TrickHLA v3

An HLA interface package for Trick

Edwin Z. Crues, Ph.D.
Daniel E. Dexter

Simulation and Graphics Branch (ER7)
NASA Johnson Space Center
2101 NASA Parkway, Houston, Texas, 77058

edwin.z.crues@nasa.gov
daniel.e.dexter@nasa.gov

January 2021

Outline

1. Introduction
2. Minimum Software Requirements
3. TrickHLA Features
4. Setting up the Environment
5. Sine Wave Simulation Example
6. Steps to Adding TrickHLA to a Trick Simulation
7. General TrickHLA Configuration
8. Simulation Multiphase Initialization
9. Data Packing and Unpacking
10. Interactions
11. Lag Compensation
12. Ownership Transfer
13. Object Deletion

Introduction

- The main design goal of TrickHLA is to allow existing Trick simulations to use the IEEE-1516 High Level Architecture (HLA) with minimal development effort.
- TrickHLA is an HLA abstraction for the Trick simulation environment.
- TrickHLA is data driven and provides a simple API for the HLA functionality.
- Trick is available from the NASA GitHub website: https://github.com/nasa/trick/wiki
- TrickHLA is available from the NASA GitHub website: https://github.com/nasa/trickhla/wiki

Minimum Software Requirements

- Trick version 17 or newer.
- IEEE-1516-2010 HLA standard compliant Runtime Infrastructure (RTI).
- TrickHLA callback interfaces must be written in C++.

TrickHLA Features

TrickHLA Features

- Primitive C/C++ data types.
- Static arrays of primitive data types.
- One-dimensional dynamic arrays of primitive data types.
- Strings and static arrays of strings.
- Opaque (hidden type) data.
- Automatically encodes/decodes the data to/from the RTI using the specified HLA encoding.
- Big/Little Endian for primitives
- HLAunicodeString, HLAASCIIstring, and HLAopaqueData for strings (i.e. char *)
- User defined packing and unpacking functions to allow for data transformations before data is sent to (pack) or after data is received from (unpack) the other federates.
- Automatically handles HLA time advancement.

TrickHLA Features

Continued

- Handles real time and non-real time simulations
- Lag Compensation
  - None
  - Sending-side
  - Receive-side
- Interactions (job safe and thread safe)
  - Receive Order (RO)
  - Timestamp Order (TSO)
- Ownership Transfer
- Multiphase Initialization, which allows simulation initialization data to be shared between federates before the simulation starts. This includes a simulation configuration object.
- Automatically synchronizes the startup of a distributed simulation without depending on specific federate start order.

TrickHLA Features
Continued

- Coordinated Federation Save & Restore.
- Late joining federates.
- Supports IEEE-1516.1-2010 (a.k.a HLA Evolved).
- Allows subscription to non-required federate data.
- Attribute preferred order (Receive or Timestamp Order) can be overridden.
- Blocking cyclic data reads (only 2 federate case supported for now).
- Notification of object deletions.

Setting up the Environment

Setting up the Environment

- Set the path to the TrickHLA directory
  - Set the environment variable TRICKHLA_HOME to the system file path for the TrickHLA source directory.
  - For example:

```
setenv TRICKHLA_HOME ${HOME}/Trick/hla/TrickHLA
```

- Add the TrickHLA models directory to the Trick compile environment
  - The Trick environment variable TRICK_CXXFLAGS must include the path to the TrickHLA models.
  - For example:

```
TRICK_CXXFLAGS += -I${TRICKHLA_HOME}/models
```

- Set up the RTI for a C-Shell environment
    - Copy the provided scripts /.rti_cshrc file to your home directory.
    - Change the RTI_HOME environment variable in the .rti_cshrc file to point to the location of your RTI installation.
    - For example:

```
setenv RTI_HOME ${HOME}/rti/prti1516e_v5.5.1
```

    - Add the following lines to your .Trick_user_cshrc file in your home directory.

```
# Perform RTI setup, which MUST occur after you set
# up your TRICK_CXXFLAGS since .rti_cshrc will appended
# entries to it.
if ( -e ${HOME}/.rti_cshrc ) then
   source ${HOME}/.rti_cshrc
endif
```

- Set up the RTI Bourne Shell environment
  - Copy the provided scripts /.rti_profile file to your home directory.
  - Change the RTI_HOME environment variable in the .rti_profile file to point to the location of your RTI installation.
  - For example:

```
set RTI_HOME=${HOME}/rti/prti1516e_v5.5.1
export RTI_HOME
```

  - Add the following lines to your .Trick_profile file in your home directory.

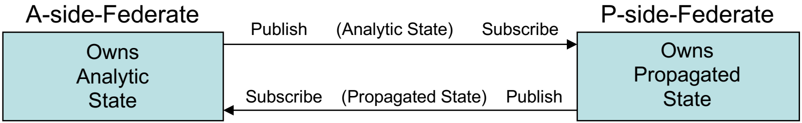
```
# Perform RTI setup, which MUST occur after you set
# up your TRICK_CXXFLAGS since .rti_profile will appended
# entries to it.
if [ -f ${HOME}/.rti_profile ] ; then
. ${HOME}/.rti_profile
fi
```

Sine Wave Simulation Example

Sine Wave Simulation Example

- A simple simulation of a sinusoid will be used to describe the TrickHLA concepts.
- The example can be found in `${TRICKHLA_HOME}/sims/TrickHLA/SIM_sine`.
- Equation of a sinusoid with a given amplitude a, angular frequency ω (period $2\pi/\omega$), and phase ϕ: $f(t) = a\sin(\omega t + \phi)$

Sinusoidal Parameter	**Analytic**	**Propagated**
Amplitude	2	1
Frequency	0.1963	0.3927
Period	32	16



Steps to Adding TrickHLA to a Trick Simulation

Steps to Adding TrickHLA to a Trick Simulation

- Adding TrickHLA to a Trick simulation consists of three steps.

  Step 1: Add the provided TrickHLA specific `THLA.sm` simulation module to your `S_define` file, and pass in the `data_cycle` and `interaction_cycle` parameter values, using the `THLA_DATA_CYCLE_TIME` and `THLA_INTERACTION_CYCLE_TIME` values defined at the top of the `S_define` file.

  Step 2: Add a generic `THLA_INIT` multiphase initialization sim object to your `S_define` file, and pass in the `TrickHLA::Manager` and `TrickHLA::Federate` objects from `THLA.sm`.

  Step 3: Configure TrickHLA through settings in your simulation RUN `input.py` file.

- Add to your S_define file:

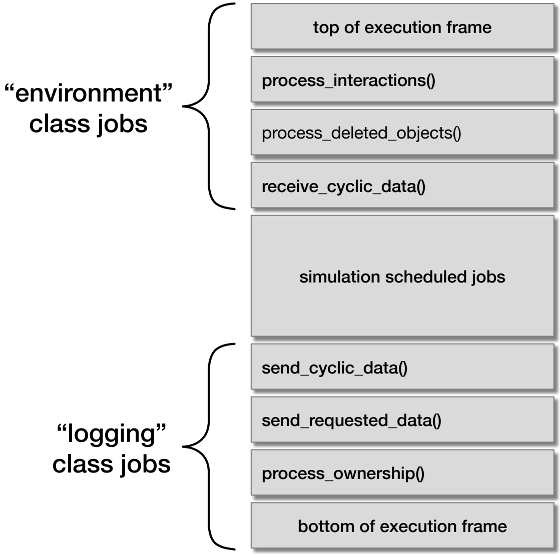
```
// Define HLA job cycle times:
#define THLA_DATA_CYCLE_TIME         0.250  // HLA data communication cycle time.
#define THLA_INTERACTION_CYCLE_TIME  0.050  // HLA Interaction cycle time.

// get sim object for generalized TrickHLA interface routines:
#include "../../../../shared/trick10/S_modules/THLA.sm"
THLASimObject THLA( THLA_DATA_CYCLE_TIME, THLA_INTERACTION_CYCLE_TIME);
```

- `THLA_DATA_CYCLE_TIME` defines the rate at which data is exchanged through the RTI.
- `THLA_INTERACTION_CYCLE_TIME` defines the rate at which received Interactions in Receive-Order (RO) will be serviced.
- Each of the above two values must be an integer multiple of Trick's real-time frame (i.e., the *n* value set via `trick.exec_set_software_frame(`*n*`)`).

