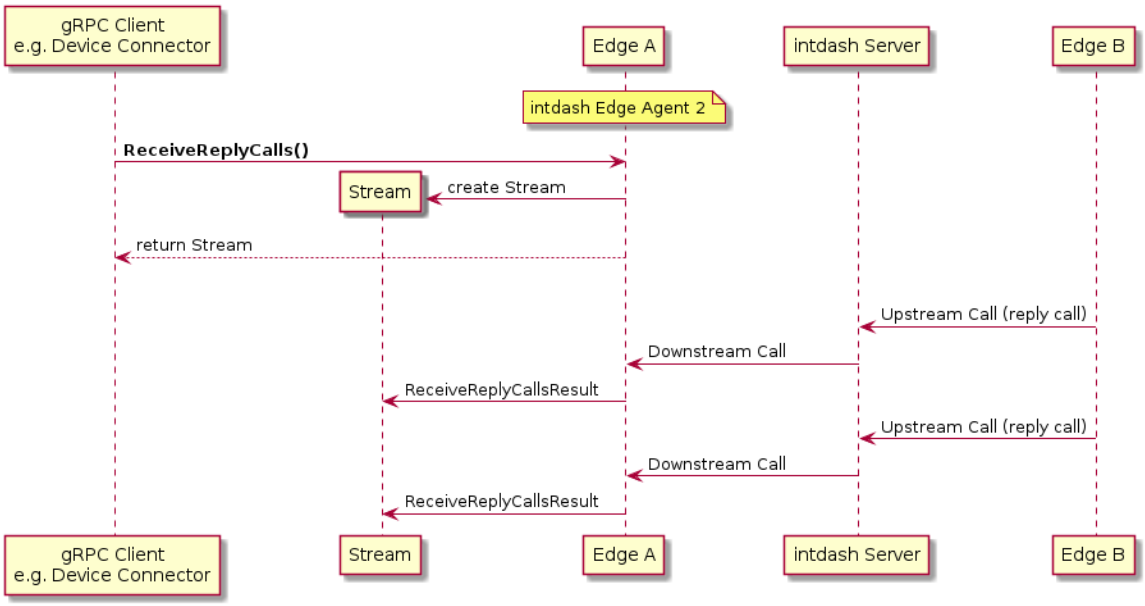



図 23　ReceiveReplyCalls

# 24 intdash-agentctl コマンド

intdash-agentctl コマンドは以下の書式で使用します。

```
intdash-agentctl [--version | -v]
intdash-agentctl [--help | -h]
intdash-agentctl <command> [<command_options>] [<arguments>]
```

**--version, -v**
: バージョンを表示します。

**--help, -h**
: ヘルプを表示します。

<command>ごとの使用方法については、以下を参照してください。

> **注釈:** コマンドや引数には、省略形が用意されている場合があります。例えば、`intdash-agentctl config upstream --list` は、`intdash-agentctl c u -l` のように省略することができます。
> 使用できる省略形を確認するには、各コマンドに `--help` オプションを付けて実行してください。

## 24.1 intdash-agentctl run

```
intdash-agentctl run [<run_command_options>]
```

ストリーマーを起動し、計測を開始します。

`<run_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、ログレベルは `information` となります。

**--address <value>**
: intdash-agentd の gRPC サーバーのアドレスを指定します。通常は、デフォルトのままでよいため指定する必要はありません。(デフォルト: localhost:50051)

**--name <name>, -n <name>**
: 計測の名前を設定します。

**--description <description>, -d <description>**
: 計測の説明を設定します。

**--timeout <seconds>, -t <seconds>**
: 計測終了までの秒数を指定します。指定しない場合、ctrl+C で終了するまで継続します。

**--e2e-caller-port <value>**
: intdash-agent-caller の gRPC サーバーに割り当てるポートを設定します。通常は、デフォルトのままでよいため指定する必要はありません。(デフォルト: localhost:50052)

**--help, -h**
: ヘルプを表示します。

## 24.2 intdash-agentctl config

```
intdash-agentctl config [<config_command_options>] <command> <command_option> [<arguments>]
```

intdash Edge Agent 2 の設定を変更します。

`<command>` では、`connection`、`upstream`、`filter upstream` などの、設定対象を示すサブコマンドを指定します。詳細については、設定対象ごとの説明を参照してください。

> **注釈:** `--modify <config_string>` `-m <config_string>` `--patch <config_string>` を使用する場合、<config_string>で指定されたフィールドのみが更新され、他のフィールドは変更されません。
> 削除できるフィールドの場合は、null を指定することで削除できます。
>
> ```
> intdash-agentctl config device-connector upstream --modify up-hello '
>  launch: null
> '
> ```

config コマンドに共通の `<config_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information レベルとなります。

**--json, -j**
: 設定の入出力を JSON 形式で行います。

**--help, -h**
: ヘルプを表示します。

> **注釈:** `intdash-agentctl config` コマンドで設定の取得、変更、削除などを実行すると、intdash Edge Agent 2 が持つ REST API に対してリクエストが行われ、設定の取得、変更、削除などが行われます。

### 24.2.1 connection サブコマンド

```
intdash-agentctl config [<config_command_options>] connection <command_option> [<arguments>]
```

intdash Edge Agent 2 と intdash サーバー間の接続について設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--get, -g**
: 接続の設定を表示します。

**--modify <config_string>, -m <config_string>, --patch <config_string>**
: 接続の設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。

  例:

```
$ intdash-agentctl config connection --modify '
    server_url: https://abc.example.com
    project_uuid: 01234567-89ab-cdef-0123-456789abcdef
  '
```

**注釈:**

- connection サブコマンドでは、REST API の `/connection` エンドポイントが使用されます。設定の詳細については、API リファレンスの Update Connection Settings を参照してください。
- `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

### 24.2.2 transport サブコマンド

```
intdash-agentctl config [<config_command_options>] transport <command_option> [<arguments>]
```

トランスポートについて設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--get, -g**
: トランスポートの設定を表示します。

**--modify <config_string>, -m <config_string>, --patch <config_string>**
: トランスポートの設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。<config_string>で指定されたフィールドのみが更新され、他のフィールドは変更されません。

例:

```
$ intdash-agentctl config transport --modify 'protocol: websocket'
```

**注釈:**

- transport サブコマンドでは、REST API の `/transport` エンドポイントが使用されます。設定の詳細については、API リファレンスの Update Transport Settings を参照してください。
- `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

### 24.2.3 upstream、downstream サブコマンド

```
intdash-agentctl config [<config_command_options>] upstream <command_option> [<arguments>]
intdash-agentctl config [<config_command_options>] downstream <command_option> [<arguments>]
```

アップストリームまたはダウンストリームについて設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--create <config_string>, -c <config_string>, --post <config_string>**

アップストリームまたはダウンストリームの設定を新規作成します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。指定しなかった設定値はデフォルトが使用されます。

例:

```
$ intdash-agentctl config upstream --create '
    id: recoverable
    enabled: true
    recover: true
    persist: true
    qos: unreliable
    flush_policy: interval
    flush_interval: 5
  '
```

**--list, -l**

アップストリーム／ダウンストリーム設定の一覧を表示します。

**--get <id>, -g <id>**

指定された ID のアップストリーム／ダウンストリームの設定を表示します。

例:

```
$ intdash-agentctl config upstream --get recoverable
```

**--modify <id> <config_string>, -m <id> <config_string>, --patch <id> <config_string>**

指定された ID のアップストリーム／ダウンストリームの設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。<config_string>で指定されたフィールドのみが更新され、他のフィールドは変更されません。

例:

```
$ intdash-agentctl config upstream --modify recoverable 'enabled: false'
```

**--delete <id>, -d <id>**

指定されたストリームを削除します。引数としてアップストリーム／ダウンストリームの ID を指定します。

**注釈:**

- upstream サブコマンドでは、REST API の `/upstreams` エンドポイントが使用されます。設定の詳細については、API リファレンスの Create Upstream Settings を参照してください。
- downstream サブコマンドでは、REST API の `/downstreams` エンドポイントが使用されます。設定の詳細については、API リファレンスの Create Downstream Settings を参照してください。
- `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

### 24.2.4 filter サブコマンド

```
intdash-agentctl config [<config_command_options>] filter upstream <command_option> [<arguments>]
intdash-agentctl config [<config_command_options>] filter downstream <command_option> [<arguments>]
```

フィルターについて設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--create <config_string>, -c <config_string>, --post <config_string>**

フィルターの設定を新規作成します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。指定しなかった設定値はデフォルトが使用されます。

例

```
$ intdash-agentctl config filter upstream --create '
    id: example
    enabled: true
    type: name
    target:
      dest_ids:
        - recoverable
      name: v1/.+
    change_to:
      dest_id: deferred
'
```

**--list, -l**

フィルター設定の一覧を表示します。

**--get <id>, -g <id>**

指定されたフィルターを表示します。引数としてフィルターの ID を指定します。

**--modify <id> <config_string>, -m <id> <config_string>, --patch <id> <config_string>**

指定された ID のフィルター設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。<config_string>で指定されたフィールドのみが更新され、他のフィールドは変更されません。

例:

```
$ intdash-agentctl config filter upstream --modify data_sampling 'enabled: false'
```

**--delete <id>, -d <id>**

指定されたフィルター設定を削除します。

**注釈:**

- filter upstream サブコマンドでは、REST API の `/filters_upstream` エンドポイントが使用されます。設定の詳細については、API リファレンスの Create Filter for Upstream を参照してください。
- filter downstream サブコマンドでは、REST API の `/filters_downstream` エンドポイントが使用されます。設定の詳細については、API リファレンス Create Filter for Downstream を参照してください。
- `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

### 24.2.5 device-connector コマンド

```
intdash-agentctl config [<config_command_options>] device-connector upstream <command_option> [<arguments>]
intdash-agentctl config [<config_command_options>] device-connector downstream <command_option> [<arguments>]
```

デバイスコネクター IPC について設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--create <config_string>, -c <config_string>, --post <config_string>**

デバイスコネクター IPC 設定を新規作成します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。指定しなかった設定値はデフォルトが使用されます。

例

```
$ intdash-agentctl config device-connector upstream --create '
  id: dc1
  data_name: group/subgroup
  dest_ids:
  - recoverable
  enabled: true
  data_name_prefix: v1/1/
  format: iscp-v2-compat
  ipc:
    path: /var/run/intdash/device_connector.fifo
    type: fifo
  launch:
    args:
    - --config
    cmd: device-connector-intdash
    environment:
    - KEY=value
'
```

**--list, -l**

デバイスコネクター IPC 設定の一覧を表示します。

**--get <id>**

指定されたデバイスコネクター IPC を表示します。

**--modify <id> <config_string>, -m <id> <config_string>, --patch <id> <config_string>**

指定された ID のデバイスコネクター IPC 設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。<config_string>で指定されたフィールドのみが更

新され、他のフィールドは変更されません。

例:

```
$ intdash-agentctl config device-connector upstream --modify dc1 'enabled: false'
```

**--delete <id>, -d <id>**

指定された ID のデバイスコネクター IPC を削除します。

> **注釈:**
> - device-connector upstream サブコマンドでは、REST API の `/device_connectors_upstream` エンドポイントが使用されます。設定の詳細については、API リファレンスの Create Device Connector IPC Settings for Upstream を参照してください。
> - device-connector downstream サブコマンドでは、REST API の `/device_connectors_downstream` エンドポイントが使用されます。設定の詳細については、API リファレンス Create Device Connector IPC Settings for Downstream を参照してください。
> - `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

### 24.2.6 deferred-upload コマンド

```
intdash-agentctl config [<config_command_options>] deferred-upload <command_option> [<arguments>]
```

遅延アップロードについて設定します。

#### 設定操作

`<command_option> [<arguments>]` で以下のように指定します。

**--get, -g**

遅延アップロードの設定を表示します。

**--modify <config_string>, -m <config_string>, --patch <config_string>**

遅延アップロードの設定を更新します。<config_string>には YAML 形式の文字列（--json を指定した場合は JSON 型式）を与えます。<config_string>で指定されたフィールドのみが更新され、他のフィールドは変更されません。

例:

```
$  intdash-agentctl config deferred-upload --modify 'priority: higher_than_realtime'
```

> **注釈:**
> - deferred-upload サブコマンドでは、REST API の `/deferred_upload` エンドポイントが使用されます。設定の詳細については、API リファレンスの Update Deferred Upload Settings を参照してください。
> - `<config_string>` では、API リファレンスに記載されているリクエストボディの JSON と同等の内容を YAML 形式で指定してください（または `--json` オプションを使用して JSON 形式で指定してください）。

## 24.3 intdash-agentctl config-file

```
intdash-agentctl config-file [<config_file_command_options>] <command> <command_option> [<arguments>]
```

config コマンドに共通の `<config_file_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information となります。

**--help, -h**
: ヘルプを表示します。

### 24.3.1 show サブコマンド

```
intdash-agentctl config-file [<config_file_command_options>] show [<command_options>]
```

intdash Edge Agent 2 の現在の設定を標準出力に書き出します。

`[<command_options>]` で以下のように指定します。

**--default**
: 現在の設定ではなくデフォルトの設定を表示します。

**--help, -h**
: ヘルプを表示します。

### 24.3.2 apply サブコマンド

```
intdash-agentctl config-file [<config_file_command_options>] apply <command_options>
```

指定されたファイルから設定を読み込み、適用します。

また、読み込まれた設定は現在使用中の設定ファイル（デフォルトでは `/var/lib/intdash/agent.yaml` 、ただし環境変数 `AGENT_LIB_DIR` が設定されている場合は `${AGENT_LIB_DIR}/agent.yaml` ）に書き込まれます。

`<command_options>` で以下のように指定します。

**--user-agent-config <value>, -c <value> (必須)**
: 使用する設定ファイルを指定します。

**--help, -h**
: ヘルプを表示します。

## 24.4 intdash-agentctl measurement

```
intdash-agentctl measurement [<measurement_command_options>] [<arguments>]
```

intdash Edge Agent 2 内にある計測のリストを古い順に表示します（一番下が最も新しいもの）。`--delete` オプションを使用することで、intdash Edge Agent 2 内のその計測に関するデータ（遅延アップロード用の送信待ちデータ含む）を削除することができます。

`<measurement_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information となります。

**--address <value>**
: intdash-agentd の gRPC サーバーのアドレスを指定します。通常は、デフォルトのままでよいため指定する必要はありません。(デフォルト: localhost:50051)

**--json, -j**
: 計測のリストを JSON 型式で表示します。

**--delete, -d**
: intdash Edge Agent 2 内の計測データを削除します。引数として、削除したい計測の UUID を指定します。

**--help, -h**
: ヘルプを表示します。

## 24.5 intdash-agentctl status

```
intdash-agentctl status [<status_command_options>]
```

アップストリーム、ダウンストリーム、遅延アップロードの現在の状態を表示します。

`<status_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information となります。

**--json, -j**
: ステータスのリストを JSON 型式で表示します。

**--help, -h**
: ヘルプを表示します。

## 24.6 intdash-agentctl ping

```
intdash-agentctl ping [<ping_command_options>]
```

intdash-agentd が正常に動作しているかどうかを確認します。

`<ping_command_options>` では、以下を使用することができます。

**--log <level>, -l <level>**
: ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

  ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information となります。

**--address <value>**
: intdash-agentd の gRPC サーバーのアドレスを指定します。通常は、デフォルトのままでよいため指定する必要はありません。(デフォルト: localhost:50051)

**--timeout <value>, -t <value>**
: 確認のタイムアウトを指定します（単位: 秒)。(デフォルト: 5)

**--help, -h**
: ヘルプを表示します。

## 24.7 intdash-agentctl about

```
intdash-agentctl about [<command_options>]
```

バージョン、各プログラムのバージョン、intdash Edge Agent 2 用の環境変数の値を表示します。

`<about_command_options>` では、以下を使用することができます。

**--help, -h**
: ヘルプを表示します。

## 24.8 intdash-agentctl help

```
intdash-agentctl help
```

intdash-agentctl コマンドの使い方を表示します。

# 25 intdash-agentd コマンド

**注釈:** intdash-agentd は、intdash Edge Agent 2 のデーモンとして使用されるプログラムです。通常、intdash-agentd の起動や終了は init スクリプトを使って行うので直接 `intdash-agentd` コマンドを実行する必要はありません。

intdash-agentd コマンドは以下の書式で使用します。

```
intdash-agentd [--version | -v]
intdash-agentd [--help | -h]
intdash-agentd <command> [<command_options>] [<arguments>]
```

**--version, -v**
　バージョンを表示します。

**--help, -h**
　ヘルプを表示します。

<command>ごとの使用方法については、以下を参照してください。

## 25.1 intdash-agentd serve

```
intdash-agentd serve [<serve_command_options>] <arguments>
```

intdash-agentd を起動します。

**注釈:** 通常、intdash-agentd の起動は init スクリプトを使って行うため、直接 `intdash-agentd serve` コマンドを実行する必要はありません。

`[<serve_command_options>]` では、以下を使用することができます。

**--log <level>, -l <level>**
　ログレベルを設定します。level には、以下を設定できます: `t[race]|d[ebug]|i[nformation]|w[arning]|e[rror]|q[uiet]`

　ログレベルは、環境変数 AGENT_LOG でも設定できます。環境変数 AGENT_LOG よりもこのコマンドオプションが優先されます。環境変数 AGENT_LOG が与えられておらず、このコマンドオプションも与えられていない場合、information レベルとなります。

**--port <value>**
　intdash-agentd の gRPC サーバーのポート番号を指定します。intdash-agentctl コマンドから計測情報にアクセスするために利用されます。通常は、デフォルトのままでよいため指定する必要はありません。(デフォルト: 50051)

**--uploader-request-size <value>**
　intdash-agentd から intdash サーバーに向けて遅延アップロードを行う際の、1 リクエストに含めるチャンクの最大サイズを指定します（単位: バイト)。(デフォルト: 1048576)

**--help, -h**
　ヘルプを表示します。

## 25.2 intdash-agentd help

```
intdash-agentd help
```

intdash-agentd コマンドの使い方を表示します。

# 26 intdash Edge Agent 2 設定一覧

`intdash-agentctl config-file show` コマンドにより出力することができる、intdash Edge Agent 2 の全設定は以下のとおりです。

**注釈:**
- 設定を出力する方法については 設定の書き出し／読み込み (p. 67) を参照してください。
- 設定可能な項目は、REST API の各エンドポイントと同じです。REST API のメッセージボディは JSON ですが、設定の書き出し／読み込み (p. 67) では同内容の YAML が使用されます。

設定は以下の部分に大別されます。

| キー | 型 | 説明 |
|---|---|---|
| connection (p. 88) | object | 接続先サーバーと認証情報の設定 |
| transport (p. 88) | object | トランスポートの設定 |
| upstream (p. 89) | object | アップストリームの設定 |
| downstream (p. 89) | object | ダウンストリームの設定 |
| filters_upstream (p. 90) | [object] | 上り方向のフィルター設定のリスト |
| filters_downstream (p. 90) | [object] | 下り方向のフィルター設定のリスト |
| device_connectors_upstream (p. 91) | [object] | 上り方向のデバイスコネクター IPC 設定のリスト |
| device_connectors_downstream (p. 91) | [object] | 下り方向のデバイスコネクター IPC 設定のリスト |
| deferred_upload (p. 92) | object | 未送信データの遅延アップロードに関する設定 |

## 26.1 設定の例

```yaml
connection:
  server_url: https://xxxxx.intdash.jp
  project_uuid: c48e3eee-0242-462f-xxxx-xxxxxxxxxxxx
  edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
  client_secret: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
transport:
  protocol: quic
upstream:
- id: recoverable
  enabled: true
  recover: true
  persist: true
  qos: unreliable
  flush_policy: interval
  flush_interval: 5
- id: h264_nal_unit_key_units
  enabled: true
  recover: true
  persist: true
  qos: partial
  flush_policy: immediately
  flush_interval: 5
- id: h264_nal_unit_extra_units
  enabled: true
  recover: true
```

(次のページに続く)

(前のページからの続き)

```yaml
  persist: true
  qos: unreliable
  flush_policy: immediately
  flush_interval: 5
downstream:
- id: down
  enabled: true
  dest_ids:
  - down-hello
  qos: unreliable
  filters:
  - src_edge_uuid: 03ace3b1-d208-xxxx-xxxxxxxxxxxx
    data_filters:
    - type: string
      name: v1/1/ab
device_connectors_upstream:
- id: up-hello
  data_name_prefix: v1/1/
  dest_ids:
  - recoverable
  enabled: true
  format: iscp-v2-compat
  ipc:
    type: fifo
    path: /var/run/intdash/up-hello.fifo
  launch:
    cmd: device-connector-intdash
    args:
    - --config
    - /tmp/dc-hello.yaml
- id: h264_nal_unit
  data_name_prefix: 101/
  dest_ids:
  - recoverable
  enabled: true
  format: iscp-v2-compat
  ipc:
    type: fifo
    path: /var/run/intdash/up-h264_nal.fifo
device_connectors_downstream:
- id: down-hello
  data_name_prefix: v1/1/
  enabled: true
  format: iscp-v2-compat
  ipc:
    type: fifo
    path: /var/run/intdash/down-hello.fifo
filters_upstream:
- id: string_to_deferred
  enabled: true
  type: type
  target:
    dest_ids:
      - recoverable
    type: string
  change_to:
    dest_id: deferred
```

(次のページに続く)

(前のページからの続き)

```
- id: h264_nal_unit_filter
  enabled: true
  type: type
  target:
    dest_ids:
      - recoverable
    type: h264_nal_unit
  change_to:
    dest_id: h264_nal_unit_extra_units
- id: h264_nal_unit_key_filter
  enabled: true
  type: h264-essential-nal-units
  target:
    dest_ids:
      - h264_nal_unit_extra_units
  change_to:
    dest_id: h264_nal_unit_key_units
filters_downstream: []
deferred_upload:
  priority: same_as_realtime
  limit_data_storage: true
  data_storage_capacity: 102400
```

## 26.2 connection

設定例：

```
connection:
  server_url: https://xxxxx.intdash.jp
  project_uuid: c48e3eee-0242-462f-xxxx-xxxxxxxxxxxx
  edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
  client_secret: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

**注釈:** 設定可能な項目は、REST API の `/connection` エンドポイントと同じです。設定の詳細については、API リファレンスの Update Connection Settings を参照してください。

## 26.3 transport

設定例：

```
transport:
  protocol: quic
```

**注釈:** 設定可能な項目は、REST API の `/transport` エンドポイントと同じです。設定の詳細については、API リファレンスの Update Transport Settings を参照してください。

## 26.4 upstream

設定例：

```yaml
upstream:
- id: recoverable
  enabled: true
  recover: true
  persist: true
  qos: unreliable
  flush_policy: interval
  flush_interval: 5
- id: h264_nal_unit_key_units
  enabled: true
  recover: true
  persist: true
  qos: partial
  flush_policy: immediately
  flush_interval: 5
- id: h264_nal_unit_extra_units
  enabled: true
  recover: true
  persist: true
  qos: unreliable
  flush_policy: immediately
  flush_interval: 5
```

**注釈:** 設定可能な項目は、REST API の `/upstreams` エンドポイントと同じです。設定の詳細については、API リファレンスの Create Upstream Settings を参照してください。

## 26.5 downstream

設定例：

```yaml
downstream:
- id: down
  enabled: true
  dest_ids:
  - down-hello
  qos: unreliable
  filters:
  - src_edge_uuid: 03ace3b1-d208-4fc3-xxxx-xxxxxxxxxxxx
    data_filters:
    - type: string
      name: v1/1/ab
```

データ ID のワイルドカード指定を含む、データ ID の詳細については iSCP の仕様を参照してください。

**注釈:** 設定可能な項目は、REST API の `/downstreams` エンドポイントと同じです。設定の詳細については、API リファレンスの Create Downstream Settings を参照してください。

## 26.6 filters_upstream

設定例：

```
filters_upstream:
- id: string_to_deferred
  enabled: true
  type: type
  target:
    dest_ids:
      - recoverable
    type: string
  change_to:
    dest_id: deferred
- id: h264_nal_unit_filter
  enabled: true
  type: type
  target:
    dest_ids:
      - recoverable
    type: h264_nal_unit
  change_to:
    dest_id: h264_nal_unit_extra_units
- id: h264_nal_unit_key_filter
  enabled: true
  type: h264-essential-nal-units
  target:
    dest_ids:
      - h264_nal_unit_extra_units
  change_to:
    dest_id: h264_nal_unit_key_units
```

**注釈:** 設定可能な項目は、REST API の `/filters_upstream` エンドポイントと同じです。設定の詳細については、API リファレンスの Create Filter for Upstream を参照してください。

## 26.7 filters_downstream

設定例：

```
filters_downstream:
- id: string_to_dc2
  enabled: true
  type: type
  target:
    dest_ids:
      - recoverable
    type: string
  change_to:
    dest_id: deferred
```

**注釈:** 設定可能な項目は、REST API の `/filters_downstream` エンドポイントと同じです。設定の詳細については、API リファレンス Create Filter for Downstream を参照してください。

## 26.8 device_connectors_upstream

設定例：

```
device_connectors_upstream
- id: up-hello
  enabled: true
  data_name_prefix: v1/1/
  format: iscp-v2-compat
  ipc:
    type: fifo
    path: /var/run/intdash/up-hello.fifo
  launch:
    cmd: device-connector-intdash
    args:
    - --config
    - /tmp/dc-hello.yaml
    environment:
    - TEST_VAR=1
```

**注釈:** 設定可能な項目は、REST API の `/device_connectors_upstream` エンドポイントと同じです。設定の詳細については、API リファレンスの Create Device Connector IPC Settings for Upstream を参照してください。

## 26.9 device_connectors_downstream

設定例：

```
device_connectors_downstream
- id: down-hello
  data_name_prefix: v1/1/
  enabled: true
  format: iscp-v2-compat
  ipc:
    type: fifo
    path: /var/run/intdash/down-hello.fifo
  launch:
    cmd: device-connector-intdash
    args:
    - --config
    - /tmp/down-hello.yaml
    environment:
    - TEST_VAR=1
```

**注釈:** 設定可能な項目は、REST API の `/device_connectors_downstream` エンドポイントと同じです。設定の詳細については、API リファレンス Create Device Connector IPC Settings for Downstream を参照してください。

## 26.10 deferred_upload

設定例：

