



# EMANE

## Developer Training

0.7.3

# Extendable Mobile Ad-hoc Emulator

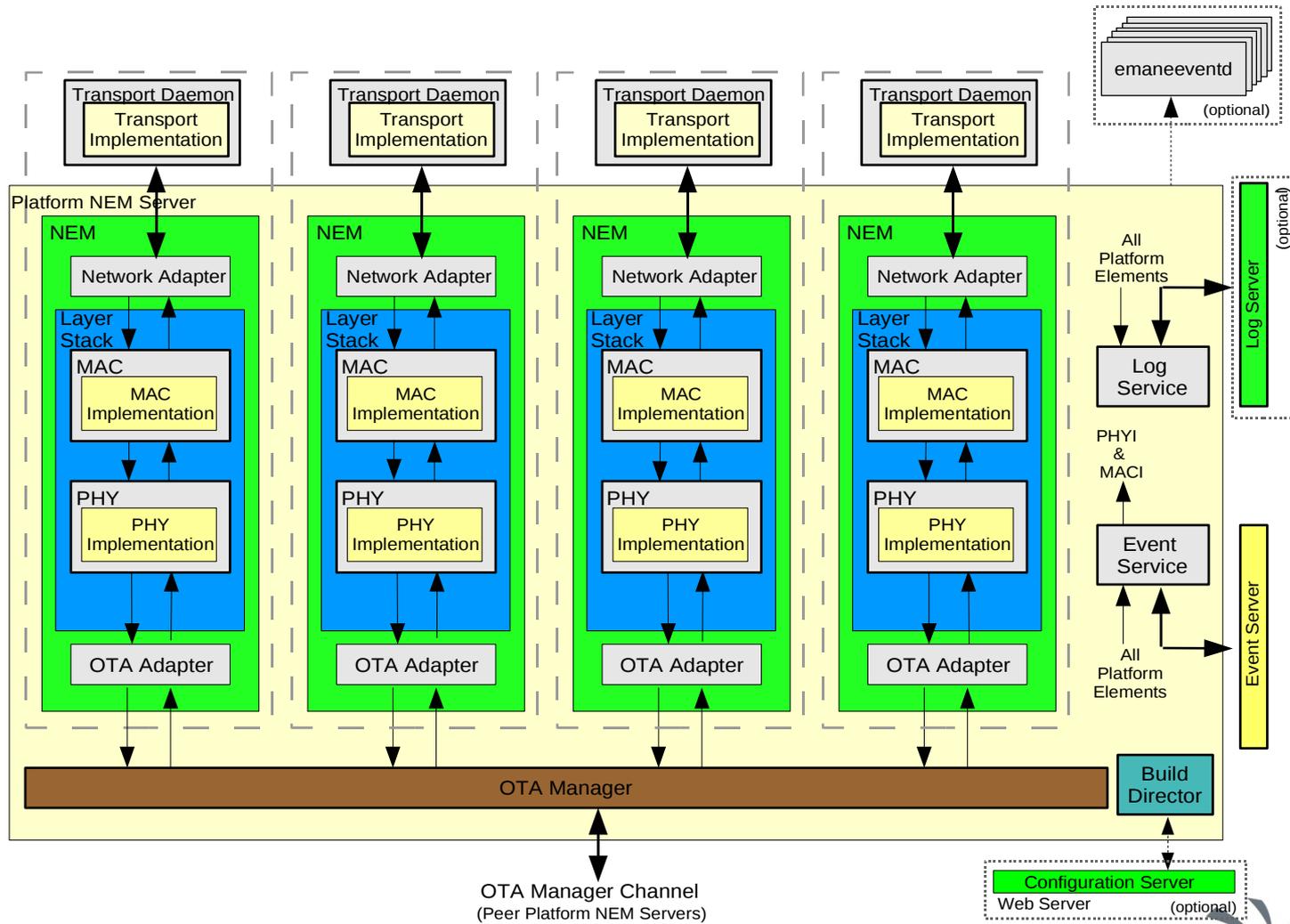
*The Extendable Mobile Ad-hoc Network Emulator (EMANE) is an open source framework which provides wireless network experimenters with a highly flexible modular environment for use during the design, development and testing of simple and complex network architectures.*

*EMANE provides a set of well-defined APIs to allow independent development of network emulation modules, emulation/application boundary interfaces and emulation environmental data distribution mechanisms.*

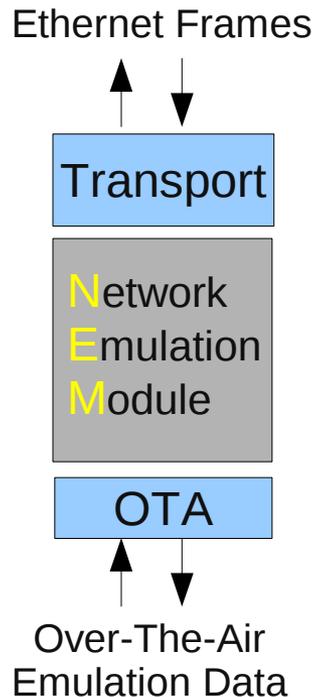
# Extendable Mobile Ad-hoc Emulator

- Supports emulation of simple as well as complex network architectures
- Supports emulation of multichannel gateways
- Supports model specific configuration and control messaging
- Provides mechanisms to bridge emulation environment control information with non-emulation aware components
- Supports large scale testbeds with the same ease as small test networks
- Supports cross platform deployment (Unix, Linux, OSX, MS Windows)