Steps to Adding TrickHLA to a Trick Simulation

TrickHLA jobs in THLA.sm



Steps to Adding TrickHLA to a Trick Simulation

Step 2: THLA_INIT Simulation Object

- Add to your S_define file:

```
##include "simconfig/include/SimpleSimConfig.hh"
class THLAInitSimObject : public Trick::SimObject {
  public:
    SimpleSimConfig simple_sim_config; // The simple simulation configuration
    THLAInitSimObject( TrickHLAManager  & thla_mngr,
                       TrickHLAFederate & thla_fed )
       : thla_manager( thla_mngr ), thla_federate( thla_fed )
    {
       // Make sure we initialize the sim config before the TrickHLAManager
       P1 ("initialization") simple_sim_config.initialize(
                                        thla_federate.known_feds_count,
                                        thla_federate.known_feds );
       // Send all the initialization data
       P100 ("initialization") thla_manager.send_init_data();
       // Wait to receive all the initialization data
       P100 ("initialization") thla_manager.receive_init_data();
       // Clear remaining initialization sync-points
       P100 ("initialization") thla_manager.clear_init_sync_points();
  private:
    TrickHLAManager  & thla_manager;
    TrickHLAFederate & thla_federate;
    THLAInitSimObject(const THLAInitSimObject & rhs); // do not allow copy
    THLAInitSimObject & operator=(const THLAInitSimObject & rhs); // do not allow assign
    THLAInitSimObject();  // do not allow default constructor
};
THLAInitSimObject THLA_INIT( THLA.manager, THLA.federate );
```

- The last step to adding TrickHLA to a simulation is to configure it through settings in your input.py file.
- The majority of the presentation will cover TrickHLA configuration for:
    1. General Configuration
    2. Multiphase Initialization
    3. Data Packing and Unpacking
    4. Interactions
    5. Ownership Transfer
    6. Lag Compensation
    7. Object Deletion

General TrickHLA Configuration

General TrickHLA Configuration

General TrickHLA configuration consists of the following:

1. General TrickHLA Configuration
2. Initializing the Federate
3. Initializing the List of Known Federates
4. Initializing the Debug Level (Optional)
5. Initializing the Data Objects
6. Initializing the Data Object Attributes

- The Pitch Central RTI Component (CRC) is initialized by specifying the host, port, and lookahead time as shown below. The host and port describe where the CRC is/will be running.
- Notice the settings are based on the simulation object name and the parameter name as they exist in the `S_define` file.
- In the `RUN_a_side/input.py` file:

```
# Configure the CRC for the Pitch RTI.
THLA.federate.local_settings = "crcHost = localhost\n crcPort = 8989"
THLA.federate.lookahead_time = 0.25  # this is THLA_DATA_CYCLE_TIME
```

General TrickHLA Configuration

Initializing the Federate

- The input.py file settings determine how each federate shares data.

```
In the RUN_a_side/input.py file:
THLA.federate.name                      = "A-side-Federate"
THLA.federate.enable_FOM_validation = False
THLA.federate.FOM_modules               =
   "S_FOMfile.xml,TrickHLAFreezeInteraction.xml"
THLA.federate.federation_name           = "SineWaveSim"
THLA.federate.time_regulating           = True
THLA.federate.time_constrained          = True
```

- In the RUN_p_side/input.py file:

```
THLA.federate.name                      = "P-side-Federate"
THLA.federate.enable_FOM_validation = False
THLA.federate.FOM_modules               =
   "S_FOMfile.xml,TrickHLAFreezeInteraction.xml"
THLA.federate.federation_name           = "SineWaveSim"
THLA.federate.time_regulating           = True
THLA.federate.time_constrained          = True
```

- A list of federates known to be in the federation is initialized next.
- The simulation will wait for all federates designated as "required" to join the simulation before continuing on. These required federates define the distributed simulation parts that MUST exist for the simulation to run.
- In both the `RUN_a_side/input.py` and `RUN_p_side_input.py` files:

```
THLA.federate.enable_known_feds      = True
THLA.federate.known_feds_count       = 2
THLA.federate.known_feds             = trick.alloc_type(
   THLA.federate.known_feds_count, "TrickHLAKnownFederate" )
THLA.federate.known_feds[0].name     = "A-side-Federate"
THLA.federate.known_feds[0].required = True
THLA.federate.known_feds[1].name     = "P-side-Federate"
THLA.federate.known_feds[1].required = True
```

General TrickHLA Configuration

Initializing the Debug Level (Optional)

- Although not required, it is recommended that you enable TrickHLA debug messages while you are getting your simulation to work with TrickHLA for the first time.
- Varying amounts of messages may be output by setting the debug level, which ranges from `THLA_LEVEL0_TRACE` (no messages) to `THLA_LEVEL11_TRACE` (all messages).
- For example:

```
THLA.manager.debug_handler.debug_level = trick.THLA_LEVEL3_TRACE
```

- The TrickHLA data objects define the data the federates will exchange.
- For the sine wave simulation each federate will publish one object containing its own state data and will subscribe to the state data of the other federate (2 objects total).
- In both the `RUN_a_side/input.py` and `RUN_p_side_input.py` files:

```
THLA.manager.obj_count = 2
THLA.manager.objects   = trick.alloc_type( THLA.manager.obj_count,
   "TrickHLAObject" )
```

- The `S_FOMfile.xml` FOM file defines the `Test` data class containing the following 8 attributes for the sine wave simulation: *Name*, *Time*, *Value*, *dvdt*, *Phase*, *Frequency*, *Amplitude*, and *Tolerance*.

General TrickHLA Configuration

Initializing the Data Objects - RUN_a

- Data object configuration in the `RUN_a_side/input.py` file:

```
# Configure the object this federate owns and will publish.
THLA.manager.objects[0].FOM_name             = "Test"
THLA.manager.objects[0].name                 = "A-side-Federate.Test"
THLA.manager.objects[0].create_HLA_instance = True
THLA.manager.objects[0].attr_count           = 8
THLA.manager.objects[0].attributes           = trick.alloc_type(
   THLA.manager.objects[0].attr_count, "TrickHLAAttribute" )

# Configure the object this federate does not own and will subscribe to.
THLA.manager.objects[1].FOM_name             = "Test"
THLA.manager.objects[1].name                 = "P-side-Federate.Test"
THLA.manager.objects[1].create_HLA_instance = False
THLA.manager.objects[1].attr_count           = 8
THLA.manager.objects[1].attributes           = trick.alloc_type(
   THLA.manager.objects[1].attr_count, "TrickHLAAttribute" )
```

- Data object configuration in the `RUNp_side/input.py` file:

```
# Configure the object this federate does not own and will subscribe to.
THLA.manager.objects[0].FOM_name             = "Test"
THLA.manager.objects[0].name                 = "A-side-Federate.Test"
THLA.manager.objects[0].create_HLA_instance = False
THLA.manager.objects[0].attr_count           = 8
THLA.manager.objects[0].attributes           = trick.alloc_type(
   THLA.manager.objects[0].attr_count, "TrickHLAAttribute" )

# Configure the object this federate owns and will publish.
THLA.manager.objects[1].FOM_name             = "Test"
THLA.manager.objects[1].name                 = "P-side-Federate.Test"
THLA.manager.objects[1].create_HLA_instance = True
THLA.manager.objects[1].attr_count           = 8
THLA.manager.objects[1].attributes           = trick.alloc_type(
   THLA.manager.objects[1].attr_count, "TrickHLAAttribute"
```

General TrickHLA Configuration

Initializing the Data Object Attributes

- The main concept to remember is that we are tying Trick simulation variables to FOM object attributes.
- TrickHLA is restricted to supporting only Trick simulation variables that are either primitive types, static array of primitives, strings, or static array of strings. (A future TrickHLA version will support aggregate data types, which will remove this restriction.)
- Supported RTI encodings of attributes include:
  - THLA_BIG_ENDIAN
  - THLA_LITTLE_ENDIAN
  - THLA_UNICODE_STRING
  - THLA_ASCII_STRING
  - THLA_OPAQUE_DATA
- All attributes must be configured in the input.py file, the following examples only show a snippet.