```
deferred_upload:
  priority: same_as_realtime
  limit_data_storage: true
  data_storage_capacity: 102400
```

**注釈:** 設定可能な項目は、REST API の `/deferred_upload` エンドポイントと同じです。設定の詳細については、API リファレンスの Update Deferred Upload Settings を参照してください。

# 27 データ ID

iSCP のデータ ID は、データ型とデータ名称により構成されます。

intdash サーバーと intdash Edge Agent2 の間では iSCP に沿ったデータフォーマットが使用されますが (A)、デバイスコネクターと intdash Edge Agent 2 の間ではそれとは異なる FIFO 用データフォーマットが使用されています (B)。

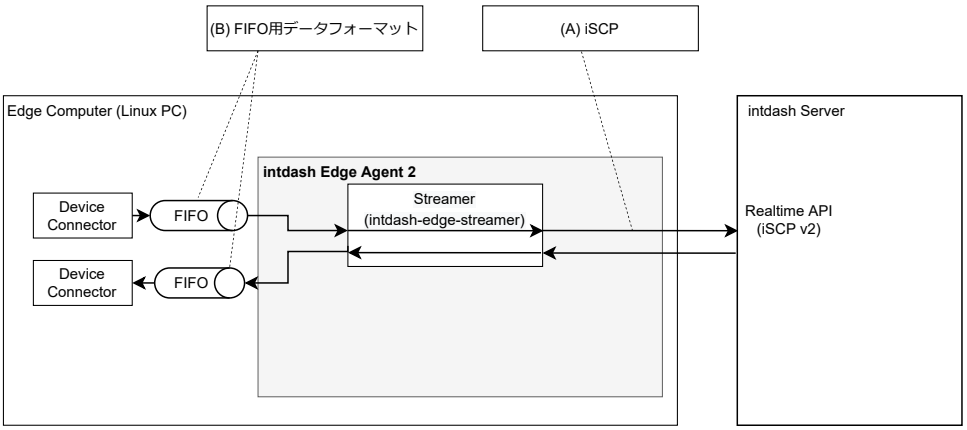



図 24　異なるフォーマット

そのため、デバイスコネクター→エッジ→サーバーの方向にデータが流れる際も、逆方向に流れる際も、intdash Edge Agent 2 においてデータフォーマットの変換が行われます。この変換の際には、データ ID の変換も行われます。

ここでは、FIFO 用データフォーマットのどの値がどのように iSCP のデータ ID に変換されるのか（およびその逆方向）について説明します。

## 27.1 アップストリーム方向の場合

intdash Edge Agent 2 がデバイスコネクターからデータを受信すると、データ型とデータ名称が以下のように決定され、データはサーバーに送信されます（推奨される iscp-v2-compat 形式の FIFO 用データフォーマットを使用する場合）。

**iSCP のデータ型**

FIFO 用データフォーマット (p. 100) 内の Data Type フィールドの値が使用されます。

**iSCP のデータ名称**

デバイスコネクター IPC 設定に含まれる `data_name_prefix` と、FIFO 用データフォーマットに含まれる Data Name フィールドの値を結合したものが使用されます。

例えば、FIFO 用データフォーマット内の Data Type フィールドが `string` 、デバイスコネクター IPC 設定の `data_name_prefix` が `abc` 、Data Name が `def` だった場合、iSCP でのデータ型は `string` 、データ名称は `abcdef` となります。

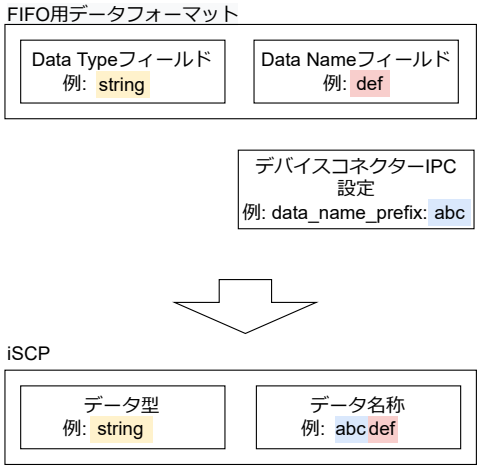



図 25　アップストリーム方向の場合

**重要:** FIFO 用フォーマットの Data Name は基本的に空文字以外を設定してください。
Data Name を空文字として、data_name_prefix も空文字とした場合、iSCP でのデータ名称も空文字になり、不正なデータ名称になります。不正なデータ名称を持つデータポイントは、intdash サーバーへの送信でエラーになります。
計測内にエラーとなったデータポイントが含まれる場合、intdash Edge Agent 2 は計測データが壊れていると判断し、それ以後、その計測について未送信データの遅延アップロードを行いません。

なお、iSCP データ名称が v1/<チャンネル>/ から始まる場合、iSCP v1 互換のための特別な処理が行われます。詳細については、iSCP v1 互換のデータ ID (p. 97) を参照してください。

**注釈:** FIFO 用データフォーマットとして logger-msg 形式（非推奨）を使用する場合は、上記とは異なります。

- iSCP のデータ型は、logger-msg 形式のデータが持つデータタイプによって一対一で決定されます。対応については以下の表を参照してください。
- iSCP のデータ名称は、ストリーマーにて以下の表のように生成されます（logger-msg フォーマットには Data Name フィールドがないため）。NMEA、CAN/CAN-FD、String 等ではペイロード内の情報を使ってデータ名称が生成されます。

| logger-msg のデータタイプ | 変換後 | |
|---|---|---|
| | iSCP のデータ型 | iSCP のデータ名称 |
| NMEA (0x10) | string/nmea | `<data_name_prefix>`<トーカ>/<メッセージ><br>ペイロード内の NMEA センテンス（例:「$GNRMC,..」）から、2〜3 文字目のトーカ（例 :「GN」）と、4〜6 文字目のメッセージ（例 :「RMC」）が抽出されます。 |
| CAN/CAN-FD (0x11) | can_frame | `<data_name_prefix><CAN ID>`<br>ペイロード内の CAN フレームから CAN ID が抽出されます。 |
| JPEG (0x12) | jpeg | `<data_name_prefix>jpeg` |
| H264 (0x1C) | h264_annex_b | `<data_name_prefix>h264_annex_b` |
| String (0x1D) | string | `<data_name_prefix><logger-msg の ID フィールド>` |
| Float (0x1E) | float64 | `<data_name_prefix><logger-msg の ID フィールド>` |
| Int (0x1F) | int64 | `<data_name_prefix><logger-msg の ID フィールド>` |
| Bytes (0x20) | bytes | `<data_name_prefix><logger-msg の ID フィールド>` |
| PCM (0x22) | pcm | `<data_name_prefix>pcm` |

## 27.2 ダウンストリーム方向の場合

intdash Edge Agent 2 がサーバーからデータを受信すると、データは以下のように FIFO 用データフォーマットに変換され、デバイスコネクターに送信されます（推奨される iscp-v2-compat 形式の FIFO 用データフォーマットを使用する場合）。

**FIFO 用データフォーマット内の Data Type**

iSCP のデータ型の値が使用されます。

**FIFO 用データフォーマット内の Data Name**

デバイスコネクター IPC 設定に含まれる `data_name_prefix` の値によって以下のように決まります。

- `data_name_prefix` に空文字以外が設定されている場合は、データポイントに付与されているデータ名称から、`data_name_prefix` が除去されます。除去処理が行われたデータポイントのみがデバイスコネクターに送信されます。
- `data_name_prefix` が空文字の場合は、データポイントに付与されているデータ名称がそのまま使用されます。

例えば、デバイスコネクター IPC 設定の `data_name_prefix` として `abc` が設定され、データポイントに付与されているデータ名称が `abcdef` だった場合、このデータポイントは `abc` に一致しますので、デバイスコネクターに送信されます。また、その際デバイスコネクターが受け取るデータポイントの Data Name フィールドは `def`

となります。

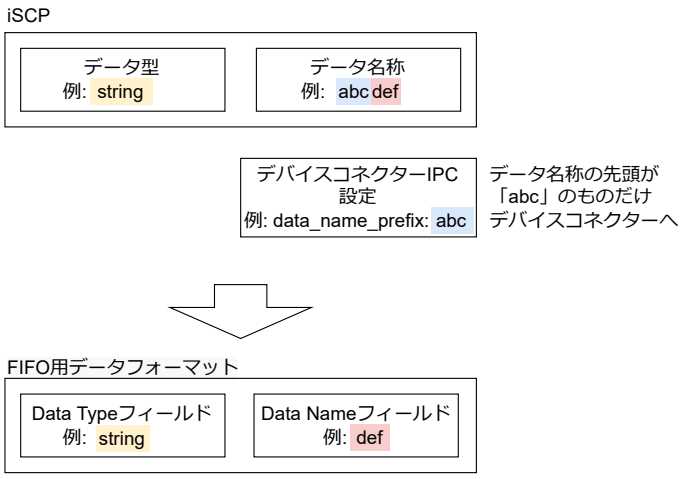



図 26　ダウンストリームの場合のデータ名称（data_name_prefix に一致した場合）

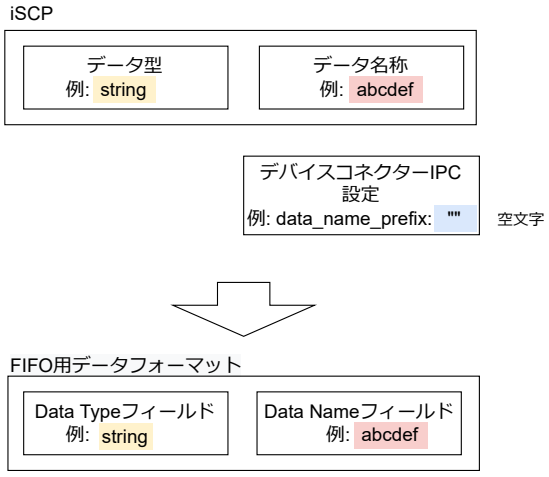



図 27　ダウンストリームの場合のデータ名称（data_name_prefix が空文字の場合）

なお、iSCP データ名称が v1/<チャンネル>/ から始まる場合、iSCP v1 互換のための特別な処理が行われます。詳細については、iSCP v1 互換のデータ ID (p. 97) を参照してください。

**注釈:** FIFO 用データフォーマットとして logger-msg 形式（非推奨）を使用する場合は、上記とは異なります。iSCP のデータ ID は以下のように logger-msg フォーマットに変換されます。

| iSCP のデータ型 | 変換後 | |
|---|---|---|
| | logger-msg のデータタイプ | logger-msg の ID フィールド |
| string/nmea | NMEA (0x10) | (ID フィールドなし) |
| can_frame | CAN/CAN-FD (0x11) | iSCP のデータ名称から先頭の `data_name_prefix` を削除したもの (CAN ID) |
| jpeg | JPEG (0x12) | (ID フィールドなし) |
| h264_annex_b | H264 (0x1C) | (ID フィールドなし) |
| string | String (0x1D) | iSCP のデータ名称から先頭の `data_name_prefix` を削除したもの |
| float64 | Float (0x1E) | iSCP のデータ名称から先頭の `data_name_prefix` を削除したもの |
| int64 | Int (0x1F) | iSCP のデータ名称から先頭の `data_name_prefix` を削除したもの |
| bytes | Bytes (0x20) | iSCP のデータ名称から先頭の `data_name_prefix` を削除したもの |
| pcm | PCM (0x22) | (ID フィールドなし) |

## 27.3 iSCP v1 互換のデータ ID

iSCP v2 と iSCP v1 とでは、データを特定する方法が異なります。iSCP v1 にはチャンネルがありますが、v2 にはチャンネルがありません。また、データ ID の仕組みも異なります。

しかし、iSCP v2 でアップストリームを行う際に以下のようなデータ名称を付与してサーバーに送信すると、サーバー側の仕組みにより、そのデータは iSCP v1 でダウンストリームすることができます。

| iSCP v2 のデータ型 | iSCP v1 と互換性を持たせるためのデータ名称 | データ名称例 |
|---|---|---|
| string/nmea | `v1/<チャンネル>/<トーカ><メッセージ>` | v1/1/GPRMC |
| can_frame | `v1/<チャンネル>/<CAN ID（最上位ビットは IDE)>` | v1/1/80000001 |
| jpeg | `v1/<チャンネル>/jpeg` | v1/1/jpeg |
| h264_annex_b | `v1/<チャンネル>/<iSCP v1 の H.264 データのタイプ>` | v1/1/01 |
| string | `v1/<チャンネル>/<iSCP v1 のデータ ID>` | v1/1/hello |
| float64 | `v1/<チャンネル>/<iSCP v1 のデータ ID>` | v1/1/hello |
| int64 | `v1/<チャンネル>/<iSCP v1 のデータ ID>` | v1/1/hello |
| bytes | `v1/<チャンネル>/<iSCP v1 のデータ ID>` | v1/1/hello |
| pcm | `v1/<チャンネル>/pcm` | v1/1/pcm |

<チャンネル> は、10 進数 0〜255 で指定してください。

**注釈:** intdash サーバーは、データ名称の最初が `v1/<10 進数 0〜255>/` になっている場合、10 進数の部分を iSCP v1 用のチャンネル番号として解釈します。また、`v1/<10 進数 0〜255>/` の後の文字列を、データ型に応じて、トーカ、メッセージ、CAN ID、または iSCP v1 のデータ ID として解釈します。

## 27.4 iSCP v1 互換のデータ名称

**注釈:** ここでは、アップストリームの場合を例にして、FIFO 用データフォーマット→ iSCP の方向で説明しますが、ダウンストリームの場合も対応関係は同様です。

intdash Edge Agent 2 において、上の表のように iSCP v1 と互換性を持たせるためのデータ名称を与えるためには、デバイスコネクター IPC 設定で、`data_name_prefix` を `v1/<チャンネル>/` としてください。その後ろの部分については、以下のとおりです。

**重要:** ただし、iscp-v2-compat 形式で h264_nal_unit 型のデータを送信したい場合は、`data_name_prefix` を `<チャンネル番号>/` とし（`v1/` は不要）、iscp-v2-compat の Data Name フィールドを `h264_nal_unit` としてください。h264_nal_unit 型は iSCP v1 に存在しないデータ型であるため、特別な指定方法となっています。

### 27.4.1 iscp-v2-compat 使用時

string/nmea、can_frame、jpeg、h264_annex_b、pcm 型の場合、`data_name_prefix` と、iscp-v2-compat フォーマットの Data Name フィールドを結合した文字列の先頭が `v1/<チャンネル>/` であるとき[1]、ストリーマーにて以下のように特別な処理が行われます。

**string/nmea 型**
: ペイロード内の NMEA センテンス（例:「$GNRMC,..」）から 2〜3 文字目のトーカ（例 :「GN」）と、4〜6 文字目のメッセージ（例 :「RMC」）が抽出され、データ名称 `v1/<チャンネル>/<トーカ><メッセージ>` が付与されます。

**can_frame 型**
: ペイロード内から CAN ID が抽出され、データ名称 `v1/<チャンネル>/<CAN ID（最上位ビットは IDE）>` が付与されます。

**jpeg 型**
: 常にデータ名称 `v1/<チャンネル>/jpeg` が付与されます。

**h264_annex_b 型**
: ペイロード内から H.264 データのタイプが抽出され、データ名称 `v1/<チャンネル>/<iSCP v1 の H.264 データのタイプ>` が付与されます。

**pcm 型**
: 常にデータ名称 `v1/<チャンネル>/pcm` が付与されます。

string、float64、int64、bytes 型の場合、ストリーマーでの特別な処理は行われません。アップストリーム時のデータ名称は `<data_name_prefix><iscp-v2-compat フォーマットの Data Name フィールド>` のようになりますので、iscp-v2-compat フォーマットの Data Name フィールドに、iSCP v1 のデータ ID として使用したい文字列を指定してください。

例えば、`data_name_prefix` を `v1/1/` とし、iscp-v2-compat フォーマットの Data Name を `abc` とすれば、`v1/1/abc` としてアップストリームが行われ、これを iSCP v1 でダウンストリームする場合は、チャンネル `1`、データ ID `abc` を指定すればよいことになります。

[1] 以下の場合はいずれも該当します。
- `data_name_prefix` が `v1/1/` で、iscp-v2-compat フォーマットの Data Name フィールドが `abc` である場合 → `v1/1/abc`
- `data_name_prefix` が `v1/` で、iscp-v2-compat フォーマットの Data Name フィールドが `1/abc` である場合 → `v1/1/abc`

### 27.4.2 logger-msg（非推奨）使用時

string/nmea、can_frame、jpeg、h264_annex_b、pcm 型の場合、data_name_prefix が v1/<チャンネル>/ になっていると、ストリーマーにて以下のように特別な処理が行われます。

**string/nmea 型**

ペイロード内の NMEA センテンス（例:「$GNRMC,..」）から 2〜3 文字目のトーカ（例 :「GN」）と、4〜6 文字目のメッセージ（例 :「RMC」）が抽出され、データ名称 v1/<チャンネル>/<トーカ><メッセージ> が付与されます。

**can_frame 型**

ペイロード内から CAN ID が抽出され、データ名称 v1/<チャンネル>/<CAN ID（最上位ビットは IDE)> が付与されます。

**jpeg 型**

常にデータ名称 v1/<チャンネル>/jpeg が付与されます。

**h264_annex_b 型**

ペイロード内から H.264 データのタイプが抽出され、データ名称 v1/<チャンネル>/<iSCP v1 の H.264 データのタイプ> が付与されます。

**pcm 型**

常にデータ名称 v1/<チャンネル>/pcm が付与されます。

string、float64、int64、bytes 型の場合、ストリーマーでの特別な処理は行われません。アップストリーム時のデータ名称は <data_name_prefix><logger-msg フォーマットの ID フィールド> のようになります。

例えば、data_name_prefix を v1/1/ とし、logger-msg フォーマットの ID フィールドを abc とすれば、v1/1/abc としてアップストリームが行われ、これを iSCP v1 でダウンストリームする場合は、チャンネル 1 、データ ID abc を指定すればよいことになります。

# 28 FIFO 用データフォーマット

intdash Edge Agent 2 とデバイスコネクターの間（デバイコネクター IPC）で使われる FIFO 用データフォーマットについて説明します。

**重要:** intdash Edge Agent 2 とデバイスコネクターの間で使われる FIFO 用データフォーマットには 2 種類あります。どちらを使用するかはデバイスコネクター IPC 設定で指定します。

- iscp-v2-compat: iSCP v2 のペイロードフォーマットに沿ったフォーマット
- logger-msg: 旧 intdash Edge Agent で使用されていたフォーマット（非推奨）

本章で説明するのは iscp-v2-compat フォーマットです。
旧フォーマット logger-msg については、旧 intdash Edge Agent のデベロッパーガイド を参照してください。

## 28.1 iscp-v2-compat の全体像

iscp-v2-compat フォーマットは以下のとおりです。

```
0                                                               31
+---------------+---------------+---------------+---------------+
|                      Timestamp (seconds)                      |
+---------------+---------------+---------------+---------------+
|                      Timestamp (nano seconds)                 |
+---------------+---------------+---------------+---------------+
|        Data Type Length       |         Data Name Length      |
+---------------+---------------+---------------+---------------+
|                           Payload Length                      |
+---------------+---------------+---------------+---------------+
|   Data Type
+-----------//--+
|   Data Name
+-----------//--+
|   Payload
+-----------//--+
```

| フィールド名 | バイト長 | エンディアン | 符号 | 値範囲 | 説明 |
|---|---|---|---|---|---|
| Timestamp (seconds) | 4 | LE | U | 0 ~ 4294967295 | このデータのタイムスタンプ（秒）。アップストリームの場合とダウンストリームの場合で意味が異なります。Timestamp フィールドの意味 (p. 101) を参照してください。 |
| Timestamp (nano seconds) | 4 | LE | U | 0 ~ 999999999 | このデータのタイムスタンプ（ナノ秒）。アップストリームの場合とダウンストリームの場合で意味が異なります。Timestamp フィールドの意味 (p. 101) を参照してください。 |
| Data Type Length | 2 | LE | U | 0 ~ 65535 | Data Type フィールドのバイト長 |
| Data Name Length | 2 | LE | U | 0 ~ 65535 | Data Name フィールドのバイト長 |
| Payload Length | 4 | LE | U | 0 ~ 4294967295 | Payload フィールドのバイト長 |
| Data Type | 可変 | - | - | - | iSCPv2 のデータ型の文字列 |
| Data Name | 可変 | - | - | - | iSCPv2 のデータ名称（もしくはデータ名称の末尾）の文字列。この値とデータ ID の関係については データ ID (p. 93) を参照してください。 |
| Payload | 可変 | - | - | - | iSCPv2 拡張仕様のペイロード |

## 28.2 Timestamp フィールドの意味

Timestamp フィールドの意味は、アップストリームの場合とダウンストリーム場合で異なります。

### 28.2.1 アップストリームの場合

POSIX の clock_gettime() の引数に CLOCK_MONOTONIC（または CLOCK_MONOTONIC_RAW）を指定して取得した時間を設定します。

> **注釈:** intdash Edge Agent 2 のデフォルト設定では、CLOCK_MONOTONIC が使用されます。CLOCK_MONOTONIC_RAW を使用する場合は、Agent コマンドでの計測を実行時に以下のように環境変数を設定してください。
>
> ```
> AGENT_CLOCK_ID=CLOCK_MONOTONIC_RAW intdash-ageentctl run
> ```

CLOCK_MONOTONIC の取得方法の例

#### C

```c
#include <time.h>
#include <stdio.h>

int main(int argc, const char* argv[])
{
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  printf("%ld%09ld ns\n", ts.tv_sec, ts.tv_nsec);
  return 0;
}
```

#### C++11

```cpp
#include <stdio.h>
#include <chrono>

int main(int argc, const char* argv[])
{
  auto steady_ts = std::chrono::steady_clock::now();
  printf("%lld ns\n",
        static_cast<long long>(std::chrono::duration_cast<std::chrono::nanoseconds>(steady_ts.time_since_
↪epoch()).count()));
  return 0;
}
```

#### Go

```go
package main

import "fmt"

/*
#include <time.h>
static unsigned long long get_nsecs(void)
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (unsigned long long)ts.tv_sec * 1000000000UL + ts.tv_nsec;
}
*/
```

（次のページに続く）

(前のページからの続き)

```
import "C"

func main() {
  monotonic := uint64(C.get_nsecs())
  fmt.Println(monotonic, "ns")
}
```

### Python

```
#!/usr/bin/env python3
import time

if __name__ == '__main__':
  now = int(time.clock_gettime(time.CLOCK_MONOTONIC) * 1_000_000_000)
  print("%d ns" % (now))
```

### Rust

```
use libc;

fn main() {
    let mut ts = libc::timespec {
        tv_sec: 0,
        tv_nsec: 0,
    };
    unsafe {
        libc::clock_gettime(libc::CLOCK_MONOTONIC, &mut ts);
    };
    println!(
        "{} ns",
        (ts.tv_sec as u64) * 1000000000 + (ts.tv_nsec as u64)
    );
}
```

## 28.2.2 ダウンストリームの場合

ダウンストリームの際の FIFO データに含まれる Timestamp フィールドには、計測の基準時刻に経過時間を足すことで得られたタイムスタンプが格納されます。

このとき、基準時刻としては、アップストリームを行うエッジがサーバーに送信した基準時刻（エッジのリアルタイムクロックによるもの、NTP によるものなど複数の基準時刻）のうち、最も高い優先度を持つものが使用されます。計測の途中で優先度が高い基準時刻が得られると、基準時刻はそれに置き換わり、それ以降のタイムスタンプは新しい基準時刻を基に計算されます。

複数のエッジ（ここではエッジ A、エッジ B とします）からのデータを 1 つの intdash Edge Agent 2 でダウンストリームする場合、エッジ A からのデータポイントの Timestamp フィールドはエッジ A が送信した基準時刻を基にしたタイムスタンプ、エッジ B からのデータポイントの Timestamp フィールドはエッジ B が送信した基準時刻を基にしたタイムスタンプです。

# 29 intdash Edge Agent 2 環境変数一覧

| 環境変数 | 説明 |
|---|---|
| AGENT_LOG | ログレベル (trace ¦ debug ¦ information ¦ warning ¦ error ¦ quiet) |
| AGENT_CLOCK_ID | 基準時刻取得に使用する時計の種類（CLOCK_MONOTONIC ¦ CLOCK_MONOTONIC_RAW) |
| AGENT_RUN_DIR | アプリケーション起動中の状態を保存するディレクトリ |
| AGENT_LIB_DIR | agent.db、agent.yaml などのアプリケーションデータを保存するディレクトリ（agent.yaml の変更を反映するにはデーモンの再起動が必要です。) |
| AGENT_CONF_DIR | アプリケーションのコンフィグレーションを保存するディレクトリ |
| AGENT_DAEMON | intdash-agentd の実行バイナリのパス |
| AGENT_STREAMER | ストリーマーの実行バイナリのパス |
| AGENT_CALLER | E2E コーラーの実行バイナリのパス |

# 30 旧 intdash Edge Agent からの移行

本製品と 旧 intdash Edge Agent は、操作や設定の方法が異なります。

これまで旧 intdash Edge Agent を使用していて、本製品 intdash Edge Agent 2 に移行する場合は、以下をお読みください。



## 30.1 旧ソフトウェアとの主な違い

intdash Edge Agent 2 が旧 intdash Edge Agent と異なる点を説明します。以下の記述は概要のみを説明するもので、すべての違いを網羅したものではありません。



### 30.1.1 使用するプロトコルの変更

**旧 intdash Edge Agent** では、リアルタイムデータの送受信用のプロトコルとして iSCP 1.0 を使用していました。

**intdash Edge Agent 2** では、iSCP 2.0 を使用します。そのため、アップストリームとダウンストリームの設定方法が変更されています。ただし、iSCP 2.0 に対応した intdash サーバーは iSCP 1.0 と 2.0 の間の相互変換機能を持っています。iSCP 1.0 と 1SCP 2.0 の間の互換仕様については、iSCP 2.0 プロトコル仕様書を参照してください。

### 30.1.2 設定方法の変更

**旧 intdash Edge Agent** では、設定の変更は設定ファイルを修正することで行いました。

**intdash Edge Agent 2** では、設定の変更は intdash-agentctl config コマンド (p. 75) で行います。ただし、`intdash-agentctl config-file show` コマンドを実行して設定をエクスポートし、ファイルとして編集することも可能です。詳細については、設定の書き出し／読み込み (p. 67) を参照してください。

### 30.1.3 「再送」は「遅延アップロード」に

**旧 intdash Edge Agent** では、Realtime クライアントまたは Bulk クライアントから送信を試みて正しく送信できなかった場合、そのデータはエッジコンピューター内に保持され、Resend クライアントから再び送信されていました。

この機能は「再送」と呼ばれ、この機能が有効なのはクライアントの設定が以下のようになっている場合でした：

- `mode` が `1`（永続化する）、かつ `noresend` が `false`（再送を行う）

また、フィルター機能を使ってデータをリアルタイム送信の対象から除外することにより、リアルタイムでの送信をせずに再送処理に回すこともできました。

**intdash Edge Agent 2** でも同様に、リアルタイム送信を試みて正しく送信できなかったデータは、エッジコンピューター内に保持され、再び送信されます。

この機能は「遅延アップロード」と呼ばれ、この機能が有効なのはアップストリームの設定が以下のようになっている場合です：

- `persist` が `true`（永続化する）、かつ `recover` が `true`（遅延アップロードを行う）

また、デバイスコネクターで取得したデータの送信先として `deferred` を指定した場合、このデータのリアルタイム送信は行われず、遅延アップロードで送信されます。詳細については 遅延アップロード (p. 61) を参照してください。

旧 intdash Edge Agent の再送用のデータと、intdash Edge Agent 2 の遅延アップロード用のデータは形式が異なり、互換性がありません。

### 30.1.4 リアルタイム送信の「クライアント」は「ストリーマー」に

**旧 intdash Edge Agent** では、intdash サーバーとの通信は「クライアント」が担っていました（送信を行う Realtime クライアントと、受信を行う Control クライアント）。

**intdash Edge Agent 2** では、リアルタイム API による送信と受信は「ストリーマー」が担います。なお、REST API による遅延アップロード（旧 intdash Edge Agent の「再送」に相当）は、「デーモン」が担います。

ストリーマーによるアップストリームとダウンストリームについては、以下のような設定が可能です。詳細については、アップストリームによる送信 (p. 38) 、ダウンストリームによる受信 (p. 45) を参照してください。

**アップストリームの設定の例**

```
upstream:
- id: upstream1            # ストリームを区別する ID
  enabled: true            # 有効／無効
  recover: true            # 送信できなかった場合に遅延アップロードするか
  persist: true            # サーバーに永続化するか
  qos: unreliable          # iSCP の QoS(Quality of Service)
  flush_policy: interval   # フラッシュするタイミング
  flush_interval: 5        # フラッシュ間隔
```

**ダウンストリームの設定の例**