# EMANE Architecture



# Emulation Stack Anatomy



- *Application/Emulation Boundary (Transport)* – Mechanism responsible for transporting data to and from the emulation space.
- *Network Emulation Module (NEM)* - Emulation implementation functionality for a given radio model
- *Over-The-Air Manager* - Provides the mechanism NEMs use to communicate

# Transport

- Realization of an emulation/application boundary interface. Provides the entry and exit point for all data routed through the emulation.

For example the two ethernet frame transports: Virtual Transport and Raw Transport are responsible for interfacing with the underlying operating system

- TunTap (linux, OS X)
- WinTap (win32)
- libpcap (linux, OS X)
- winpcap (win32)

Supports:

- IPv4
  - IPv6
  - Unicast
  - Broadcast
  - Multicast
  - Throughput limitation
- Tunnels transport data as opaque payload to the corresponding NEM.

# Network Emulation Module

- NEMs composed of two components

## MAC Layer – Medium Access Control Layer emulation functionality

- CSMA
- TDMA & Hybrid schemes
- Queue Management, Discard, QOS
- Adaptive Transmission Protocols (Power and Data Rate)
- Packet treatment based on BER, SINR and packet size

## PHY Layer – Physical Layer emulation functionality

- Filter out of band packets
- Directional antenna support
- Waveform Timing
- Noise Estimation
- Half duplex operations
- Transmit Power and antenna gain

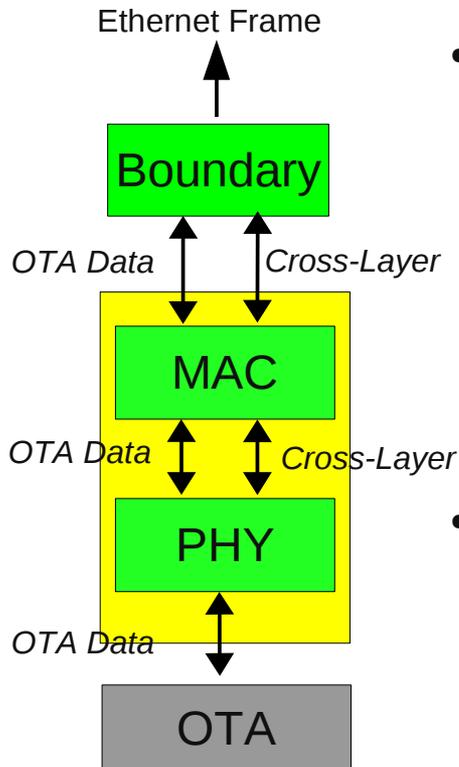
# Over-The-Air Manager

- Messaging infrastructure to deliver emulation radio model messages to all nodes in the deployment
  - Uses a multicast channel for message distribution for NEMs hosted by an NEM server different than that of the source NEM
  - Uses thread shared memory for message distribution for NEMs hosted by the same NEM server
- All messages are delivered to every NEM participating in the emulation (PHY Layer activity)
  - Provides the ability to model complex PHY phenomena such as RF interference
  - Multiple OTA multicast channels can be used to reduce overhead
  - OTA multicast channel can be disabled when utilizing a single NEM server



# Cross-Layer Communication

- Each stack layer has both a data and control path
- Control messages are only valid between contiguous layers



Examples of control message use:

- Per packet RSSI (PHY to MAC)
  - Carrier Sense (PHY to MAC)
  - Transmission Control (MAC to PHY)
- Layer data destined for a corresponding layer uses the opaque data path for messaging
    - Data can be placed in a layer specific header of an existing downstream data message
    - A new data message may be generated just for layer specific messaging

# Emulation Events

- Emulation data is distributed in realtime to NEMs by the EMANE Event Service. Event data is distributed using the Event Multicast channel.
- Emulation components that generate events are called Event Generators.
  - Events are distributed as opaque data
  - Only Event Generators and components subscribed to the specific events process the data
- Emulation event data is also available to non emulation components through the EMANE Event Daemon
  - An event agent plugin API exposes all transmitted events for external processing without exposing the mechanisms used to transmit the data

# EMANE Applications

- *emane* – NEM Platform Server application. Creates and manages one or more NEMs.

Input XML: platform file

- *emanetransportd* – Transport Daemon application. Creates and manages one or more transports.

Input XML: transportdaemon file

- *emaneeventd* – Event Daemon application. Creates and manages one or more event agents.

