EC Protocol Bridge Developer Guide

Release 3.0

Contents

- Overview and High-level Architecture
- Plug-in Technical Overview
- Plug-in Data Inputs & Outputs
- Data Exchange Examples
- Plug-in Development
- Sample Plug-in Code

Overview and High-level Architecture

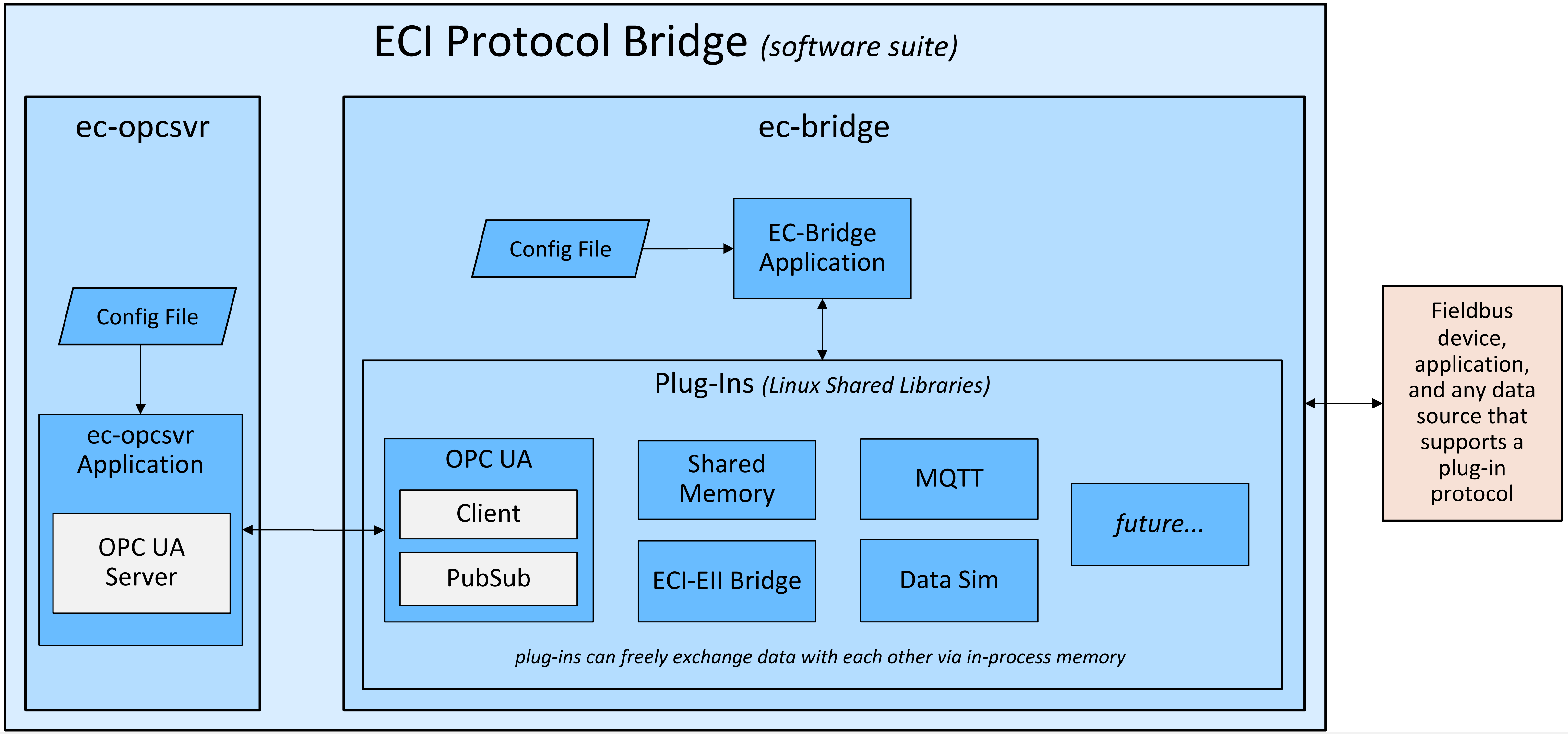
Protocol Bridge Overview

The purpose of the Protocol Bridge is to provide a real-time, deterministic data pipeline. Data is typically input from a source having a particular data protocol and output to destination having a different data protocol. For instance, the data may be input from an OPC UA server and output as an MQTT publication.

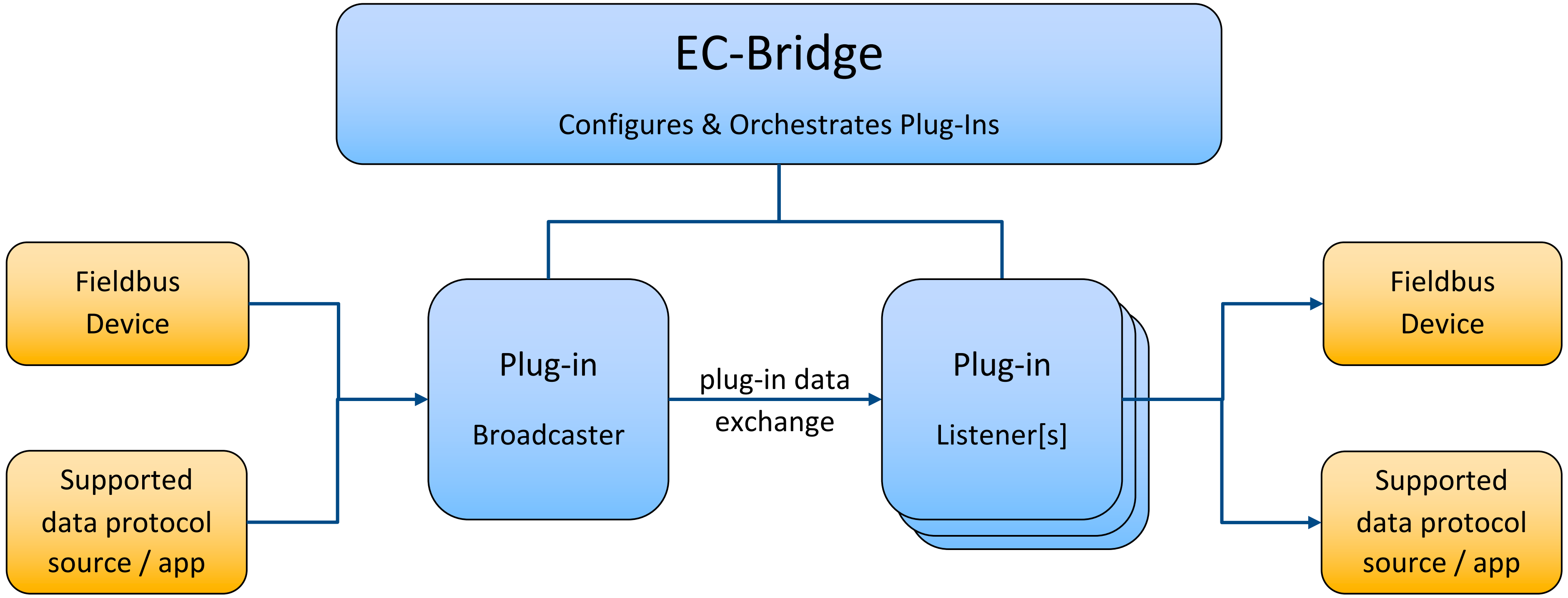
The Protocol Bridge application used a plug-in design. Its Manager provides a framework which dynamically loads and configures plug-in modules. These plug-ins extend the base functionality of the application. Typically, each plug-in provides capabilities for a particular data protocol. For instance, one plug-in provides OPC UA, another provides MQTT, and so on. Since plug-ins are specifically designed to exchange data with each other, transformation of data from one protocol to another is very simple and efficient.

The behavior of the Protocol Bridge is controlled by a configuration file. It is not necessary for an end-user to write any code. When executing the application, the name of a configuration file is provided as a command-line argument, ex. *ec-bridge config.yaml*. The configuration file specifies which plug-ins are loaded, the runtime parameters for each, and how data is routed from “broadcast” plug-ins to “listener” plug-ins. Multiple configuration files may be created for different use cases.