```
downstream:
- id: downstream1              # ストリームを区別する ID
  enabled: true                # 有効／無効
  dest_ids:                    # 受信したデータを渡す先のデバイスコネクター IPC
  - down-hello
  filters:                     # 受信対象とするデータの指定
  - src_edge_uuid: 03ace3b1-d208-xxxx-xxxxxxexample
    data_filters:
    - type: string
      name: v1/1/ab
```

### 30.1.5 FIFO 用フォーマットの変更

**旧 intdash Edge Agent** では、デバイスコネクターと intdash Edge Agent の間でデータを送受信する際、logger-msg という専用フォーマットを使用し、タイムスタンプを CLOCK_MONOTONIC_RAW で取得していました。

**intdash Edge Agent 2** では、新しい専用フォーマット iscp-v2-compat を使用し、タイムスタンプを CLOCK_MONOTONIC で取得します。

ただし、デバイスコネクター IPC に関する設定（アップストリーム用 (p. 91)、ダウンストリーム用 (p. 91)）で `format: logger-msg` のように指定すると、旧フォーマット logger-msg を使用することが可能です（非推奨）。

### 30.1.6 RAW データ保存の廃止

**旧 intdash Edge Agent** では、デバイスコネクターから入力されたデータはすべて RAW データとして保存されていました。

**intdash Edge Agent 2** には RAW データを保存する機能はありません。

なお、RAW データの保存と、リアルタイム送信できなかったデータをローカルに保存する機能は別の機能です。intdash Edge Agent 2 でも、リアルタイム送信できなかったデータはローカルに保存され、遅延アップロード (p. 106) が行われます。

### 30.1.7 デバイスコネクターとの接続設定の変更

**旧 intdash Edge Agent** では、デバイスコネクターを起動してデータを取得するため設定は、マネージャーの設定ファイルの `logger` の項目で設定していました。

**intdash Edge Agent 2** では、デバイスコネクターからデータを取得するため設定は、「デバイスコネクター IPC 設定」(IPC = InterProcess Communication) で設定します。デバイスコネクター IPC に関する設定の説明（アップストリーム用 (p. 91)、ダウンストリーム用 (p. 91)）を参照してください。

### 30.1.8 付属デバイスコネクターの変更

**旧 intdash Edge Agent** では、デバイスコネクターとして、intdash-edge-logger が付属していました。

**intdash Edge Agent2** では、device-connector-intdash (p. 140) が付属します。

device-connector-intdash では、外部デバイスからのデータを上述の新データフォーマット (iscp-v2-compat (p. 100)) に簡単に変換することができます。また、アプトポッド製のデバイスコネクター作成用フレームワークである Device Connector Framework (p. 182) を採用しているため、データの取得、加工、書き出しの処理を柔軟に定義することができます。独自の処理を行うための拡張開発も容易です。

### 30.1.9 Status プラグインの廃止

**旧 intdash Edge Agent** では、Status プラグインを使ってエッジコンピューターのステータス情報を収集することができました。このステータス情報をサーバーに送信することで、遠隔からエッジコンピューターのステータスを確認することができました。

**intdash Edge Agent 2** には Status プラグインはありません。代わりに、付属デバイスコネクター device-connector-intdash と付属設定ファイル `/etc/dc_conf/device_inventory.yml` を使用することで、エッジコンピューターのステータス情報を収集することができます。詳細については ステータスを送信する (p. 137) を参照してください。

## 30.2 設定の移行方法

既存の旧 intdash Edge Agent の設定を intdash Edge Agent 2 に移行する方法を以下で説明します。各設定の詳細については、本ドキュメントの各設定項目のページを参照してください。

### 30.2.1 接続先サーバー URL の設定

#### intdash Edge Agent での設定

`clients` フィールドが持つ以下のフィールドで設定していました。リアルタイム送信用クライアント、再送用クライアント、受信用クライアントなど複数のクライアントを配列指定する必要があり、各要素ごとに同一の設定値を書き込む必要がありました。

- `clients[].connection.host`
- `clients[].connection.path`

#### intdash Edge Agent 2 での設定

`connection.server_url` に URL 形式で記載します。

### 30.2.2 プロジェクト UUID の設定

#### intdash Edge Agent での設定

`clients` フィールドが持つ `clients[].project_uuid` フィールドで設定していました。リアルタイム送信用クライアント、再送用クライアント、受信用クライアントなど複数のクライアントに同一のプロジェクト UUID を書き込む必要がありました。

#### intdash Edge Agent 2 での設定

`connection.project_uuid` で設定します。

### 30.2.3 送信元エッジ認証情報の設定

#### intdash Edge Agent での設定

`clients` フィールドが持つ以下のフィールドで設定していました。リアルタイム送信用クライアント、再送用クライアント、受信用クライアントなど複数のクライアントに同一の設定値を書き込む必要がありました。

- `clients[].my_id` (エッジ UUID)
- `clients[].my_secret` (クライアントシークレット)
- `clients[].my_token` (エッジトークン、API トークン)

#### intdash Edge Agent 2 での設定

以下の項目で設定します。

- `connection.edge_uuid` (エッジ UUID)
- `connection.client_secret` (クライアントシークレット)

### 30.2.4 アップストリームの設定

#### intdash Edge Agent での設定

`clients` フィールドが持つ以下のフィールドを組み合わせて設定していました。

- `clients[].mode`
- `clients[].unit_flush_cycle`
- `clients[].noresend`
- `clients[].store_flushtime`
- `clients[].store_flushsize`
- `clients[].store_cycle`

#### intdash Edge Agent 2 での設定

upstream で設定します。永続化するか（ persist ）、リアルタイム送信で到達しなかったデータを後回収するか（ recover ）などが指定できます。また、送信時のバッファ設定は flush_policy で指定します。

### 30.2.5 ダウンストリームの設定

#### intdash Edge Agent での設定

clients フィールドが持つ以下のフィールドを組み合わせて設定していました。

- clients[].ctlr_ids
- clients[].ctlr_flts.channel
- clients[].ctlr_flts.dtype
- clients[].ctlr_flts.ids

#### intdash Edge Agent 2 での設定

downstream で設定します。どのエッジのどのデータを受信するかは、filters により指定します。

> **注釈:** filters は、iSCP プロトコルのダウンストリームフィルター仕様に則っています。ダウンストリームフィルターの詳細については、iSCP 2.0 プロトコル仕様書を参照してください。
> また、現在は iSCP 1.0 から iSCP 2.0 への移行期のため、新旧プロトコル間での相互変換が運用されています。iSCP 1.0 と 1SCP 2.0 の間の互換仕様については、intdash API/SDK ドキュメントの「iSCP 1.0 と iSCP 2.0 の互換仕様」を参照してください。

### 30.2.6 デバイスコネクターの設定

#### intdash Edge Agent での設定

loggers フィールドで設定していました。デバイスコネクターとの間の FIFO のパスは、loggers[].connections[].fifo_tx や loggers[].connections[].fifo_rx で設定していました。

#### intdash Edge Agent 2 での設定

device_connectors_upstream または device_connectors_downstream で設定します。デバイスコネクターとの間の FIFO のパスは、ipc.path で指定します。

### 30.2.7 フィルターの設定

#### intdash Edge Agent での設定

manager.filters フィールドで設定していました。

#### intdash Edge Agent 2 での設定

`filters_upstream` または `filters_downstream` で設定します。フィルター機能を使用することで、条件にマッチしたものだけをリアルタイム送信したり、後回収に回したりすることができます。

### 30.2.8 再送・後回収の設定

#### intdash Edge Agent での設定

以下のフィールドを組み合わせて設定していました。

- `manager.required_space`
- `clients[].resend_cycle`

#### intdash Edge Agent 2 での設定

`deferred_upload` で設定します。

# 31 intdash Edge Agent 2 リリースノート

## 31.1 v1.2.1

### 31.1.1 Enhancements

- デーモンが管理する DB のアクセス効率を改善しました

### 31.1.2 Bug Fixes

- 計測の遅延アップロードの途中で正しい総データポイント数をサーバーに送信するようにしました。これにより、遅延アップロードの途中でもサーバー側でデータポイント回収率を正しく計算できるようになります。
- 遅延アップロードの際、サーバーから HTTP ステータスコード 502/503/504 が返った場合、遅延アップロードを停止するのではなくリトライするようにしました
- 計測終了と共にデーモンを終了した場合、次のデーモン起動時までサーバーに計測終了通知が送られなかったのを、計測終了時に通知するようにしました
- ローカルストレージ内の計測の削除を行ったときに他の計測の遅延アップロード処理も停止することがあったのを修正しました

## 31.2 v1.2.0

**警告:** このバージョンから、アップストリーム用の デバイスコネクター IPC (p. 39) にある `data_name_prefix` の扱い方が変わっています。
v1.2.0 未満 のバージョンでは、アップストリーム用のデバイスコネクター IPC 設定にある `data_name_prefix` の末尾がセパレータ文字（/）でない場合、自動的にセパレータ文字（/）が付加されていました。これに対し v1.2.0 以上のバージョンでは、`data_name_prefix` がそのまま使用されます。
そのため、v1.2.0 未満で使用していた設定を v1.2.0 以上で使用する場合、変更が必要な場合があります。

| 設定とデータ | | 結果として付与されるデータ名称 | | v1.2.0 以上へのアップデート時の設定変更 |
|---|---|---|---|---|
| デバイスコネクター IPC の data_name_prefix | FIFO 用データフォーマット内の Data Name フィールド | v1.2.0 未満使用時 | v1.2.0 以上使用時 | |
| v1/1 | any_name | v1/1/any_name | v1/1any_name | **必要** |
| v1/1/ | any_name | v1/1/any_name | v1/1/any_name | 不要 |
| v1/1 | 空文字 | v1/1 | v1/1 | 不要 |
| abc | def | abc/def | abcdef | **必要** |

なお、結果として付与されるデータ名称（`data_name_prefix + Data Name` フィールド）が空文字の場合、データ名称が不正であるためそのデータは intdash サーバーに保存されません。

### 31.2.1 Breaking Changes

- data_name_prefix の扱い方を変更しました

### 31.2.2 Features

- ダウンストリーム用のデバイスコネクター IPC 設定に data_name_prefix を追加しました
- E2E Call API を追加しました

### 31.2.3 Enhancements

- デーモンが管理する DB 内の計測データを定期的に vacuum するようにしました
- intdash サーバーと通信できない場合に計測開始までに時間がかかるのを修正しました
- rename フィルターの change_to.dest_id を必須項目ではなくしました

### 31.2.4 Bug Fixes

- フィルターで src_id を指定した時に設定内容が反映されないのを修正しました
- 未送信の計測を削除すると遅延アップロード処理が行われないことがあるのを修正しました
- デーモンが管理する DB 内の計測データが自動削除される際、順番が新しい順になっていたのを修正しました
- デーモンが管理する DB のパスを変更すると、ディスク空き容量チェックの対象パーティションが正しくない状態になるのを修正しました
- サーバーに送信するデータポイント数が正しくない値になることがあったのを修正しました
- リアルタイム送信で intdash サーバーからエラーが返ったときに未送信データとして遅延アップロードしていなかったのを修正しました

# 32 V4L で取得した画像を JPEG で送信する

付属デバイスコネクター device-connector-intdash を使用してカメラから JPEG 画像を取得し、リアルタイム送信するための設定例です。

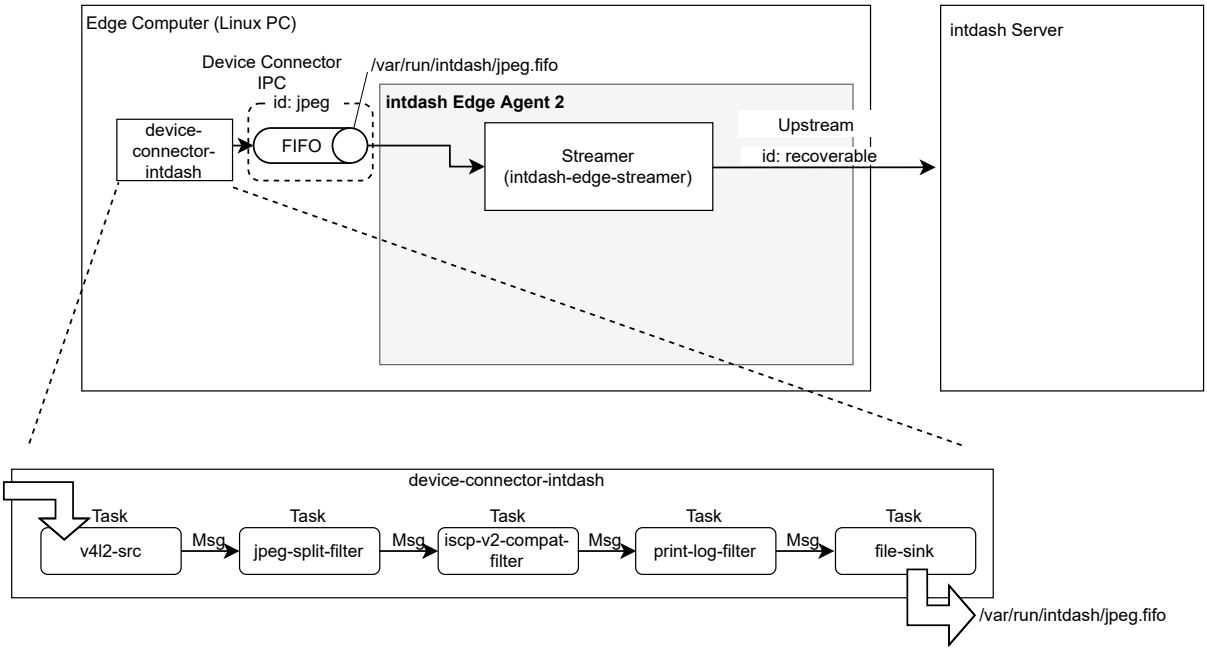



図 28　JPEG 画像を送信するための設定例

## 32.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 32.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: jpeg
    data_name_prefix: v1/101/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/jpeg.fifo
    launch:
```

(次のページに続く)

(前のページからの続き)

```
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/jpeg.yml
      environment:
      - DC_V4L2_SRC_CONF_PATH=/dev/video0
      - DC_V4L2_SRC_WIDTH=320
      - DC_V4L2_SRC_HEIGHT=240
      - DC_V4L2_SRC_FPS=5
      - DC_PRINT_LOG_FILTER_CONF_TAG=jpeg
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/jpeg.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/jpeg.yml (p. 174) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。
- device-connector-intdash から `/var/run/intdash/jpeg.fifo` を介して得られたデータは、`v1/101/` というデータ名称プレフィックスを与えられて、`recoverable` という ID を持つアップストリームに送信されます。

## 32.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

v1 互換データとして送っているため、Edge Finder で確認することができます。

# 33 V4L で取得した画像を H.264 で送信する

付属デバイスコネクター device-connector-intdash を使用して、カメラで取得した画像を GStreamer を使って H.264 に変換し、リアルタイム送信するための設定例です。

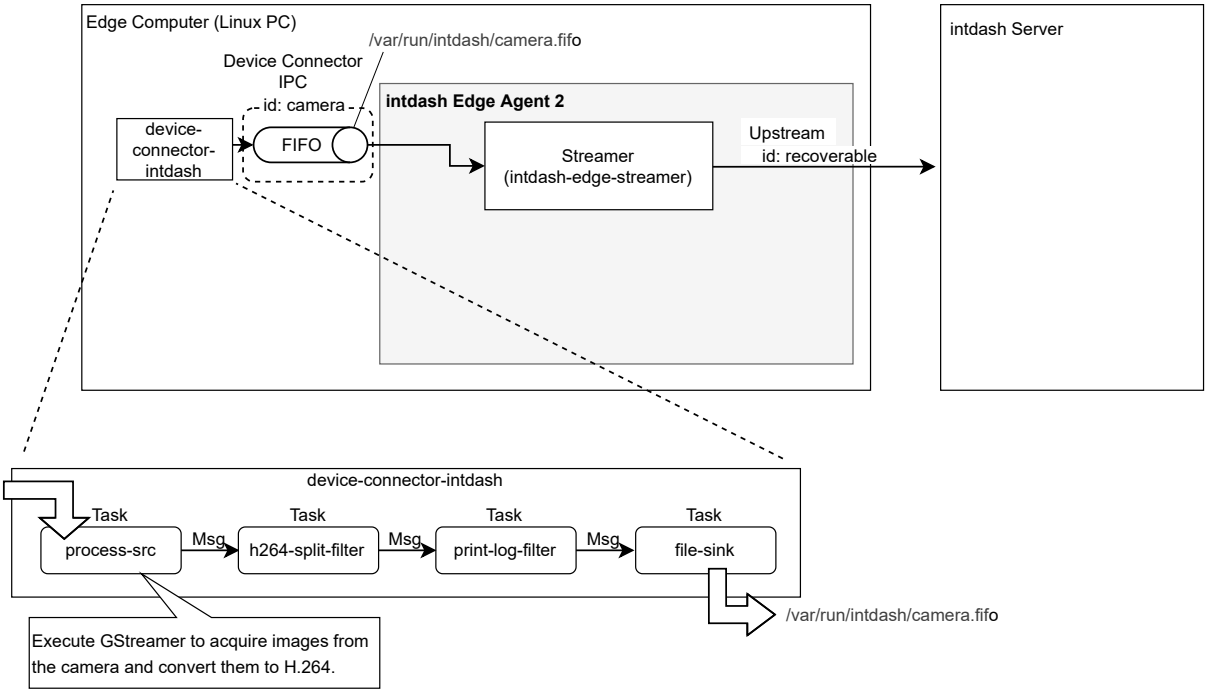



図 29　カメラからのデータを H.264 に変換して送信するための設定例

## 33.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 33.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。

EDGEPLANT T1 など、OpenMAX によるエンコード処理が可能な環境では、以下のコマンドを実行してください（それ以外の環境については下の注釈を参照してください）。

```
$ intdash-agentctl config device-connector up --create '
    id: camera
    data_name_prefix: v1/101/
    format: iscp-v2-compat
    dest_ids:
      - recoverable
    ipc:
```

（次のページに続く）

(前のページからの続き)

```
      type: fifo
      path: /var/run/intdash/camera.fifo
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/gstreamer_h264.yml
      environment:
      - DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! queue ! jpegdec ! omxh264enc control-rate=2 iframeinterval=15␣
↪bitrate=3000000 insert-sps-pps=true insert-vui=true ! video/x-h264,stream-format=byte-stream ! queue !␣
↪h264parse ! queue ! fdsink fd=1"
      - DC_H264_SPLIT_FILTER_CONF_DELAY_MS=100
      - DC_PRINT_LOG_FILTER_CONF_TAG=camera
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/camera.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/gstreamer_h264.yml (p. 171) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。

  `DC_PROCESS_SRC_CONF_COMMAND` は、device-connector-intdash の process-src エレメントで実行されるコマンドです。`/dev/video0` から JPEG 画像を取得し H.264 に変換する GStreamer のコマンドを指定しています。

  > **注釈:** 上の例は OpenMAX によるエンコード処理が可能な環境のための設定です。それ以外の環境では、`DC_PROCESS_SRC_CONF_COMMAND` を以下のように設定してください。
  >
  > **VA-API によるエンコード処理が可能な環境の場合:**
  >
  > ```
  > DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
  > ↪width=1920,height=1080,framerate=15/1 ! queue ! vaapijpegdec ! queue ! vaapipostproc ! queue !␣
  > ↪vaapih264enc rate-control=2 bitrate=3000 max-bframes=0 keyframe-period=15 ! fdsink fd=1"
  > ```
  >
  > **OpenMAX、VA-API とも利用できない環境の場合**（GPU を利用しないため CPU 処理負荷が高くなります）：
  >
  > ```
  > DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
  > ↪width=1920,height=1080,framerate=15/1 ! queue ! jpegdec ! videoconvert ! queue ! x264enc ! fdsink␣
  > ↪fd=1"
  > ```

- device-connector-intdash から `/var/run/intdash/camera.fifo` を介して得られたデータは、`v1/101/` というデータ名称プリフィックスを与えられて、`recoverable` という ID を持つアップストリームに送信されます。

## 33.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

v1 互換データとして送っているため、Edge Finder で確認することができます。

# 34 H.264 動画を NALU ごとに送信する

iSCP で定義されている H.264 NAL Unit データ型を使って、H.264 の動画をリアルタイム送信するための設定例です。

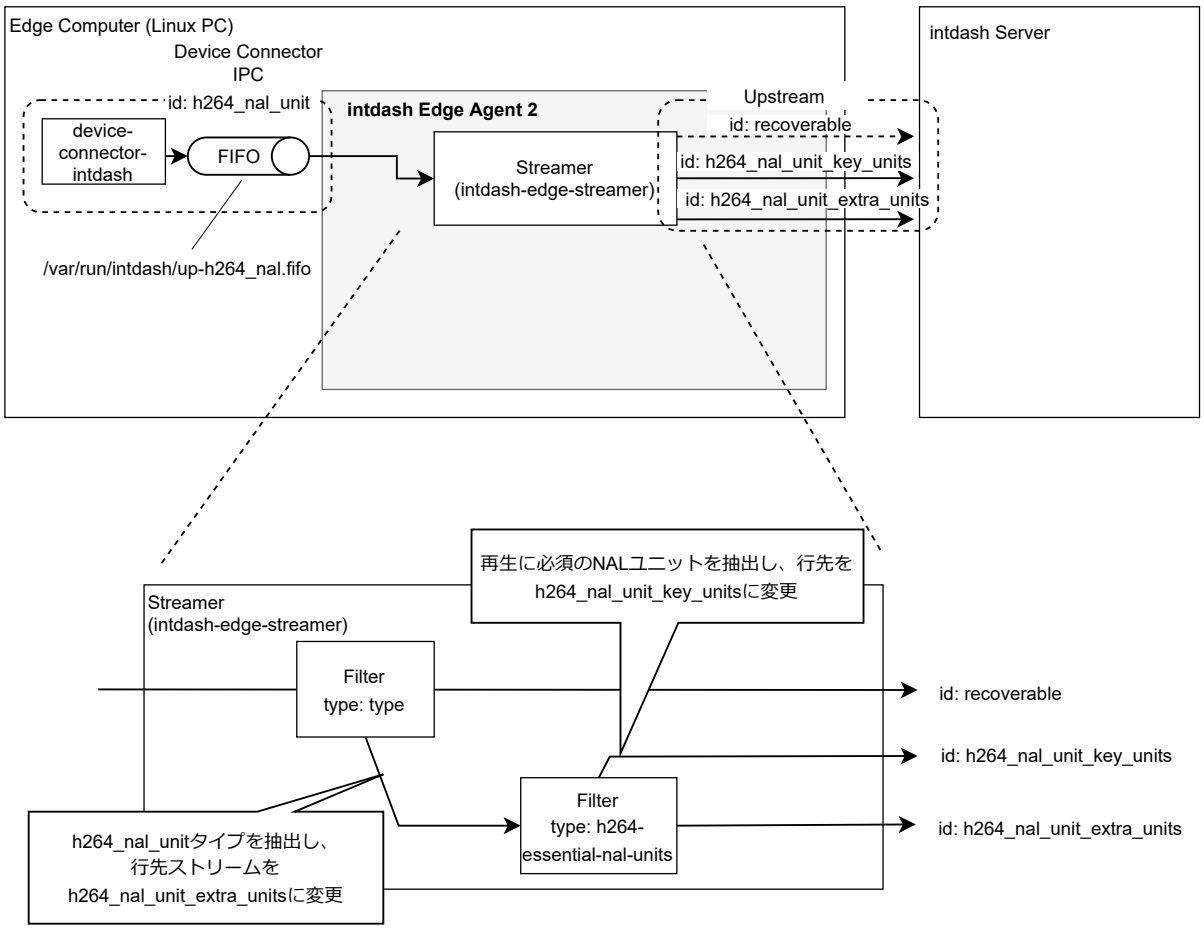



図 30　H.264 NAL Unit データ型を使って、H.264 の動画をリアルタイム送信するための設定例

H.264 NAL Unit データ型を使い適切に設定されたストリームで送信することで、受信側の Visual M2M Data Visualizer では、伝送欠損せず受信できた最低限のデータのみを使った動画の再生が可能になります。データが欠損した個所はブロックノイズになったり、緑色の表示になったりしますが、これにより、送信エッジ側の送信帯域が狭い場合でも、リアルタイム性が高い状態で動画を再生できます。

**重要:**

- この設定例は、アップストリーム h264_nal_unit_extra_units で、信頼性のない接続（`qos: unreliable`）を使用することで効果が発揮されます。そのため、トランスポート (p. 37) として、QUIC を使用する必要があります。QUIC を使用するためにはサーバー側で設定が必要です。
- H.264 NAL Unit データ型は、H.264 の NALUnit 単位でデータポイントを送信するためのものです。Data Visualizer はこの方法で送信されたデータの再生が可能です。Data Visualizer 以外のアプリケーションを使って再生するには、この送信方法に合わせた再生処理を実装する必要があります。

## 34.1 トランスポートの設定

H.264 NAL Unit に適した送信をするために、トランスポートを設定します。

```
$ intdash-agentctl config transport --modify '
    protocol: quic
  '
```

## 34.2 アップストリームの設定

H.264 NAL Unit に適した送信をするために、以下の 2 種類のアップストリームを作成します。

| アップストリームの ID | 概要 |
|---|---|
| h264_nal_unit_key_units | 信頼性のあるアップストリームを使用して NAL Unit 単位でデータを送信します。このストリームを使用して、再生に必須となる SPS/PPS/IDR の NAL Unit を送信します。 |
| h264_nal_unit_extra_units | 信頼性のないアップストリームを使用して NAL Unit 単位でデータを送信します。このストリームを使用して、nonIDR の NAL Unit を送信します。 |

これら 2 つのアップストリームを作成するには、以下のコマンドを実行してください。

```
$ intdash-agentctl config up --create '
    id: h264_nal_unit_key_units
    enabled: true
    recover: true
    persist: true
    qos: partial
    flush_policy: immediately
  '
$ intdash-agentctl config up --create '
    id: h264_nal_unit_extra_units
    enabled: true
    recover: true
    persist: true
    qos: unreliable
    flush_policy: immediately
  '
```

## 34.3 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

> **注意:** `DC_PROCESS_SRC_CONF_COMMAND` では、GStreamer を使って動画を取得し、変換するコマンドを指定します。以下の例では GStreamer において omxh264enc を使用しています。omxh264enc は、OpenMAX 対応 GPU を使って動画エンコーディングを行います。
>
> パイプライン設定ファイルのサンプル /etc/dc_conf/gstreamer_h264_nalunit.yml (p. 171) 内の例も参考にして、使用するハードウェアに合ったコマンドを指定してください。

```
$ intdash-agentctl config device-connector up --create '
    id: h264_nal_unit
    data_name_prefix: 101/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-h264_nal.fifo
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/gstreamer_h264_nalunit.yml
      environment:
      - DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! jpegdec ! omxh264enc control-rate=2 iframeinterval=15␣
↪bitrate=3000000 bit-packetization=true slice-header-spacing=1200 insert-sps-pps=true insert-vui=true !␣
↪video/x-h264,stream-format=byte-stream ! queue ! h264parse ! queue ! fdsink fd=1"
      - DC_H264_NALUNIT_SPLIT_FILTER_CONF_DELAY_MS=100
      - DC_PRINT_LOG_FILTER_CONF_TAG=video0
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/up-h264_nal.fifo
'
```

- launch で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/gstreamer_h264_nalunit.yml (p. 171) を指定しています。また、パイプライン設定で使用する環境変数を environment で与えています。
- device-connector-intdash から /var/run/intdash/up-h264_nal.fifo を介して得られたデータは、101/ というデータ名称プレフィックスを与えられて、recoverable という ID を持つアップストリームに送信されます。

**注釈:** h264_nal_unit は iSCP v1 には存在しないデータ型であるため、Data Visualizer で表示できるようにするために、特別な data_name_prefix を設定しています。

## 34.4 フィルターの設定

h264_nal_unit データ型のデータポイントの行先を、さきほど作成した別のアップストリーム h264_nal_unit_extra_units に変更するフィルターを設定します。

```
$ intdash-agentctl config filter up --create '
    id: h264_nal_unit_filter
    enabled: true
    type: type
    target:
      type: h264_nal_unit
      dest_ids:
        - recoverable
    change_to:
      dest_id: h264_nal_unit_extra_units
'
```

次に、再生に必須であるデータポイントの行先を、さらに別のアップストリーム h264_nal_unit_key_units に変更するフィルターを設定します。

```
$ intdash-agentctl config filter up --create '
    id: h264_nal_unit_key_filter
    enabled: true
    type: h264-essential-nal-units
    target:
      dest_ids:
        - h264_nal_unit_extra_units
    change_to:
      dest_id: h264_nal_unit_key_units
'
```

**注釈:** `h264-essential-nal-units` タイプのフィルターは、再生に必須の NAL ユニット（SPS/PPS/IDR）を抽出するために存在する特別なフィルターです。

## 34.5 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

**注釈:** 送信帯域が狭くなった場合にどうなるかを実際に試す場合には、Linux の tc コマンドを使用するのが便利です。NIC eth0 の送信帯域を 2700kbit/sec に制限する例:

```
$ tc qdisc add dev eth0 handle 10: root tbf rate 2700kbit burst 10kb limit 10kb
```

上のコマンドで設定した帯域制限を解除するには以下を実行します:

```
$ tc qdisc del dev eth0 root
```

# 35 EDGEPLANT ANALOG-USB I/F からデータを取得する

付属デバイスコネクター device-connector-intdash を使用して EDGEPLANT ANALOG-USB Interface からアナログデータを取得し、リアルタイム送信するための設定例です。

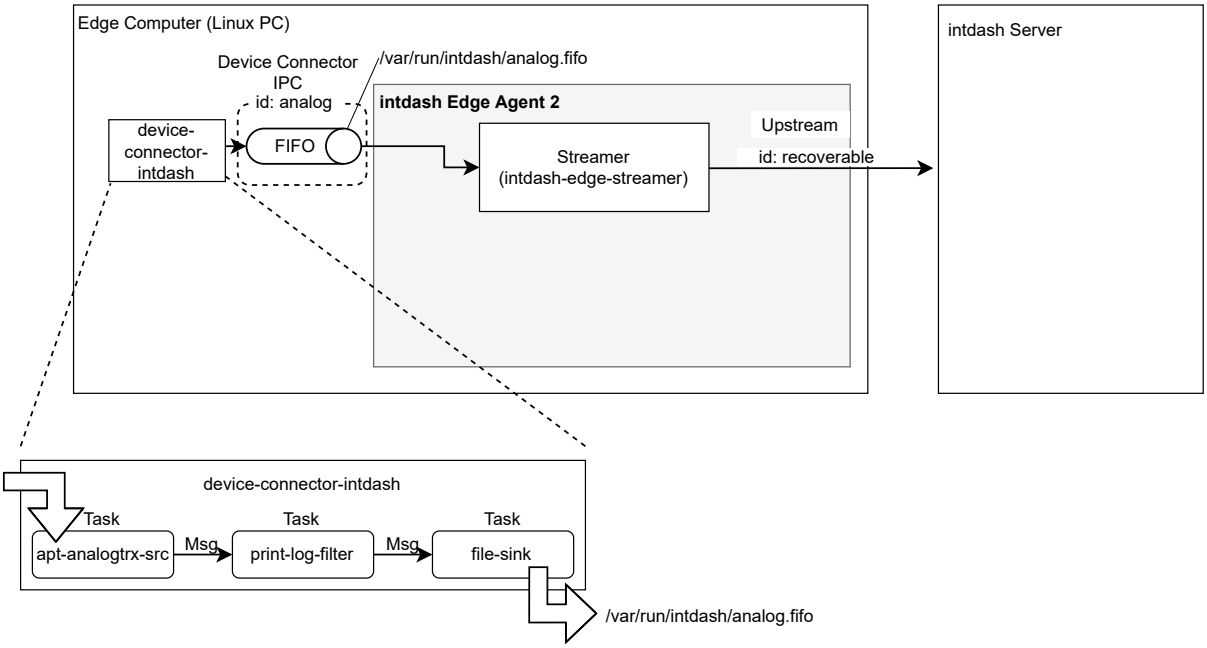



図 31　EDGEPLANT ANALOG-USB Interface からのデータを送信するための設定例

## 35.1 ストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 35.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: analog
    data_name_prefix: v1/101/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/analog.fifo
```

(次のページに続く)

(前のページからの続き)