General TrickHLA Configuration

Initializing the Data Object Attributes - RUN_a

- An example of attribute data the “A-side-Federate” owns and will publish (notice this is for object index “0”), in the `RUN_a_side/input.py` file:

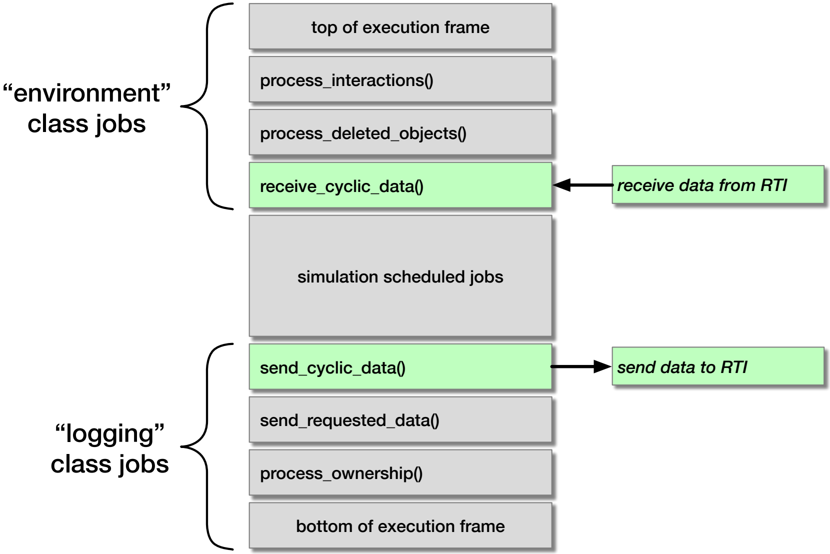
```
THLA.manager.objects[0].attributes[0].FOM_name        = "Time"
THLA.manager.objects[0].attributes[0].trick_name      = "A.sim_data.time"
THLA.manager.objects[0].attributes[0].config          = trick.THLA_CYCLIC
THLA.manager.objects[0].attributes[0].publish         = True
THLA.manager.objects[0].attributes[0].locally_owned   = True
THLA.manager.objects[0].attributes[0].rti_encoding    = trick.THLA_LITTLE_ENDIAN
...
THLA.manager.objects[0].attributes[7].FOM_name        = "Name"
THLA.manager.objects[0].attributes[7].trick_name      = "A.sim_data.name"
THLA.manager.objects[0].attributes[7].config          = trick.THLA_INITIALIZE + trick.THLA_CYCLIC
THLA.manager.objects[0].attributes[7].publish         = True
THLA.manager.objects[0].attributes[7].locally_owned   = True
THLA.manager.objects[0].attributes[7].rti_encoding    = trick.THLA_UNICODE_STRING
```

- An example of attribute data the "A-side-Federate" does not own the state of and will subscribe to (notice this is for object index "1"), in the `RUN_a_side/input.py` file:

```
THLA.manager.objects[1].attributes[0].FOM_name       = "Time"
THLA.manager.objects[1].attributes[0].trick_name     = "P.sim_data.time"
THLA.manager.objects[1].attributes[0].config         = trick.THLA_CYCLIC
THLA.manager.objects[1].attributes[0].subscribe      = True
THLA.manager.objects[1].attributes[0].locally_owned  = False
THLA.manager.objects[1].attributes[0].rti_encoding   = trick.THLA_LITTLE_ENDIAN
...
THLA.manager.objects[1].attributes[7].FOM_name       = "Name"
THLA.manager.objects[1].attributes[7].trick_name     = "P.sim_data.name"
THLA.manager.objects[1].attributes[7].config         = trick.THLA_INITIALIZE + trick.THLA_CYCLIC
THLA.manager.objects[1].attributes[7].subscribe      = True
THLA.manager.objects[1].attributes[7].locally_owned  = False
THLA.manager.objects[1].attributes[7].rti_encoding   = trick.THLA_UNICODE_STRING
```

General TrickHLA Configuration

TrickHLA jobs in THLA.sm



Simulation Multiphase Initialization

Simulation Multiphase Initialization

- TrickHLA supports multiphase simulation initialization where data is exchanged between federates before the simulation starts running.
- An attribute is used for multiphase simulation initialization by specifying `THLA_INITIALIZE` for the attribute's config field (as in the previous example shown in the `RUN_a_side/input.py` file) :

```
THLA.manager.objects[1].attributes[7].config = trick.THLA_INITIALIZE
                                             + trick.THLA_CYCLIC
```

Simulation Multiphase Initialization

S_define: Public data and constructor

- This is similar to our generic THLA_INIT simulation object, but using a multiphase approach:

```
. . .
// Include the simple simulation configuration object definition.
##include "simconfig/include/SimpleSimConfig.hh"
. . .

//==========================================================================
// SIM_OBJECT: THLA_INIT  (TrickHLA multi-phase initialization sim-object)
//==========================================================================
class THLAInitSimObj : public Trick::SimObject {

 public:

   TrickHLA::SimTimeline      sim_timeline;
   TrickHLA::ScenarioTimeline scenario_timeline;

   THLAInitSimObj( TrickHLA::Manager  & thla_mngr,
                   TrickHLA::Federate & thla_fed )
      : scenario_timeline( sim_timeline, 0.0, 0.0 ),
        thla_manager( thla_mngr ),
        thla_federate( thla_fed )
   {
```

- Declare the initialization jobs.

```
    //------------------------------------------------------------------------
    // NOTE: Initialization phase numbers must be greater than P60
    // (i.e. P_HLA_INIT) so that the initialization jobs run after the
    // P60 THLA.manager->initialize() job.
    //------------------------------------------------------------------------
    // Data will only be sent if this federate owns it.
    P100 ("initialization") thla_manager.send_init_data( "A-side-Federate.Test" );

    // Data will only be received if it is remotely owned by another federate.
    P100 ("initialization") thla_manager.receive_init_data( "A-side-Federate.Test" );

    // Do some processing here if needed...

    // Wait for all federates to reach this sync-point.
    P100 ("initialization") thla_manager.wait_for_init_sync_point( "Phase1" );

    // Data will only be sent if this federate owns it.
    P200 ("initialization") thla_manager.send_init_data( "P-side-Federate.Test" );

    // Data will only be received if it is remotely owned by another federate.
    P200 ("initialization") thla_manager.receive_init_data( "P-side-Federate.Test" );

    // Do some processing here if needed...

    // Wait for all federates to reach this sync-point.
    P200 ("initialization") thla_manager.wait_for_init_sync_point( "Phase2" );
}
```

Simulation Multiphase Initialization

S_define: Private data and instantiation

- Protect from copies and do not permit a default constructor.

```
 private:
   TrickHLA::Manager  & thla_manager;
   TrickHLA::Federate & thla_federate;

   // Do not allow the implicit copy constructor or assignment operator.
   THLAInitSimObj(const THLAInitSimObj & rhs);
   THLAInitSimObj & operator=(const THLAInitSimObj & rhs);

   // Do not allow the default constructor.
   THLAInitSimObj();
};
. . .
// Intantiate the initialization simulation object.
THLAInitSimObject THLA_INIT( THLA.manager, THLA.federate );
. . .
```

Data Packing and Unpacking

Data Packing and Unpacking

- TrickHLA supports a data packing and unpacking mechanism so that data transformations can be applied to the data before being sent to (pack) or after received from (unpack) another federate.
- For example, your simulation uses a phase variable in radians but the FOM specifies the phase variable will be exchanged between federates in degrees.
- Packing/unpacking is added to your simulation by extending the `TrickHLAPacking` class and implementing the `pack()` and `unpack()` functions.
- TrickHLA will automatically call your `pack()` and `unpack()` functions at the appropriate time.
- Adding packing/unpacking to your simulation consists of three steps ...