Input XML: eventdaemon file

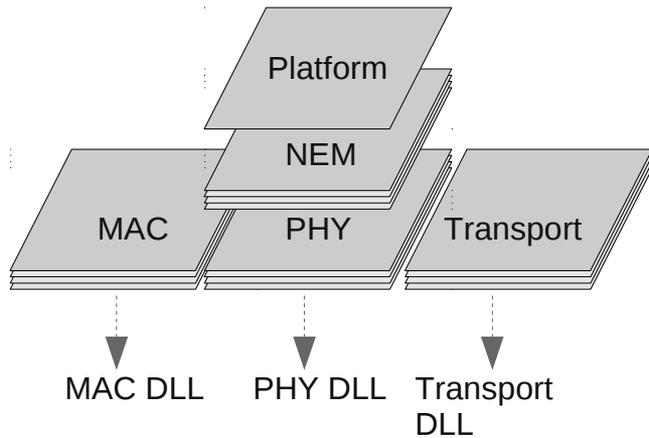
- *emaneeventservice* – Event Service application. Creates and manages one or more event generators.

Input XML: eventservice file

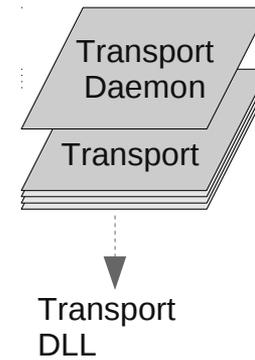


# EMANE XML Hierarchy

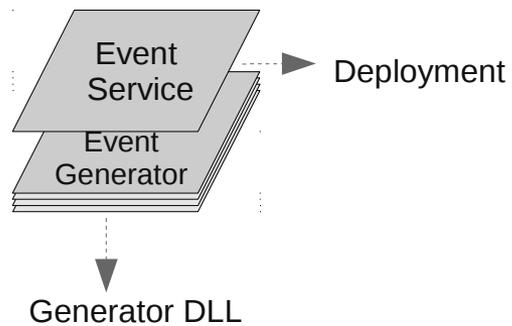
## NEM Platform Server



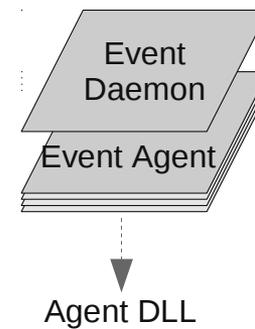
## Transport Daemon



## Event Service



## Event Daemon



# Constructing NEM Layers

- `EMANE::NEMDirector` is responsible for parsing the Platform Server configuration and determining configuration item values based on the referenced NEM and NEM Layer component configurations.

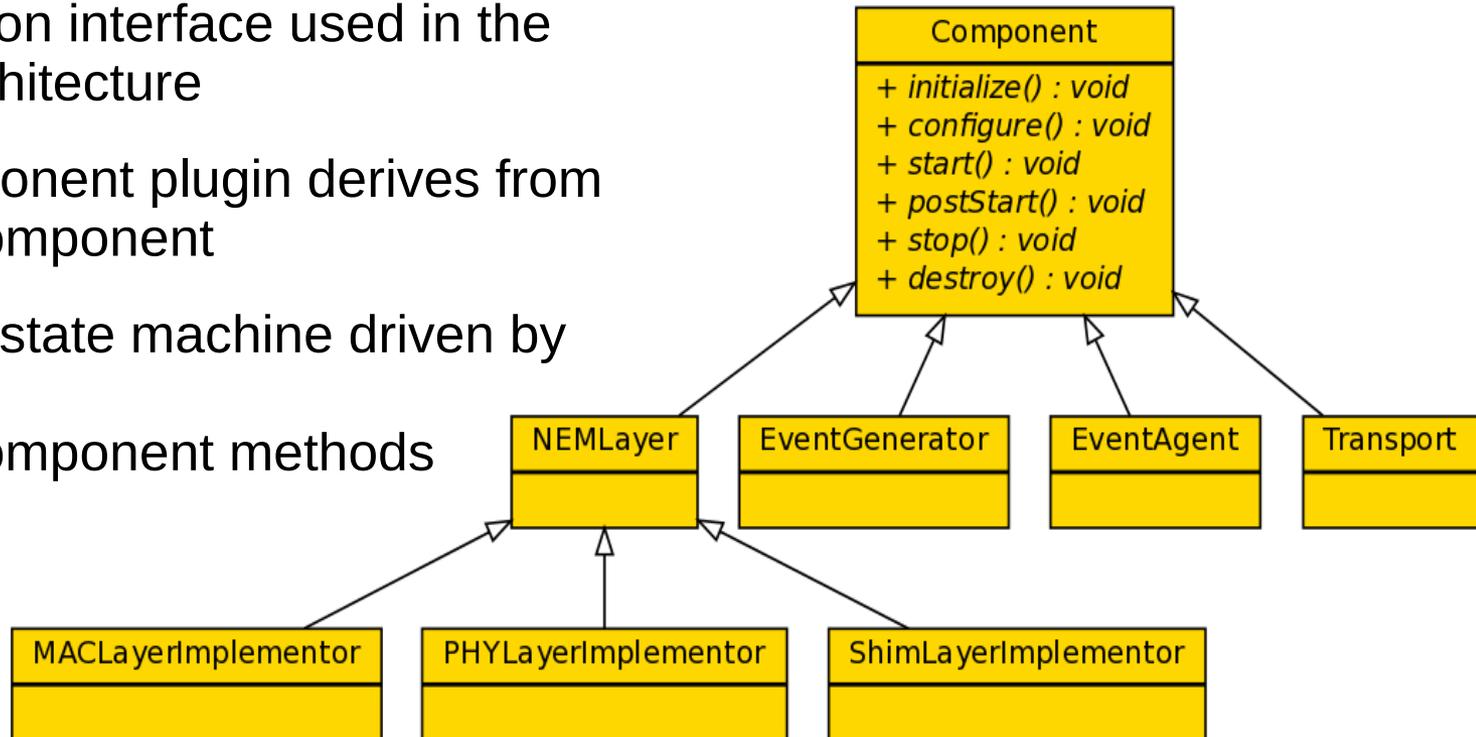
[emane/src/nemdirector.h](#)