```
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/apt_analog.yml
      environment:
      - DC_APT_ANALOGTRX_SRC_CONF_PATH=/dev/apt-usb/by-id/usb-xxx
      - DC_APT_ANALOGTRX_SRC_CONF_TIMESTAMP_MODE=device
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_SEND_RATE=1250000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_0=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_1=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_2=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_3=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_4=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_5=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_6=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_7=true
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_0=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_1=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_2=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_3=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_4=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_5=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_6=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_7=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_8=-5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_0=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_1=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_2=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_3=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_4=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_5=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_6=5000
      - DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_7=5000
      - DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_ENABLED=false
      - DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_WAVEFORM_TYPE=0
      - DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_VOLTAGE=20
      - DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_FREQUENCY=1000
      - DC_PRINT_LOG_FILTER_CONF_TAG=analog
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/analog.fifo
,
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/apt_analog.yml (p. 166) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。
- device-connector-intdash から `/var/run/intdash/analog.fifo` を介して得られたデータは、`v1/101/` というデータ名称プレフィックスを与えられて、`recoverable` という ID を持つストリームに送信されます。

## 35.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 36 EDGEPLANT CAN-USB I/F からデータを取得する

付属デバイスコネクター device-connector-intdash を使用して EDGEPLANT CAN-USB Interface から CAN データを取得し、リアルタイム送信するための設定例です。

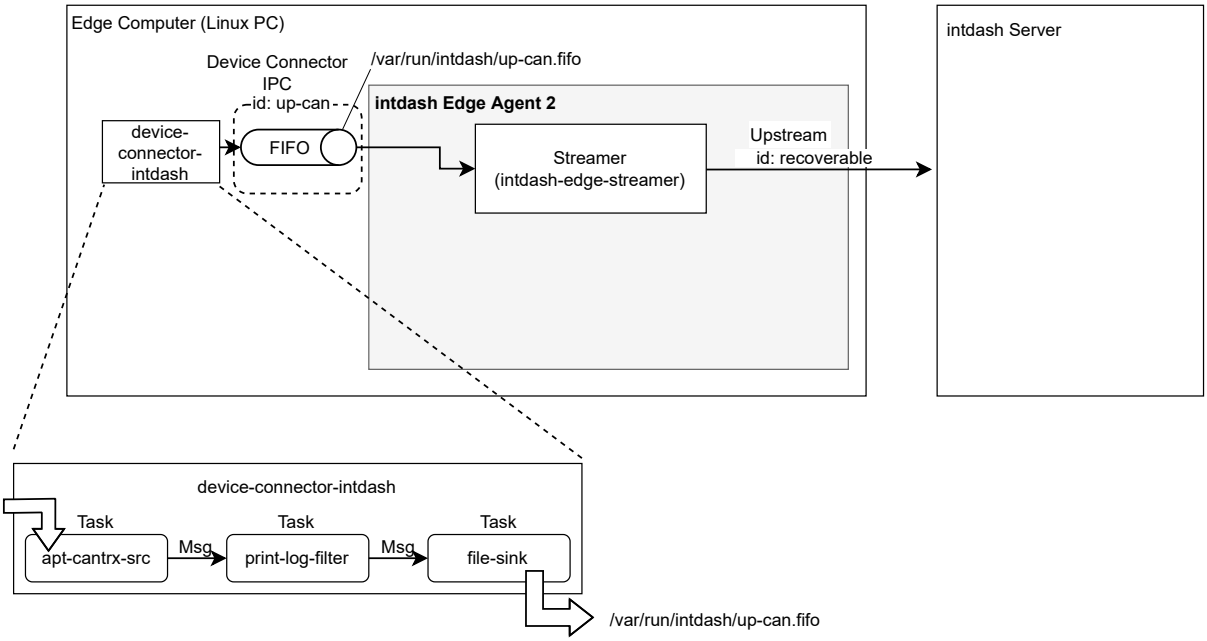



図 32　EDGEPLANT CAN-USB Interface からのデータを送信するための設定例

## 36.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 36.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: up-can
    data_name_prefix: can/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-can.fifo
```

(次のページに続く)

(前のページからの続き)

```
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/apt_cantrx-up.yml
      environment:
      - DC_APT_CANTRX_SRC_CONF_PATH=/dev/apt-usb/by-id/usb-xxx
      - DC_APT_CANTRX_SRC_CONF_LISTENONLY=1
      - DC_APT_CANTRX_SRC_CONF_BAUDRATE=500
      - DC_APT_CANTRX_SRC_CONF_TIMESTAMP_MODE=device
      - DC_PRINT_LOG_FILTER_CONF_TAG=up-can
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/up-can.fifo
'
```

- launch で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash を実行するための設定は、パイプライン設定ファイル /etc/dc_conf/apt_cantrx-up.yml として与えています（このファイルは次の手順で作成します）。また、パイプライン設定ファイルで使用するための環境変数を与えています（ environment ）。
- device-connector-intdash から /var/run/intdash/up-can.fifo を介して得られたデータは、can というデータ名称プレフィックスを与えられて、recoverable という ID を持つアップストリームに送信されます。

## 36.3 device-connector-intdash のパイプライン設定

device-connector-intdash 用のパイプライン設定ファイルを以下の内容で作成し、/etc/dc_conf/apt_cantrx-up.yml として保存します。

```
before_task:
  # sync timestamp
  - mkdir -p /var/lock/intdash
  - |
    BASETIME_CLOCK_ID=$DC_CLOCK_ID
    meas-hook --lockfile /var/lock/intdash/dc_apt_usbtrx.lock --command "
      if command -v apt_usbtrx_timesync.sh     > /dev/null 2>&1 ; then apt_usbtrx_timesync.sh;     exit 0; fi;
      if command -v apt_usbtrx_timesync_all.sh > /dev/null 2>&1 ; then apt_usbtrx_timesync_all.sh; exit 0; fi;
      echo \"ERROR: timestamp script not found\";
      exit 1;
    "

after_task:
  - rm -f /var/lock/intdash/dc_apt_usbtrx.lock

tasks:
  - id: 1
    element: apt-cantrx-src
    conf:
      clock_id: CLOCK_MONOTONIC
      path: $(DC_APT_CANTRX_SRC_CONF_PATH)
      listenonly: $(DC_APT_CANTRX_SRC_CONF_LISTENONLY)
      baudrate: $(DC_APT_CANTRX_SRC_CONF_BAUDRATE)
      timestamp_mode: $(DC_APT_CANTRX_SRC_CONF_TIMESTAMP_MODE)

  - id: 2
    element: print-log-filter
    from: [[1]]
```

(次のページに続く)

(前のページからの続き)

```
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 3
    element: file-sink
    from: [[2]]
    conf:
      flush_size: 10
      path: $(DC_FILE_SINK_CONF_PATH)
```

- before_task では、EDGEPLANT CAN-USB Interface の同期処理の初期化を行っています。複数の device-connector-intdash を使用する場合に対応するためにプロセス間排他を用いています。aftre_script でその後片付けをしています。( before_task では、定義されたタスクが開始される前に実行されるコマンドを、after_task では、タスク終了後に実行されるコマンドを指定します。)
- apt-cantrx-src エレメントでは、環境変数として与えられた値を使って EDGEPLANT CAN-USB Interface からデータを取得します。(参考: apt-analogtrx-src (p. 147) )
- print-log-filter では標準エラー出力にログを出力します。(参考: print-log-filter (p. 193) )
- file-sink では、$(DC_FILE_SINK_CONF_PATH) に FIFO 用データフォーマットのデータが書き出されます。これを intdash Edge Agent 2 が読み取ります。(参考: file-sink (p. 196) )

## 36.4 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 37 EDGEPLANT CAN-USB I/F へデータを出力する

リアルタイムでダウンストリームした CAN データを、device-connector-intdash を使用して EDGEPLANT CAN-USB Interface に書き込むための設定例です。

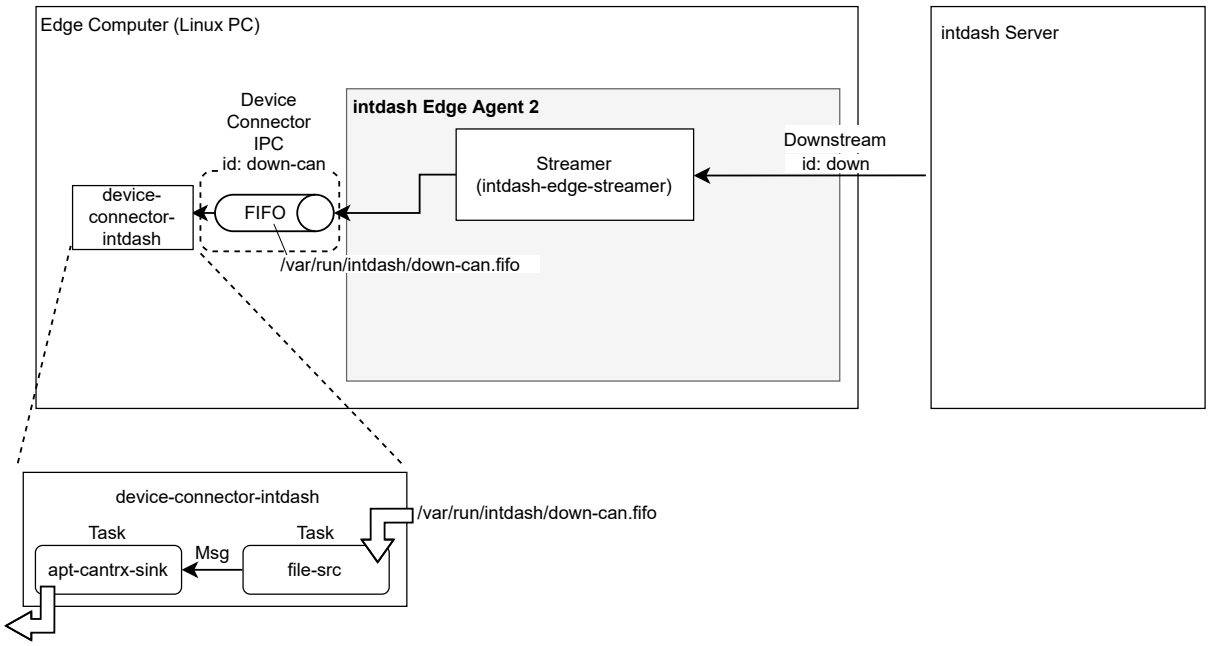



図 33　ダウンストリームした CAN データを EDGEPLANT CAN-USB Interface に書き込むための設定例

## 37.1 ダウンストリームの設定

以下のコマンドを実行して、down という ID を持つダウンストリームを作成します。

送信元が 03ace3b1-d208-4fc3-9763-a502923ca6ab で、データ名称が can のデータのみを受信するようダウンストリームフィルターが設定してあります。

```
$ intdash-agentctl config downstream --create '
    id: down
    enabled: true
    dest_ids:
     - down-can
    filters:
     - src_edge_uuid: 03ace3b1-d208-4fc3-9763-a502923ca6ab
       data_filters:
       - type: can_frame
         name: can
  '
```

サーバーからダウンストリームされたデータは、デバイスコネクター IPC down-hello に送られます。デバイスコネクター IPC down-hello はこのあとの手順で設定します。

## 37.2 デバイスコネクター IPC の設定

デバイスコネクターにデータを渡すためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector down --create '
    id: down-can
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/down-can.fifo
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/apt_cantrx-down.yml
      environment:
      - DC_APT_CANTRX_SINK_CONF_PATH=/dev/apt-usb/by-id/usb-xxx
      - DC_APT_CANTRX_SINK_CONF_LISTENONLY=1
      - DC_APT_CANTRX_SINK_CONF_BAUDRATE=500
      - DC_APT_CANTRX_SINK_CONF_TIMESTAMP_MODE=device
      - DC_FILE_SRC_CONF_PATH=/var/run/intdash/down-can.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash を実行するための設定は、パイプライン設定ファイル `/etc/dc_conf/apt_cantrx-down.yml` として与えています（このファイルは次の手順で作成します）。また、パイプライン設定ファイルで使用するための環境変数を与えています（ `environment` ）。
- ダウンストリームから `/var/run/intdash/down-can.fifo` を介してデバイスコネクターに送られます。

## 37.3 device-connector-intdash のパイプライン設定

device-connector-intdash 用のパイプライン設定ファイルを以下の内容で作成し、`/etc/dc_conf/apt_cantrx-down.yml` として保存します。

```
before_task:
  # sync timestamp
  - mkdir -p /var/lock/intdash
  - |
    BASETIME_CLOCK_ID=$DC_CLOCK_ID
    meas-hook --lockfile /var/lock/intdash/dc_apt_usbtrx.lock --command "
      if command -v apt_usbtrx_timesync.sh     > /dev/null 2>&1 ; then apt_usbtrx_timesync.sh;     exit 0; fi;
      if command -v apt_usbtrx_timesync_all.sh > /dev/null 2>&1 ; then apt_usbtrx_timesync_all.sh; exit 0; fi;
      echo \"ERROR: timestamp script not found\";
      exit 1;
    "

after_task:
  - rm -f /var/lock/intdash/dc_apt_usbtrx.lock

tasks:
  - id: 10
    element: file-src
    conf:
      path: $(DC_FILE_SRC_CONF_PATH)

  - id: 11
```

(次のページに続く)

(前のページからの続き)

```
    element: apt-cantrx-sink
    from: [[10]]
    conf:
      clock_id: CLOCK_MONOTONIC
      path: $(DC_APT_CANTRX_SINK_CONF_PATH)
      listenonly: $(DC_APT_CANTRX_SINK_CONF_LISTENONLY)
      baudrate: $(DC_APT_CANTRX_SINK_CONF_BAUDRATE)
      timestamp_mode: $(DC_APT_CANTRX_SINK_CONF_TIMESTAMP_MODE)
```

- file-src では、intdash Edge Agent 2 によって書き出されたデータを$(DC_FILE_SINK_CONF_PATH)から読みます。(参考: file-src (p. 190))
- apt-cantrx-sink エレメントでは、環境変数として与えられた値を使って EDGEPLANT CAN-USB Interface にデータを渡します。(参考: apt-cantrx-src (p. 149))

## 37.4 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 38 u-blox GNSS モジュールから UBX メッセージを取得する

付属デバイスコネクター device-connector-intdash を使用して UBX プロトコルのメッセージを取得し、リアルタイム送信するための設定例です。

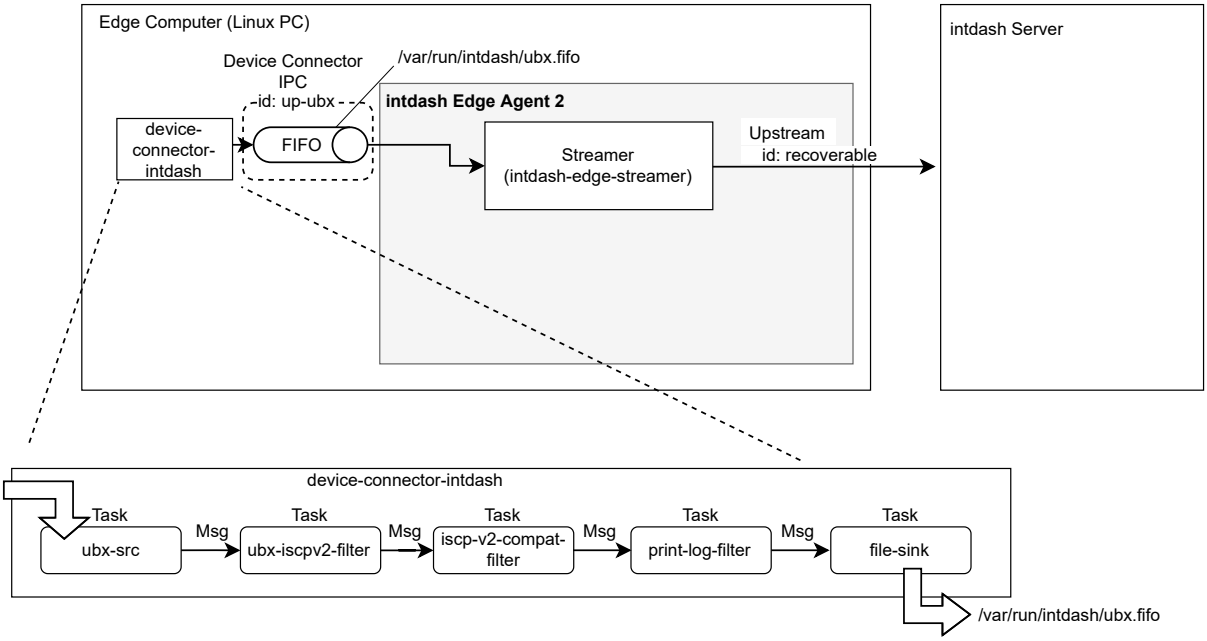



図 34　UBX プロトコルのメッセージを送信するための設定例

## 38.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 38.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: up-ubx
    data_name_prefix: ubx/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/up-ubx.fifo
    launch:
```

(次のページに続く)

(前のページからの続き)

```
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/ubx.yml
      environment:
      - DC_UBX_SRC_CONF_PATH=/dev/ttyTHS1
      - DC_UBX_SRC_CONF_BAUD_RATE=57600
      - DC_UBX_SRC_CONF_MEAS_RATE_MS=200
      - DC_UBX_SRC_CONF_NAV_RATE=1
      - DC_UBX_SRC_CONF_HIGH_NAV_RATE_HZ=5
      - DC_UBX_SRC_CONF_ESF_STATUS_RATE=1
      - DC_UBX_SRC_CONF_HNR_ATT_RATE=1
      - DC_UBX_SRC_CONF_HNR_INS_RATE=1
      - DC_UBX_SRC_CONF_HNR_PVT_RATE=1
      - DC_UBX_SRC_CONF_NAV_STATUS_RATE=1
      - DC_PRINT_LOG_FILTER_CONF_TAG=ubx
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/up-ubx.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/ubx.yml (p. 177) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。
- 特に、`DC_UBX_SRC_CONF_PATH` は u-blox GNSS モジュールのデバイスパス、`DC_UBX_SRC_CONF_BAUD_RATE` は使用されるボーレートです。使用するデバイスに合った値を設定してください。
- device-connector-intdash から `/var/run/intdash/up-ubx.fifo` を介して得られたデータは、`ubx` というデータ名称プレフィックスを与えられて、`recoverable` という ID を持つアップストリームに送信されます。

## 38.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 39 NMEA データを取得する

付属デバイスコネクター device-connector-intdash を使用して NMEA データを取得し、リアルタイム送信するための設定例です。

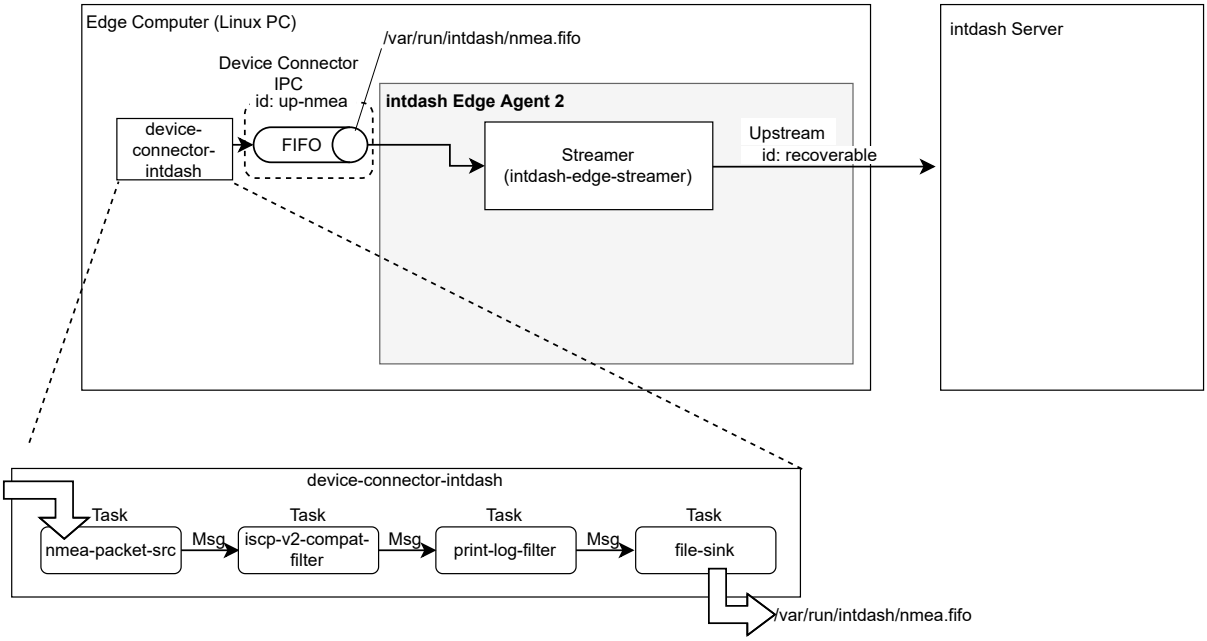



図 35　NMEA データを送信するための設定例

## 39.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 39.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: up-nmea
    data_name_prefix: nmea/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/nmea.fifo
    launch:
```

(次のページに続く)

(前のページからの続き)

```
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/nmea.yml
      environment:
      - DC_NMEA_PACKET_SRC_CONF_PATH=/dev/ttyTHS1
      - DC_NMEA_PACKET_SRC_CONF_BAUDRATE=57600
      - DC_PRINT_LOG_FILTER_CONF_TAG=nmea
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/nmea.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/nmea.yml (p. 175) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。
- 特に、`DC_NMEA_PACKET_SRC_CONF_PATH` は GPS デバイスのデバイスパス、`DC_NMEA_PACKET_SRC_CONF_BAUDRATE` は使用されるボーレートです。使用するデバイスに合った値を設定してください。
- device-connector-intdash から `/var/run/intdash/nmea.fifo` を介して得られたデータは、`nmea` というデータ名称プレフィックスを与えられて、`recoverable` という ID を持つアップストリームに送信されます。

**注釈:** パイプラインのエレメント nmea-packet-src は GPS デバイスの初期化は行いません。詳細については、nmea-packet-src (p. 150) の注釈を参照してください。

## 39.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 40 EDGEPLANT T1 で音声データを取得する

EDGEPLANT T1 のオンボードの音声ジャックに接続されたマイクから、付属デバイスコネクター device-connector-intdash を使用して音声データを取得し、リアルタイム送信するための設定例です。

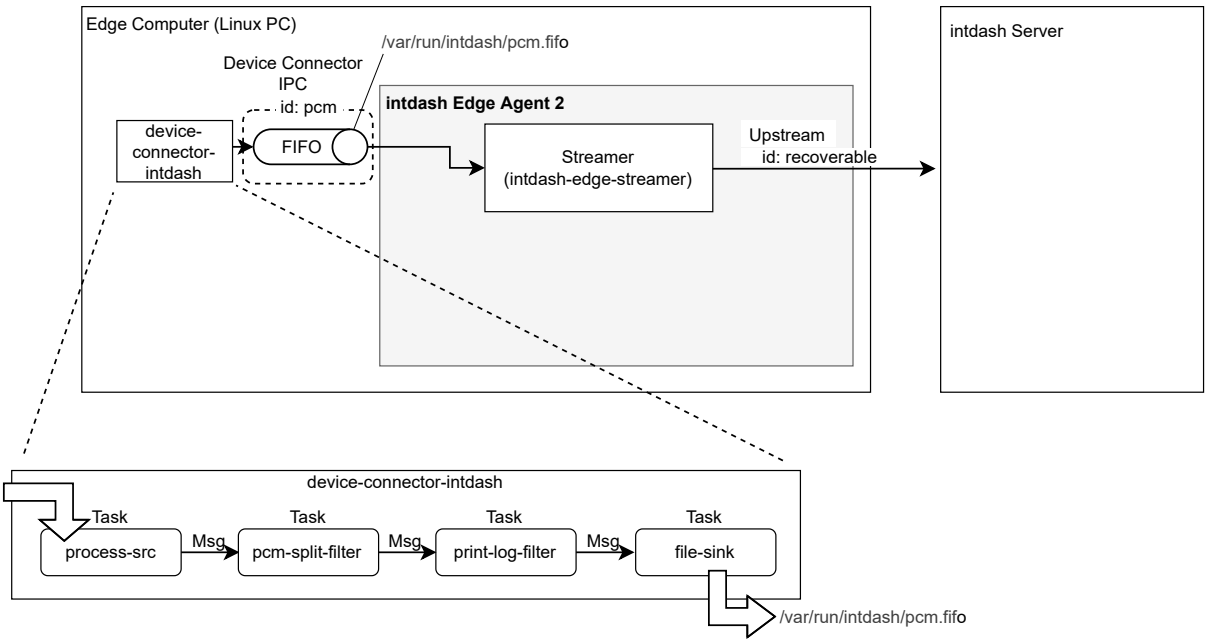



図 36　音声データを送信するための設定例

## 40.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 40.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

> **注意:** DC_PROCESS_SRC_CONF_COMMAND では、GStreamer を使って音声を取得し、変換するコマンドを指定します。パイプライン設定ファイルのサンプル /etc/dc_conf/gstreamer_pcm.yml (p. 173) 内の例も参考にして、使用するハードウェアに合ったものを指定してください。

```
$ intdash-agentctl config device-connector up --create '
    id: pcm
    data_name_prefix: v1/150/
    format: iscp-v2-compat
```

(次のページに続く)

(前のページからの続き)

```
    dest_ids:
      - recoverable
    ipc:
      type: fifo
      path: /var/run/intdash/pcm.fifo
    launch:
      cmd: device-connector-intdash
      args:
      - --config
      - /etc/dc_conf/gstreamer_pcm.yml
      environment:
      - DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q alsasrc device=hw:1 ! audioconvert ! audio/x-raw,
↪format=S32LE,rate=48000,channels=1 ! fdsink fd=1"
      - DC_PCM_SPLIT_FILTER_CONF_DELAY_MS=0
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_ELEMENT="y Jack-state"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_IFACE="mixer"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_FORMAT="S32LE"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_RATE=48000
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_CHANNELS=1
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_IFACE="mixer"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_ELEMENT="y Mic Capture Volume"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_VALUE=20
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_BOOST_ELEMENT="y Mic Boost Capture Volume"
      - DC_PCM_SPLIT_FILTER_CONF_AUDIO_BOOST_VALUE=1
      - DC_PCM_SPLIT_FILTER_CONF_PATH=/dev/snd/by-path/platform-sound
      - DC_PRINT_LOG_FILTER_CONF_TAG=pcm
      - DC_FILE_SINK_CONF_PATH=/var/run/intdash/pcm.fifo
'
```

- `launch` で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/gstreamer_pcm.yml (p. 173) を指定しています。また、パイプライン設定で使用する環境変数を `environment` で与えています。
- device-connector-intdash から `/var/run/intdash/pcm.fifo` を介して得られたデータは、`v1/150/` というデータ名称プレフィックスを与えられて、`recoverable` という ID を持つアップストリームに送信されます。

## 40.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

# 41　ステータスを送信する

付属デバイスコネクター device-connector-intdash を使用してエッジデバイスのステータス情報（CPU、メモリ、ストレージ、ネットワークなど）を取得し、JSON 文字列で送信するための設定例です。

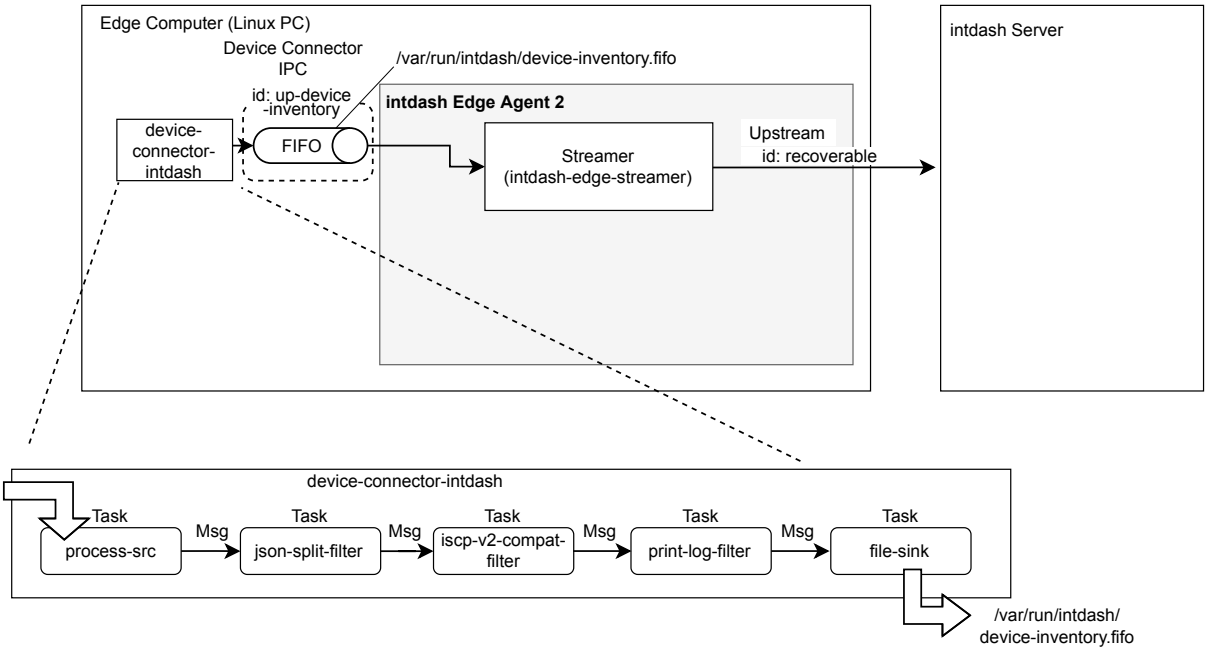



図 37　エッジデバイスのステータス情報を送信するための設定例

## 41.1 アップストリームの設定

以下のコマンドを実行して、recoverable という ID を持つアップストリームを作成します。指定しているのは ID のみのため、他の設定値はデフォルトのとおりになります。