- Step 1: Extend the TrickHLA::Packing class and implement the pack() and unpack() functions in SinePacking.hh :

```
#include SineData.hh
#include "TrickHLA/include/TrickHLAPacking.hh"

class SinePacking : public TrickHLAPacking
{
  public:
   // Initialize the packing object.
   void initialize( SineData * sim_data );
   // From the TrickHLAPacking class.
   virtual void initialize_callback( TrickHLAObject * obj );

   // From the TrickHLAPacking class.
   virtual void pack();
   // From the TrickHLAPacking class.
   virtual void unpack();

  private:
   SineData * sim_data; // -- Simulation data.
   double phase_deg;    // d  Phase offset in degrees.
};
```

- Example in SinePacking.cpp :

```
void SinePacking::initialize( // RETURN: -- None.
      SineData * sim_data)    // IN:     -- Simulation data.
{
   this->sim_data = sim_data;
}
void SinePacking::pack()    // RETURN: -- None.
{
   // For this example to show how to use the Packing API's, we
   // will assume that the phase shared between federates is in
   // degrees so convert it from radians to degrees.
   phase_deg = sim_data->get_phase() * 180.0 / M_PI;
}
void SinePacking::unpack()  // RETURN: -- None.
{
   // For this example to show how to use the Packing API's, we
   // will assume that the phase shared between federates is in
   // degrees so convert it back from degrees to radians.
   sim_data->set_phase( phase_deg * M_PI / 180.0 );
}
```

- Step 2: In the S_define file add your packing object to each simulation object that needs to have its data packed/unpacked.
- Make sure to initialize your packing object if it needs it.

```
class ASimObject : public Trick::SimObject {
   ...
   SineData     sim_data;
   SinePacking  packing;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     ...
   }
};

class PSimObject : public Trick::SimObject {
   ...
   SineData     sim_data;
   SinePacking  packing;

   PSimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     ...
   }
};
```

- Step 3: Configure the data object in the input.py file to use your packing object by setting the data object's packing field. For example in the `RUN_a_side/input.py` file:

```
THLA.manager.objects[0].packing = A.packing
THLA.manager.objects[1].packing = P.packing
```

- Don't forget to update the attribute `trick_name` field to use the correct Trick simulation variable (i.e. the transformed data from the `pack()` and `unpack()` functions). To use the simulation data phase in radians:

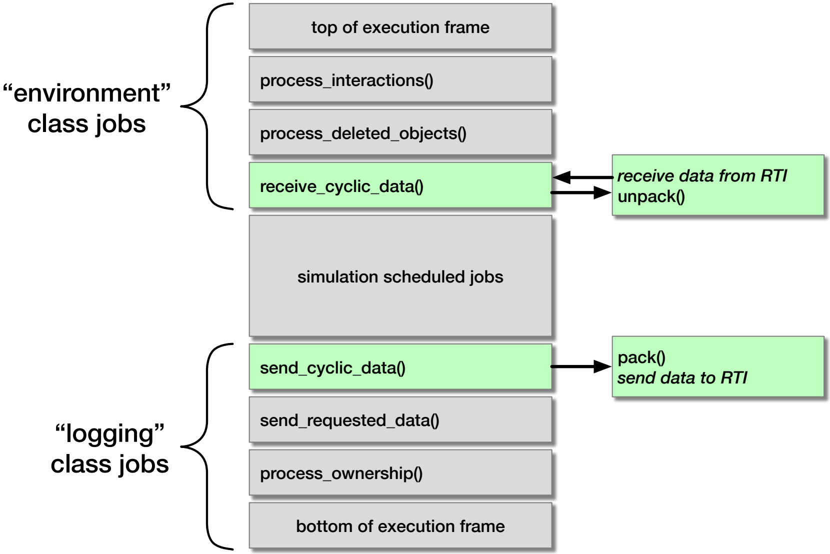
```
THLA.manager.objects[0].attributes[3].trick_name = "A.sim_data.phase"
THLA.manager.objects[1].attributes[3].trick_name = \P.sim_data.phase"
```

- To use the transformed phase in degrees to be shared with the other federates:

```
THLA.manager.objects[0].attributes[3].trick_name = "A.packing.phase_deg"
THLA.manager.objects[1].attributes[3].trick_name = \P.packing.phase_deg"
```

Data Packing and Unpacking

TrickHLA jobs in THLA.sm



Interactions

Interactions

- TrickHLA supports interactions that are either Timestamp Order (TSO) or Receive Order (RO).
- Configuring TrickHLA interactions involves three steps:
  - Step 1: Extend the `TrickHLA::InteractionHandler` class and implement the virtual `receive_interaction()` function.
  - Step 2: Add your interaction-handler object to each simulation object that needs to process interactions in your `S_define` file.
  - Step 3: Configure the interaction-handlers in the `input.py` file.
  - Step 4: Send interactions at specified times using the `input.py` file or programmatically within your simulation.

Interactions

Step 1: Extend the `TrickHLA::InteractionHandler` Class

- Step 1: This is a snippet of the base class from `TrickHLA::InteractionHandler.hh` that you must extend and then implement the virtual `receive_interaction()` function:

```
class TrickHLA::InteractionHandler
{
  ...
  public:
   ...
   virtual void initialize_callback( TrickHLAInteraction * inter );

   bool send_interaction();                          // Receive Order
   bool send_interaction( double send_time ); // Timestamp Order

   // This is a virtual function and must be defined by a full class.
   virtual void receive_interaction();
   ...
};
```

- Extended in SineInteractionHandler.hh:

```
#include "TrickHLA/include/TrickHLAInteractionHandler.hh"

class SineInteractionHandler : public TrickHLAInteractionHandler
{
  ...
  public:
   ...
   void send_sine_interaction( double sim_time );

   virtual void receive_interaction();

   char *   name;    // -- Example of a unique name to identify the interaction handler.
   char *   message; // -- Example of a static array of strings.

  protected:
   double   time;        // s  Example of floating-point data.
   int      year;        // -- Example of integer data.
   int      send_cnt;    // -- The number of times an interaction is sent.
   int      receive_cnt; // -- The number of times an interaction was received.};
};
```