Protocol Bridge Architecture Overview

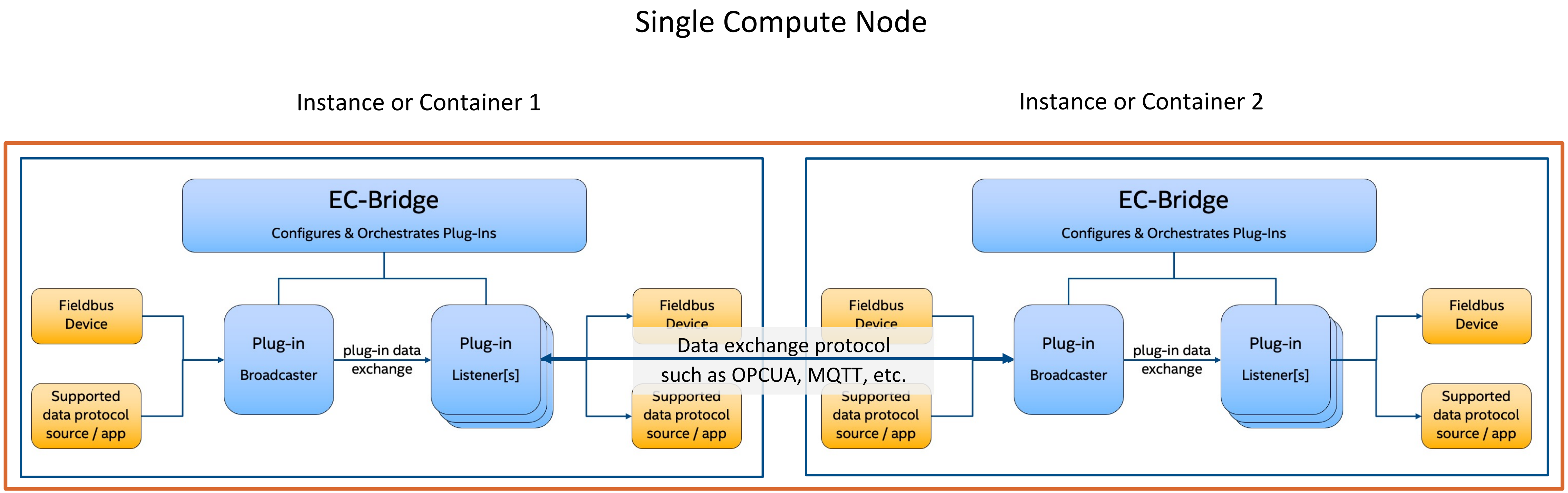

ECI Protocol Bridge *(software suite)*
ec-opcsvr
Config File
ec-opcsvr Application
OPC UA Server
ec-bridge
Config File
EC-Bridge Application
Plug-Ins *(Linux Shared Libraries)*
OPC UA
Client
PubSub
Shared Memory
MQTT
future...
ECI-EII Bridge
Data Sim
plug-ins can freely exchange data with each other via in-process memory
Fieldbus device, application, and any data source that supports a plug-in protocol

EC-Bridge Architecture Detail


EC-Bridge
Configures & Orchestrates Plug-Ins
Fieldbus Device
Supported data protocol source / app
Plug-in Broadcaster
plug-in data exchange
Plug-in Listener[s]
Fieldbus Device
Supported data protocol source / app

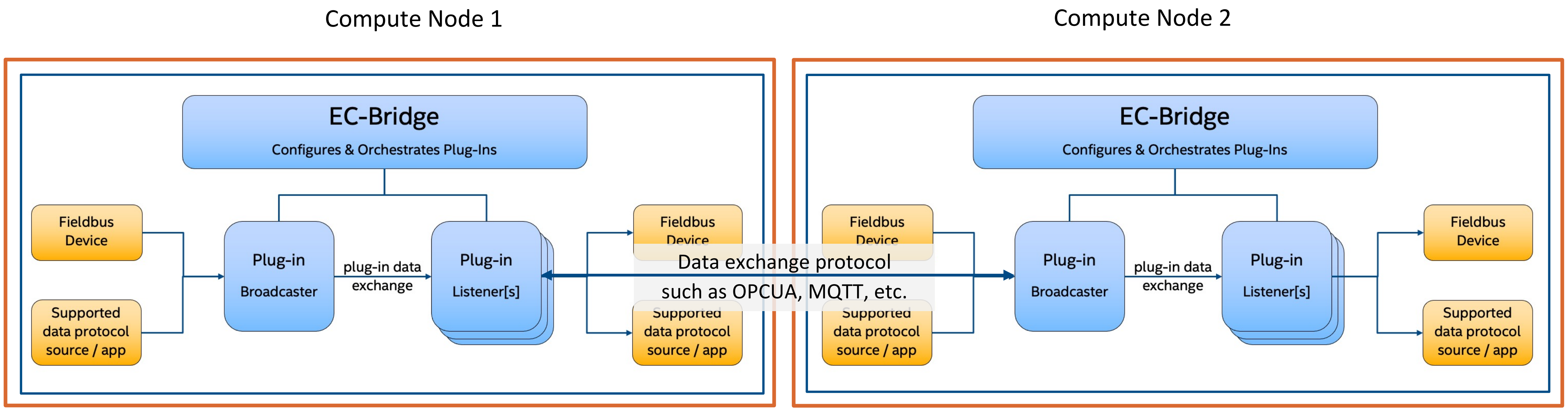
- Supported data protocols implemented as plug-ins
- Data flows defined via configuration file
- Zero-code implementation for end-user
- Multi-threaded, deterministic, soft real-time data exchange

Comms between Instances or Containers



Comms between Compute Nodes/Devices

Multiple Compute Nodes


Compute Node 1
Compute Node 2
EC-Bridge
Configures & Orchestrates Plug-Ins
Fieldbus Device
Supported data protocol source / app
Plug-in Broadcaster
plug-in data exchange
Plug-in Listener[s]
Data exchange protocol such as OPCUA, MQTT, etc.

Plug-in Technical Overview

Developing a Plug-in Overview

At a high-level, developing a Protocol Bridge plug-in includes the following steps:

- Design the plug-in to adapt its execution dynamically at runtime from parameters received from a configuration file.
- Design its execution to perform well in a real-time, multi-threaded environment.
- Design the code to follow the API callback lifecycle:
  - plg_initialize()
    - Read in and parse the configuration parameters supplied to this function from a specified configuration file.
    - Validate the parameters are correct individually and make sense collectively together. If not return an error code to prevent further execution.
    - Dynamically configure the plug-in based upon these parameters, allocating resources as needed.
  - plg_start()
    - Allocate any remaining resources that are needed, establish any necessary connections, etc.
    - Launch a real-time thread that will input data externally from a data source. That may include for instance reading from a server or subscribing to a service. Broadcast that data, using functions supplied by the framework, to make it available to other plug-ins to consume.
    - Launch a real-time thread to listen for data being received from other plug-ins, using functions supplied by the framework to read and parsed it, and output it externally. That may include for instance writing to a server or publishing to a service.
    - Launch any other required threads and begin processing until informed by the framework to stop execution.
  - plg_stop()
    - Stop all processes that were launched in plg_start() and release resources and connections.
  - plg_terminate()
    - Final cleanup of any remaining resources and prepare for termination of plug-in execution.
- Create one or more configuration files for thoroughly testing the plug-in. Validate that data is input and output from external sources as expected, is properly exchanged with other plug-ins and meets all performance benchmarks.

Datasets and Fields

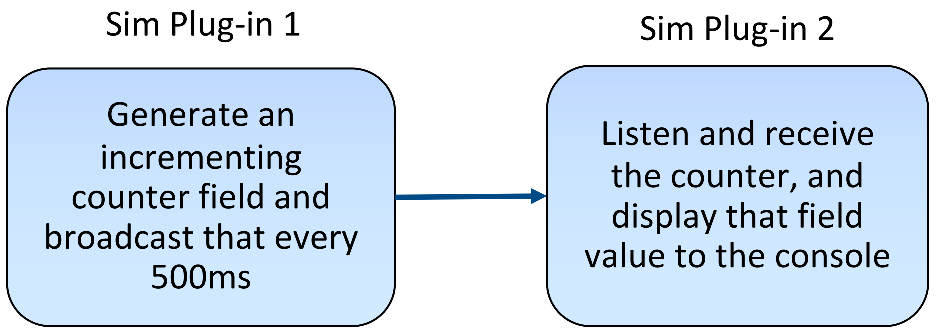
A Dataset, along with the Fields it contains, define a fundamental unit of data for all ingest, export and exchange operations. Each field within a dataset is given a unique identifier and a defined datatype. The number of fields and their position within a Dataset is fixed. Data is ingested and exported in this order as well as packed into internal Data Buffers for exchange from on Plug-in to another. This uniformity form a “contract” that every Plug-in can rely upon and what makes data exchange straight forward and easy to configure.