```
$ intdash-agentctl config up --create '
    id: recoverable
  '
```

## 41.2 デバイスコネクター IPC の設定

デバイスコネクターからのデータを受け取るためにデバイスコネクター IPC を追加します。以下のコマンドを実行してください。

```
$ intdash-agentctl config device-connector up --create '
    id: up-device-inventory
    data_name_prefix: v1/255/
    dest_ids:
      - recoverable
    format: iscp-v2-compat
    ipc:
      type: fifo
      path: /var/run/intdash/device-inventory.fifo
    launch:
```

（次のページに続く）

(前のページからの続き)

```
    cmd: device-connector-intdash
    args:
    - --config
    - /etc/dc_conf/device_inventory.yml
    environment:
    - DC_PROCESS_SRC_CONF_COMMAND_INTERVAL=30
    - DC_PRINT_LOG_FILTER_CONF_TAG=device-inventory
    - DC_FILE_SINK_CONF_PATH=/var/run/intdash/device-inventory.fifo
,
```

- launch で、device-connector-intdash を起動するように設定しています。
- device-connector-intdash のパイプライン設定として、付属の /etc/dc_conf/device_inventory.yml (p. 169) を指定しています。また、パイプライン設定で使用する環境変数を environment で与えています。
- DC_PROCESS_SRC_CONF_COMMAND_INTERVAL はステータスの送信間隔（秒）です。30 秒以上で指定することを推奨します。
- device-connector-intdash から /var/run/intdash/device-inventory.fifo を介して得られたデータは、v1/255/ というデータ名称プレフィックスを与えられて、recoverable という ID を持つアップストリームに送信されます。

## 41.3 ストリーマーの起動

以上の設定ができたらストリーマーを起動します。

```
$ intdash-agentctl run
```

**警告:** このデバイスコネクターでは、df や mmcli などのコマンドによりステータス情報が取得され、送信されます。ただし、以下に該当する古いバージョンのディストリビューションでは、いくつかのコマンドのデータが送信されません。

| ディストリビューション | 情報を送信できないコマンド |
|---|---|
| Debian:9 | mmcli、ip |
| Debian:10 | mmcli |
| Ubuntu:18.04 | mmcli |

**警告:** 以下条件に該当する場合、送信する情報が多くなりデータの取得に時間がかかることがあります。

| 条件 | 例 |
|---|---|
| ip コマンドで送信するネットワークインターフェースが多い場合 | Docker を利用している |
| lsusb コマンドで送信するデバイス情報が多い場合 | 多数の USB デバイスが接続されている |
| df コマンドで送信するストレージ情報が多い場合 | 多数のストレージが接続されている |

**注釈:** Docker 環境で利用する場合、以下の点に注意してください。

- mmcli の情報を取得するために `/var/run/dbus` のバインドマウントが必要です。
- df のマウントポイント `/` の情報は、Docker のストレージドライバの情報となります。通常はホストの `/var/lib/docker/overlay2` が存在するパーティションの情報となります。
- ip で取得できる情報はコンテナからアクセスできるネットワークインターフェースの情報のみとなります。ホストの情報も取得したい場合は、コンテナ実行時に `--net=host` を指定してください。

# 42 付属デバイスコネクター概要

intdash Edge Agent 2 には、device-connector-intdash というデバイスコネクターが付属しています。

device-connector-intdash を使用すると、アプトポッド製 EDGEPLANT CAN-USB Interface や、EDGEPLANT ANALOG-USB Interface を intdash Edge Agent 2 に接続することができます。

また device-connector-intdash では、さまざまなデータを FIFO 用フォーマット (iscp-v2-compat (p. 100)) に変換することができます。

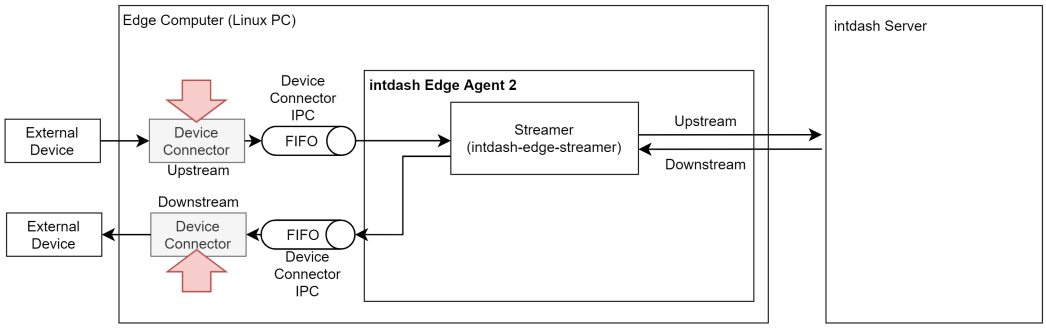



図 38　デバイスコネクター

### パイプラインとエレメント

付属デバイスコネクター device-connector-intdash は Device Connector Framework (p. 182) を使って開発されており、パイプラインアーキテクチャーを採用しています。

device-connector-intdash 内で行われる一連の処理は、パイプラインとして設定します。パイプラインを構成する個別の処理（データを取得する処理、加工する処理、書き出す処理）は、それぞれエレメントという部品を使って表現されます。適切なエレメントをつないでパイプラインを設定することにより、さまざまな処理を行うことができます。

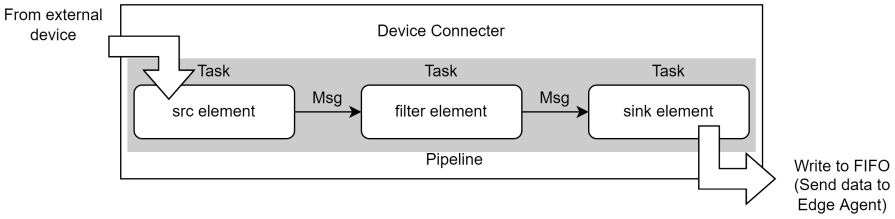



図 39　デバイスから intdash Edge Agent 2 へ入力するパイプライン

device-connector-intdash には 多くのエレメントが付属 (p. 147) していますので、これらを組み合わせることで多様な処理が可能です。また、Device Connector Framework を使って開発されているため、フレームワークに従って必要なエレメントを独自に追加することも可能です。

- パイプライン設定ファイルの書き方については、Device Connector Framework のドキュメントの パイプライン設定ファイル (p. 187) を参照してください。
- device-connector-intdash には、パイプライン設定ファイルのサンプルが付属しています。パイプライン設定サンプル一覧 (p. 166) を参照してください。
- 独自エレメントの開発については、独自エレメントの開発 (p. 198) を参照してください。

# 43 インストール

**注意:** device-connector-intdash は、intdash Edge Agent 2 をインストールした際に依存パッケージとしてインストールされています。インストールされていることを確認するには、以下のコマンドを実行してください。

```
device-connector-intdash --version
```

device-connector-intdash がインストール済みの場合は、本インストール手順は不要です。

device-connector-intdash を正しくインストールするには以下の要件を満たす必要があります。

**ディストリビューションとアーキテクチャー**

| ディストリビューション | バージョン | アーキテクチャー |
|---|---|---|
| Ubuntu | 22.04(LTS), 20.04(LTS), 18.04(LTS) | x86_64 (or amd64), armhf, arm64 |
| Debian | 11, 10, 9 | x86_64 (or amd64), armhf, arm64 |

device-connector-intdash は、アプトポッドの公開リポジトリで「device-connector-intdash」パッケージとして提供されています。device-connector-intdash をインストールするには、インストール先エッジデバイスのターミナルで以下のコマンドを実行します。

1. アプトポッドの公開リポジトリの設定を行います。

   コマンド内の `${DISTRIBUTION}` には、ご使用の環境に応じて、`ubuntu` または `debian` を指定してください。

   ```
   $ sudo apt-get update
   $ sudo apt-get install -y \
       apt-transport-https \
       ca-certificates \
       curl \
       gnupg-agent \
       lsb-release
   $ sudo mkdir -p /etc/apt/keyrings
   $ curl -fsSL https://repository.aptpod.jp/intdash-edge/linux/${DISTRIBUTION}/gpg | \
       sudo gpg --dearmor -o /etc/apt/keyrings/intdash-edge.gpg
   $ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/intdash-edge.gpg] \
       https://repository.aptpod.jp/intdash-edge/linux/${DISTRIBUTION} \
       $(lsb_release -cs) \
       stable" \
       | sudo tee /etc/apt/sources.list.d/intdash-edge.list
   $ sudo apt-get update
   ```

2. device-connector-intdash をインストールします。

   ```
   $ sudo apt-get install -y device-connector-intdash
   ```

**注釈:** 上記のインストール手順では、推奨される依存パッケージも含めてインストールされます。プラグイン開発を目的に、必要最低限の機能のみインストールしたい場合は、`apt-get install` コマンドに `--no-install-recommends` オプションを付けて実行してください。

```
$ sudo apt-get install -y --no-install-recommends device-connector-intdash
```

推奨パッケージがインストールされていない場合、以下のエレメントを利用できません。これらのエレメントをパイプラインで利用したい場合は、`--no-install-recommends` を指定せずに device-connector-intdash をインストールしてください。

- apt-analogtrx-src
- apt-cantrx-src
- h264-split-filter
- pcm-split-filter
- apt-cantrx-sink

**注釈:** 利用可能な device-connector-intdash のバージョンは、アプトポッドの公開リポジトリの設定後に以下のコマンドで確認が可能です。
表示例：

```
$ apt policy device-connector-intdash
  device-connector-intdash:
  Installed: 2.0.0
  Candidate: 2.0.0
  Version table:
  *** 2.0.0 500
        500 https://repository.aptpod.jp/intdash-edge/linux/ubuntu focal/stable amd64 Packages
        100 /var/lib/dpkg/status
```

# 44 チュートリアル

このチュートリアルでは、付属デバイスコネクター device-connector-intdash からデータを取得し、サーバーに送信します。

**注釈:** 付属デバイスコネクターは、intdash Edge Agent 2 をインストールした際に依存パッケージとしてインストールされています。インストールされていることを確認するには、以下のコマンドを実行してください。

```
device-connector-intdash --version
```

もし device-connector-intdash が見つからない場合は、intdash-edge-agent パッケージを再インストールしてください。

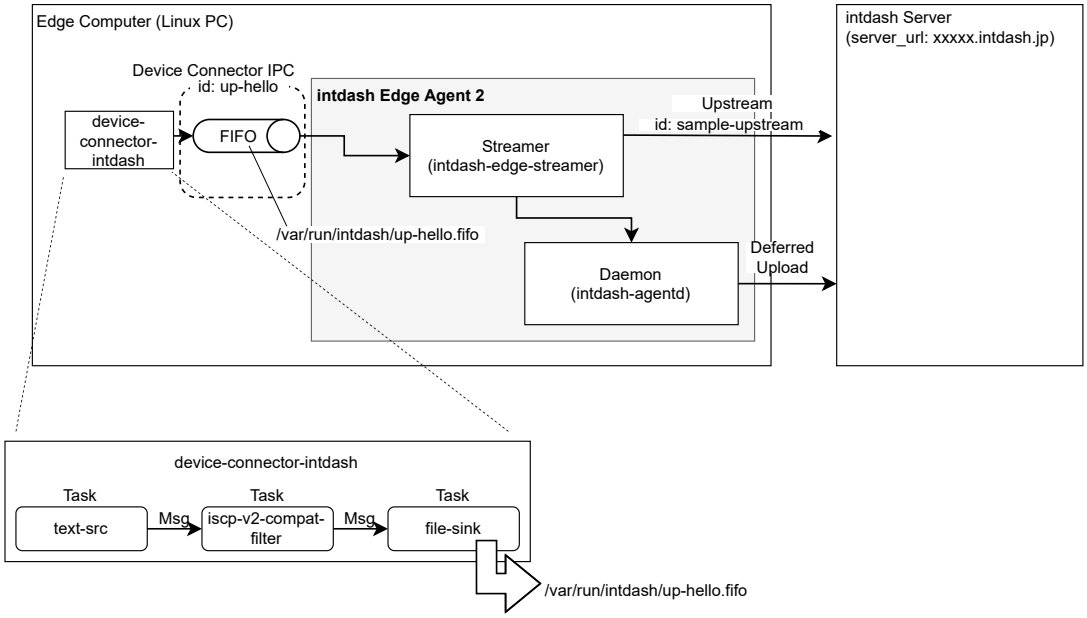



図 40　チュートリアル概要（デバイスコネクター以外はチュートリアル 1 と同じ）

## 44.1 事前準備

このチュートリアルを実行するには以下が必要です。

- チュートリアル 1 (p. 19) でセットアップ済みのアップストリーム用エッジ

**注釈:** このチュートリアルでは、エッジは 1 つしか使用しません。

## 44.2 パイプライン設定の作成

device-connector-intdash では、データをデバイスから受け取り intdash Edge Agent 2 に渡すまでの流れを「パイプライン設定ファイル」で設定します。

ここでは、以下の内容のパイプライン設定ファイルを作成し、`/tmp/dc-hello.yaml` として保存してください。

```yaml
tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello from device connector!"
      interval_ms: 100

  - id: 2
    element: iscp-v2-compat-filter
    from: [[ 1 ]]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        string:
          name: ab

  - id: 3
    element: file-sink
    from: [[ 2 ]]
    conf:
      path: "/var/run/intdash/up-hello.fifo"
```

このパイプライン設定ファイルでは、以下が定義されています。

- 「Hello from device connector!」というテキストを 100 ミリ秒おきに生成する（タスク ID: 1）
- FIFO 用データフォーマットに変換し、タイムスタンプ、型、ID を与える（タスク ID: 2）
- iscp-v2-compat フォーマットで FIFO に書き出す（タスク ID: 3）

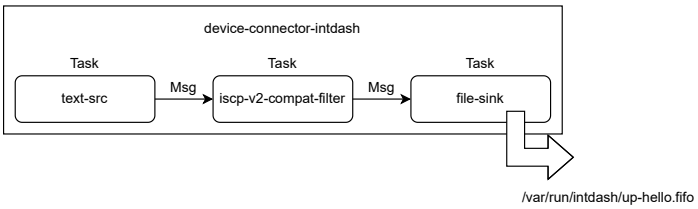



図 41　intdash-device-connector のパイプライン設定

## 44.3 ストリーマーの起動

以下のコマンドを実行して、ストリーマーを起動します。

```
$ intdash-agentctl run
```

これによりストリーマーは、`/var/run/intdash/up-hello.fifo` への入力を待ち受けている状態になります。

次に、新しいターミナルを開いて、デバイスコネクターを起動します。

```
$ device-connector-intdash --config /tmp/dc-hello.yaml
```

これにより、device-connector-intdash は 100 ミリ秒ごとに `/var/run/intdash/up-hello.fifo` に文字列を書き込みます。intdash Edge Agent 2 はそれを受け取り、intdash サーバーに送信します。

> **注釈:** ストリーマーが起動されたときに連動してデバイスコネクターを起動させることも可能です。デバイスコネクター IPC (p. 39) の `launch.cmd` を参照してください。

## 44.4 ウェブアプリでの確認

ウェブブラウザーで Edge Finder を開き、使用しているエッジのトラフィック画面を表示します。データが送信されていることを確認します。


図 42　Edge Finder でトラフィックを確認する

確認が済んだら、ストリーマーとデバイスコネクターを起動したターミナルで、それぞれ Ctrl+C を押して終了します。

# 45 パイプラインの設定方法

付属デバイスコネクター device-connecto-intdash は、パイプライン設定ファイルで設定します。

## 45.1 設定ファイルのフォーマット

パイプライン設定ファイルのフォーマットと内容については、Device Connector Framework のドキュメントの パイプライン設定ファイル (p. 187) を参照してください。また、インストールされる サンプル (p. 166) も参考にしてください。

## 45.2 環境変数の使用

device-connector-intdash では、パイプライン設定ファイルの設定値において、環境変数を展開して使用することができます。環境変数を展開するには、パイプライン設定ファイル内で `$(VARIABLE_NAME)` の形式で記述します。

例として、以下のような設定ファイル（conf.yaml）を考えます。`path` に `$(MY_FIFO_NAME)` が含まれています。

```
...

- id: 3
  element: file-sink
  from: [[2]]
  conf:
    flush_size: 10
    path: /var/run/intdash/$(MY_FIFO_NAME).fifo
```

その上で、この設定ファイルを使ってデバイスコネクターを実行する際に環境変数を与えます。

```
$ MY_FIFO_NAME=device123 device-connector-intdash --config conf.yaml
```

そうすると、この場合、`path` の値が `/var/run/intdash/device123.fifo` となります。

# 46 エレメント一覧 (device-connector-intdash)

device-connector-intdash には以下のエレメントが含まれています。パイプライン設定ファイルでは、これらを組み合わせてデータ取得から書き出しまでのパイプラインを設定します。

含まれているエレメントには、device-connector-intdash に固有のものと、Device Connector Framework から継承しているものがあります。Device Connector Framework の詳細については、デバイスコネクター開発フレームワーク (p. 182) を参照してください。

| | device-connector-intdash のエレメント | Device Connector Framework の基本エレメント[1] |
|---|---|---|
| src エレメント | • apt-analogtrx-src (p. 147)<br>• apt-cantrx-src (p. 149)<br>• nmea-packet-src (p. 150)<br>• ubx-src (p. 151)<br>• v4l2-src (p. 153) | • file-src (p. 190)<br>• process-src (p. 191)<br>• repeat-process-src (p. 191)<br>• text-src (p. 192)<br>• tee-src (p. 192) |
| filter エレメント | • h264-split-filter (p. 153)<br>• iscp-v2-compat-filter (p. 154)<br>• jpeg-split-filter (p. 161)<br>• pcm-split-filter (p. 161)<br>• ubx-iscpv2-filter (p. 164) | • print-log-filter (p. 193)<br>• split-by-delimiter-filter (p. 193)<br>• split-by-fixed-size-filter (p. 194)<br>• stat-filter (p. 194)<br>• tee-filter (p. 195) |
| sink エレメント | • apt-cantrx-sink (p. 164) | • file-sink (p. 196)<br>• null-sink (p. 197)<br>• stdout-sink (p. 197) |

## 46.1 src エレメント

以下の src エレメントを使用できます（アルファベット順）。

### 46.1.1 apt-analogtrx-src

エッジデバイスに接続された EDGEPLANT ANALOG-USB Interface からデータを取得するときに使用する src エレメントです。

このエレメントは、デバイスパスにより指定された EDGEPLANT ANALOG-USB Interface から専用カーネルモジュール経由でデータを取得します。そして iscp-v2-compat フォーマット（bytes 型）で送出します。

> **注釈:** EDGEPLANT ANALOG-USB Interface 用のカーネルモジュールについては、アプトポッドのウェブサイト の周辺機器についてのページを参照してください。

**エレメントの種類**

src

[1] エレメント名をクリックすると Device Connector Framework の説明ページに移動します。

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

カスタムタイプ "iscp-v2-compat-msgs"（iscp-v2-compat フォーマット（bytes 型）を複数個連結したメッセージ）

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの `conf:` 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | ANALOG-USB Interface のデバイスパス (例: `/dev/apt-usb/by-id/usb-xxx` ) |
| input_send_rate | 整数 ( `10` / `100` / `1000` / `10000` / `156250` / `312500` / `625000` / `1250000` / `2500000` / `5000000` / `10000000` ) | ANALOG-USB Interface のサンプリング周波数 [mHz](全ポート共通) |
| input_enabled | boolean(`true` / `false`) 8 個の配列 | ANALOG-USB Interface のアナログ入力ポートごと (全 8 ポート) の有効／無効 |
| input_voltage_min | 整数 8 個の配列 ( `0` / `-5000` / `-10000` ) | ANALOG-USB Interface のアナログ入力ポートごと（全 8 ポート）の入力電圧 [mV] の最小値 |
| input_voltage_max | 整数 8 個の配列 ( `5000` / `10000` ) | ANALOG-USB Interface のアナログ入力ポートごと（全 8 ポート）の入力電圧 [mV] の最大値<br>ただし、同一ポートの最小値と最大値の組み合わせは以下のいずれかである必要があります。<br>• min:-10000, max:10000<br>• min:-5000, max:5000<br>• min:0, max:5000<br>上記に一致しない場合は実行時にエラーになります。 |
| output_enabled | `true` / `false` | ANALOG-USB Interface のアナログ出力ポートの有効／無効 |
| output_voltage | 整数 (20〜5000（分解能 20）) | ANALOG-USB Interface のアナログ出力ポートから出力する信号の電圧 [mV] |
| output_wave-form_type | 整数 (`0` / `1` / `2` / `3` / `16` ) | ANALOG-USB Interface のアナログ出力ポートから出力する信号の波形（0:擬似ランダム信号、1:正弦波、2:三角波、3:矩形波、16:固定） |
| output_frequency | 整数 （1000〜100000（分解能 1000）） | ANALOG-USB Interface のアナログ出力ポートから出力する信号波形の周波数 [mHz] |
| clock_id | 文字列 (`CLOCK_MONOTONIC` / `CLOCK_MONOTONIC_RAW`) | タイムスタンプを算出するときにシステムからどのように時刻を取得するか<br>• `CLOCK_MONOTONIC`: NTP が行う段階的な調整の影響を受ける<br>• `CLOCK_MONOTONIC_RAW`: NTP が行う段階的な調整の影響を受けない<br>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。 |

次のページに続く

表 1 – 前のページからの続き

| 項目 | 設定値 | 説明 |
|---|---|---|
| timestamp_mode | 文字列 (`device` / `host`) デフォルト: `device` | 受信したデータに対して、タイムスタンプをどのように付与するか<br>• `device`: ANALOG-USB Interface 上で、ANALOG-USB Interface のクロックを元にタイムスタンプを付与する<br>• `host`: エッジデバイスが ANALOG-USB Interface からのデータを USB 経由で受信したタイミングで、エッジデバイスのクロックを元にタイムスタンプを付与する |

### 46.1.2 apt-cantrx-src

エッジデバイスに接続された EDGEPLANT CAN-USB Interface からデータを取得するときに使用する src エレメントです。

このエレメントは、デバイスパスにより指定された EDGEPLANT CAN-Interface から、専用カーネルモジュール経由でデータを取得します。そして、iscp-v2-compat フォーマット（can_frame 型）で送出します。

> **注釈:** EDGEPLANT CAN-USB Interface 用のカーネルモジュールについては、アプトポッドのウェブサイト の周辺機器についてのページを参照してください。

**エレメントの種類**
src

**次のエレメントに送信するポートの数**
1

**送信ポートでの送信形式**
カスタムタイプ "iscp-v2-compat-msgs"（iscp-v2-compat フォーマット（can_frame 型）を複数個連結したメッセージ）

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの `conf`: 以下に記述します。

> **重要:** EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して apt-cantrx-src と apt-cantrx-sink を同時に使用する場合は、apt-cantrx-src と apt-cantrx-sink の設定をすべて同じにしてください。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | CAN-USB Interface のデバイスパス（例: `/dev/apt-usb/by-id/usb-xxx`） |
| baudrate | 整数 (`125` / `250` / `500` / `10000`) | CAN 通信のボーレート [kbps] |
| silent | `true` / `false` | CAN-USB Interface から CAN バスへの ACK の送信<br>• `true`: ACK を送信しない<br>• `false`: ACK を送信する |

次のページに続く

表 2 – 前のページからの続き

| 項目 | 設定値 | 説明 |
|---|---|---|
| clock_id | 文字列 (`CLOCK_MONOTONIC` / `CLOCK_MONOTONIC_RAW`) | タイムスタンプを算出するときにシステムからどのように時刻を取得するか<br>• `CLOCK_MONOTONIC`: NTP が行う段階的な調整の影響を受ける<br>• `CLOCK_MONOTONIC_RAW`: NTP が行う段階的な調整の影響を受けない<br>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。 |
| timestamp_mode | 文字列 (`device` / `host`) デフォルト: `device` | 受信したデータに対して、タイムスタンプをどのように付与するか<br>• `device`: CAN-USB Interface 上で、ANALOG-USB Interface のクロックを元にタイムスタンプを付与する<br>• `host`: エッジデバイスが CAN-USB Interface からのデータを USB 経由で受信したタイミングで、エッジデバイスのクロックを元にタイムスタンプを付与する |

### 46.1.3 nmea-packet-src

このエレメントでは、デバイスパスにより指定された TTY デバイスから TTY ドライバー経由で NMEA データを取得します。

**エレメントの種類**

src

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

MIME タイプ "text/plain" (1 つの NMEA センテンスが 1 つのメッセージ)

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | GPS デバイスのデバイスパス（例: `/dev/ttyUSB9`） |
| baudrate | 整数（`0`, `50`, `75`, `110`, `134`, `150`, `200`, `300`, `600`, `1200`, `1800`, `2400`, `4800`, `9600`, `19200`, `38400`, `57600`, `115200`, `230400`） | ボーレート (例:`57600`)<br>termios で定義された左記いずれかの値である必要があります。 |

**注釈:**

- nmea-packet-src エレメントは、GPS デバイスの初期化は行いません。あらかじめ GPS デバイス（tty デバイス）が所定のボーレートで NMEA を出力する状態にしてください。
- EDGEPLATNT T1 で L4T を使用する場合は、以下のようにすることで GPS デバイスを初期化してデータを取得することができます。
  1. edgeplant-l4t-tools サービスを有効にします。方法については、EDGEPLANT T1 デベロッパーガイド を参照してください。
  2. パイプライン設定ファイルに以下を追加します。