- Extended in SineInteractionHandler.cpp:

```
void SineInteractionHandler::send_sine_interaction( // RETURN: -- None.
   double sim_time)        // IN: s Current simulation time.
{
   time = sim_time;    // Update the time with the simulation time.
   ostringstream msg;  // Create a message to send
   msg << "Interaction from:\"" << ((name != NULL) ? name : "Unknown") << "\" "
       << "Send-count:" << (send_cnt + 1);
   if ( ( message != NULL ) && TMM_is_alloced( message ) ) {
      TMM_delete_var_a( message );
   }
   message = TMM_strdup( (char *)msg.str().c_str() );

   double lookahead_time = get_fed_lookahead().getDoubleTime();
   double timestamp = time + lookahead_time;
   // Notify the parent interaction handler to send the interaction using
   // Timestamp Order (TSO) at the current simulation time plus the lookahead_time.
   bool was_sent = this->TrickHLAInteractionHandler::send_interaction( timestamp );
   if ( was_sent ) {
      cout << "++++SENDING++++ SineInteractionHandler::send_sine_interaction() << endl
           << "  name:'" << ((name != NULL) ? name : "NULL") << "'" << endl
           << "  message:'" << ((message != NULL) ? message : "NULL") << "'" << endl
           << "  timestamp:" << timestamp;
      send_cnt++; // Update send count, just used for the message in this example
   } else {
      cout << "+-+-NOT SENT-+-+ SineInteractionHandler::send_sine_interaction()" << endl
}
```

- Extended in SineInteractionHandler.cpp (Continued):

```
void SineInteractionHandler::receive_interaction() // RETURN: -- None.
{
   receive_cnt++;
   double sim_time = exec_get_sim_time();

   cout << "++++RECEIVING++++ SineInteractionHandler::receive_interaction()" << endl
        << "  name:'" << ((name != NULL) ? name : "NULL") << "'" << endl
        << "  message:'" << ((message != NULL) ? message : "NULL") << "'" << endl
        << "  message length:" << ((message != NULL) ? strlen(message) : 0) << endl
        << "  sim_time:" << sim_time << endl
        << "  time:" << time << endl
        << "  year:" << year << endl
        << "  receive_cnt:" << receive_cnt << endl;
}
```

- Step 2: In the `S_define` file add your interaction-handler object to each simulation object that needs to process interactions.

```
class ASimObject : public Trick::SimObject {

   SineData     sim_data;
   SinePacking            packing;
   SineInteractionHandler interaction_handler;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     ...
   }
};

class PSimObject : public Trick::SimObject {
   ...
   SineData     sim_data;
   SinePacking            packing;
   SineInteractionHandler interaction_handler;

   PSimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     ...
   }
};
```

- Step 3: Example in the RUN_a_side/input.py file:

```
# We are taking advantage of the input file to specify a unique name for each handler
A.interaction_handler.name = "A-side: A.interaction_handler.name"
P.interaction_handler.name = "A-side: P.interaction_handler.name"

# Trick HLA Interactions and Parameters.
THLA.manager.inter_count  = 1
THLA.manager.interactions = trick.alloc_type( THLA.manager.inter_count, "TrickHLAInteraction" )

THLA.manager.interactions[0].FOM_name    = "Communication"
THLA.manager.interactions[0].publish     = True
THLA.manager.interactions[0].subscribe   = False
THLA.manager.interactions[0].handler     = A.interaction_handler
THLA.manager.interactions[0].param_count = 3
THLA.manager.interactions[0].parameters  = trick.alloc_type( THLA.manager.interactions[0].param_count,
                                           "TrickHLAParameter" )
THLA.manager.interactions[0].parameters[0].FOM_name    = "Message"
THLA.manager.interactions[0].parameters[0].trick_name = "A.interaction_handler.message"
THLA.manager.interactions[0].parameters[0].rti_encoding = trick.THLA_UNICODE_STRING

THLA.manager.interactions[0].parameters[1].FOM_name    = "time"
THLA.manager.interactions[0].parameters[1].trick_name = "A.interaction_handler.time"
THLA.manager.interactions[0].parameters[1].rti_encoding = trick.THLA_LITTLE_ENDIAN

THLA.manager.interactions[0].parameters[2].FOM_name    = "year"
THLA.manager.interactions[0].parameters[2].trick_name = "A.interaction_handler.year"
THLA.manager.interactions[0].parameters[2].rti_encoding = trick.THLA_LITTLE_ENDIAN
```

- Step 3: Example in the RUN_p_side/input.py file:

```
# We are taking advantage of the input file to specify a unique name for each handler
A.interaction_handler.name = P-side: A.interaction_handler.name"
P.interaction_handler.name = P-side: P.interaction_handler.name"

# TrickHLA Interactions and Parameters.
THLA.manager.inter_count  = 1
THLA.manager.interactions = trick.alloc_type( THLA.manager.inter_count, "TrickHLAInteraction" )

THLA.manager.interactions[0].FOM_name    = "Communication"
THLA.manager.interactions[0].publish     = False
THLA.manager.interactions[0].subscribe   = True
THLA.manager.interactions[0].handler     = P.interaction_handler
THLA.manager.interactions[0].param_count = 3
THLA.manager.interactions[0].parameters  = trick.alloc_type( THLA.manager.interactions[0].param_count,
                                           "TrickHLAParameter" )
THLA.manager.interactions[0].parameters[0].FOM_name    = "Message"
THLA.manager.interactions[0].parameters[0].trick_name = "P.interaction_handler.message"
THLA.manager.interactions[0].parameters[0].rti_encoding = trick.THLA_UNICODE_STRING

THLA.manager.interactions[0].parameters[1].FOM_name    = "time"
THLA.manager.interactions[0].parameters[1].trick_name = "P.interaction_handler.time"
THLA.manager.interactions[0].parameters[1].rti_encoding = trick.THLA_LITTLE_ENDIAN

THLA.manager.interactions[0].parameters[2].FOM_name    = "year"
THLA.manager.interactions[0].parameters[2].trick_name = "P.interaction_handler.year"
THLA.manager.interactions[0].parameters[2].rti_encoding = trick.THLA_LITTLE_ENDIAN
```

Interactions

Step 4: Sending Interactions Using the Input File

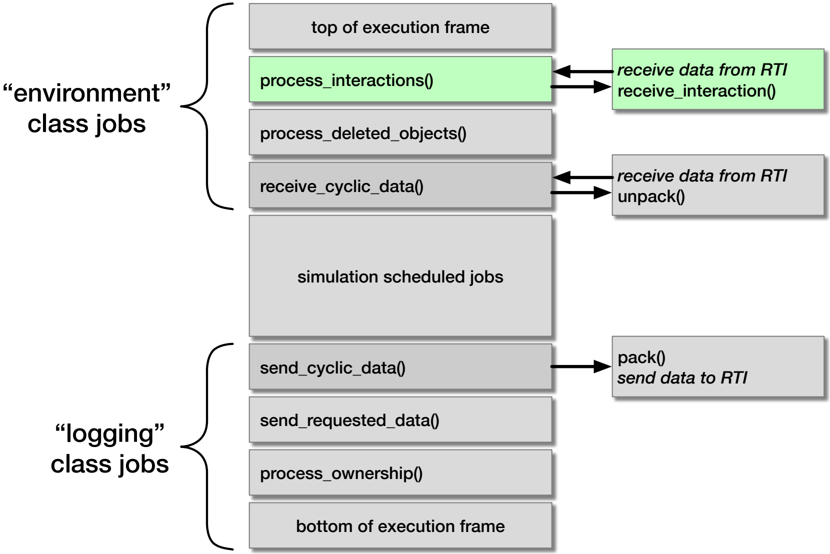
- Step 4: Add entries to the `input.py` file for when you want to send an interaction. For example, to send an interaction at time 10.0:

```
trick.add_read( 10.0, A.interaction_handler.send_sine_interaction(10.0) )
```

- Sending an interaction can be initiated three different ways:
  - Programmatically, call one of the `send_interaction()` functions inherited by your interaction-handler.
  - As a job in your simulation specified in the `S_define` file, which in turn calls `send_interaction()` as above.
  - As a job in your simulation called at a specific time from the `input.py` file, which in turn calls `send_interaction()` as above.
- TrickHLA will automatically process received interactions and call the `receive_interaction()` function of the appropriate user defined interaction-handler.
- TrickHLA handles interactions in a thread-safe and Trick simulation environment job-safe way.