The example below show a configuration file snippet that contains the definition of a dataset, named ds1, along with two data fields, a 32-bit integer and a float, named fld1 and fld2 respectively.

```
dataset:
  -
    dataset-id: ds1
    dataset-fields:
      -
        datafld-id: fld1
        datatype: int32
      -
        datafld-id: fld2
        datatype: float
```

Config file to exchange data between two plug-ins

```
settings:
  log-level: LOG_DEBUG
plugins:
  -
    plugin-id: plg-sim
    filename: libplgsim.so
    dataset:
      -
        dataset-id: sim-ds
        dataset-fields:
          -
            datafld-id: counter
            datatype: int32
    configuration:
      interval-us: 500000
  -
    plugin-id: plg-sim2
    filename: libplgsim.so
    dataset:
      -
        dataset-id: sim-ds2
        listener-dataset-id: sim-ds
    configuration:
```

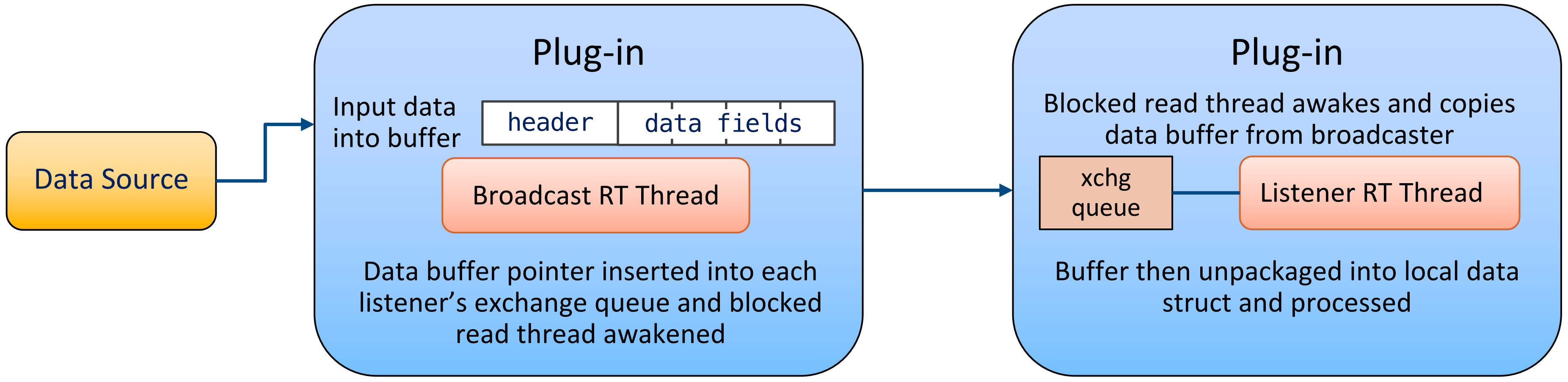

Sim Plug-in 1
Generate an incrementing counter field and broadcast that every 500ms
Sim Plug-in 2
Listen and receive the counter, and display that field value to the console

Notes:

- For details of available configuration parameters for each plug-in, please reference the main ECI documentation, section → Components and Features of ECI → Industrial Protocols & Bridging Communication → Edge-Control Protocol Bridge
- For information on other sample configuration files, please reference the main ECI documentation, section → Components and Features of ECI → Industrial Protocols & Bridging Communication → EC Protocol Bridge Example Configurations

Plug-in Deterministic Data Exchange

A plug-in shares data with another by packaging that data into data buffer, queueing it and then signaling each listener plug-in that data is available by unblocking its read thread. That data is received, a local copy is made, and it's then unpackaged and processed. On a properly tuned preempt-RT platform the latency and jitter numbers for this data exchange are predictable but will vary based on many factors and can be as low as 5µs.

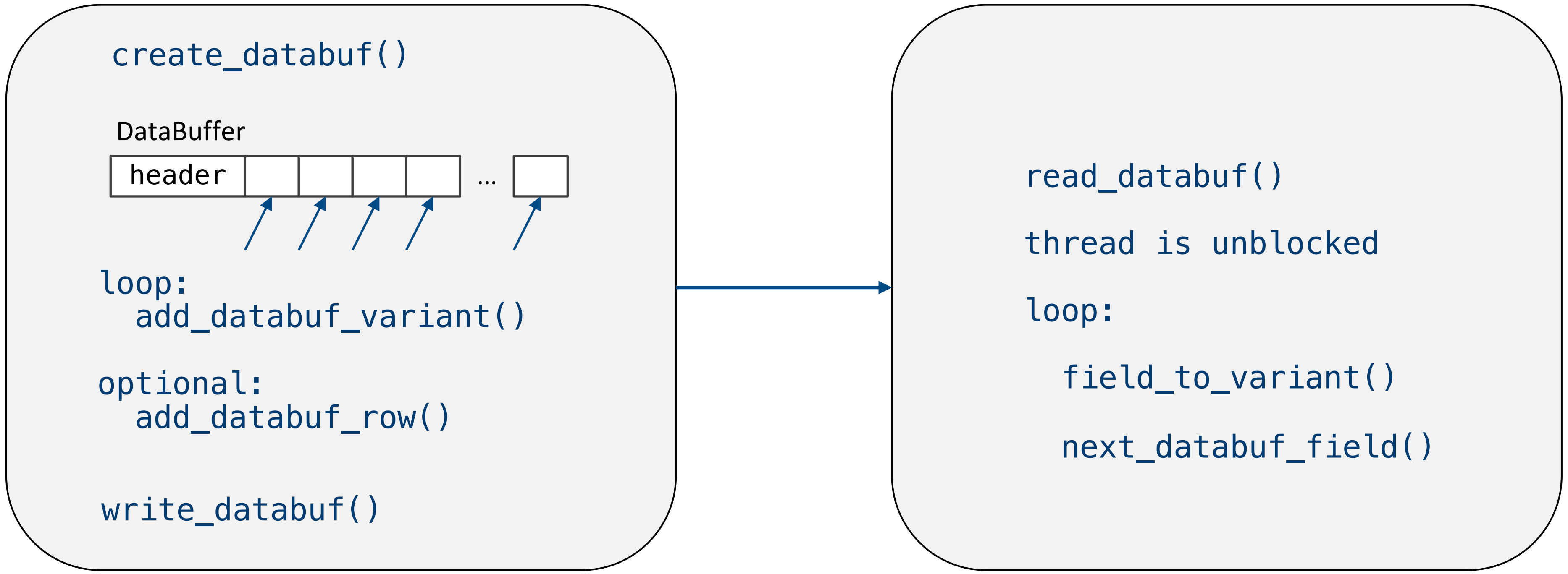

Data Source
Plug-in
Input data into buffer
header
data fields
Broadcast RT Thread
Data buffer pointer inserted into each listener's exchange queue and blocked read thread awakened
Plug-in
Blocked read thread awakes and copies data buffer from broadcaster
xchg queue
Listener RT Thread
Buffer then unpackaged into local data struct and processed

- Input data into a data buffer
- Store data pointer into plug-in's ring queue
- Unblock read thread of each listener
- Data buffer freed when all listeners have read, or when an active ring buffer entry is overwritten (and data loss logged)

- Copy data buffer from broadcaster
- Decrement broadcast's queue listener count
- Unpackage and process data buffer

Queue access managed via pthread mutex

Data Buffer Packaging/Unpackaging API Details



Determinism Configuration Parameters

dx-core-affinity: This parameter represents the CPU core to assign to the real-time blocking read thread for the listener. Some plug-ins also use this value for their real-time broadcast thread. Plug-ins are free to define their own related parameters as well such as **cli-core-affinity** used by OPC UA.