```
before_task:
  - while true; do if test $(stty -F /dev/ttyTHS1 speed) = 57600; then break; fi; sleep 1; done
  - sleep 3
```

### 46.1.4 ubx-src

u-blox GNSS モジュールから UBX プロトコルのメッセージを取得し、そのままバイナリデータとして送出します。

取得対象の UBX プロトコルのメッセージは以下のとおりです。

- UBX-ESF-STATUS
- UBX-HNR-STATUS
- UBX-HNR-ATT
- UBX-HNR-INS
- UBX-HNR-PVT

**重要:** u-blox GNSS モジュールとして NEO-M8U のみをサポートしています。NEO-M8U を搭載していないエッジデバイス（ VTC1910-S など）はサポートしていません。

**注釈:** エッジコンピューターの起動時から GPS 捕捉できない状態が継続すると、座標 (0, 0) が送信されます。

**エレメントの種類**

src

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | u-blox GNSS モジュールのデバイスパス<br>(例: `/dev/ttyTHS1` ) |
| baud_rate | 整数（ `9600`, `19200`, `38400`, `57600`, `115200`, `230400`, `460800` ） | ボーレート（例:`57600`) |

次のページに続く

表 3 – 前のページからの続き

| 項目 | 設定値 | 説明 |
|---|---|---|
| meas_rate_ms | 小数 | GNSS 測定周期 (ミリ秒) (例: 200)<br>50 ミリ秒以上の値である必要があります。<br>以下のメッセージの送信頻度に影響します。<br>• UBX-ESF-STATUS<br>• UBX-HNR-STATUS |
| nav_rate | 整数 | ナビゲーションソリューションに対する測定頻度 (例: 1)<br>127 以下の値である必要があります。n を指定した場合、各ナビゲーションソリューションに対して n 回の測定を行います（n が大きくなると、送信頻度が下がります)。<br>以下のメッセージの送信頻度に影響します。<br>• UBX-ESF-STATUS<br>• UBX-HNR-STATUS |
| esf_status_rate | 整数 | UBX-ESF-STATUS メッセージ送信頻度 (例: 1)<br>n を指定した場合は、n * nav_rate * meas_rate_ms の周期で UBX-ESF-STATUS メッセージが送信されます（n が大きくなると、送信頻度が下がります)。<br>0 を指定した場合は、送信されません。 |
| nav_status_rate | 整数 | UBX-NAV-STATUS メッセージ送信頻度 (例: 1)<br>n を指定した場合は、n * nav_rate * meas_rate_ms の周期で UBX-NAV-STATUS メッセージが送信されます（n が大きくなると、送信頻度が下がります)。<br>0 を指定した場合は、送信されません。 |
| high_nav_rate_hz | 整数 | UBX-HNR メッセージの送信間隔 (周波数) (例: 5)<br>20 以下の値である必要があります。<br>以下のメッセージの送信頻度に影響します。<br>• UBX-HNR-ATT<br>• UBX-HNR-INS<br>• UBX-HNR-PVT |
| hnr_att_rate | 整数 | UBX-HNR-ATT メッセージの送信頻度 (例: 1)<br>n を指定した場合は、high_nav_rate_hz / n の周波数で UBX-HNR-ATT メッセージが送信されます（n が大きくなると、送信頻度が下がります)。<br>0 を指定した場合は、送信されません。 |
| hnr_ins_rate | 整数 | UBX-HNR-INS メッセージの送信頻度 (例: 1)<br>n を指定した場合は、high_nav_rate_hz / n の周波数で UBX-HNR-INS メッセージが送信されます（n が大きくなると、送信頻度が下がります)。<br>0 を指定した場合は、送信されません。 |
| hnr_pvt_rate | 整数 | UBX-HNR-PVT メッセージの送信頻度 (例: 1)<br>n を指定した場合は、high_nav_rate_hz / n の周波数で UBX-HNR-PVT メッセージが送信されます（n が大きくなると、送信頻度が下がります)。<br>0 を指定した場合は、送信されません。 |

**注釈:** 各設定項目の詳細については、u-blox GNSS モジュールのドキュメント "u-blox 8 / u-blox M8 Receiver description" を参照してください。

### 46.1.5 v4l2-src

指定されたカメラから V4L2 (Video for Linux 2) 経由で H.264 形式の動画データを受け取り、h264_annex_b 型の iscp-v2-compat フォーマット で送出します。

**エレメントの種類**
src

**次のエレメントに送信するポートの数**
1

**送信ポートでの送信形式**
カスタムタイプ "iscp-v2-compat-msg" (h264_annex_b 型)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの `conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | デバイスパス（例: `/dev/video0`） |
| type | 文字列（`h264` / `jpeg`） | ドライバーから取得するデータの種類。<br>• `h264`: H.264 のデータを取得します。<br>• `jpeg`: JPEG データを取得します。 |
| width | 整数 | 画像幅。カメラが対応している画像幅を指定してください。 |
| height | 整数 | 画像高さ。カメラが対応している画像高さを指定してください。 |
| fps | 整数 | FPS。カメラが対応している FPS を指定してください。 |

## 46.2 filter エレメント

以下の filter エレメントを使用できます（アルファベット順)。

### 46.2.1 h264-split-filter

メッセージを受け取り、H.264 形式の動画データとしてパースし、iscp-v2-compat フォーマット (h264_annex_b 型）で送出します。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
H.264 Annex B 形式のバイナリストリームを任意の箇所で区切ったバイト列

**送信ポートでの送信形式**

カスタムタイプ "iscp-v2-compat-msg" (h264_annex_b 型)

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| clock_id | 文字列 (`CLOCK_MONOTONIC` / `CLOCK_MONOTONIC_RAW`) | タイムスタンプを算出するときにシステムからどのように時刻を取得するか<br>• `CLOCK_MONOTONIC`: NTP が行う段階的な調整の影響を受ける<br>• `CLOCK_MONOTONIC_RAW`: NTP が行う段階的な調整の影響を受けない<br>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。 |
| delay_ms | 小数 | タイムスタンプを付与する際に差し引くオフセット (カメラ処理時間)<br>例えば、カメラ内での処理に 100 ミリ秒かかる場合は、「100」を指定します。これにより、タイムスタンプ付与時に 100 ミリ秒前の時刻が使用されます。<br>設定例:<br>`conf:`<br>`  clock_id: CLOCK_MONOTONIC`<br>`  delay_ms: 100` |
| name | 文字列 | 送出する iscp-v2-compat-msg の Data Name フィールドに設定する文字列<br>デフォルトは `h264_annex_b` です。 |

### 46.2.2 iscp-v2-compat-filter

データを指定されたフォーマットでパースし、指定に応じてタイムスタンプを付与し、can_frame / jpeg / string/nmea / bytes / float64 / int64 / string いずれかの型の iscp-v2-compat フォーマット で送出します。

**エレメントの種類**

filter

**前のエレメントから受信するポートの数**

1

**次のエレメントに送信するポートの数**

1

**受信ポートでの受信形式**

任意（ `timestamp` および `convert_rule` の指定に適合していること）

**送信ポートでの送信形式**

カスタムタイプ "iscp-v2-compat-msg"（can_frame 型 / jpeg 型 / string/nmea 型 / bytes 型 / float64 型 / int64 型 / string 型）

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| timestamp | 文字列 (stamped)、または、タイムスタンプ付与方法を定義したオブジェクト | 受信するメッセージにタイムスタンプ（受信データのタイムスタンプ (p. 155)）が含まれているか<br>• stamped: タイムスタンプが含まれている（このタイムスタンプを出力時にも付与します。）<br>`conf:`<br>`  timestamp: stamped`<br>• stamp: タイムスタンプが含まれていない（出力時には clock_id の指定に従ってタイムスタンプを付与します。）<br>`conf:`<br>`  timestamp:`<br>`  stamp:`<br>`  clock_id: CLOCK_MONOTONIC`<br>clock_id の指定:<br>• CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける時計を使用する<br>• CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない時計を使用する<br>clock_id は、intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。 |
| convert_rule | 変換ルールを定義したオブジェクト | iscp-v2-compat フォーマットに変換するためのルールを指定します。詳細は以下を参照してください。<br>• can (p. 156)<br>• jpeg (p. 156)<br>• nmea (p. 157)<br>• bytes (p. 157)<br>• float64 (p. 158)<br>• int64 (p. 159)<br>• string (p. 160) |

### 受信データのタイムスタンプ

受信するメッセージの先頭にタイムスタンプが追加されている場合は、設定項目 timestamp の値を stamped とします。この場合、受信したメッセージの先頭 8 バイトがタイムスタンプとしてパースされます。期待するフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                      Timestamp (seconds)                      |
+---------------+---------------+---------------+---------------+
|                      Timestamp (nano seconds)                 |
+---------------+---------------+---------------+---------------+
|                      Message
+---------------+---------------+------------//--+
```

この Timestamp (seconds) フィールド、Timestamp (nano seconds) フィールドの値は、それぞれ iscp-v2-compat での出力時に同名のフィールドに出力されます。

Message は、設定項目 `convert_rule` で指定された任意のフォーマットのデータです。

### CAN データ用変換ルール（can）

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule:
    can:
      lower_hex: true
```

この変換ルールでは以下の項目を設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| lower_hex | boolean(`true` / `false`) | 出力する iscp-v2-compat フォーマットの Data Name フィールドに入れる 16 進数数値表現を小文字で行うか |

CAN データ用の変換で期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|            CAN_ID  + EFF/RTR/ERR flags (Little Endian)        |
+---------------+---------------+---------------+---------------+
|      DLC      |      DATA
+---------------+---------------+-----------//--+
```

- CAN_ID + EFF/RTR/ERR flags: Linux SocketCan の can_frame 構造体 の `can_id`。
- DLC: Linux SocketCan の can_frame 構造体 の `len`。
- DATA: Linux SocketCan の can_frame 構造体 の `data`。

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `can_frame`
- Data Name: 上記入力データ内の CAN_ID（16 進 8 桁の文字列表現）（ただし EFF フラグはオフになります）
- Payload: 上記入力データ内の DATA フィールドの内容

### JPEG 用変換ルール（jpeg）

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule:
    jpeg:
      name: "jpeg"
```

この変換ルールでは以下の項目を設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | 文字列 | 送出する iscp-v2-compat-msg の Data Name フィールドに設定する文字列<br>デフォルトは `jpeg` です。 |

JPEG 用の変換では、入力されたメッセージを、JPEG (ITU-T Rec. T.81 ¦ ISO/IEC 10918-1, 10918-2) としてパースします。期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                     JPEG
+---------------+---------------+-----------//--+
```

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `jpeg`
- Data Name: 設定項目 `name` に指定された文字列。
- Payload: 上記入力データ内の JPEG フィールドの内容

### NMEA データ用変換ルール（nmea）

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule: nmea
```

NMEA データ用の変換では、入力されたメッセージを NMEA 0183 としてパースします。期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                     NMEA
+---------------+---------------+-----------//--+
```

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `string/nmea`
- Data Name: `<NMEA のトーカ>/<NMEA のメッセージ>`（上記入力データ内の NMEA フィールドから抽出されたもの）
- Payload: 上記入力データ内の NMEA フィールドの内容

この変換ルールには設定項目はありません。

### Bytes データ用変換ルール（bytes）

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    bytes:
      name: null
```

この変換ルールでは以下の項目が設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | string または null | 出力する iscp-v2-compat フォーマットの Data Name フィールドに入れる文字列 |

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                     Bytes
+---------------+---------------+-----------//--+
```

設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|  Name Length  |                     Name
+---------------+---------------+---------------+-----------//--+
|                             Bytes
+---------------+---------------+-----------//--+
```

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: bytes
- Data Name: 設定項目 name に指定された文字列。または、name が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の Bytes フィールドの内容

### Float64 データ用変換ルール（float64）

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    float64:
      name: null
```

この変換ルールでは以下の項目が設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | string または null | 出力する iscp-v2-compat フォーマットの Data Name フィールドに入れる文字列 |

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                                                               |
+                       Float (Little Endian)                   +
|                                                               |
+---------------+---------------+---------------+---------------+
```

設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマッ

トは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|  Name Length  |                     Name                      |
+---------------+---------------+---------------+-----------//--+
|                                                               |
+                      Float (Little Endian)                    +
|                                                               |
+---------------+---------------+---------------+---------------+
```

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `float64`
- Data Name: 設定項目 `name` に指定された文字列。または、`name` が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の Float フィールドの内容

### int64 データ用変換ルール（int64）

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    int64:
      name: null
```

この変換ルールでは以下の項目が設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | string または null | 出力する iscp-v2-compat フォーマットの Data Name フィールドに入れる文字列 |

設定項目の `name` に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                                                               |
+                       Int (Little Endian)                     +
|                                                               |
+---------------+---------------+---------------+---------------+
```

設定項目の `name` が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|  Name Length  |                     Name                      |
+---------------+---------------+---------------+-----------//--+
|                                                               |
+                       Int (Little Endian)                     +
|                                                               |
+---------------+---------------+---------------+---------------+
```

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `int64`
- Data Name: 設定項目 `name` に指定された文字列。または、`name` が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の Int フィールドの内容

### String データ用変換ルール（string）

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    string:
      name: abc
```

この変換ルールでは以下の項目が設定可能です。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | string または null | 出力する iscp-v2-compat フォーマットの Data Name フィールドに入れる文字列 |

設定項目の `name` に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|                     String
+---------------+---------------+-----------//--+
```

設定項目の `name` が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

```
0                                                               4
+---------------+---------------+---------------+---------------+
|  Name Length  |                     Name                      |
+---------------+---------------+---------------+-----------//--+
|                     String
+---------------+---------------+-----------//--+
```

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compat の各フィールドは以下のようになります。

- Data Type: `string`
- Data Name: 設定項目 `name` に指定された文字列。または、`name` が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の String フィールドの内容

### 46.2.3 jpeg-split-filter

jpeg-split-filter は、受信したバイナリストリームを JPEG が複数並んでいるバイナリとしてパースして、1 枚ごとの JPEG に分けて送信します。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
JPEG フォーマットのバイナリを並べたバイナリストリームを任意の箇所で区切ったバイト列

**送信ポートでの送信形式**
MIME タイプ "image/jpeg" (1 枚の画像が 1 つのメッセージ)

エレメント独自の設定項目はありません。

### 46.2.4 pcm-split-filter

GStreamer から出力された音声データを受け取り、bytes 型の iscp-v2-compat フォーマット に変換して送出します。

> **重要:** エッジデバイスとして T1 / VTC1910-S 以外を使用する場合は正しく設定を行えない可能性があります。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
任意（前のタスクで process-src エレメントを使って GStreamer から音声を出力させ、その出力を本タスクで受け取るようにします。）

GStreamer からの出力は Raw Audio Media Types における以下のタイプとしてください。

- `format`: S16LE / S32LE / F32LE
- `rate`: 48000
- `channels`: 1
- `channel-mask`: 使わない（GStreamer コマンドに設定しないこと）
- `layout`: 使わない（GStreamer コマンドに設定しないこと）

**送信ポートでの送信形式**
カスタムタイプ "iscp-v2-compat-msg" (bytes 型)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの `conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| clock_id | 文字列 (`CLOCK_MONOTONIC` / `CLOCK_MONOTONIC_RAW`) | タイムスタンプを算出するときにシステムからどのように時刻を取得するか<br>• `CLOCK_MONOTONIC`: NTP が行う段階的な調整の影響を受ける<br>• `CLOCK_MONOTONIC_RAW`: NTP が行う段階的な調整の影響を受けない<br>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。 |
| delay_ms | 小数 | データ打刻のオフセットを設定入力します。音声が発生してからデバイスコネクターにデータが届くまでに 100 ミリ秒かかる場合は、`100` を指定します。ほとんどの場合 `0` で問題ありません。 |
| audio_element | 文字列 (`Mic Jack`/ `y Jack-state`) | コンピューターのマイクジャックを使用する場合、マイクジャックの状態（抜き差し）を示すデバイスエレメント名を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合: `Mic Jack`<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: `y Jack-state` |
| audio_iface | 文字列 (`card` /`mixer`) | 音声を制御するデバイスの種別を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合: `card`<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: `mixer` |
| audio_format | 文字列 (`S16LE` / `S32LE` / `F32LE`) | 音声データの形式を選択します<br>• `S16LE`: signed 16 bit little-endian<br>• `S32LE`: signed 32 bit little-endian<br>• `F32LE`: floating-point little-endian |
| audio_rate | 整数 (`48000`)（固定） | サンプリング周波数 [Hz] |
| audio_channels | 整数 (`1`)（固定） | チャンネルの数。 |
| audio_volume_iface | 文字列 (`mixer`) | 音量を設定する場合、音量を制御するデバイスの種別を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合: `mixer`<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: `mixer` |
| audio_volume_element | 文字列 (`Capture Volume`/ `y Mic Capture Volume`) | 音量を設定する場合、音量を示すデバイスエレメント名を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合: `Capture Volume`<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: `y Mic Capture Volume` |

次のページに続く

表 11 – 前のページからの続き

| 項目 | 設定値 | 説明 |
|---|---|---|
| audio_volume_value | 整数 | 音量を設定する場合、音量として設定する値を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合:<br>　• 範囲: 0 - 46<br>　• デフォルト: 28<br>　• 0 のときに 16.00 dB 設定<br>　• 値を 1 上げるごとに 1.00 dB 上昇<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合:<br>　• 範囲: 0 - 20<br>　• デフォルト: 20<br>　• 0 のときに 0.00 dB (音量上昇なし、等倍) 設定<br>　• 値を 1 上げるごとに 1.00 dB 上昇 |
| audio_boost_element | 文字列 (`Mic Boost Volume`/ `y Mic Boost Capture Volume`) | 音量を設定する場合、ブーストを示すデバイスエレメント名を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合: `Mic Boost Volume`<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: `y Mic Boost Capture Volume` |
| audio_boost_value | 整数 | 音量を設定する場合、ブーストとして設定する値を入力します。<br>• Terminal System のコンピューターとして VTC 1910-S を使用する場合:<br>　• 範囲: 0 - 3<br>　• デフォルト: 0<br>　• 0 のときに 0.00 dB 設定<br>　• 値を 1 上げるごとに 10.00 dB 上昇<br>• Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合:<br>　• 範囲: 0 - 3<br>　• デフォルト: 1<br>　• 0: -20.00 dB (Mute)<br>　• 1: 0.00 dB (gain なし)<br>　• 2: 20.00 dB<br>　• 3: 30.00 dB |
| name | 文字列 | 送出する iscp-v2-compat-msg の Data Name フィールドに設定する文字列<br>デフォルトは `pcm` です。 |

### 46.2.5 ubx-iscpv2-filter

ubx-src から UBX メッセージを受け取り、メッセージをパースして Data Name をバイナリデータの先頭に付与し、iscp-v2-compat-filter に送信します。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
MIME タイプ "application/octet-stream"（ubx-src が送信したメッセージ）

**送信ポートでの送信形式**
MIME タイプ "application/octet-stream"（iscp-v2-compat-filter が受信するメッセージ）

エレメント独自の設定項目はありません。

## 46.3 sink エレメント

以下の sink エレメントを使用できます（アルファベット順）。

### 46.3.1 apt-cantrx-sink

メッセージを受け取り、iscp-v2-compat フォーマット（can_frame 型）としてパースし、専用カーネルモジュール経由で EDGEPLANT CAN-USB Interface に送出します。

> **注釈:** EDGEPLANT CAN-USB Interface 用のカーネルモジュールについては、アプトポッドのウェブサイト の周辺機器についてのページを参照してください。

**エレメントの種類**
sink

**前のエレメントから受信するポートの数**
1

**受信ポートでの受信形式**
任意（iscp-v2-compat フォーマット（can_frame 型）を複数個連結したバイナリストリーム）

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

> **重要:** EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して apt-cantrx-src と apt-cantrx-sink を同時に使用する場合は、apt-cantrx-src と apt-cantrx-sink の設定をすべて同じにしてください。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | CAN-USB Interface のデバイスパス（例: `/dev/apt-usb/by-id/usb-xxx`） |
| baudrate | 整数（`125` / `250` / `500` / `10000`） | CAN 通信のボーレート [kbps] |

次のページに続く

表 12 – 前のページからの続き

| 項目 | 設定値 | 説明 |
|---|---|---|
| silent | `true` / `false` | CAN-USB Interface から CAN バスに ACK を送信しない設定の有効／無効（ `true`:ACK なし、`false`:ACK あり） |
| clock_id | 文字列 (`CLOCK_MONOTONIC` / `CLOCK_MONOTONIC_RAW`) | タイムスタンプを算出するときにシステムからどのように時刻を取得するか<br>本エレメントはタイムスタンプを付与しないため、任意の値で構いません。<br>ただし、EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して、apt-cantrx-src と本エレメントを同時に使用する場合は、apt-cantrx-src と同じ設定にしてください。 |
| timestamp_mode | 文字列 (`device` / `host` )<br>デフォルト: `device` | 受信したデータに対して、タイムスタンプをどのように付与するか<br>本エレメントはタイムスタンプを付与しないため、任意の値で構いません。<br>ただし、EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して、apt-cantrx-src と本エレメントを同時に使用する場合は、apt-cantrx-src と同じ設定にしてください。 |

# 47 パイプライン設定サンプル一覧

付属デバイスコネクター device-connector-intdash がインストールされると、パイプライン設定ファイルのサンプルが /etc/dc_conf ディレクトリにインストールされます。

| パイプライン設定ファイル | 説明 |
|---|---|
| apt_analog.yml (p. 166) | ANALOG-USB Interface のデータを送信する |
| apt_cantrx.yml (p. 168) | CAN-USB Interface のデータを送受信する |
| device_inventory.yml (p. 169) | エッジデバイスのステータス情報（CPU、メモリ、ストレージ、ネットワークなど）を送信する ( 詳細 (p. 169) ) |
| gstreamer_h264.yml (p. 171) | GStreamer から出力される H.264 の動画を送信する |
| gstreamer_h264_nalunit.yml (p. 171) | GStreamer から出力される H.264 の動画を、iSCP で定義されている H.264 NAL Unit データ型を使って送信する |
| gstreamer_jpeg.yml (p. 172) | GStreamer から出力される MJPEG 形式の動画を送信する |
| gstreamer_pcm.yml (p. 173) | GStreamer から出力される音声を送信する |
| jpeg.yml (p. 174) | V4L2（Video for Linux 2）経由で取得した MJPEG 形式の動画を送信する |
| nmea.yml (p. 175) | TTY デバイスから取得した NMEA データを送信する |
| repeat_process_json.yml (p. 175) | プロセスから標準出力された JSON 文字列を送信する |
| repeat_process_string.yml (p. 176) | プロセスから標準出力された文字列を送信する |
| ubx.yml (p. 177) | u-blox GNSS モジュールからの情報（UBX プロトコルのメッセージ）を送信する |
| v4lh264.yml (p. 178) | V4L2（Video for Linux 2）経由で取得した H.264 の動画を送信する |

## 47.1 apt_analog.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/apt_analog.yml は、ANALOG-USB Interface のデータを送信するためのパイプライン設定ファイルです。

```
before_task:
  # sync timestamp
  - mkdir -p /var/lock/intdash
  - |
    BASETIME_CLOCK_ID=$DC_CLOCK_ID
    meas-hook --lockfile /var/lock/intdash/dc_apt_usbtrx.lock --command "
      if command -v apt_usbtrx_timesync.sh     > /dev/null 2>&1 ; then apt_usbtrx_timesync.sh;     exit 0; fi;
      if command -v apt_usbtrx_timesync_all.sh > /dev/null 2>&1 ; then apt_usbtrx_timesync_all.sh; exit 0; fi;
      echo \"ERROR: timestamp script not found\";
      exit 1;
    "

after_task:
  - rm -f /var/lock/intdash/dc_apt_usbtrx.lock

tasks:
  - id: 1
    element: apt-analogtrx-src
    conf:
```

(次のページに続く)

(前のページからの続き)

```
    clock_id: CLOCK_MONOTONIC
    path: $(DC_APT_ANALOGTRX_SRC_CONF_PATH)
    timestamp_mode: $(DC_APT_ANALOGTRX_SRC_CONF_TIMESTAMP_MODE)
    input_send_rate: $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_SEND_RATE)
    input_enabled:
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_0)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_1)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_2)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_3)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_4)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_5)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_6)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_ENABLED_7)
    input_voltage_min:
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_0)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_1)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_2)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_3)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_4)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_5)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_6)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MIN_7)
    input_voltage_max:
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_0)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_1)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_2)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_3)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_4)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_5)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_6)
      - $(DC_APT_ANALOGTRX_SRC_CONF_INPUT_VOLTAGE_MAX_7)
    output_enabled: $(DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_ENABLED)
    output_waveform_type: $(DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_WAVEFORM_TYPE)
    output_voltage: $(DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_VOLTAGE)
    output_frequency: $(DC_APT_ANALOGTRX_SRC_CONF_OUTPUT_FREQUENCY)

  - id: 2
    element: print-log-filter
    from: [[1]]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 3
    element: file-sink
    from: [[2]]
    conf:
      flush_size: 10
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.2 apt_cantrx.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/apt_cantrx.yml` は、CAN-USB Interface のデータを送受信するためのパイプライン設定ファイルです。

```
before_task:
  # sync timestamp
  - mkdir -p /var/lock/intdash
  - |
    BASETIME_CLOCK_ID=$DC_CLOCK_ID
    meas-hook --lockfile /var/lock/intdash/dc_apt_usbtrx.lock --command "
      if command -v apt_usbtrx_timesync.sh     > /dev/null 2>&1 ; then apt_usbtrx_timesync.sh;     exit 0; fi;
      if command -v apt_usbtrx_timesync_all.sh > /dev/null 2>&1 ; then apt_usbtrx_timesync_all.sh; exit 0; fi;
      echo \"ERROR: timestamp script not found\";
      exit 1;
    "

after_task:
  - rm -f /var/lock/intdash/dc_apt_usbtrx.lock

tasks:
  - id: 1
    element: apt-cantrx-src
    conf:
      clock_id: CLOCK_MONOTONIC
      path: $(DC_APT_CANTRX_SRC_CONF_PATH)
      listenonly: $(DC_APT_CANTRX_SRC_CONF_LISTENONLY)
      baudrate: $(DC_APT_CANTRX_SRC_CONF_BAUDRATE)
      timestamp_mode: $(DC_APT_CANTRX_SRC_CONF_TIMESTAMP_MODE)
      # If you want to change the data_name settings, please uncomment, default is as follows.
      #data_name_conf:
      #  with_ide: false
      #  lower_case: true

  - id: 2
    element: print-log-filter
    from: [[1]]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 3
    element: file-sink
    from: [[2]]
    conf:
      flush_size: 10
      path: $(DC_FILE_SINK_CONF_PATH)

  - id: 10
    element: file-src
    conf:
      path: $(DC_FILE_SRC_CONF_PATH)

  - id: 11
    element: apt-cantrx-sink
    from: [[10]]
    conf:
```

(次のページに続く)

(前のページからの続き)

```
      clock_id: CLOCK_MONOTONIC
      path: $(DC_APT_CANTRX_SINK_CONF_PATH)
      listenonly: $(DC_APT_CANTRX_SINK_CONF_LISTENONLY)
      baudrate: $(DC_APT_CANTRX_SINK_CONF_BAUDRATE)
      timestamp_mode: $(DC_APT_CANTRX_SINK_CONF_TIMESTAMP_MODE)
```

## 47.3 device_inventory.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/device_inventory.yml は、エッジデバイスのステータス情報（CPU、メモリ、ストレージ、ネットワークなど）を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: process-src
    conf:
      # Set DC_PROCESS_SRC_CONF_COMMAND_INTERVAL to 30s or more. Shorter values may not ensure sends at the␣
↪configured intervals.
      command: /etc/dc_conf/scripts/device_inventory.sh --interval $(DC_PROCESS_SRC_CONF_COMMAND_INTERVAL)
      buffer_size: 65535
      retry: true
      retry_interval_ms: 1000

  - id: 2
    element: json-split-filter
    from: [ [1] ]

  - id: 3
    element: iscp-v2-compat-filter
    from: [ [2] ]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        string:
          name: device_inventory

  - id: 4
    element: print-log-filter
    from: [ [3] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 5
    element: file-sink
    from: [ [4] ]
    conf:
      path: $(DC_FILE_SINK_CONF_PATH)
```

### device_inventory.yml についての補足

パイプライン設定ファイル `device_inventory.yml` は、エッジデバイスのステータス情報を以下のコマンドを利用して取得します。

| コマンド | 説明 |
|---|---|
| df | ストレージ情報 |
| ip | ネットワーク統計情報 |
| lsusb | USB デバイス情報 |
| mmcli | モバイル回線情報[1] |
| ping | RTT(Round Trip Time) 情報 |
| vmstat | システムの統計情報 |

**警告:** このデバイスコネクターでは、df や mmcli などのコマンドによりステータス情報が取得され、送信されます。ただし、以下に該当する古いバージョンのディストリビューションでは、いくつかのコマンドのデータが送信されません。

| ディストリビューション | 情報を送信できないコマンド |
|---|---|
| Debian:9 | mmcli、ip |
| Debian:10 | mmcli |
| Ubuntu:18.04 | mmcli |

**警告:** 以下条件に該当する場合、送信する情報が多くなりデータの取得に時間がかかることがあります。

| 条件 | 例 |
|---|---|
| ip コマンドで送信するネットワークインターフェースが多い場合 | Docker を利用している |
| lsusb コマンドで送信するデバイス情報が多い場合 | 多数の USB デバイスが接続されている |
| df コマンドで送信するストレージ情報が多い場合 | 多数のストレージが接続されている |

**注釈:** Docker 環境で利用する場合、以下の点に注意してください。

- mmcli の情報を取得するために `/var/run/dbus` のバインドマウントが必要です。
- df のマウントポイント `/` の情報は、Docker のストレージドライバの情報となります。通常はホストの `/var/lib/docker/overlay2` が存在するパーティションの情報となります。
- ip で取得できる情報はコンテナからアクセスできるネットワークインターフェースの情報のみとなります。ホストの情報も取得したい場合は、コンテナ実行時に `--net=host` を指定してください。

[1] 現在接続しているバンド情報（active-band）の取得には非対応です。

## 47.4 gstreamer_h264.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/gstreamer_h264.yml は、GStreamer から出力される H.264 の動画を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: process-src
    conf:
      # Example of vaapih264enc: gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! queue ! vaapijpegdec ! queue ! vaapipostproc ! queue ! vaapih264enc␣
↪rate-control=2 bitrate=3000 max-bframes=0 keyframe-period=15 ! fdsink fd=1
      # Example of omxh264enc: gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! jpegdec ! omxh264enc control-rate=2 iframeinterval=15␣
↪bitrate=3000000 insert-sps-pps=true insert-vui=true ! video/x-h264,stream-format=byte-stream ! queue !␣
↪h264parse ! queue ! fdsink fd=1
      command: $(DC_PROCESS_SRC_CONF_COMMAND)

  - id: 2
    element: h264-split-filter
    from: [ [1] ]
    conf:
      clock_id: CLOCK_MONOTONIC
      delay_ms: $(DC_H264_SPLIT_FILTER_CONF_DELAY_MS)

  - id: 3
    element: print-log-filter
    from: [ [2] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 4
    element: file-sink
    from: [ [3] ]
    conf:
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.5 gstreamer_h264_nalunit.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/gstreamer_h264_nalunit.yml は、GStreamer から出力される H.264 の動画を、iSCP で定義されている H.264 NAL Unit データ型を使って送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: process-src
    conf:
      # Example of vaapih264enc: gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! queue ! vaapijpegdec ! queue ! vaapipostproc ! queue ! vaapih264enc␣
↪rate-control=2 bitrate=3000 max-bframes=0 keyframe-period=15 num-slices=32 ! fdsink fd=1
      # Example of omxh264enc: gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,
↪width=1920,height=1080,framerate=15/1 ! jpegdec ! omxh264enc control-rate=2 iframeinterval=15␣
↪bitrate=3000000 bit-packetization=true slice-header-spacing=1200 insert-sps-pps=true insert-vui=true !␣
↪video/x-h264,stream-format=byte-stream ! queue ! h264parse ! queue ! fdsink fd=1
```