Interactions

TrickHLA jobs in THLA.sm



Lag Compensation

Lag Compensation

- An in depth discussion of lag/latency compensation goes beyond the scope of this training. Please see the following paper for more detailed information on the subject:

  R. Phillips, E. Crues, Time Management Issues and Approaches for Real Time HLA Based Simulations, Proceedings of the Fall 2005 Simulation Interoperability Workshop and Conference, Fall 2005.

- Lag Compensation is an attempt to correct for the time difference between when a state is published at one federate and received at another federate.
- Using TrickHLA lag compensation consists of four steps:
  - Step 1: Extend the `TrickHLALagCompensation` class and implement the `sending_lag_compensation()` and `receive_lag_compensation()` virtual functions.
  - Step 2: In the `S_define` file add a lag-compensation object to each simulation object that needs to perform lag compensation.
  - Step 3: Configure lag compensation in the `input.py` file.
  - Step 4: Update the object attributes in the `input.py` file to use the lag compensated Trick simulation variable.

- Step 1: Extend the `TrickHLALagCompensation` class and implement the `sending_lag_compensation()` and `receive_lag_compensation()` virtual functions. Example in `SineLagCompensation.hh`:

```
#include "SineData.hh"
#include "TrickHLA/include/TrickHLALagCompensation.hh"

class SineLagCompensation : public TrickHLALagCompensation
{
...
public:
   // From the TrickHLALagCompensation class.
   virtual void sending_lag_compensation( double current_time, double dt );

   // From the TrickHLALagCompensation class.
   virtual void receive_lag_compensation( double current_time, double dt );
...
};
```

- TrickHLA will automatically call the send or receive lag compensation functions at the appropriate time.

- Step 2: In the S_define file add a lag-compensation object to each simulation object that needs to perform lag compensation. Make sure to initialize your lag-compensation object if it needs it.

```
class ASimObject : public Trick::SimObject {
   ...
   SineData      sim_data;
   SineData      lag_comp_data;
   SinePacking            packing;
   SineLagCompensation    lag_compensation;
   SineInteractionHandler interaction_handler;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     P50 ("initialization") lag_compensation.initialize( &sim_data,
                                                          &lag_comp_data );
     ...
   }
};
```

Lag Compensation
Step 3: Configuration

- Step 3: Configure lag compensation in the `input.py` file.

```
THLA.manager.objects[0].lag_comp      = A.lag_compensation
THLA.manager.objects[0].lag_comp_type = trick.THLA_LAG_COMP_SENDING
```

- The supported lag compensation types are:
  - THLA_LAG_COMP_NONE (default)
  - THLA_LAG_COMP_SENDING
  - THLA_LAG_COMP_RECEIVE

- Step 4: Update the attribute `trick_name` field in the `input.py` file to use the lag compensated Trick simulation variable.
- If the lag compensation type is `THLA_LAG_COMP_NONE`:

```
THLA.manager.objects[0].attributes[1].FOM_name   = "Value"
THLA.manager.objects[0].attributes[1].trick_name = "A.sim_data.value"
```

- If the lag compensation type is `THLA_LAG_COMP_SENDING` or `THLA_LAG_COMP_RECEIVE`:

```
THLA.manager.objects[0].attributes[1].FOM_name   = "Value"
THLA.manager.objects[0].attributes[1].trick_name = "A.lag_comp_data.value"
```

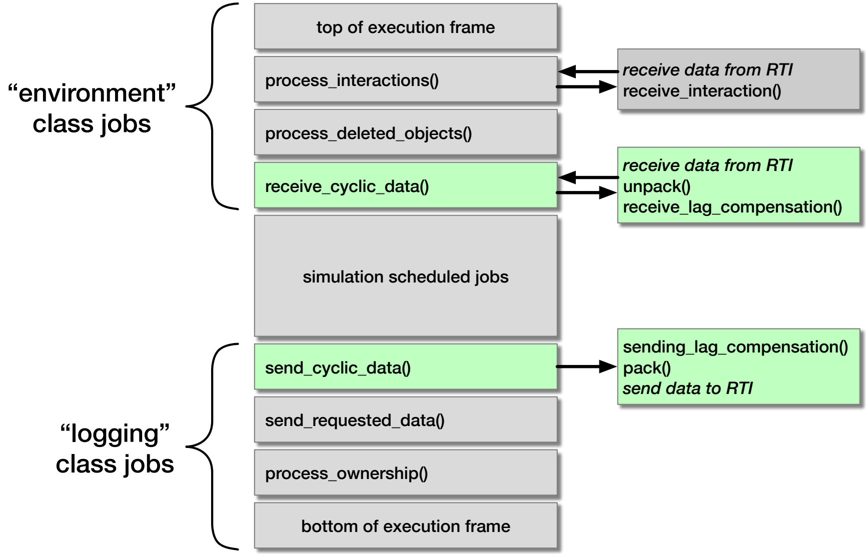
- The five logical combinations of lag compensation between two federates is shown in the table below:

Lag Comp. Type	A-side-Federate	P-side-Federate
None	Publish: THLA_LAG_COMP_NONE Subscribe: THLA_LAG_COMP_NONE	Subscribe: THLA_LAG_COMP_NONE Publish: THLA_LAG_COMP_NONE
Sending-side	Publish: THLA_LAG_COMP_SENDING Publish: THLA_LAG_COMP_SENDING	Subscribe: THLA_LAG_COMP_NONE Subscribe: THLA_LAG_COMP_NONE
Receive-side	Publish: THLA_LAG_COMP_NONE Publish: THLA_LAG_COMP_NONE	Subscribe: THLA_LAG_COMP_RECEIVE Subscribe: THLA_LAG_COMP_RECEIVE
Sending- and Receive-side	Publish: THLA_LAG_COMP_SENDING Subscribe: THLA_LAG_COMP_NONE	Subscribe: THLA_LAG_COMP_RECEIVE Publish: THLA_LAG_COMP_NONE
Sending- and Receive-side	Publish: THLA_LAG_COMP_NONE Publish: THLA_LAG_COMP_SENDING	Subscribe: THLA_LAG_COMP_NONE Subscribe: THLA_LAG_COMP_RECEIVE

- "Publish" refers to the attributes the federate is configured to send data for.
- "Subscribe" refers to the attributes the federate is configured to receive data for.

Lag Compensation

TrickHLA jobs in THLA.sm



Ownership Transfer

Ownership Transfer

- TrickHLA supports ownership transfer down to the individual attribute.
- Ownership of attributes can be Pulled from the owning federate or federates.
- Ownership of attributes can be Pushed to any accepting federate or federates.
- Multiple Push/Pull requests can be scheduled for different times and for specific attributes or all attributes.
- Automatically handles attribute state publication until attribute ownership has been transferred.
- Pushing attribute ownership will result in the RTI deciding which federate(s) get ownership of which attributes. You don't have control over which federate you get to push attribute ownership to.
- Pulling attribute ownership gives you control over which federate gets ownership of a particular attribute.
- There are two approaches to using ownership transfer:
  - Programmatically, which requires more programming effort.
  - Entries in the `input.py` file, which requires the least effort.