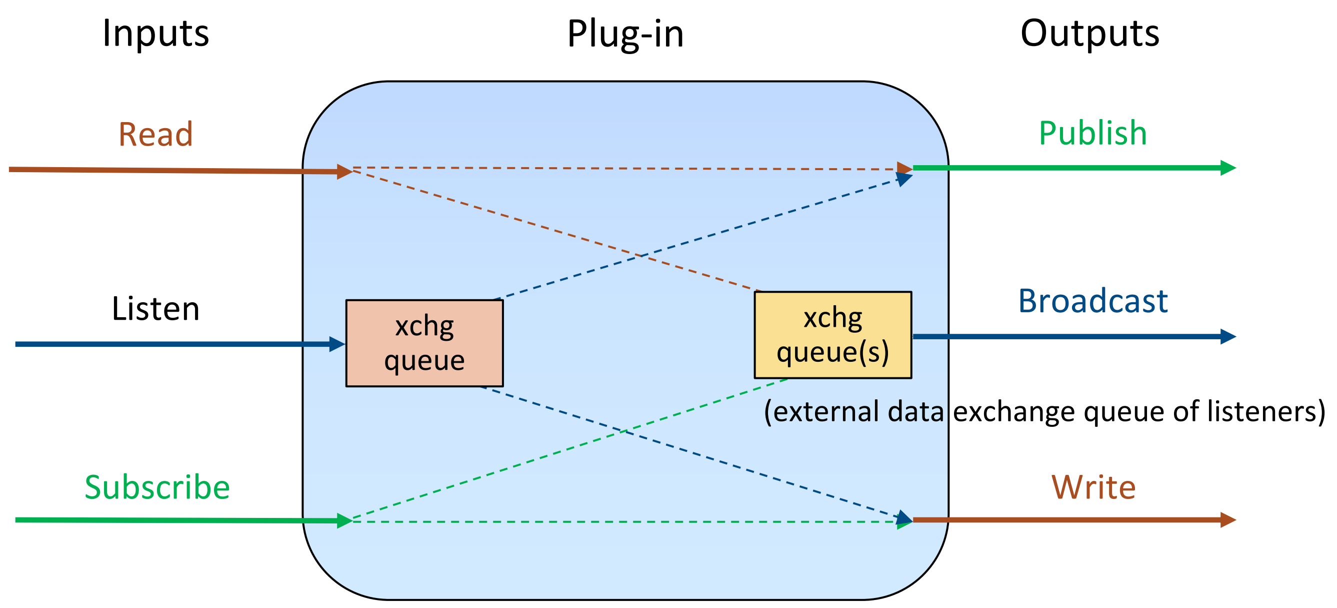
dx-sched-priority: This parameter represents the priority assign to the real-time blocking read thread for the listener. Some plug-ins also use this value for their real-time broadcast thread. Plug-ins are free to define their own related parameters as well such as **cli-sched-priority** used by OPC UA.

sync-start-delay-ms: This parameter is used to synchronize the timing loop for plug-ins by providing a single clock which can be used to synchronize processing. Participating plug-ins will wait until this delay expires until beginning their processing.

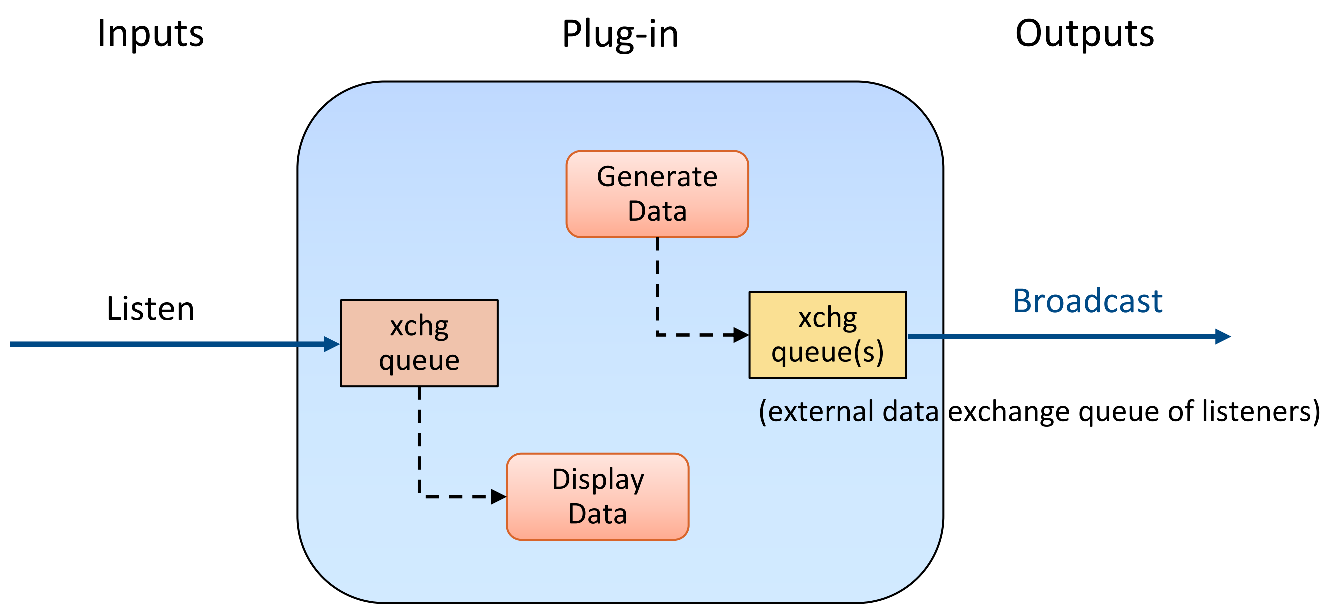
sync-start-offset-ms: This parameter is used by participating plug-ins and represents the offset in milliseconds from the synchronized start delay clock. Plug-ins are free to define their own related parameters as well such as **svr-sync-start-offset-ms**, **pub-sync-start-offset-ms** and **sub-sync-start-offset-ms** used by OPC UA.

Plug-in Data Inputs & Outputs

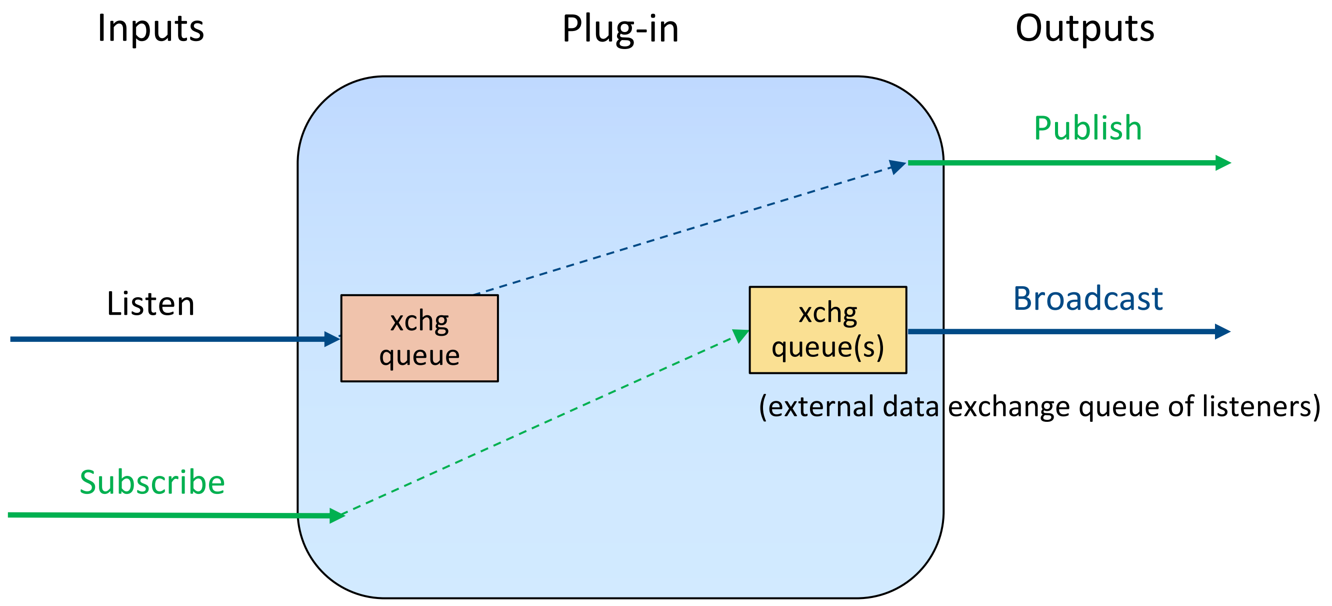
All Possible Data Inputs and Outputs for a Plug-in