- Uses `EMANE::NEMBuilder` to instantiate all the required components.
- `EMANE::NEMBuilder` is the only location within the infrastructure where the actual NEM component plugin type is known.

[emane/src/nembuilder.h](#)

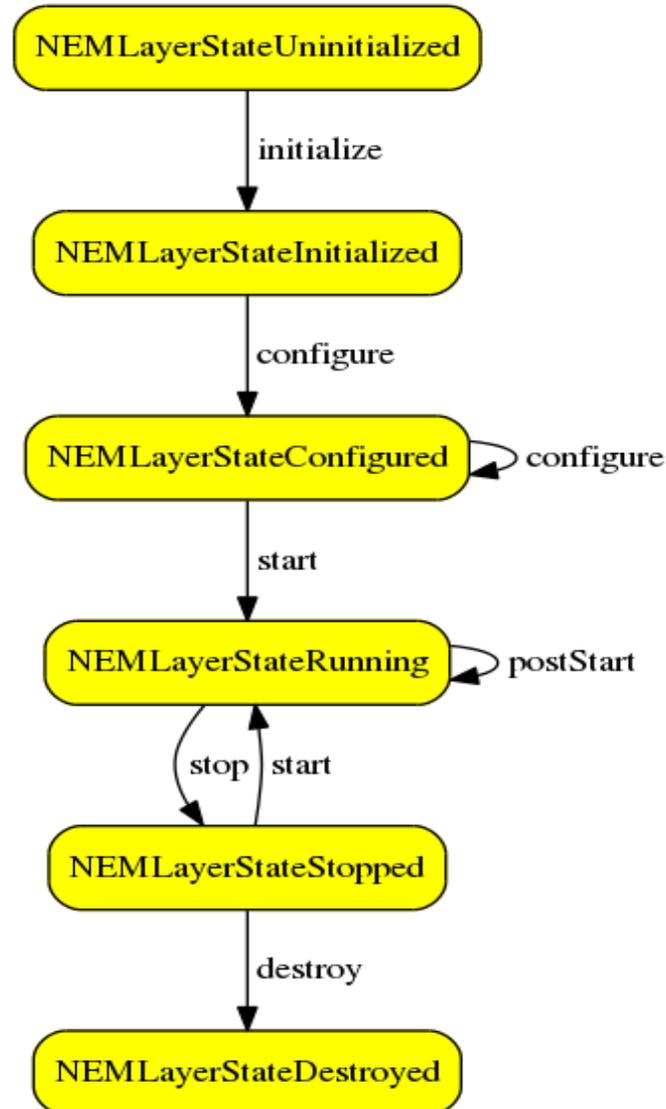
# EMANE::Component

- Most common interface used in the EMANE architecture
- Every component plugin derives from EMANE::Component
- NEM Layer state machine driven by dispatching EMANE::Component methods



[emane/include/emane/emanecomponent.h](#)

# NEM Layer State Diagram



# ConfigurationDefinition Elements

Name	Type	Description
bDefault_	bool	true if the parameter is required
bRequired_	bool	true if value specified should be used as the default
pzValue_	char *	Null terminated string containing the parameter name
uiCount_	unsigned int	Total number of parameter instances allowed or 0 for unlimited
pzValue_	char *	Null terminated string representation of the value
pzType_	char *	Null terminated string containing the parameter type (optional)
pzDescription	char *	Null terminated string containing the parameter description (optional)

# Implementing ConfigurationDefinition

- Declare a `ConfigurationDefinition` array containing the desired configuration parameters, descriptions and default values. Each parameter entry must be marked as either required or optional using the `bRequired_` element.

```
const EMANE::ConfigurationDefinition defs[] =
{
    // req, default, count, name,      value, type, description */
    {false, true,  1,  "traceenable", "off", 0,  "turn on packet trace"},
    {false, true,  1,  "maxstore",    "10", 0,  "max trace id amount"},
    {false, false, 0,  "ignorenode", 0,  0,  "do not trace dest node"},
    {true, false, 1,  "floatvalue", 0,  0,  "unused value"},
    {0,0,0,0,0,0,0,0},
};
```