(次のページに続く)

(前のページからの続き)

```
      command: $(DC_PROCESS_SRC_CONF_COMMAND)
  - id: 2
    element: h264-nalunit-split-filter
    from: [ [1] ]
    conf:
      clock_id: CLOCK_MONOTONIC
      delay_ms: $(DC_H264_NALUNIT_SPLIT_FILTER_CONF_DELAY_MS)

  - id: 3
    element: print-log-filter
    from: [ [2] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 4
    element: file-sink
    from: [ [3] ]
    conf:
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.6 gstreamer_jpeg.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/gstreamer_jpeg.yml` は、GStreamer から出力される MJPEG 形式の動画を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: process-src
    conf:
      # Example : gst-launch-1.0 -q v4l2src device=/dev/video0 ! videorate ! image/jpeg,width=320,height=240,
↪framerate=15/1 ! queue ! fdsink fd=1
      command: $(DC_PROCESS_SRC_CONF_COMMAND)

  - id: 2
    element: jpeg-split-filter
    from: [ [1] ]

  - id: 3
    element: iscp-v2-compat-filter
    from: [ [2] ]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        # Starting from v2.2.0, it is required that 'jpeg' be specified as a map.
        jpeg:
          name: "jpeg"

  - id: 4
    element: print-log-filter
    from: [ [3] ]
    conf:
```

(次のページに続く)

(前のページからの続き)

```
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 5
    element: file-sink
    from: [ [4] ]
    conf:
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.7 gstreamer_pcm.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/gstreamer_pcm.yml` は、GStreamer から出力される音声を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: process-src
    conf:
      # Example of VTC 1910-S onboard audio:
      #   DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q alsasrc device=hw:0 ! audioconvert ! audio/x-raw,
↪format=S16LE,rate=48000,channels=1 ! fdsink fd=1"
      #   DC_PCM_SPLIT_FILTER_CONF_PATH="/dev/snd/by-path/pci-0000:00:1b.0"
      # Example of EDGEPLANT T1 onboard audio:
      #   DC_PROCESS_SRC_CONF_COMMAND="gst-launch-1.0 -q alsasrc device=hw:1 ! audioconvert ! audio/x-raw,
↪format=S16LE,rate=48000,channels=1 ! fdsink fd=1"
      #   DC_PCM_SPLIT_FILTER_CONF_PATH="/dev/snd/by-path/platform-sound"
      command: /etc/dc_conf/scripts/gstreamer_pcm.sh "$(DC_PROCESS_SRC_CONF_COMMAND)" "$(DC_PCM_SPLIT_FILTER_
↪CONF_PATH)"

  - id: 2
    element: pcm-split-filter
    from: [ [1] ]
    conf:
      clock_id: CLOCK_MONOTONIC
      delay_ms: $(DC_PCM_SPLIT_FILTER_CONF_DELAY_MS)
      audio_element: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_ELEMENT)
      audio_iface: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_IFACE)
      audio_format: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_FORMAT)
      audio_rate: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_RATE)
      audio_channels: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_CHANNELS)
      audio_volume_iface: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_IFACE)
      audio_volume_element: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_ELEMENT)
      audio_volume_value: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_VOLUME_VALUE)
      audio_boost_element: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_BOOST_ELEMENT)
      audio_boost_value: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_BOOST_VALUE)
      audio_sync_period_s: $(DC_PCM_SPLIT_FILTER_CONF_AUDIO_SYNC_PERIOD_S)
      path: $(DC_PCM_SPLIT_FILTER_CONF_PATH)

  - id: 3
    element: print-log-filter
    from: [ [2] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
```

(次のページに続く)

(前のページからの続き)

```
      output: stderr

  - id: 4
    element: file-sink
    from: [ [3] ]
    conf:
      create: true
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.8 jpeg.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/jpeg.yml` は、V4L2 (Video for Linux 2) 経由で取得した MJPEG 形式の動画を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: v4l2-src
    conf:
      path: $(DC_V4L2_SRC_CONF_PATH)
      type: jpeg
      width: $(DC_V4L2_SRC_WIDTH)
      height: $(DC_V4L2_SRC_HEIGHT)
      fps: $(DC_V4L2_SRC_FPS)

  - id: 2
    element: jpeg-split-filter
    from: [ [1] ]

  - id: 3
    element: iscp-v2-compat-filter
    from: [ [2] ]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        # Starting from v2.2.0, it is required that 'jpeg' be specified as a map.
        jpeg:
          name: "jpeg"

  - id: 4
    element: print-log-filter
    from: [ [3] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 5
    element: file-sink
    from: [ [4] ]
    conf:
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.9 nmea.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/nmea.yml は、TTY デバイスから取得した NMEA データを送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: nmea-packet-src
    conf:
      path: $(DC_NMEA_PACKET_SRC_CONF_PATH)
      baudrate: $(DC_NMEA_PACKET_SRC_CONF_BAUDRATE)

  - id: 2
    element: iscp-v2-compat-filter
    from: [ [1] ]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule: nmea

  - id: 3
    element: print-log-filter
    from: [ [2] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 4
    element: file-sink
    from: [ [3] ]
    conf:
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.10 repeat_process_json.yml

device-connector-intdash とともにインストールされる /etc/dc_conf/repeat_process_json.yml は、プロセスから標準出力された JSON 文字列を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: repeat-process-src
    conf:
      command: $(DC_REPEAT_PROCESS_SRC_CONF_COMMAND)
      interval_ms: $(DC_REPEAT_PROCESS_SRC_CONF_INTERVAL_MS)

  - id: 2
    element: json-split-filter
    from: [ [1] ]

  - id: 3
    element: iscp-v2-compat-filter
    from: [ [2] ]
    conf:
      timestamp:
        stamp:
```

(次のページに続く)

(前のページからの続き)

```
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        string:
          name: $(DC_ISCP_V2_COMPAT_FILTER_CONF_CONVERT_RULE_STRING_NAME)

  - id: 4
    element: print-log-filter
    from: [ [3] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 5
    element: file-sink
    from: [ [4] ]
    conf:
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.11 repeat_process_string.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/repeat_process_string.yml` は、プロセスから標準出力された文字列を送信するためのパイプライン設定ファイルです。

```
tasks:
  - id: 1
    element: repeat-process-src
    conf:
      command: $(DC_REPEAT_PROCESS_SRC_CONF_COMMAND)
      interval_ms: $(DC_REPEAT_PROCESS_SRC_CONF_INTERVAL_MS)

  - id: 2
    element: iscp-v2-compat-filter
    from: [[1]]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        string:
          name: $(DC_ISCP_V2_COMPAT_FILTER_CONF_CONVERT_RULE_STRING_NAME)

  - id: 3
    element: print-log-filter
    from: [[2]]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 4
    element: file-sink
    from: [[3]]
    conf:
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.12 ubx.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/ubx.yml` は、u-blox GNSS モジュールからの情報（UBX プロトコルのメッセージ）を送信するためのパイプライン設定ファイルです。

```yaml
tasks:
  - id: 1
    element: ubx-src
    conf:
      # Exapmle of VTC1910: /dev/ttyS2
      # Example of EDGEPLANT T1: /dev/ttyTHS1
      path: $(DC_UBX_SRC_CONF_PATH)
      baud_rate: $(DC_UBX_SRC_CONF_BAUD_RATE)
      meas_rate_ms: $(DC_UBX_SRC_CONF_MEAS_RATE_MS)
      nav_rate: $(DC_UBX_SRC_CONF_NAV_RATE)
      high_nav_rate_hz: $(DC_UBX_SRC_CONF_HIGH_NAV_RATE_HZ)
      esf_status_rate: $(DC_UBX_SRC_CONF_ESF_STATUS_RATE)
      hnr_att_rate: $(DC_UBX_SRC_CONF_HNR_ATT_RATE)
      hnr_ins_rate: $(DC_UBX_SRC_CONF_HNR_INS_RATE)
      hnr_pvt_rate: $(DC_UBX_SRC_CONF_HNR_PVT_RATE)
      nav_status_rate: $(DC_UBX_SRC_CONF_NAV_STATUS_RATE)

  - id: 2
    element: ubx-iscpv2-filter
    from: [[1]]

  - id: 3
    element: iscp-v2-compat-filter
    from: [[2]]
    conf:
      timestamp:
        stamp:
          clock_id: CLOCK_MONOTONIC
      convert_rule:
        bytes: {}

  - id: 4
    element: print-log-filter
    from: [[3]]
    conf:
      duration_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 5
    element: file-sink
    from: [[4]]
    conf:
      path: $(DC_FILE_SINK_CONF_PATH)
```

## 47.13 v4lh264.yml

device-connector-intdash とともにインストールされる `/etc/dc_conf/v4lh264.yml` は、V4L2（Video for Linux 2）経由で取得した H.264 の動画を送信するためのパイプライン設定ファイルです。

```yaml
tasks:
  - id: 1
    element: v4l2-src
    conf:
      path: $(DC_V4L2_SRC_CONF_PATH)
      type: h264
      width: $(DC_V4L2_SRC_CONF_WIDTH)
      height: $(DC_V4L2_SRC_CONF_HEIGHT)
      fps: $(DC_V4L2_SRC_CONF_FPS)

  - id: 2
    element: h264-split-filter
    from: [ [1] ]
    conf:
      clock_id: CLOCK_MONOTONIC

  - id: 3
    element: print-log-filter
    from: [ [2] ]
    conf:
      interval_ms: 10000
      tag: $(DC_PRINT_LOG_FILTER_CONF_TAG)
      output: stderr

  - id: 4
    element: file-sink
    from: [ [3] ]
    conf:
      flush_size: 100
      path: $(DC_FILE_SINK_CONF_PATH)
```

# 48 独自エレメントの開発

device-connector-intdash に独自のエレメントを追加したい場合、プラグイン形式で開発します。プラグインの開発方法について、デバイスコネクター開発フレームワークの説明 (p. 182) を参照してください。

device-connector-intdash は Rust で開発されているため、プラグインの開発には Cargo ビルドシステムを使用し、エレメントの実装は Rust または C で行います。

## 48.1 プラグイン開発用プロジェクトの作成

device-connector-intdash 向けプラグインを開発する場合、以下の表を参照し、device-connector-intdash のパッケージバージョンに合ったテンプレートを使用してプロジェクトを作成してください。

| device-connector-intdash パッケージのバージョン | 本体の Rust バージョン | テンプレートのタグ |
|---|---|---|
| 2.0.0 以降 | 1.65.0 | v2.2.1 以降 |

プロジェクトは以下のコマンドで作成します。

```
cargo generate --git https://github.com/aptpod/device-connector-template.git --tag <テンプレートのタグ>
```

Rust または C で開発を行い、作成したプロジェクトにある Dockerfile を使用してクロスビルドすることで、プラグインを生成できます。詳細については、作成したプロジェクトの README.md を参照してください。

**重要:** device-connector-intdash パッケージを利用してプラグイン開発を行う場合、制限事項 (p. 210) により、Rust のバージョンを本体とプラグインで合わせる必要があります。

## 48.2 依存クレートの追加時の注意点

`Cargo.toml` の `[dependecies]` にクレートを追加してビルドした際に、以下のようなエラーが出る場合があります。

```
error: package `time v0.3.29` cannot be built because it requires rustc 1.67.0 or newer, while the currently␣
↪active rustc version is 1.65.0
Either upgrade to rustc 1.67.0 or newer, or use
cargo update -p time@0.3.29 --precise ver
where `ver` is the latest version of `time` supporting rustc 1.65.0
```

この場合、依存クレートのバージョンをクロスビルド環境の Rust バージョンでビルド可能なものに更新する必要があります。以下のコマンドでクレートのバージョンを更新してください。

例: time クレートのバージョンを Rust バージョン 1.65.0 でビルド可能なバージョン 0.3.23 に更新する

```
cargo update -p time@0.3.29 --precise 0.3.23
```

**注釈:** クレートのバージョンは crates.io で検索できます。クレートによっては、クレートのバージョンでサポートする Rust バージョンを確認できます。(例：time )

# 49 リリースノート

## 49.1 device-connector-intdash v2.2.0

⚠ **警告:** このバージョンから、以下のエレメントが送出する iscp-v2-compat-msg の Data Name フィールドの値が変わっています。

- h264-split-filter (p. 153)
- iscp-v2-compat-filter JPEG 用変換ルール (p. 156)
- pcm-split-filter (p. 161)

v2.2.0 未満 のバージョンでは、iscp-v2-compat-msg の Data Name フィールドは空文字でした。
v2.2.0 では、パイプラインのタスク設定で `name` に指定されている値が Data Name フィールドに書き込まれます。指定がない場合は以下のデフォルト値が使用されます。

| エレメント | `name` が未設定の場合に Data Name フィールドに設定されるデフォルト値 |
|---|---|
| h264-split-filter | `h264_annex_b` |
| iscp-v2-compat-filter JPEG 用変換ルール | `jpeg` |
| pcm-split-filter | `pcm` |

⚠ **警告:** このバージョンから、iscp-v2-compat-filter JPEG 用変換ルール（jpeg）(p. 156) の設定ファイル記載方法が変わっています。
新しい設定の記載方法の例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule:
    jpeg:
      name: ""
```

古い設定の記載方法の例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule: jpeg
```

古い記載方法ではエラーになります。必ず設定ファイルの更新を行なってください。

### 49.1.1 Breaking Changes

- iscp-v2-compat-msg を送出するエレメントは基本的にデータ名称を持つように変更しました

### 49.1.2 Features

- 新しいエレメントと設定ファイルを追加しました

# 50 デバイスコネクター開発フレームワーク概要

intdash Edge Agent 2 用デバイスコネクターの開発には Device Connector Framework を使用することができます。

- Device Connector Framework により作成されたデバイスコネクターでは、デバイスへの接続やデータ収集、収集したデータに対する処理は、それぞれ「エレメント (Element)」として定義されます。ユーザーは、エレメントを組み合わせて、データが流れるパイプラインを設定することができます。設定はパイプライン設定ファイルで行います。
- デバイスコネクターを実行すると、パイプライン設定ファイルで指定された各エレメントは、同期または非同期に動作する「タスク（Task)」として実行されます。また、タスク間でのメッセージパッシングはデバイスコネクターにより管理されます。

これにより、柔軟にエレメントを組み合わせて、データ（メッセージ）が流れるパイプラインを作成することができます。

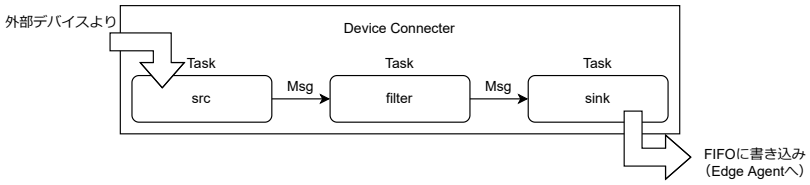



図 43　デバイスから intdash Edge Agent へ入力するパイプラインの例

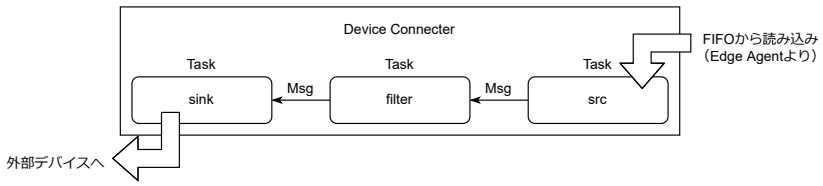



図 44　intdash Edge Agent からデバイスへ出力するパイプラインの例

なお、ファイルからの読み込み、ファイルへの書き出しなどの基本的な処理を行うエレメントは Device connector Framework に含まれています（基本エレメント)。

# 51 用語

Device Connector Framework で使用される用語について説明します。

## 51.1 メッセージ

タスク間で送受信されるデータです。

タスク間で送受信されるメッセージは、何らかの型を持ちます。型情報は、MIME タイプ（例：`MIME` タイプ `"image/jpeg"`）またはカスタムタイプ（例：カスタムタイプ `"iscp-v2-compat-msg"`）です。

各ポートがどの型のメッセージを受け取れるかは、エレメントにおいて定義されます。

送信元のタスクは、メッセージを最初に送信するとき、そのメッセージの型をデバイスコネクターに通知します。デバイスコネクターは、メッセージを受信するポートがその型を受け入れ可能かどうかを判定し、受け入れができない場合はエラーとします。

データを、MIME タイプ "image/jpeg"、MIME タイプ "text/plain"、カスタムタイプ "iscp-v2-compat-msg" 等で送出するエレメントの場合は、1 枚の画像、1 行のテキスト、1 つのデータポイントのように、意味を持つ単位を 1 つのメッセージとして送出します。

それに対し、MIME タイプ "application/octet-stream" で送出するエレメントの場合は、送出するデータは区切りのないバイナリストリームであるため、意味的な区切りとは関係なく送信上の都合でセグメントに分割して送信します。この場合のセグメントもここではメッセージと呼びます。

## 51.2 エレメント

メッセージの生成やメッセージの処理を定義したものです。デバイスコネクター内にメッセージを生成する「src エレメント」、受け取るだけの「sink エレメント」、送受信を共に行う「filter エレメント」があります。

エレメントは、実行時には「タスク」として実行されます。

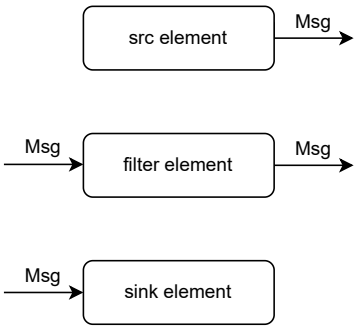



図 45　エレメントの種類（src、filter、sink）

デバイスコネクター内のパイプラインでは、以下の順序で処理が行われます。

- src エレメントにおいて、デバイスコネクターの外部（例えばセンサー）からデータを取得して、メッセージを生成
- filter エレメントにおいて、メッセージを意味のある単位に区切る、変換する、などの処理を行う
- sink エレメントにおいて、デバイスコネクターの外部（例えば intdash Edge Agent）にデータを出力する

**注釈:** アプトポッドにより提供されるエレメントは以下のような名前になっています。

- src エレメント: `*-src`
- filter エレメント: `*-filter`
- sink エレメント: `*-sink`

例えば、入力を jpeg のフレームごとのメッセージにするのは filter エレメントの役目で、jpeg-split-filter と名付けています。

## 51.3 タスク

エレメントにおいて定義された処理が実際に実行されるとき、その処理はタスクと呼ばれます。タスクは、メッセージの送受信が必要になったときに起動されます。

## 51.4 ポート

エレメントには、0 個以上 255 個以下の受信ポートと、0 個以上 255 個以下の送信ポートを定義することができます。複数のポートを使うことによって、ポートごとに別の型のメッセージを送受信させることができます。

- 複数の送信ポートからのメッセージを、1 つの受信ポートで受信することができます。
- 1 つの送信ポートから、複数の受信ポートに向けてメッセージを送信することはできません。

**注釈:** Device Connector Framework v1.0 現在、送受信ともに 0 番ポート 1 つのみを使用することができます。今後のリリースで、順次複数のポートに対応する予定です。

## 51.5 プラグイン

Device Connector Framework は、プラグインによるエレメントの追加をサポートしています。プラグインは共有ライブラリ (.so) の形式で開発し、実行時にロードされるようにします（ダイナミックリンク）。

# 52 デバイスコネクターのビルド

この章は、Device Connector Framework を使用してデバイスコネクターをビルドし、動作確認するチュートリアルです。ここでビルドされるデバイスコネクターには、基本エレメントが含まれます。基本エレメントについては、エレメント一覧 (Device Connector Framework) (p. 190) を参照してください。

## 52.1 開発環境を準備する

Device Connector Framework を使ってデバイスコネクターをビルドするには、Rust の開発環境が必要です。Rust の開発環境をインストールしてください。

## 52.2 基本エレメントのみでビルドする

Device Connector Framework には基本エレメントのソースコードが付属しています。

以下のコマンドを実行することにより、Device Connector Framework を取得し、基本エレメントを内部に含んだデバイスコネクターをビルドすることができます。

```
git clone https://github.com/aptpod/device-connector-framework.git
cd device-connector-framework
cargo build --release
```

生成された、`./target/release/device-connector-run` という実行ファイルが、基本エレメントを含むデバイスコネクターです。

## 52.3 パイプライン設定を作成する

Device Connector Framework により作成されたデバイスコネクターを使用するには、行いたい処理をパイプラインとして定義する必要があります。

例として、以下のパイプライン設定ファイル `conf.yml` をカレントディレクトリに用意します。ここでは、文字列を 100ms ごとに送出するエレメント (text-src) に、受信内容を標準出力に書き出すエレメント (stdout-sink) をつないで、パイプラインを作成しています。

```yaml
tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello, World!"
      interval_ms: 100

  - id: 2
    element: stdout-sink
    from: [[ 1 ]]
    conf:
      separator: "\n"
```

## 52.4 デバイスコネクターを実行する

以下のコマンドでデバイスコネクターを実行します。

```
./target/release/device-connector-run --config conf.yml
```

すると、パイプライン設定ファイルで指定した文字列が 100ms ごとに出力されます。

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

終了するには Ctrl-C を押してください。

以上で、デバイスコネクターの動作確認は終了です。

この例では文字列を標準出力に出力しましたが、代わりに、intdash Edge Agent 用のバイナリフォーマットで FIFO に出力し、intdash Edge Agent がその FIFO から読み取るようにすることでデータを渡すことができます。

# 53 パイプライン設定ファイル

パイプライン設定ファイルは、デバイスコネクターで行いたい処理をエレメントの組み合わせで定義したものです。

以下の設定例では、`text-src` エレメントにより定義されたタスクから 100ms ごとに“Hello, World!”という文字列を送出し、`stdout-sink` エレメントにより定義されたタスクがそれを受け取って標準出力に出力します。

また、プラグインファイル `libdc_my_plugin0.so` をロードするように設定しています。

```
runner:
  channel_capacity: 16

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_my_plugin0.so

tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello, World!"
      interval_ms: 100

  - id: 2
    element: stdout-sink
    from: [[ 1 ]]
    conf:
      separator: "\n"
```

設定項目の詳細は以下のとおりです。

## 53.1 runner の設定

パイプライン設定ファイルの `runner:` には、デバイスコネクターの実行パラメータを記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| channel_capacity | 整数値 | スレッド間のメッセージ送信に用いるチャンネルの容量（メッセージの個数）です。これを超える容量のメッセージを送信しようとすると、受信側でメッセージが処理されるまで送信側のスレッドは処理がブロックされます。省略した場合デフォルトの値が使用されます。 |

## 53.2 bg_processes の設定

```
bg_processes:
  - command: /background/process/path args
```

バックグラウンドで実行するコマンドを指定します。タスクの開始時に before_task の前に起動されます。

## 53.3 before_task の設定

```
before_task:
  - command1
  - command2
```

タスクの開始前に実行されるコマンドを指定します。bg_processes と異なり、起動されたプロセスが終了するまでタスクの開始はブロックされます。

## 53.4 after_task の設定

```
after_task:
  - command1
  - command2
```

デバイスコネクターの終了時、タスクの終了後に実行されるコマンドを指定します。

## 53.5 plugin の設定

パイプライン設定ファイルの plugin: には、実行時にロードするプラグインについて設定します。

独自エレメントをプラグイン形式で作成した場合、ここでそのファイルを指定します。独自エレメントについては、独自エレメントの開発 (p. 198) を参照してください。

| 項目 | 設定値 | 説明 |
|---|---|---|
| plugin_files | 文字列の配列 | ロードしたいプライグインファイル (例:libdc_xxx.so) へのパスのリストです。 |

## 53.6 tasks の設定

タスクの設定は、tasks: 内に並列に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| id | 整数値 | タスクに割り当てる ID です。これはこのパイプライン上で一意でなくてはなりません。 |
| element | 文字列 | このタスクに割り当てるエレメントを指定します。 |
| from | 文字列の二次元配列 | このタスクが受け取るメッセージの送信元を指定します。<br>`[["<送信元タスク ID>:<送信元ポート番号>"], ... ]`<br>メッセージは非同期に送られます。<br>例えば、`[[ "1:0" ], [ "2:1" ]]` は、以下を意味します。<br>• 0 番ポートが「タスク ID1 の 0 番ポート」からメッセージを受け取る<br>• 1 番ポートが「タスク ID2 の 1 番ポート」からメッセージを受け取る<br>ポート番号が 0 の場合、ポート番号を省略して `"1:0"` を `"1"` のように記述することができます。したがって、`[[ "1:0" ], [ "2:1" ]]` は `[[ "1" ], [ "2:1" ]]` と同義です。 |

**重要:** `from` に設定する送信元のタスク ID は、自身のタスク ID より小さい数字である必要があります。

エレメントによっては、独自の設定項目も存在します。エレメント独自の設定項目は、`conf:` に記述してください。

# 54 エレメント一覧 (Device Connector Framework)

Device Connector Framework には、パイプラインを作成するための基本的なエレメントが付属しています。

| | Device Connector Framework の基本エレメント |
|---|---|
| src エレメント | • file-src (p. 190)<br>• process-src (p. 191)<br>• repeat-process-src (p. 191)<br>• text-src (p. 192)<br>• tee-src (p. 192) |
| filter エレメント | • print-log-filter (p. 193)<br>• split-by-delimiter-filter (p. 193)<br>• split-by-fixed-size-filter (p. 194)<br>• stat-filter (p. 194)<br>• tee-filter (p. 195) |
| sink エレメント | • file-sink (p. 196)<br>• null-sink (p. 197)<br>• stdout-sink (p. 197) |

**注意:** device-connector-intdash には、これ以外のエレメントも付属しています。device-connector-intdash のエレメント一覧 (p. 147) を参照してください。

## 54.1 src エレメント

以下の src エレメントを使用できます（アルファベット順)。

### 54.1.1 file-src

Rust ドキュメント

指定されたパスのファイルを読み込み、内容をメッセージとして送出します。

**エレメントの種類**

src

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | 読み取るファイルのパス |

### 54.1.2 process-src

Rust ドキュメント

プロセスを起動し、起動したプロセスの標準出力をバッファサイズに収まるように区切って送出します。

**エレメントの種類**

src

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| program | 文字列 | 実行ファイル |
| args | 文字列の配列 | 実行ファイルに渡す引数 |
| command | 文字列 | 実行ファイルと引数を 1 文字列で指定したもの。program と args が指定されなかった場合に使用。 |
| buffer_size | 整数 | バッファサイズ (バイト)。デフォルトは 255。 |
| retry | boolean(`true` / `false`) | 実行ファイルが終了したときに再起動を行う（`true`）か行わない（`false`）か。デフォルトは行わない。 |
| retry_interval_ms | 小数 | 実行ファイルが終了したときに再起動を行う場合、何ミリ秒後に再起動するか。 |