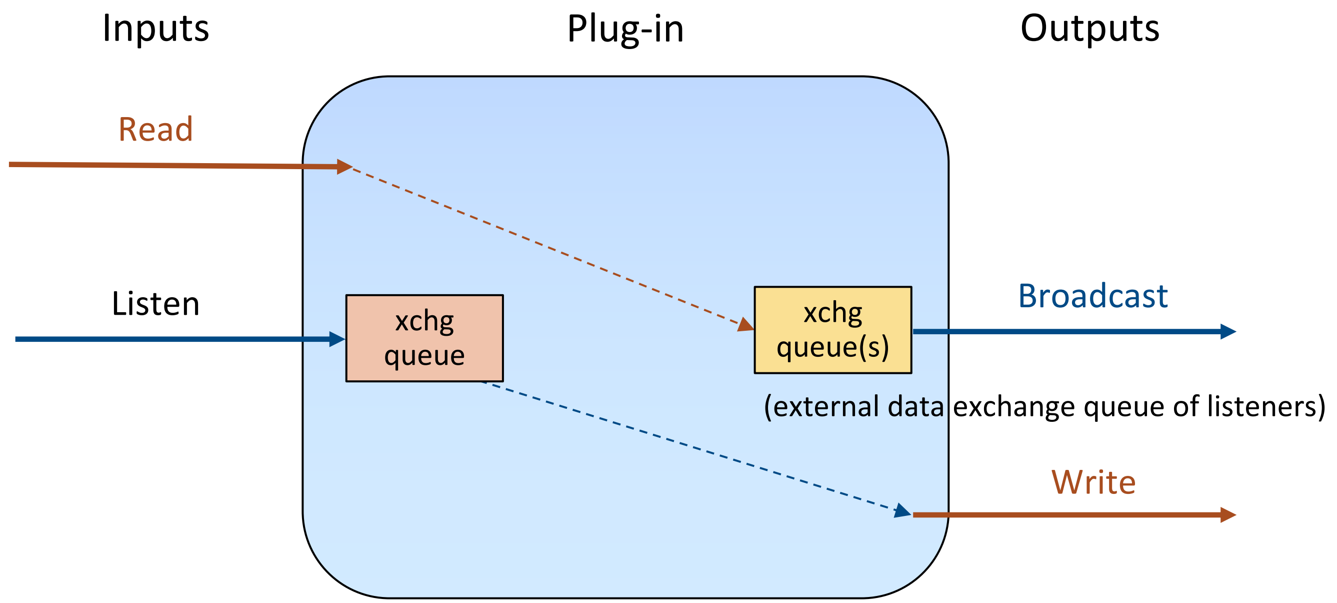
I/O for Data Simulator Plug-in



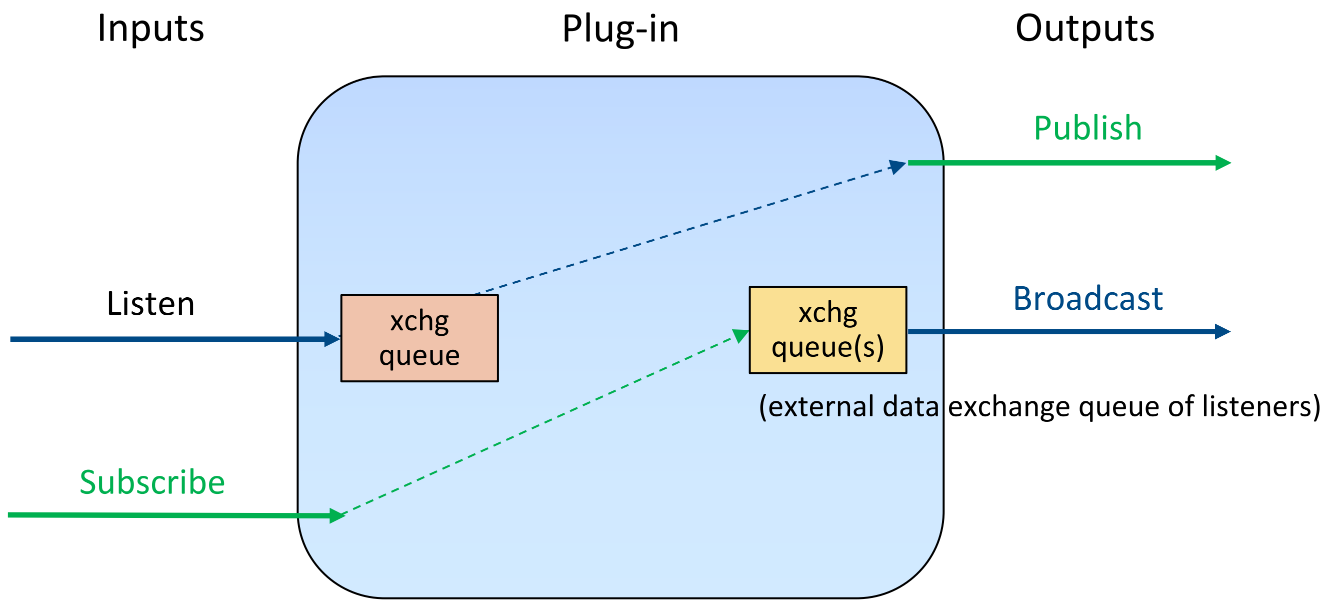
I/O for MQTT Plug-in



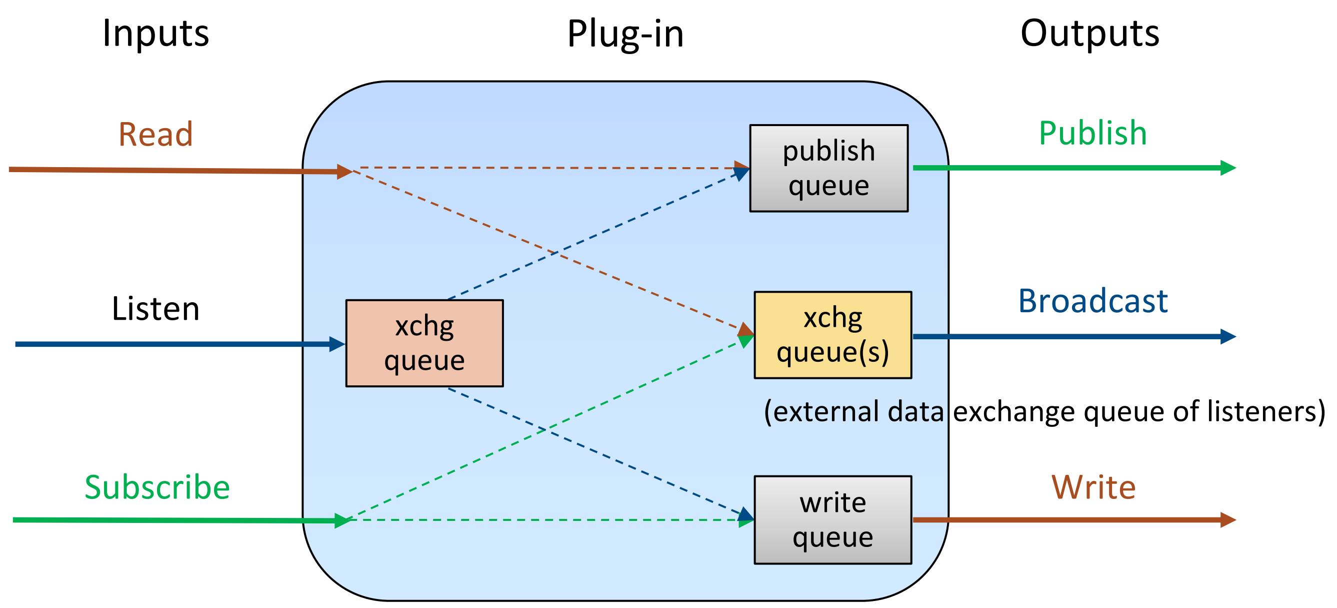
I/O for Shared Memory Plug-in



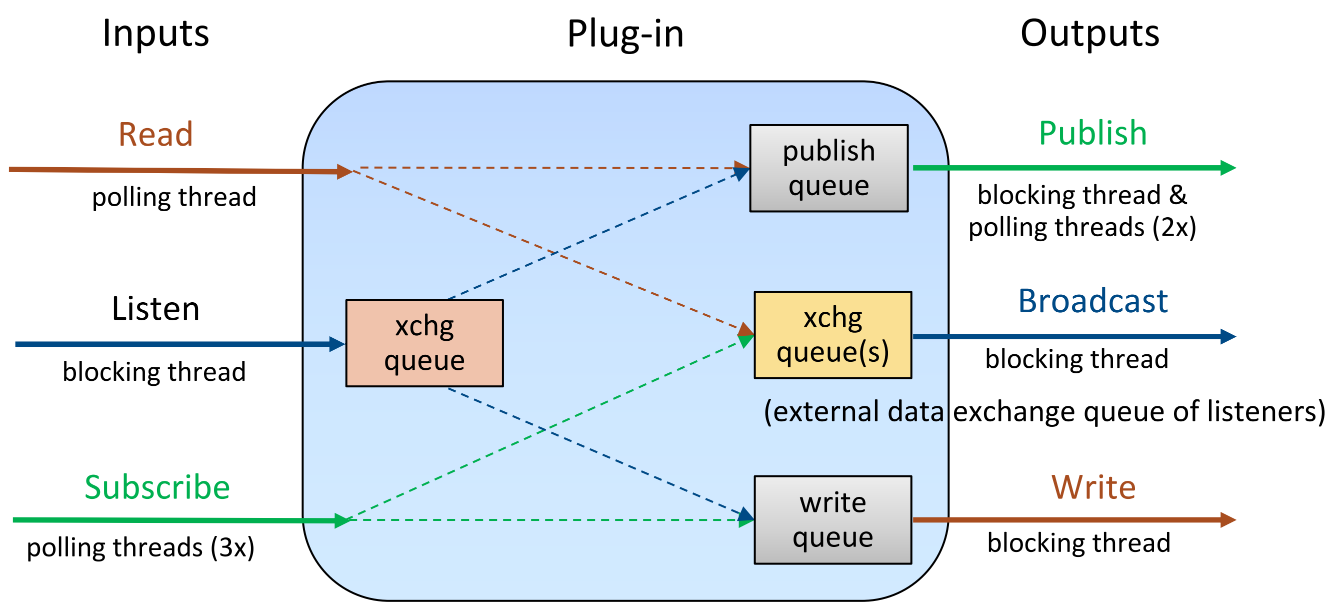
I/O for EMB Plug-in



I/O for OPC UA Plug-in

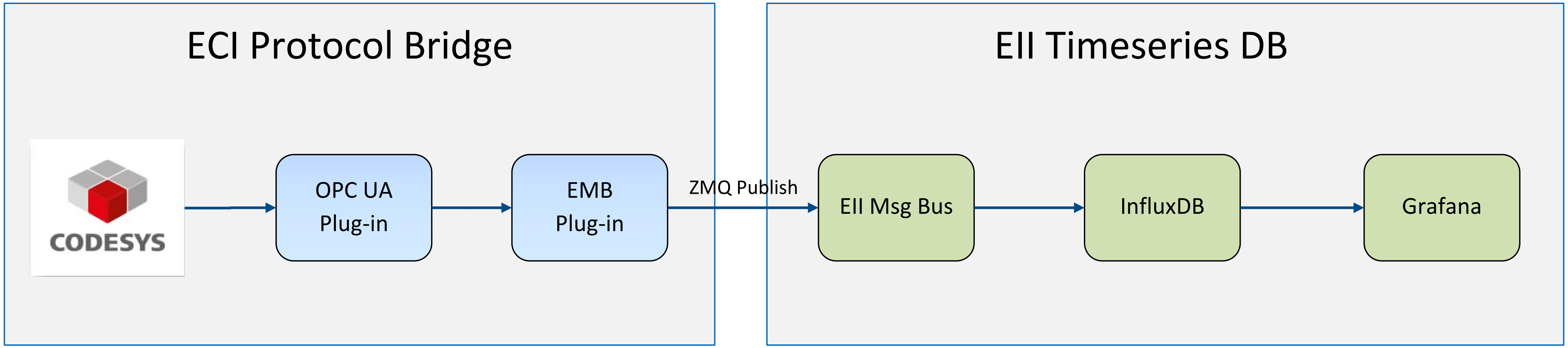


I/O for OPC UA Plug-in, with threading details

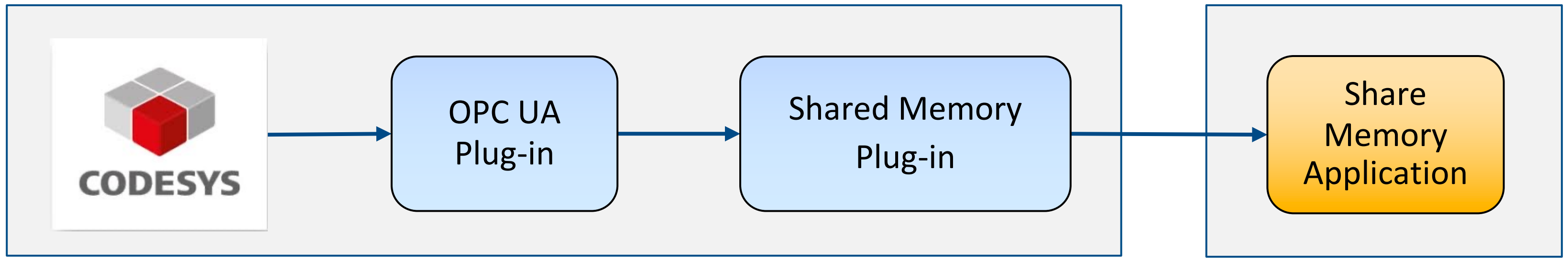


Data Exchange Examples

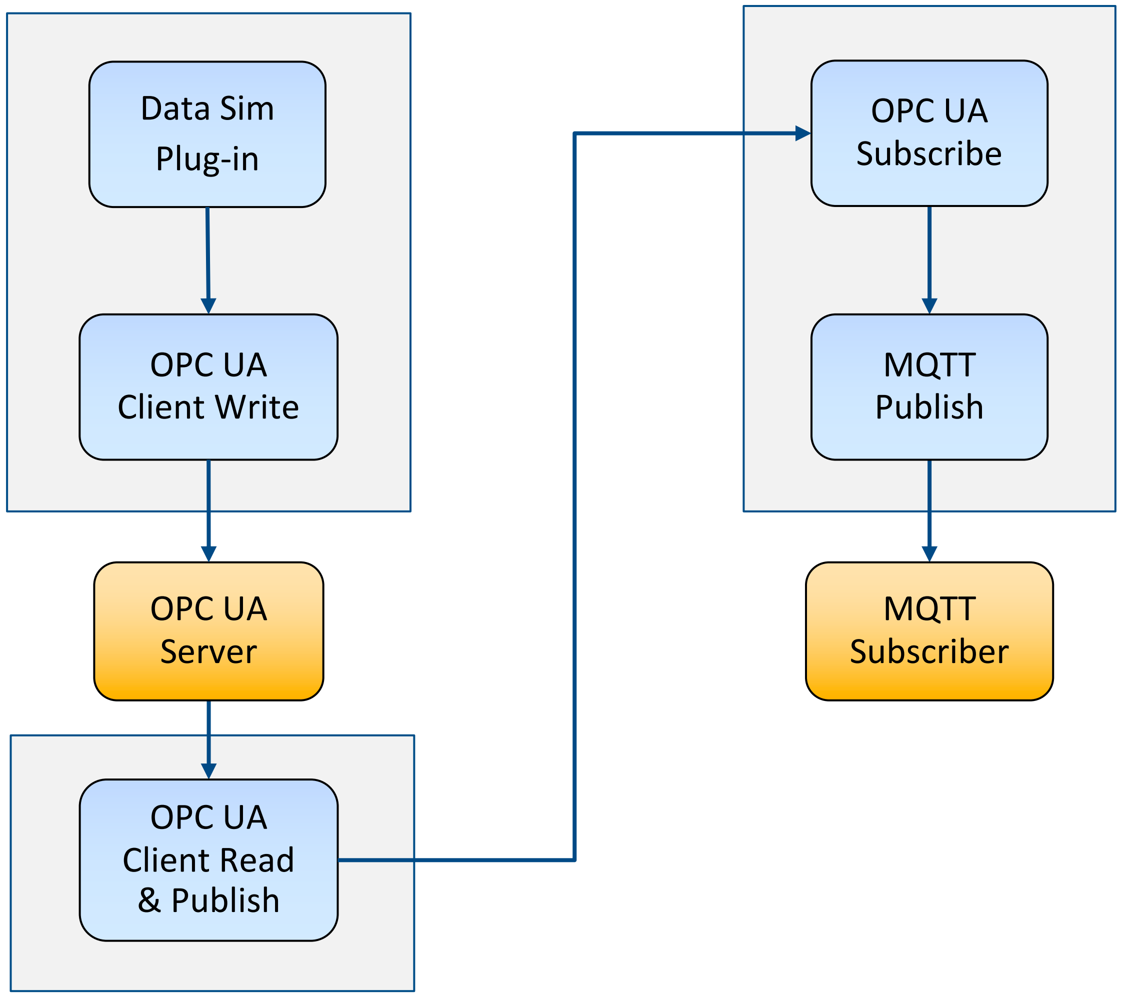
Example for Codesys to EII Timeseries DB



Example for Codesys to Shared Memory



Example for OPC UA Client/Server with PubSub and MQTT



Plug-in Development

Development Guidelines

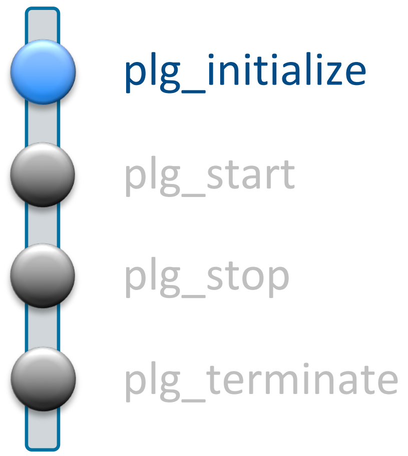
Developers should keep the following guidelines in mind when developing Plug-ins:

- It is mandatory that all API callbacks be implemented, and all version information variables be specified, otherwise the shared library will not be recognized as a plug-in and will not be loaded.
- It is critical to understand that a plug-in is a Linux dynamic shared library. The plug-in code itself, as well as all 3rd party libraries that it utilizes, must run properly in this context. This means that it's designed to be fully reentrant and thread-safe, as well as not making calls to any static libraries.
- A plug-in must return as quickly as possible from all calls to its standard API functions. It should spawn additional worker threads as necessary to perform ongoing processing.
- A plug-in must set a return code in all calls to its standard API functions to notify the framework whether execution was successful or not.
- Use valgrind, or equivalent tool, at every step of the way during development. It can be exceeding difficult to track down and resolve memory leak issues, especially in a multithreaded dynamic share library environment. It is critical that all memory issues be cleaned up immediately and not delay resolving issues until later.
- Remember that plug-ins should be real-time and deterministic, so code must be as quick and efficient as possible.
- Generously use logging throughout the code for easier debug and use logging level to control runtime output.