# Implementing ConfigurationDefinition

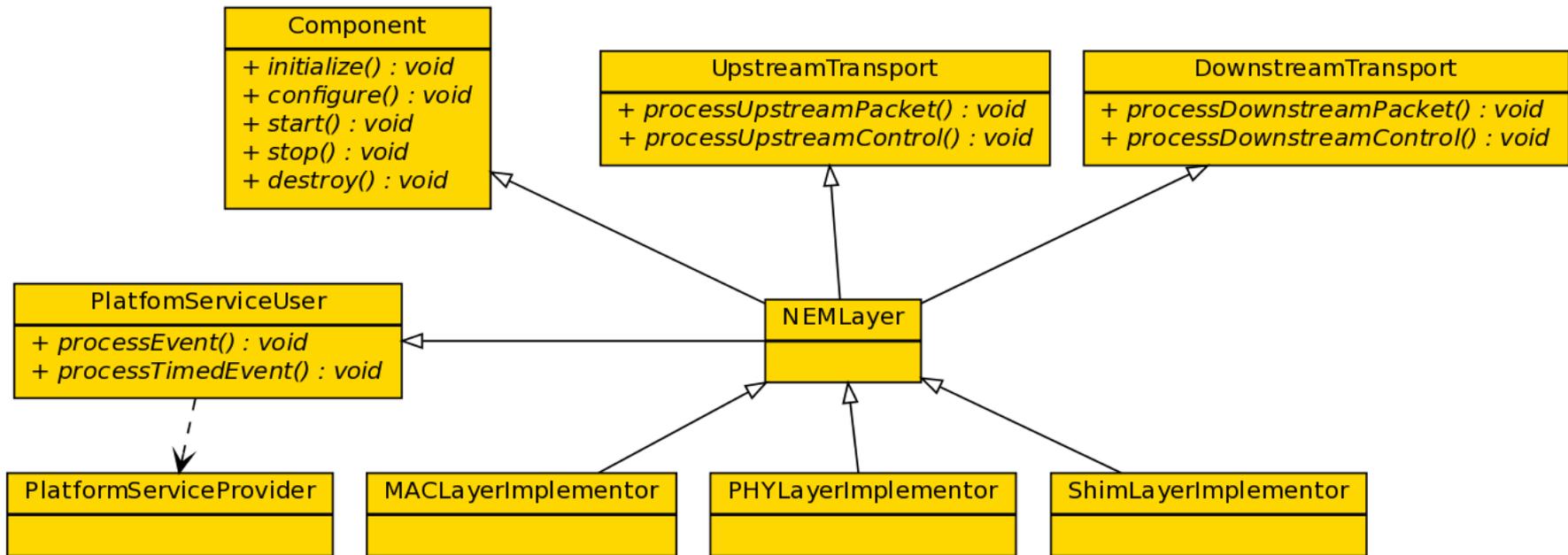
- Load the `ConfigurationDefinition` array into the `configRequirements_` attribute using the `loadConfigurationDefinition` function in the component constructor.
- Use the `Component::configure` implementation to parse the parameters in the component configure method. This step is only necessary if you are specializing the configure method.
- In the component start method iterate through the `configRequirements_` attribute to access the configuration parameters. Remember to throw an `EMANE::StartException` if parameters are missing or found to contain erroneous values.

# Implementing ConfigurationDefinition

- Make sure the component XML contains the appropriate configuration parameters

```
1: <? xml version = " 1.0 " encoding = " UTF -8 " ? >  
2: <! DOCTYPE shim SYSTEM " file: ///usr/share/emane/dtd/shim .dtd " >  
3: < shim name = " Devel Training Shim " library = " devtrainingshim02" >  
4: < param name = " floatvalue " value = " 2.7182818284 " / >  
5: </ shim >
```

# NEM Layer Anatomy



[emane/include/emane/emanenemlayer.h](#)

# NEM Layer Anatomy

- Two Types of NEM Component Stacks: *structured* and *unstructured*
  - Structured – One PHY Layer, one MAC Layer, and zero or more Shim Layers
  - Unstructured – Zero or one PHY Layer, zero or one MAC Layer, zero or more Shim Layers
- Main difference between the two types is a relaxing of internal NEM verification checks