- The state of the publish, subscribe and locally_owned attribute fields (in the `input.py` file) affect ownership transfer.

```
THLA.manager.objects[0].attributes[1].publish       = True
THLA.manager.objects[0].attributes[1].subscribe     = True
THLA.manager.objects[0].attributes[1].locally_owned = True
```

Pulish	**Locally Owned**	**Push Ownership to Another Federate**	**Pull Ownership from Another Federate**	**Another Federate Wants to Pull Ownership**	**Another Federate Wants to Push Ownership**
True	True	Yes	No	Yes	No
True	False	No	Yes	No	Yes
False	True	Yes	No	Yes	No
False	False	No	No	No	No

Subscribe	**Will receive attribute reflections when locally_owned == false?**
True	Yes
False	No

- The approach of programmatically performing ownership transfers requires more programming effort and has three steps:
  - Step 1: Extend the `TrickHLAOwnershipHandler` class.
  - Step 2: In the `S_define` file add your ownership-handler to each simulation object that needs to process ownership transfers.
  - Step 3: Configure ownership transfer in the `input.py` file.
- The approach of using the input.py file for ownership transfer requires the least effort and has three steps:
  - Step 1: In the `S_define` file add the default ownership-handler to each simulation object that needs to process ownership transfers.
  - Step 2: Configure ownership transfer in the input.py file.
  - Step 3: Add entries to the `input.py` file for when you want to push or pull ownership.

Ownership Transfer

Programmatic Approach – Step 1

- Step 1: This is a snippet of the base class from `TrickHLAOwnershipHandler.hh` that you must extend.

```
class TrickHLAOwnershipHandler
{
    ...
  public:
   virtual void initialize_callback( TrickHLAObject * obj );

   string get_object_name();
   string get_object_FOM_name();

   int get_attribute_count();
   VectorOfStrings get_attribute_FOM_names() const;

   bool is_locally_owned( const char * attribute_FOM_name );
   bool is_remotely_owned( const char * attribute_FOM_name );

   bool is_published( const char * attribute_FOM_name );
   bool is_subscribed( const char * attribute_FOM_name );

   void pull_ownership();
   void pull_ownership( double time );
   void pull_ownership( const char * attribute_FOM_name );
   void pull_ownership( const char * attribute_FOM_name, double time );

   void push_ownership();
   void push_ownership( double time );
   void push_ownership( const char * attribute_FOM_name );
   void push_ownership( const char * attribute_FOM_name, double time );
   ...
};
```

Ownership Transfer

Programmatic Approach – Step 1 Continued

- Example in `SineOwnershipHandler.hh` :

```
#include "TrickHLA/include/TrickHLAOwnershipHandler.hh"

class SineOwnershipHandler : public TrickHLAOwnershipHandler
{
  ...
  public:
   // We override this function so that we can initialize ownership
   // transfer of some attributes at a specific time.
   virtual void initialize_callback( TrickHLAObject * obj );

};
```

Ownership Transfer

Programmatic Approach – Step 1 Continued

- Example in `SineOwnershipHandler.cpp`:

```
void SineOwnershipHandler::initialize_callback( // RETURN: -- None.
   TrickHLAObject * obj ) // IN: -- Associated object for attribute ownership.
{
   // Make sure we call the original function so that the callback is initialized.
   this->TrickHLAOwnershipHandler::initialize_callback( obj );

   // Examples showing how to Pull all attributes.
   pull_ownership();          // As soon as possible for all attributes.
   pull_ownership( 3.0 );
   // Examples showing how to Pull specific attributes.
   pull_ownership( "Time" );  // As soon as possible for this attribute.
   pull_ownership( "Value", 6.1 );

   // Examples showing how to Push all attributes.
   push_ownership();          // As soon as possible for all attributes.
   push_ownership( 5.0 );
   // Examples showing how to Push specific attributes.
   push_ownership( "Time" );  // As soon as possible for this attribute.
   push_ownership( "Value", 6.1 );
}
```

Note: This example is using the handler to set up pushes and pulls at initialization time. Instead you could programmatically call `push_ownership()` or `pull_ownership()` at any time.

Ownership Transfer

Programmatic Approach – Step 2

- Step 2: In the `S_define` file add your custom ownership-handler to each simulation object that needs to process ownership transfers.

```
class ASimObject : public Trick::SimObject {
   ...
   SineData      sim_data;
   SineData      lag_comp_data;
   SineOwnershipHandler   ownership_handler;
   SinePacking            packing;
   SineLagCompensation    lag_compensation;
   SineInteractionHandler interaction_handler;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     P50 ("initialization") lag_compensation.initialize( &sim_data,
                                                          &lag_comp_data );
     ...
   }
};
```

Ownership Transfer

Programmatic Approach – Step 3

- Step 3: Configure the data object in the `input.py` file to use your ownership-handler object by setting the data object's ownership field.
- For example in the `RUN_a_side/input.py` file:

```
THLA.manager.objects[0].packing   = A.packing
THLA.manager.objects[0].ownership = A.ownership_handler
```

Ownership Transfer

Input File Approach – Step 1

- Step 1: In the `S_define` file add the default ownership-handler to each simulation object that needs to process ownership transfers.

```
class ASimObject : public Trick::SimObject {
   ...
   SineData      sim_data;
   SineData      lag_comp_data;
   TrickHLAOwnershipHandler ownership_handler;
   SinePacking               packing;
   SineLagCompensation       lag_compensation;
   SineInteractionHandler    interaction_handler;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     P50 ("initialization") lag_compensation.initialize( &sim_data,
                                                          &lag_comp_data );
     ...
   }
};
```

Ownership Transfer

Input File Approach – Step 2

- Step 2: Configure the data object in the `input.py` file to use your ownership-handler object (which in this case is the TrickHLA handler) by setting the data object's ownership field.
- For example in the `RUN_a_side/input.py` file:

```
THLA.manager.objects[0].packing   = A.packing
THLA.manager.objects[0].ownership = A.ownership_handler
```

Ownership Transfer

Input File Approach – Step 3

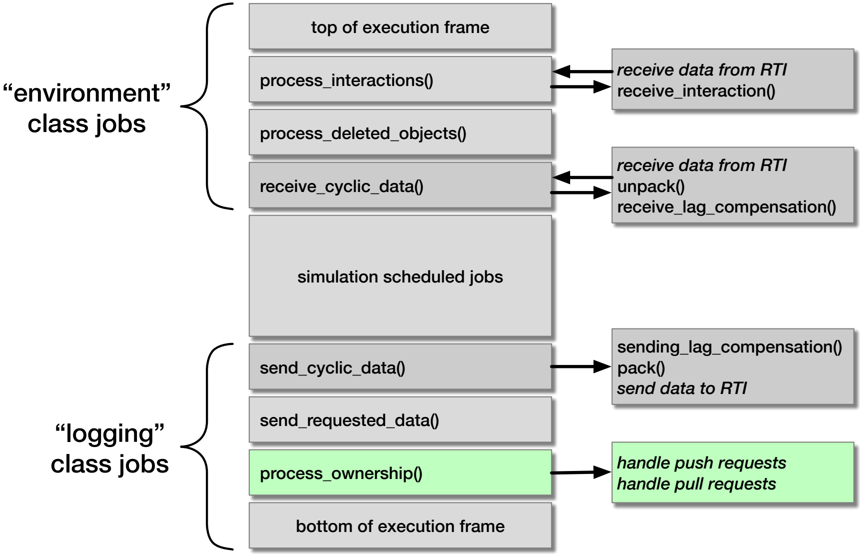
- Step 3: Add entries to the input.py file for when you want to push or pull ownership.
- For example:

```
# Push ownership of the A-side-Federate.Test object attributes
# at the RTI time of 4.0 seconds
trick.add_read(4.0, \A.ownership_handler.push_ownership()")

# Pull back ownership of the A-side-Federate.Test object attributes
# at the RTI time of 8.0 seconds.
trick.add_read(8.0, \A.ownership_handler.pull_ownership()")
```

Ownership Transfer

TrickHLA jobs in THLA.sm



Ownership Transfer

PITFALL - Handling Mixed Ownership

- With ownership transfer, it is possible to have an object containing attributes you own and attributes you don't own.
- TrickHLA knows to only send attributes you own, and only receive attributes you don't own. However ...
- Your `unpack()` or `receive_lag_compensation()` functions run AFTER TrickHLA receives data, so they could accidently override your simulation state for attributes that you own and publish. This results in corrupted simulation data!
- To avoid overriding your simulation state, the solution is to determine if the attribute is owned by another federate in your `unpack()` and `receive_lag_compensation()` functions.