### 54.1.3 repeat-process-src

Rust ドキュメント

プロセスを繰り返し起動し、そのプロセスの標準出力をメッセージとして送出します。1 度の起動で出力された標準出力が 1 つのメッセージになります。

**エレメントの種類**

src

**次のエレメントに送信するポートの数**

1

**送信ポートでの送信形式**

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| program | 文字列 | 実行ファイル |
| args | 文字列の配列 | 実行ファイルに渡す引数 |
| command | 文字列 | 実行ファイルと引数を 1 文字列で指定したもの。program と args が指定されなかった場合に使用。 |
| interval_ms | 小数 | 出力頻度をミリ秒にて指定 |

### 54.1.4 tee-src

Rust ドキュメント

tee-filter (p. 195) で複製されたメッセージを受け取り、それをそのまま送出します。

**エレメントの種類**
src

**次のエレメントに送信するポートの数**
1

**送信ポートでの送信形式**
MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf:` 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | 文字列 | どの tee-filter (p. 195) からのメッセージを受け取るかを識別するための文字列 |

### 54.1.5 text-src

Rust ドキュメント

指定されたテキストを送出します。

**エレメントの種類**
src

**次のエレメントに送信するポートの数**
1

**送信ポートでの送信形式**
MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf:` 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| text | 文字列 | メッセージのテキスト |
| interval_ms | 整数値 | メッセージ生成の間隔をミリ秒にて指定 |
| repeat | 整数値 | 「メッセージ送出のタイミングごとに、メッセージを何回送出するか」を指定します。デフォルトは 1 です例えば `interval_ms: 100` かつ `repeat: 5` の場合、100 ミリ秒経過ごとに 5 回メッセージを送出します。 |

## 54.2 filter エレメント

以下の filter エレメントを使用できます（アルファベット順）。

### 54.2.1 print-log-filter

Rust ドキュメント

受け取ったメッセージの数とサイズの情報をログとして出力します。ログの出力先としてデバイスコネクターのログまたは標準エラーを選択できます。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
任意

**送信ポートでの送信形式**
任意（受け取ったメッセージをそのまま送信ポートに書き出します）

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| interval_ms | 小数 | ログを出力するミリ秒単位の周期（例：1000）<br>例えば「1000」を指定すると、受け取ったメッセージ数と合計サイズを、1000 ミリ秒ごとにログとして出力します。 |
| tag | string | ログを識別できるように各行の先頭に出力されるタグ。<br>`connector1` と指定した場合は、以下のように出力されます。<br>`[connector1] 100 msgs, 1000 bytes` |
| output | string（`log-trace` / `stderr`） | ログの出力先を指定します。<br>`log-trace`: デバイスコネクターのログの仕組みを使用して出力します。ログレベルは `trace` となります。<br>`stderr`: 標準エラーに出力します。 |

### 54.2.2 split-by-delimiter-filter

Rust ドキュメント

受信したメッセージを、区切り文字で区切り、区切りごとにメッセージとして送出します。送出に区切り文字は含みません。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
任意

**送信ポートでの送信形式**
任意（受け取ったメッセージをそのまま送信ポートに書き出します）

エレメント独自の設定項目は以下のとおりです。conf：以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| delimiter | 文字列 | 区切り文字。デフォルトは改行（ \n ） |
| limit_size | 整数 | 区切り文字が検出されるまでに保持できるメッセージの長さ (バイト)。デフォルトは 1GiB |

### 54.2.3 split-by-fixed-size-filter

Rust ドキュメント

受信したメッセージを、指定された長さで区切り、区切りごとにメッセージとして送出します。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
任意

**送信ポートでの送信形式**
任意（受け取ったメッセージをそのまま送信ポートに書き出します）

エレメント独自の設定項目は以下のとおりです。conf：以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| size | 整数 | 1 メッセージの長さ (バイト) |

### 54.2.4 stat-filter

Rust ドキュメント

データを受け取った回数やサイズを確認するためのエレメントです。メッセージを受け取り、それをそのまま送出します。受け取ったメッセージの回数とサイズを記録し、その統計情報を標準エラー出力に書き出します。

**エレメントの種類**
filter

**前のエレメントから受信するポートの数**
1

**次のエレメントに送信するポートの数**
1

**受信ポートでの受信形式**
　任意

**送信ポートでの送信形式**
　受信ポートに入力されたデータをそのまま出力

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| interval_ms | 整数 | 出力頻度をミリ秒にて指定 |

### 54.2.5 tee-filter

Rust ドキュメント

受信したメッセージをそのまま送信ポートから送信します。その際にメッセージを複製します。複製されたメッセージは tee-src (p. 192) で使用することができます。

**エレメントの種類**
　filter

**前のエレメントから受信するポートの数**
　1

**次のエレメントに送信するポートの数**
　1

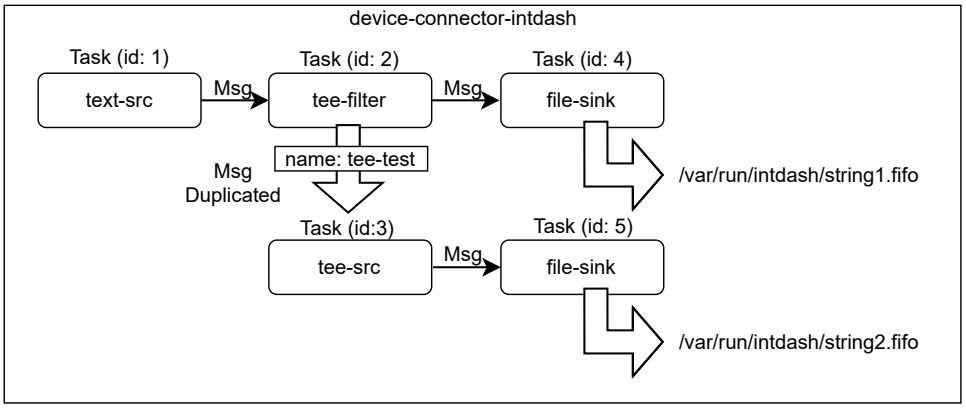
**受信ポートでの受信形式**
　任意

**送信ポートでの送信形式**
　任意（受け取ったメッセージをそのまま送信ポートに書き出します）

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| name | 文字列 | どの tee-src (p. 192) にメッセージを送るかを識別するための文字列 |
| channel_capacity | 文字列 | 送出するメッセージをバッファリングできる数 |

**設定例:**



```yaml
tasks:
  - id: 1
    element: text-src
    conf:
      text: hello
      interval_ms: 1000

  - id: 2
    element: tee-filter
    from: [[1]]
    conf:
      name: tee-test

  - id: 3
    element: tee-src
    conf:
      name: tee-test

  - id: 4
    element: file-sink
    from: [[2]]
    conf:
      path: /var/run/intdash/string1.fifo

  - id: 5
    element: file-sink
    from: [[3]]
    conf:
      path: /var/run/intdash/string2.fifo
```

## 54.3 sink エレメント

以下の sink エレメントを使用できます（アルファベット順）。

### 54.3.1 file-sink

Rust ドキュメント

指定されたパスのファイルに、受け取ったメッセージを書き出します。

**エレメントの種類**
: sink

**前のエレメントから受信するポートの数**
: 1

**受信ポートでの受信形式**
: 任意（受け取ったデータをそのままファイルに書き出します）

エレメント独自の設定項目は以下のとおりです。`conf`：以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| path | 文字列 | 出力先のファイルパス |
| create | true/false | true の場合、出力先のファイルがない場合は作成する。デフォルトは false |
| separator | 文字列 | メッセージ出力ごとに挿入するテキスト。デフォルトは空 |
| flush_size | 整数 | 入力をフラッシュするまでにバッファにためるデータ量 (バイト)。デフォルトは 0 |

### 54.3.2 null-sink

Rust ドキュメント

メッセージを受け取りますが、それを何にも使わずに破棄します。

**エレメントの種類**
sink

**前のエレメントから受信するポートの数**
1

**受信ポートでの受信形式**
任意

エレメント独自の設定項目はありません。

### 54.3.3 stdout-sink

Rust ドキュメント

受け取ったメッセージを標準出力に書き出します。

**エレメントの種類**
sink

**前のエレメントから受信するポートの数**
1

**受信ポートでの受信形式**
任意

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

| 項目 | 設定値 | 説明 |
|---|---|---|
| separator | 文字列 | メッセージ出力ごとに挿入するテキスト。デフォルトは空 |
| flush_size | 整数 | 入力をフラッシュするまでにバッファにためるデータ量 (バイト)。デフォルトは 0 |

# 55 独自エレメントの開発

Device Connector Framework で開発されたデバイスコネクターに機能を追加したい場合、Rust または C で独自エレメントを開発し、パイプラインに組み込むことができます。

## 55.1 独自エレメントの開発方法

独自エレメントを使用する方法には、以下の 2 つがあります。



### 55.1.1 プラグインとして作成する

開発した独自エレメントを共有ライブラリ (.so) の形式でビルドし、それをデバイスコネクター実行時にプラグインとして読み込んで使用する方法です。1 つのプラグインには複数のエレメントを含むことができます。

プラグインを使用する場合は、パイプライン設定ファイルの `plugin.plugin_files` で以下のようにプラグインファイル名の配列で指定します。これにより、プラグインが読み込まれ、プラグイン内の独自エレメントを使用できるようになります。

```yaml
runner:
  ..

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_xxxxxx.so
    - /path/to/my/plugin1/libdc_yyyyyy.so

tasks:
  - id: 1
    element: hello-src # defined in loaded plugin
  ..
```

独自エレメントをプラグインにする場合、開発は Rust または C で行います。

詳細については Rust による独自エレメント開発 (p. 199) または C による独自エレメント開発 (p. 205) を参照してください。

### 55.1.2 デバイスコネクターの実行ファイルに含める

デバイスコネクター本体をビルドする際に、独自エレメントも含めた実行ファイルとしてビルドする方法です。この場合、デバイスコネクターの実行ファイル単体で動作しますので、パイプライン設定ファイルの `plugin.plugin_files` を記述する必要はありません。ただし、独自エレメントを修正するたびにデバイスコネクターの本体ごと再ビルドが必要になります。

独自エレメントをデバイスコネクターの実行ファイルに含める場合、開発は Rust で行います。

詳細については Rust による独自エレメント開発 (p. 199) を参照してください。

## 55.2 Rust による独自エレメント開発

Rust で独自エレメントを開発する手順は、以下のとおりです。

Rust で開発したエレメントは プラグインとして使用する (p. 198) ことも、デバイスコネクターの実行ファイルに含める (p. 199) ことも可能です。

どちらの場合も、以下の エレメントをプラグインにする／実行ファイルに含める (p. 203) の前までの手順は同じです。

### 55.2.1 準備

エレメントの開発のために、cargo-generate 用のテンプレート が用意されています。

> **注釈:** cargo-generate は、既存の Rust プロジェクトをテンプレートにして新しいプロジェクトを作成するためのツールです。インストールしていない場合、次のコマンドでインストールしてください。
>
> ```
> cargo install cargo-generate
> ```

次のコマンドでデバイスコネクターの新しいプロジェクトを作成します。新しいプロジェクトの名前の入力を求められるので、任意の名前を入力してください。

```
cargo generate --git https://github.com/aptpod/device-connector-template.git
```

これで新しいプロジェクトが作成されます。

このプロジェクトには、以下の 2 つのエレメントのソースコードがあらかじめ含まれています。

- `src/hello.rs` (`hello-src`)
- `src/hexdump.rs` (`hexdump-sink`)

### 55.2.2 `ElementBuildable` トレイトの利用

Rust では、型に何らかの属性や実装を付与するためにトレイトと呼ばれる機能が提供されています。独自エレメントを定義するには、そのエレメントにおいて `ElementBuildable` トレイトを実装する必要があります。

`ElementBuildable` トレイトの定義は以下のとおりです。

```rust
pub trait ElementBuildable: Sized + Send + 'static {
    type Config: DeserializeOwned;
    const NAME: &'static str;
    const RECV_PORTS: Port = 0;
    const SEND_PORTS: Port = 0;
    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        Vec::new()
```

(次のページに続く)

(前のページからの続き)

```
    }
    fn new(conf: Self::Config) -> Result<Self, Error>;
    fn next(&mut self, pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult;
    fn finalizer(&mut self) -> Result<Option<ElementFinalizer>, crate::error::Error>;
}
```

**Config**
: このエレメントを構築するのに必要な設定項目を指定します。デシリアライズ可能なデータ型が指定できますが、一般的には `#[derive(Debug, Deserialize)]` を指定した構造体にします。設定を受け取らない場合、`device_connector::EmptyElementConf` を指定します。

**NAME**
: エレメントの名前を指定します。他のエレメントとの重複は許されません。他のエレメントの名前と重複していると、実行時にエラーになります。

**RECV_PORTS**
: メッセージを受け取るポートの数です。src エレメントの場合は 0 を指定します。filter または sink エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

**SEND_PORTS**
: メッセージを送出するポートの数です。sink エレメントの場合は 0 を指定します。src または filter エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

**acceptable_msg_types()**
: 受け取ることのできるメッセージの型情報の配列を返す関数です。メッセージを受け取らない src エレメントの場合は実装する必要はありません。

**new()**
: `Config` で指定した設定情報を受け取り、エレメントを実際に構築して返すメソッドです。

**next()**
: データを受け取るための `pipeline` を受け取り、エレメントの実行結果を返します。`next()` は、`MsgBuf` に有効なデータを書き出すか終了するまで、処理を返さないように実装します。

**finalizer()**
: プロセス終了時に実行されるクロージャを返すメソッドです。このクロージャではエレメント固有の終了処理を定義します。このメソッドを実装しない場合、終了処理を行わない挙動（デフォルト）となります。

### 55.2.3 HelloSrcElement を作る

実例として、device-connector-template に含まれているサンプルのエレメント `HelloSrcElement` を見ていきます。

このエレメントは、一定時間ごとにテキストを送信します。

```
use device_connector::{error::Error, ElementBuildable, ElementResult, MsgType, Pipeline, Port};
use serde_derive::Deserialize;
use std::io::Write;
use std::thread::sleep;
use std::time::Duration;

// ElementBuildable を実装するための対象となる型
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
}

// HelloSrcElement が受け取る設定の定義
```

(次のページに続く)

(前のページからの続き)

```
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}

// ElementBuildable の実装
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;

    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes())?;

        Ok(ElementValue::MsgBuf)
    }
}
```

まず、本体である HelloSrcElement を定義します。

```
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
}
```

エレメントを実装する場合、実行に必要なデータを構造体のメンバとして持ちます。ここでは、送りたいテキストを含む HelloSrcElementConf をメンバとして持ちます。

HelloSrcElementConf は、エレメントを構築・実行するための設定を持ちます。

```
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}
```

構造体の宣言に付随する各属性は、パイプライン設定ファイルからのデシリアライズを可能にするためのものです。基本的に、この例の通りの属性を付与すれば問題ありません。

エレメントを構築可能にするために、ElementBuildable を実装します。

```
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;
```

(次のページに続く)

(前のページからの続き)

```
    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    ..
}
```

Config にはさきほど定義した HelloSrcElementConf を、NAME にはエレメントの名前である "hello-src" を指定します。この名前はパイプライン設定ファイルにおいてエレメントを指定する時に使われるもので、他のエレメントと重複しないものを選ぶ必要があります。

このエレメントは 1 つの送信ポートを持つので、SEND_PORTS に 1 を指定します。

new() には、受け取った設定 (HelloSrcElementConf) を元に、HelloSrcElement を生成するためのコードを記述します。

new() は HelloSrcElementConf 型の値 conf（パイプライン設定ファイルからパースされた設定）を受け取り、その設定に基づいて HelloSrcElement を作成して返却します。

filter エレメントまたは sink エレメントを実装する場合、どの型のメッセージを受け取れるかを示すために acceptable_msg_types() を実装しなければなりません。実装しない場合、どのようなデータも受け取れないエレメントとなります。

最後に、HelloSrcElement がメッセージを生成するためのメソッド next を定義します。

```
impl ElementBuildable for HelloSrcElement {
    ..

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes())?;

        Ok(ElementValue::MsgBuf)
    }
}
```

src エレメントとして実装するために、next() メソッドの内部を記述していきます。

```
pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;
```

送出メッセージの型情報を pipeline.check_send_msg_type に渡し、受信側で受け取りができるのかを判定します。このエレメントはテキストを送出するので、"text/plain" という MIME 情報を渡します。

```
sleep(Duration::from_millis(100));
```

データ量を適当な量に制限するため、ここでは 100ms の間 sleep します。

```
let mut buf = pipeline.msg_buf(0);
buf.write_all(self.conf.text.as_bytes())?;

Ok(ElementValue::MsgBuf)
```

設定で与えられたテキストをメッセージにするために、まずは Pipeline::msg_buf() を呼び出して、MsgBuf 型のバッファを用意します。

Pipeline::msg_buf() では、メッセージの送信に用いるポート番号を引数にします（大抵の場合は 0 を指定します）。

MsgBuf は std::io::Write を実装しているため、write_all() を用いてバッファにテキストを書き込みます。

関数の最後の Ok(ElementValue::MsgBuf) は、MsgBuf に書き込んだデータが送信先のタスクに送信されるよう指定するためのものです。

返却されたメッセージは、エレメントが独立したスレッドで動作している場合、関連付けられたタスクへチャンネルを用いて送信されます。エレメントが同期的に呼び出されている場合、呼び出し元のタスクの実行へ戻ります。

### 55.2.4 エレメントをプラグインにする／実行ファイルに含める

Rust で実装したエレメントを動作可能にする方法は、プラグインにする場合と、デバイスコネクターの実行ファイルに含める場合とで異なります。

#### プラグインにする場合

実装したエレメントをプラグインにする場合、define_dc_load!() マクロを使用します。例として、テンプレートの src/lib.rs を見てみます。

```
mod hello;

// Implement plugin interface.
device_connector::define_dc_load!(hello::HelloSrcElement);
```

ここでは、さきほど実装した HelloSrcElement をプラグインに登録しています。define_dc_load!() マクロには、複数のエレメントを渡すことも可能です。プラグインを開発したい場合、このような記述を lib.rs に記述します。

#### デバイスコネクターの実行ファイルに含める場合

Rust で実装したエレメントをデバイスコネクターの実行ファイルに含めるには、ElementBank に登録します。

ElementBuildable を実装した型を append_from_buildable() により登録し、その後に Runner を構築することで、エレメントが利用できるようになります。

例として、テンプレートの src/main.rs は以下のようになっています。

```
fn main() -> Result<()> {
    // ログシステムの初期化
    env_logger::init();

    // パイプライン設定ファイルの読み込み
    let opts: Opts = Opts::parse();
    let conf = Conf::read_from_file(&opts.config)?;

    // ElementBank の作成
    let mut bank = ElementBank::new();

    // プラグインのロード
    let loaded_plugin = LoadedPlugin::from_conf(&conf.plugin)?;

    // 実装した HelloSrcElement の登録
    bank.append_from_buildable::<hello::HelloSrcElement>();

    // runner の作成
```

（次のページに続く）

(前のページからの続き)

```
    let mut runner_builder = RunnerBuilder::new(&bank, &loaded_plugin, &conf.runner);
    runner_builder.append_from_conf(&conf.tasks)?;
    let runner = runner_builder.build()?;

    // 起動
    runner.run()?;

    Ok(())
}
```

これは通常の Rust の main 関数であるため、ここに追記することでデバイスコネクターが動作する前後の挙動をカスタマイズすることも可能です。

### 55.2.5 ビルドする

テンプレートを使用した場合、実装した拡張は以下のコマンドでビルドできます。

```
cargo build --release
```

これにより、target/release に、以下の 2 種類の成果物が生成されます。いずれかを使用してください。

- libdc_xxxxxx.so：HelloSrcElement を含むプラグインファイル（共有ライブラリ）。使用するには、パイプライン設定ファイルの plugin.plugin_files で、プラグインファイル名を指定してください（例 (p. 198)）。
- xxxxxx-run：HelloSrcElement を含むデバイスコネクターの実行ファイル。このデバイスコネクターを実行すれば、プラグインなしで HelloSrcElement を使用できます。plugin.plugin_files の設定は必要ありません。

### 55.2.6 （補足）sink エレメントの例

上の説明では例として src エレメントを作成しましたが、sink エレメントを作成する場合、例えば以下のようになります。

このエレメント（hexdump-sink）では、受け取った各メッセージを標準エラーに出力します。

> **注釈:** このエレメント（hexdump-sink）は device-connector-template に含まれます。

```
use device_connector::EmptyElementConf;
use device_connector::{
    ElementBuildable, ElementResult, Error, MsgReceiver, MsgType, Pipeline, Port,
};

pub struct HexdumpSinkElement {}

impl ElementBuildable for HexdumpSinkElement {
    type Config = EmptyElementConf;

    const NAME: &'static str = "hexdump-sink";
    const RECV_PORTS: Port = 1;
    const SEND_PORTS: Port = 0;

    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        vec![vec![MsgType::any()]]
    }
```

(次のページに続く)

(前のページからの続き)

```
    fn new(_conf: Self::Config) -> Result<Self, Error> {
        Ok(Self {})
    }

    fn next(&mut self, _pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult {
        loop {
            let msg = receiver.recv(0)?;
            let bytes = msg.as_bytes();
            eprintln!(
                "msg ={}",
                bytes
                    .iter()
                    .map(|x| format!(" {:02X}", x))
                    .collect::<String>()
            );
        }
    }
}
```

## 55.3 C による独自エレメント開発

Device Connector Framework では、Rust または C で独自エレメントを開発し、パイプラインに組み込むことができます。ここでは、C インターフェイスを使って独自エレメントを開発する方法を説明します。

C で開発する場合は、独自エレメントはプラグインとして作成します。

### 55.3.1 準備

C による開発を行うためには、以下の 2 つのファイルが必要です。

- Device Connector Framework のリポジトリに含まれる インクルードファイル `common/include/device_connector.h`
- 静的リンク用のライブラリファイル `libdevice_connector_common.a` 。以下のコマンドによりビルドしてください。

  ```
  git clone https://github.com/aptpod/device-connector-framework.git
  cd device-connector-framework
  cargo build -p device-connector-common --release
  ```

  このコマンドにより、`target/release` 以下に `libdevice_connector_common.a` が生成されます。
  これを任意のディレクトリにコピーしてください。

  ```
  cp target/release/libdevice_connector_common.a </path/to/library_dir>
  ```

### 55.3.2 エレメントを実装する

例として "example-plugin" という src エレメントを実装します。このエレメントは設定を持たず、1 秒間隔で hello, world from plugin という文字列のメッセージを送信します。

以下のコードブロックを example_plugin.c という名前のファイルとして保存してください。

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <device_connector.h>

#define N_ELEMENT 1
#define PLUGIN_NAME "example-plugin"

typedef struct {
    const char* text;
} ExamplePlugin;

void *example_plugin_new(const char *config);
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver);
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer);
void example_plugin_free(void* element);

bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);

    plugin->version = "0.1.0";
    plugin->n_element = N_ELEMENT;

    DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);

    // Element settings
    elements[0].name = PLUGIN_NAME;
    elements[0].recv_ports = 0;
    elements[0].send_ports = 1;
    elements[0].acceptable_msg_types = NULL;
    elements[0].config_format = "json";
    elements[0].new_ = example_plugin_new;
    elements[0].next = example_plugin_next;
    elements[0].finalizer = example_plugin_finalizer;
    elements[0].free = example_plugin_free;

    plugin->elements = elements;
    return true;
}

void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}

DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;

    if (!dc_pipeline_send_msg_type_checked(pipeline)) {
```

(次のページに続く)

(前のページからの続き)

```
     DcMsgType msg_type;
     if (dc_msg_type_new("mime:text/plain", &msg_type)) {
         dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
     }
    }

    sleep(1);

    DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

    const uint8_t *data = (const uint8_t *)example_plugin->text;
    const size_t len = strlen(example_plugin->text);
    dc_msg_buf_write(msg_buf, data, len);

    return DcElementResult_MsgBuf;
}

bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {
    return true;
}

void example_plugin_free(void* element) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
    free(example_plugin);
}
```

以下、このソースコードについて解説していきます。

```
#include <device_connector.h>
```

プラグイン開発のために device_connector.h をインクルードします。

```
typedef struct {
    const char* text;
} ExamplePlugin;
```

エレメントの本体を定義します。

```
bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);
```

プラグインをロードする時に呼び出される dc_load 関数を定義します。最初に、dc_init にプラグインの名称を渡します。

```
plugin->version = "0.1.0";
plugin->n_element = N_ELEMENT;
```

このプラグインがターゲットとするデバイスコネクターのバージョンと、読み込ませたいエレメントの数を指定します。

```
DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);
```

DcElement の配列を malloc で用意します。この長さは読み込ませたいエレメントの数 (N_ELEMENT) と同じです。

```
elements[0].name = PLUGIN_NAME;          // エレメントの名前（実行時に他のエレメントと重複しているとエラーになる）
elements[0].recv_ports = 0;              // 受信用ポートの数
elements[0].send_ports = 1;              // 送信用ポートの数
```

(次のページに続く)

(前のページからの続き)

```
elements[0].acceptable_msg_types = NULL; // 受け取ることのできるデータ型。何も受信しない場合は NULL
elements[0].config_format = "json";      // 設定フォーマット (json or yaml)。このフォーマットが new_に指定した
関数に渡される
elements[0].new_ = example_plugin_new;   // エレメントを生成する関数
elements[0].next = example_plugin_next;  // エレメントを実行する関数
elements[0].finalizer = example_plugin_finalizer;  // プロセス終了時呼び出されるファイナライザを設定する関数
elements[0].free = example_plugin_free;  // エレメントを終了・解放する関数
```

エレメントの詳細を定義します。

```
    plugin->elements = elements;
    return true;
}
```

plugin->elements に DcElement の配列を設定し、dc_load が成功したら true を返します。

```
void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}
```

エレメントを生成する関数です。config には、パイプライン設定ファイルに記述されたのエレメントの設定（conf フィールドの値）が、config_format で指定したフォーマットに変換され文字列として渡されます。（この例では config の値は使用していません。）

ExamplePlugin のための領域を malloc で確保し、初期化後に void ポインタとして返します。失敗時には NULL を返却します。

```
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
```

example_plugin_next はエレメントを実行するための関数です。受け取った element を ExamplePlugin * にキャストします。

```
if (!dc_pipeline_send_msg_type_checked(pipeline)) {
    DcMsgType msg_type;
    if (dc_msg_type_new("mime:text/plain", &msg_type)) {
        dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
    }
}
```

dc_pipeline_send_msg_type_checked() で送信するメッセージの型チェックが行われているか調べ、行われていなければ、DcMsgType を作成して dc_pipeline_check_send_msg_type() に渡します。

```
sleep(1);

DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

const uint8_t *data = (const uint8_t *)example_plugin->text;
const size_t len = strlen(example_plugin->text);
dc_msg_buf_write(msg_buf, data, len);
```

1 秒間スリープした後、msg_buf を取得し、送信したいデータを dc_msg_buf_write() で書き込みます。ここで書き込むのは example_plugin_new で設定したテキストです。

```
    return DcElementResult_MsgBuf;
}
```

DcElementResult_MsgBuf を返し、msg_buf に書き込んだデータを送信することを示します。

```
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {
    return true;
}
```

プロセス終了時に呼び出されるファイナライザを登録するための関数です。ここでは特に何も行いませんが、プロセス終了時にエレメントが占有するリソースを解放する必要がある場合、ファイナライザに記述します。

```
void example_plugin_free(void* element) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
    free(example_plugin);
}
```

終了処理を記述します。ここでは malloc() で確保した領域を free() に渡すだけです。

### 55.3.3 コンパイルする

上記の example_plugin.c を、GCC でコンパイルするには以下のコマンドを実行します。

```
gcc -I/include_dir -Wall -O2 -fPIC -shared -L/library_dir \
    -o libdc_example_plugin.so example_plugin.c -ldevice_connector_common
```

-I オプションで、device_connector.h ファイルが含まれるディレクトリを、-L オプションで、libdevice_connector_common.a ファイルが含まれるディレクトリを指定してください。

これにより、プラグインファイル libdc_example_plugin.so が生成されます。使用するには、パイプライン設定ファイルの plugin.plugin_files で、プラグインファイル名を指定してください（例 (p. 198)）。

# 56 制限事項

Device Connector Framework には、以下のような制限事項があります。

**エレメント利用者向け**

- タスクの設定において、メッセージの送信元のタスク `from` を設定するときは、送信元のタスク ID は自身のタスク ID より小さい数字である必要があります。
- 番号が 1 以上の送信ポートについては未実装です。

**独自エレメントの開発者向け**

- 本体とプラグインで、利用するメモリアロケータが異なる場合の動作は未定義です。この問題は、Rust の将来のバージョンでメモリアロケータを固定化できれば回避できます。メモリアロケータの設定を故意に変えなければ問題は起こりません。
- Rust の `Vec` と `String` を、プラグインの FFI（Foreign Function Interface）間で受け渡しているため、もしこの ABI（Application Binary Interface）に変更があれば、コンパイラのバージョンを本体とプラグインで合わせておかないと、プラグインが壊れる可能性があります。