API Callback Functions

plg_initialize()	equivalent to an object constructor
plg_start()	begin processing
plg_stop ()	stop processing
plg_terminate()	equivalent to an object destructor

plg_initialize



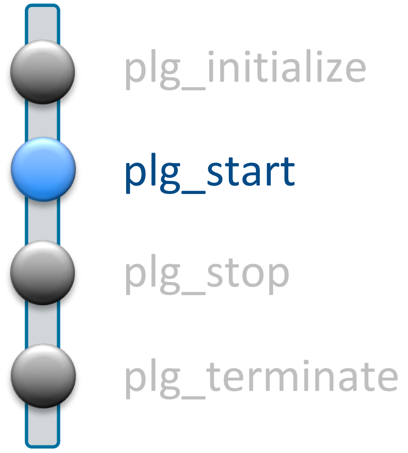
The initialize API method provides the plug-in the opportunity to allocate resources and configure itself.

This method, along with all others, is launch in its own thread, and receives a context parameter that is managed by the framework. It receives all configuration info that was specified in the config file. Although not required, it’s strongly suggested that the plug-in use this method to parse its configuration information into local structs to make accessing that information easier, and to also allocate memory for resources that will be used during its operation.

The plug-in should use the context parameter to store its heap-type data. The use of source-level and static variables must be avoided to ensure that it is fully reentrant and thread-safe.

This method, like all others, should set a return value so that the framework knows whether it has properly executed or not.

plg_start

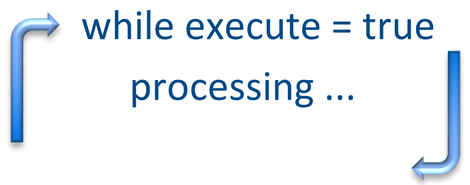


The start method provides the opportunity for the plug-in to start up its asynchronous process thread[s] and enter its processing loop.

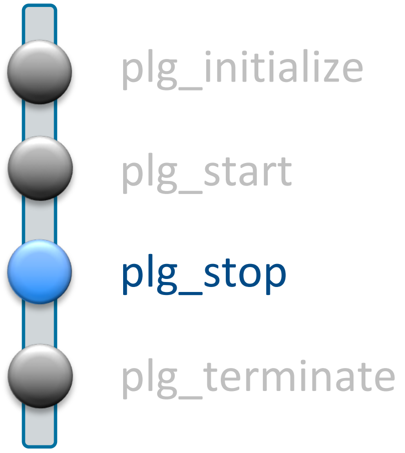
At a minimum, a plug-in should spawn a real-time blocking read thread that listens for data being sent from other plug-ins. These details are described at the end of this API section.

A plug-in should spawn threads necessary to ingest and export data to external applications and data sources. The specifics of the data protocol determines those reads and writes and/or publish and subscribe methods that are needed.

The processing loop continues to run until a variable flag, that is set by the stop() method, indicates that it should terminate.



plg_stop



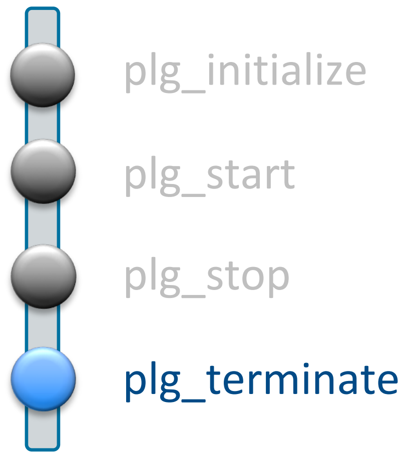
The stop method provides the opportunity for the plug-in to terminate the processing loop.

Typically, the plug-in will simply set a variable to signal the process loop (launched in the start method) that it should end.

example:

```
execute = false      // stop process loop

while execute = true
   processing ...
```

plg_terminate



The terminate method provides the opportunity for the plug-in to release all resources that have been allocated and perform all other clean up that is necessary before it is terminated by the framework.

Use valgrind, or equivalent tool, to ensure that all memory allocations have been freed.

Sample Plug-in Code

Sample Plug-in header file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include ”plgshared.h"

// Version Information
char VI_NAME[] = "Test Plug-in";
char VI_DESCRIPTION[] = "This is the sample test plug-in";
int  VI_MAJOR = 0;
int  VI_MINOR = 9;
int  VI_BUILD = 0;
int  VI_YEAR = 2020;
int  VI_MONTH = 10;
int  VI_DAY = 14;

void plg_initialize(InitParam *);
void plg_start(Plugin *);
void plg_stop(Plugin *);
void plg_terminate(Plugin *);
```

Sample Plug-in source code

```
#include "plugin.h"

// Convenience macros
#define PLG_LOGE(fmt, ...) (writelog)(LOG_ERROR, __FILE__, fmt, ##__VA_ARGS__)
#define PLG_LOGW(fmt, ...) (writelog)(LOG_WARNING, __FILE__, fmt, ##__VA_ARGS__)
#define PLG_LOGI(fmt, ...) (writelog)(LOG_INFO, __FILE__, fmt, ##__VA_ARGS__)
#define PLG_LOGD(fmt, ...) (writelog)(LOG_DEBUG, __FILE__, fmt, ##__VA_ARGS__)
#define PLG_LOGT(fmt, ...) (writelog)(LOG_TRACE, __FILE__, fmt, ##__VA_ARGS__)
#define PLG_LOGV(fmt, ...) (writelog)(LOG_VERBOSE, __FILE__, fmt, ##__VA_ARGS__)

typedef struct Vars {
    PlgFnc *fnc;
    bool running;
    uint8_t dx_core_affinity;
    uint8_t dx_sched_priority;
} Vars;

static void (*writelog)(enum LOG_LEVEL, const char*, const char*, ...);   // Convenience function
```

plg_initialize() callback

```
/*
 * Called by the plug-in framework first to launch and configure the plug-in
 */

void plg_initialize(InitParam *param) {
    PLG_LOGT("%s() called for, %s\n", __func__, VI_NAME);
    writelog = param->plg_fnc->writelog;

    Vars *vars = malloc(sizeof(Vars));
    vars->fnc = param->plg_fnc;
    vars->running = true;
    vars->dx_core_affinity = 1;
    vars->dx_sched_priority = 60;
    param->self->vars = vars;

    Configuration *cfg;
    DL_FOREACH(param->self->configuration, cfg) {
        if (cfg->level == 0 && strcasecmp(cfg->key, "dx-core-affinity") == 0) {
            vars->dx_core_affinity = atoi(cfg->val);
        } else if (cfg->level == 0 && strcasecmp(cfg->key, "dx-sched-priority") == 0) {
            vars->dx_sched_priority = atoi(cfg->val);
        }
    }
    *param->self->thrd_rtn = PLG_SUCCESS;
}
```

plg_start() callback

```
/*
 * Called by the plug-in framework after initialize to start the process loop
 */

void plg_start(Plugin *self) {
    PLG_LOGT("%s() called for, %s\n", __func__, VI_NAME);

    Vars *vars = (Vars*)self->vars;
    PLG_RC thrd_rtn = PLG_SUCCESS;

    // Launch the listener thread which runs until plug-in is shutdown
    pthread_t thread = vars->fnc->create_rt_thread(vars->dx_sched_priority, vars->dx_core_affinity,
            plg_listener, "MyListener", self);

    if (thread == 0) {
        thrd_rtn = PLG_ERROR_THREAD;
    } else {
        // Wait for thread to end
        pthread_join(thread, NULL);
    }

    self->thrd_rtn = thrd_rtn;
}
```

Sample Listener Thread function

```
static void* plg_listener(void *arg) { // Fnc must match name specified when created (ex. plg_listener)
    Plugin *self = (Plugin*) arg;
    Vars *vars = (Vars*) self->vars;
    DataBuffer *databuf_recv = NULL;

    while (vars->running) {  // Loop until plug-in is stopped
        vars->fnc->read_databuf(self, false, true, &databuf);  // Receive data (blocking read)
        if (!databuf_recv) continue;  // Check that valid data buffer was received
        uint8_t *buf_fld = databuf_recv->buffer;  // Set pointer to the first field in the data buffer