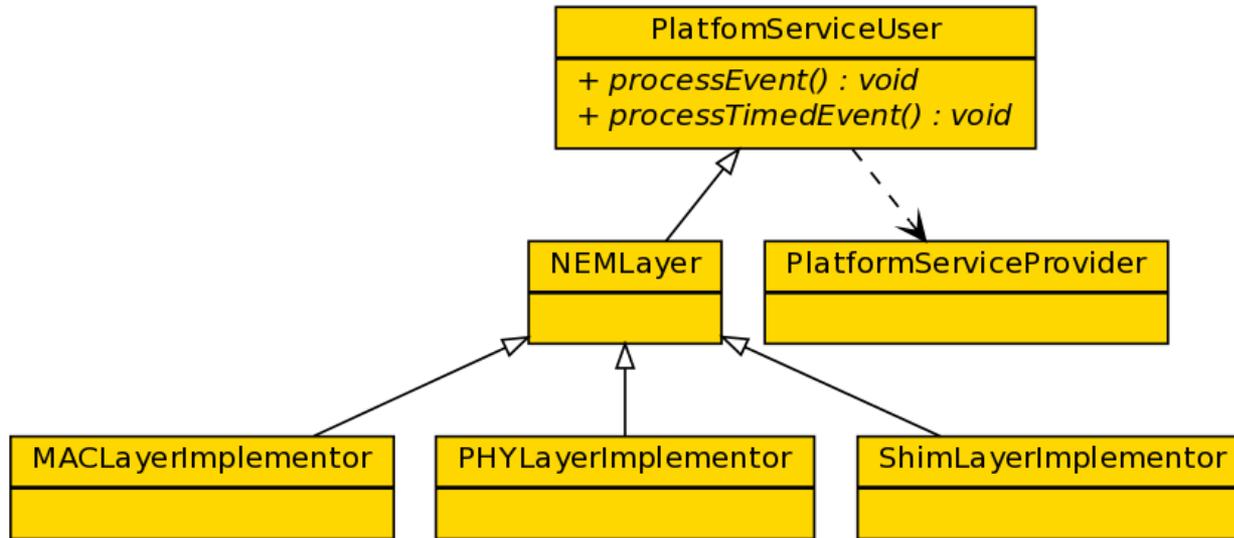
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nem SYSTEM "file:///usr/share/emane/dtd/nem.dtd">
<nem name="Devel Training NEM" type="structured">
  <mac definition="devtrainingmac.xml"/>
  <phy definition="devtrainingphy.xml"/>
  <transport definition="transvirtual.xml"/>
</nem>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nem SYSTEM "file:///usr/share/emane/dtd/nem.dtd">
<nem name="Devel Training NEM" type="unstructured">
  <shim definition="devtrainingshim.xml"/>
  <transport definition="transvirtual.xml"/>
</nem>
```

# NEM Layer Communication

- NEM Layer components communicate generically using the `EMANE::UpstreamTransport` and `EMANE::DownstreamTransport` interfaces
  - `emane/include/emane/emaneupstreamtransport.h`
  - `emane/include/emane/emanedownstreamtransport.h`
- `EMANE::UpstreamTransport` – packets and control entering the NEM from the OTA boundary traveling toward/exiting the NEM at the Transport Boundary
  - Uses `EMANE::UpstreamPacket`
    - `emane/include/emane/emaneupstreampacket.h`
- `EMANE::DownstreamTransport` – packets and control entering the NEM from the Transport boundary traveling toward/exiting the NEM at the OTA boundary
  - Uses `EMANE::DownstreamPacket`
    - `emane/include/emane/emanedownstreampacket.h`

# Event Processing



- All components derive from `EMANE::PlatformServiceUser`

[emane/include/emane/emanepatformserviceuser.h](#)

- `EMANE::EventServiceHandler` – provides `processEvent` which is used to process emulation events

[emane/include/emane/emaneeventservicehandler.h](#)