NOTE: the same scenario for your `pack()` or `send_lag_compensation()` functions is not a problem because they run BEFORE TrickHLA sends data, and because TrickHLA will not send data you don't own, no harm done.

Ownership Transfer

PITFALL - Handling Mixed Ownership in Unpacking

- Step 1: Add a `TrickHLA::Attribute` reference for each of your simulation state attributes. Example in `SinePacking.hh`:

```
#include "SineData.hh"
#include "TrickHLA/include/TrickHLAAttribute.hh"
#include "TrickHLA/include/TrickHLAPacking.hh"

class SinePacking : public TrickHLAPacking
{
  public:
   ...
   // Initialize the packing object.
   void initialize( SineData * sim_data );
   // From the TrickHLAPacking class.
   virtual void initialize_callback( TrickHLAObject * obj );

   // From the TrickHLAPacking class.
   virtual void pack();
   // From the TrickHLAPacking class.
   virtual void unpack();

  private:
   SineData * sim_data; // -- Simulation data.
   double phase_deg;    // d  Phase offset in degrees.

   TrickHLAAttribute * phase_attr; // ** Reference to Phase TrickHLAAttribute.
};
```

- Step 2: Override the `initialize_callback()` function to set the attribute references.
- Example in `SinePacking.cpp`:

```
void SinePacking::initialize_callback( // RETURN: -- None.
   TrickHLAObject * obj ) // IN: -- Object associated with this packing class.
{
   // We must call the original function so that the callback is initialized.
   this->TrickHLAPacking::initialize_callback( obj );

   // Get a reference to the TrickHLAAttribute for the "Phase" FOM attribute.
   // We do this here so that we only do the attribute lookup once instead of
   // looking it up every time the unpack function is called.
   phase_attr = get_attribute_and_validate( "Phase" );
}
```

- Step 3: Check the attribute ownership in the unpack() function.

```
void SinePacking::unpack() // RETURN: -- None.
{
   // If the HLA phase attribute has changed and is remotely owned (i.e. is
   // coming from another federate) then override our simulation state with the
   // incoming value. If we locally own the "Phase" attribute then we do not
   // want to override it's value. If we did not do this check then we would be
   // overriding state of something we own and publish with whatever value
   // happen to be in the "phase_deg" local variable, which would cause data
   // corruption of the state. We always need to do this check because
   // ownership transfers could happen at any time or the data could be at a
   // different rate.
   if ( phase_attr->is_received() ) {
      // For this example to show how to use the Packing API's, we will
      // assume that the phase shared between federates is in degrees so
      // covert it back from degrees to radians.
      sim_data->set_phase( phase_deg * M_PI / 180.0 );
   }
}
```

Object Deletion

Object Deletion

- TrickHLA supports notification of HLA object deletions.
- An HLA object can be deleted as a result of a federate resigning or by explicit deletion by the federate.
- Using TrickHLA object deletion consists of three steps:
  - Step 1: Extend the `TrickHLA::ObjectDeleted` class and implement the `deleted()` virtual function.
  - Step 2: In the `S_define` file add an object-deleted object to each simulation object that needs to be notified of a deletion.
  - Step 3: Configure object deleted in the `input.py` file.

Ownership Transfer

Step 1: Extend the TrickHLA::ObjectDeleted Class

- Step 1: Extend the `TrickHLA::ObjectDeleted` class and implement the `deleted()` virtual function.
- Example in `SineObjectDeleted.hh`:

```
#include "TrickHLA/include/TrickHLAObjectDeleted.hh"

class SineObjectDeleted : public TrickHLAObjectDeleted
{
...
  public:
...
   void deleted(                    // RETURN: -- None.
        TrickHLAObject * obj );     // IN:     -- Deleted object.
};
```

Ownership Transfer

Step 1: Extend the TrickHLA::ObjectDeleted Class (Continued)

- Step 1 continued: Example in SineObjectDeleted.cpp:

```
void SineObjectDeleted::deleted( // RETURN: -- None.
   TrickHLAObject * obj)         // IN: -- Object which was deleted.
{
   std::ostringstream msg;
   msg << "SineObjectDeleted::deleted() Object '" << obj->get_name()
       << "' deleted from the federation.";
   send_hs( stdout, (char *) msg.str().c_str() );
}
```

Ownership Transfer

Step 2: Add Object-deleted Object to S_define

- Step 2: In the `S_define` file add an object-deleted object to each simulation object that needs to be notified of a deletion.

```
class ASimObject : public Trick::SimObject {
   ...
   SineData     sim_data;
   SineData     lag_comp_data;
   SineOwnershipHandler   ownership_handler;
   SinePacking            packing;
   SineLagCompensation    lag_compensation;
   SineInteractionHandler interaction_handler;
   SineObjectDeleted      obj_deleted_callback;

   ASimObject() {
     P50 ("initialization") packing.initialize( &sim_data );
     P50 ("initialization") lag_compensation.initialize( &sim_data,
                                                          &lag_comp_data );
     ...
   }
};
```

- Step 3: Configure object deleted in the `input` file.
- In the `RUN_a_side/input.py` file:

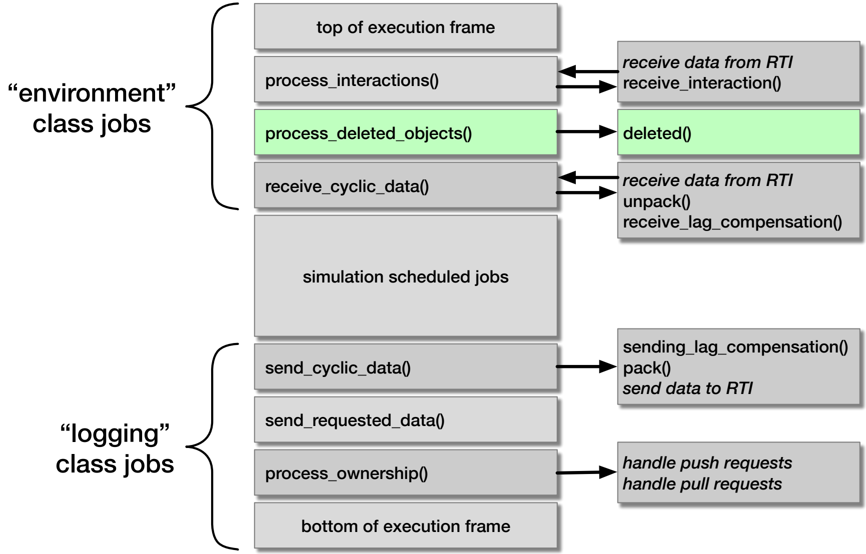
```
THLA.manager.objects[0].deleted = A.obj_deleted_callback
THLA.manager.objects[1].deleted = P.obj_deleted_callback
```

- In the `RUN_p_side/input.py` file:

```
THLA.manager.objects[0].deleted = A.obj_deleted_callback
THLA.manager.objects[1].deleted = P.obj_deleted_callback
```

Object Deletion

TrickHLA jobs in THLA.sm



TrickHLA v3

An HLA interface package for Trick

Edwin Z. Crues, Ph.D.
Daniel E. Dexter

Simulation and Graphics Branch (ER7)
NASA Johnson Space Center
2101 NASA Parkway, Houston, Texas, 77058

edwin.z.crues@nasa.gov
daniel.e.dexter@nasa.gov

January 2021