        DataField *datafld;
        DL_FOREACH((*(Dataset**)databuf->dataset)->dataflds, datafld) {  // Foreach data buffer field
            if (!buf_fld) break;

            Variant var;
            vars->fnc->field_to_variant(datafld, buf_fld, &var);

            // This example simply outputs the data field to the console. An actual implementation would process the data.
            eci_display_variant(datafld->datafld_id, &var);

            buf_fld = vars->fnc->next_databuf_field(buf_fld);
        }
        free(databuf_recv);
    }
    return NULL;
}
```

plg_stop() callback

```
/*
 * Called by the plug-in framework first when shutting down to stop the process loop
 */

void plg_stop(Plugin *self) {
    PLG_LOGT("%s() called for, %s\n", __func__, VI_NAME);

    Vars *vars = (Vars*)self->vars;
    vars->running = false;

    *self->thrd_rtn = PLG_SUCCESS;
}
```

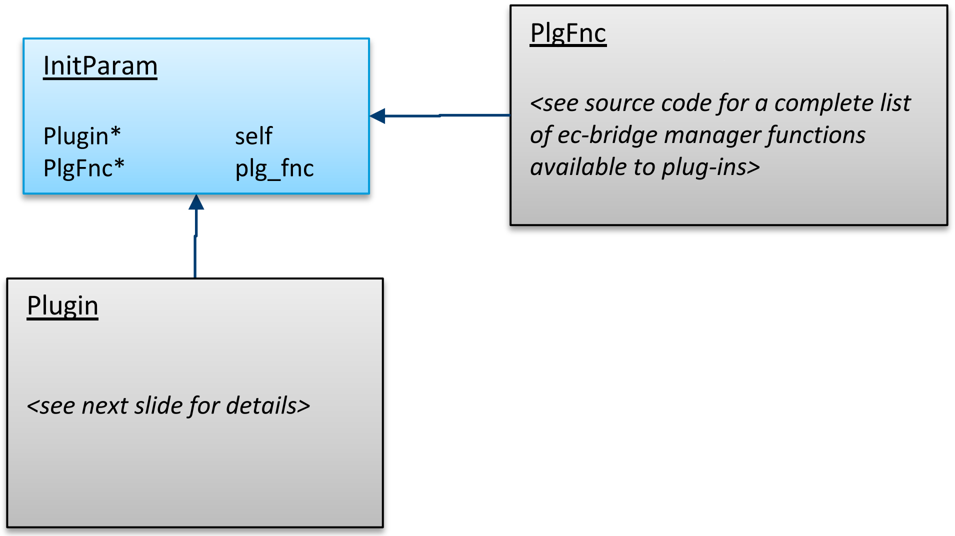
plg_terminate() callback

```
/*
 * Called by the plug-in framework when terminating the plug-in
 */

void plg_terminate(Plugin *self) {
    PLG_LOGT("%s() called for, %s\n", __func__, VI_NAME);

    free(self->vars);

    *self->thrd_rtn = PLG_SUCCESS;
}
```

InitParam data struct param passed to plg_initialize()

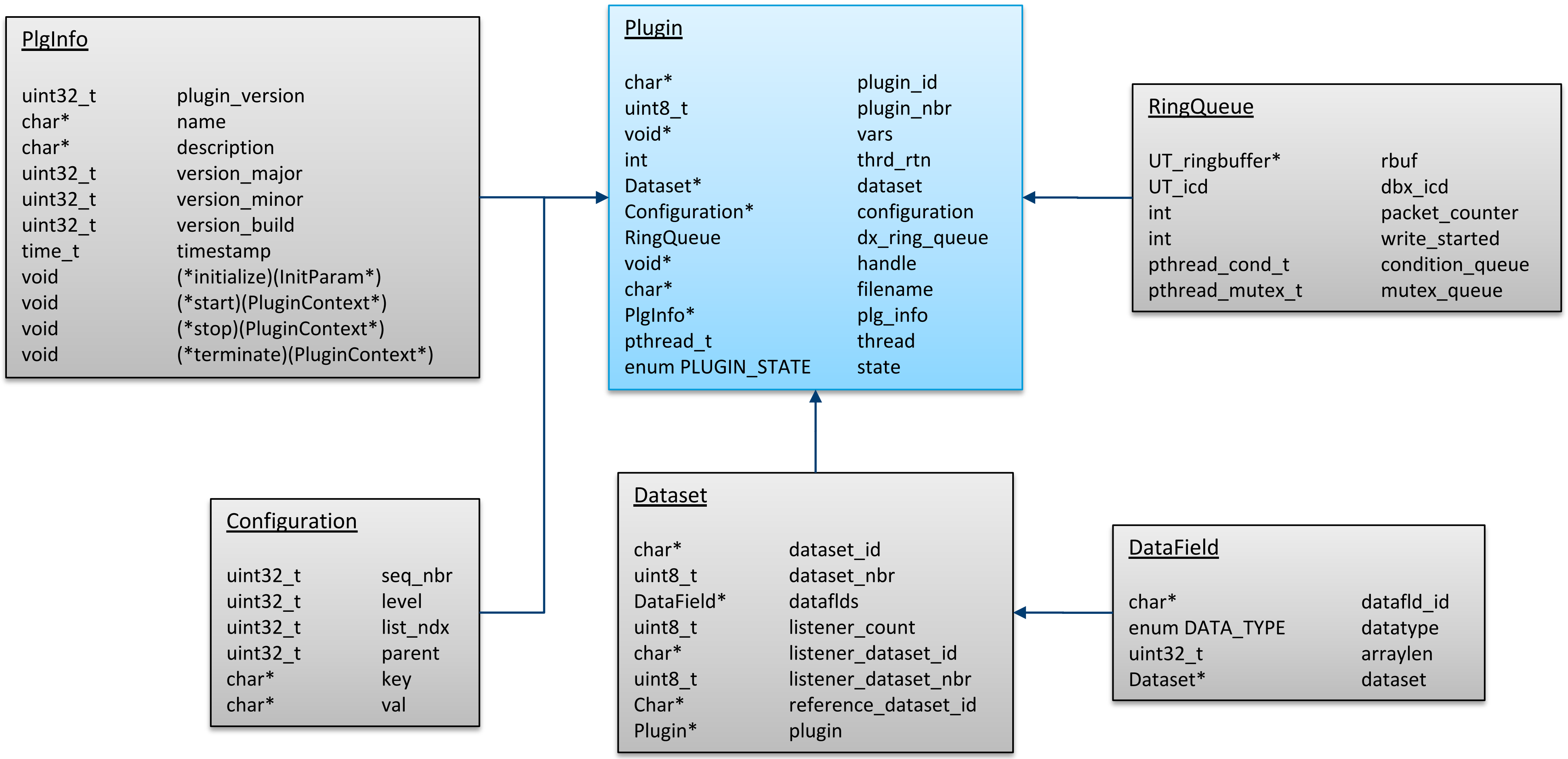
plg_initialize()

plg_start()
plg_stop()
plg_terminate()



InitParam
Plugin* self
PlgFnc* plg_fnc
PlgFnc
<see source code for a complete list of ec-bridge manager functions available to plug-ins>
Plugin
<see next slide for details>

Plugin data struct param passed to most API callbacks



PlgInfo

uint32_t	plugin_version
char*	name
char*	description
uint32_t	version_major
uint32_t	version_minor
uint32_t	version_build
time_t	timestamp
void	(*initialize)(InitParam*)
void	(*start)(PluginContext*)
void	(*stop)(PluginContext*)
void	(*terminate)(PluginContext*)

Plugin

char*	plugin_id
uint8_t	plugin_nbr
void*	vars
int	thrd_rtn
Dataset*	dataset
Configuration*	configuration
RingQueue	dx_ring_queue
void*	handle
char*	filename
PlgInfo*	plg_info
pthread_t	thread
enum PLUGIN_STATE	state

RingQueue

UT_ringbuffer*	rbuf
UT_icd	dbx_icd
int	packet_counter
int	write_started
pthread_cond_t	condition_queue
pthread_mutex_t	mutex_queue

Configuration

uint32_t	seq_nbr
uint32_t	level
uint32_t	list_ndx
uint32_t	parent
char*	key
char*	val

Dataset

char*	dataset_id
uint8_t	dataset_nbr
DataField*	dataflds
uint8_t	listener_count
char*	listener_dataset_id
uint8_t	listener_dataset_nbr
Char*	reference_dataset_id
Plugin*	plugin

DataField

char*	datafld_id
enum DATA_TYPE	datatype
uint32_t	arraylen
Dataset*	dataset