- `EMANE::TimerServiceHandler` – provides `processTimedEvent` which is used to process timer events

[emane/include/emane/emanetimerservicehandler.h](#)

# Message Processing

- `EMANE::NEMQueuedLayer` provides message queuing mechanisms to decouple inter-layer communication

`emane/src/nemqueuedlayer.h`

- Queued messages:
  - `processUpstreamPacket`
  - `processUpstreamControl`
  - `processEvent`
  - `processDownstreamPacket`
  - `processDownstreamControl`
  - `processTimedEvent`
- Ensures all calls to component message processing and event processing methods happen serially within an individual component's dedicated message processing thread

`EMANE::NEMQueuedLayer::processWorkQueue()`

# Platform Service

- `EMANE::PlatformServiceProvider` provides a single interface for a set of services that are available to every component

[emane/include/emane/emanepatformserviceprovider.h](#)

- Log Service provides the ability to log messages at various levels

[emane/include/emane/emanelogserviceprovider.h](#)

- Event Service provides the ability to send events

[emane/include/emane/emaneeventserviceprovider.h](#)

- Statistic Service provides the ability to create statistics for use in debugging and performance evaluation

[emane/include/emane/emanestatisticserviceprovider.h](#)

- Timer Service provides the ability to schedule one-shot and reoccurring timer events

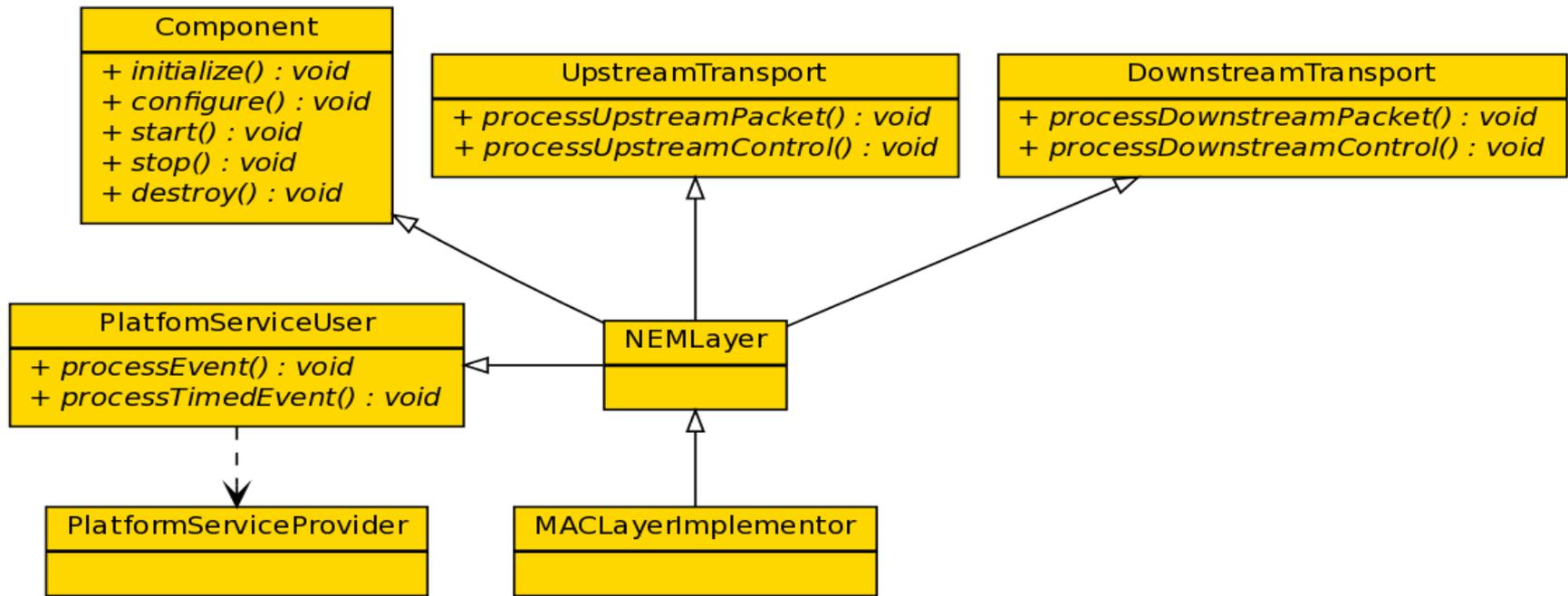
[emane/include/emane/emanetimerserviceprovider.h](#)

- Random Number Service provides the ability for a component to access random number generation in a thread-safe manner

[emane/include/emane/emanerandomnumberserviceprovider.h](#)



# MAC Layer API



- `EMANE::MACLayerImplementor` further specializes `processUpstreamPacket` and `sendDownstreamPacket`

[emane/include/emane/emanemaclayerimpl.h](#)

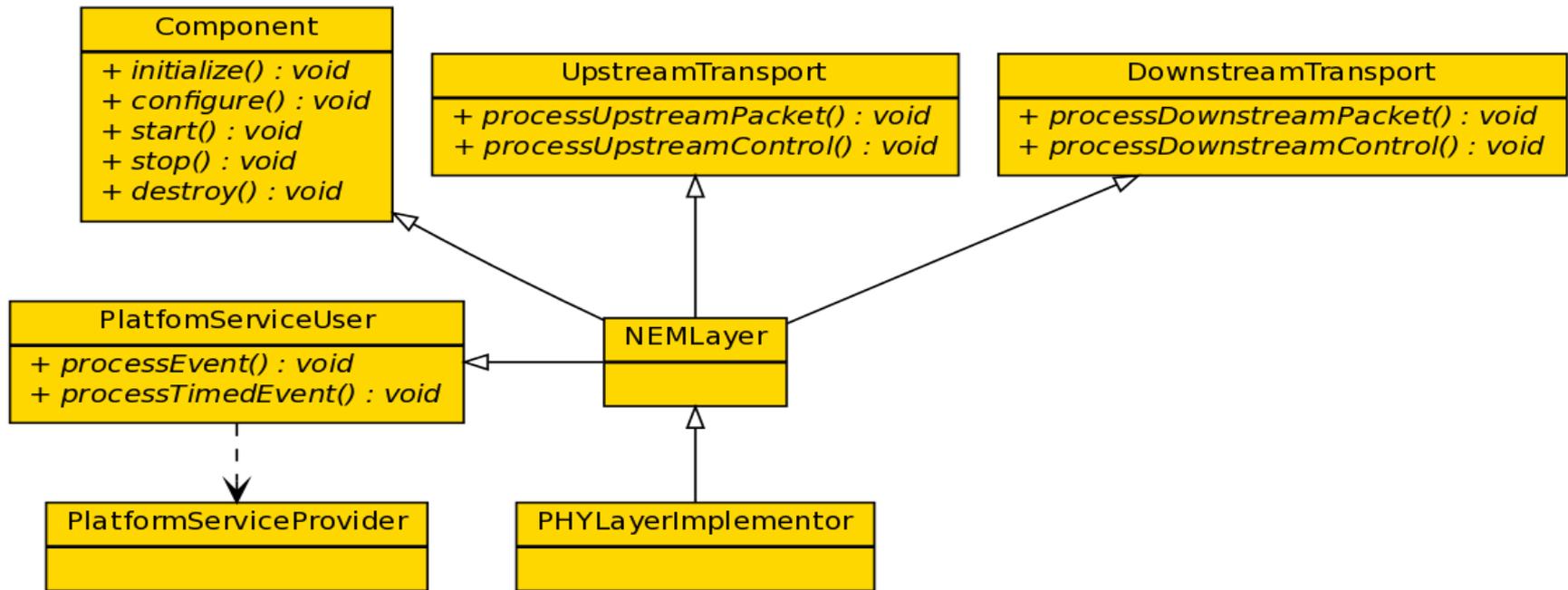
- Adds an `EMANE::CommonMACHeader` parameter

[emane/include/emane/emanecommonmacheader.h](#)

# Implementing a MAC Layer

- Create a class derived from `EMANE::MACLayerImplementor`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the component.
- Expose the new MAC Layer to the EMANE infrastructure using the `DECLARE_MAC_LAYER` macro.
- Create a MAC definition XML file containing the configuration parameters and library name for the new MAC Layer implementation.
- Create an NEM XML definition file using the new MAC definition.

# PHY Layer API



- `EMANE::PHYLayerImplementor` further specializes `processUpstreamPacket` and `sendDownstreamPacket`

[emane/include/emane/emanephylayerimpl.h](#)

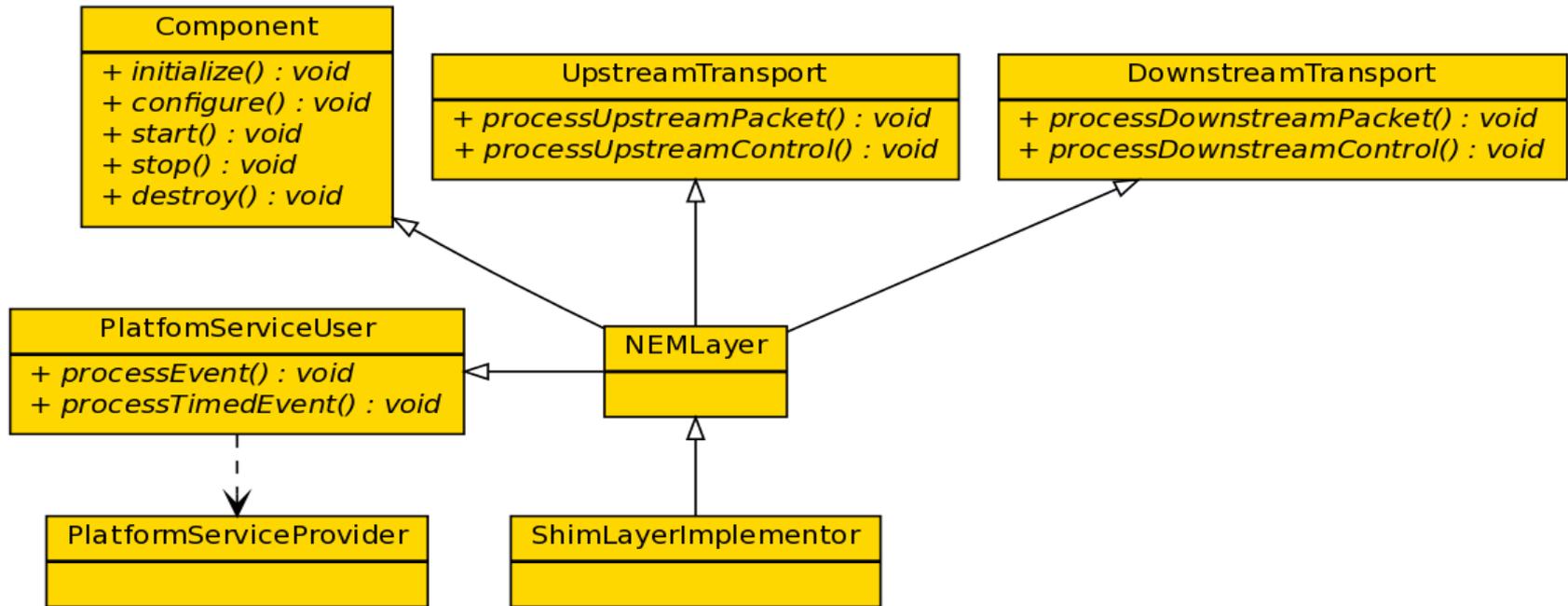
- Adds an `EMANE::CommonPHYHeader` parameter

[emane/include/emane/emanecommonphyheader.h](#)

# Implementing a PHY Layer

- Create a class derived from `EMANE::PHYLayerImplementor`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the component.
- Expose the new PHY Layer to the EMANE infrastructure using the `DECLARE_PHY_LAYER` macro.
- Create a PHY definition XML file containing the configuration parameters and library name for the new PHY Layer implementation.
- Create an NEM XML definition file using the new PHY definition.

# Shim Layer API

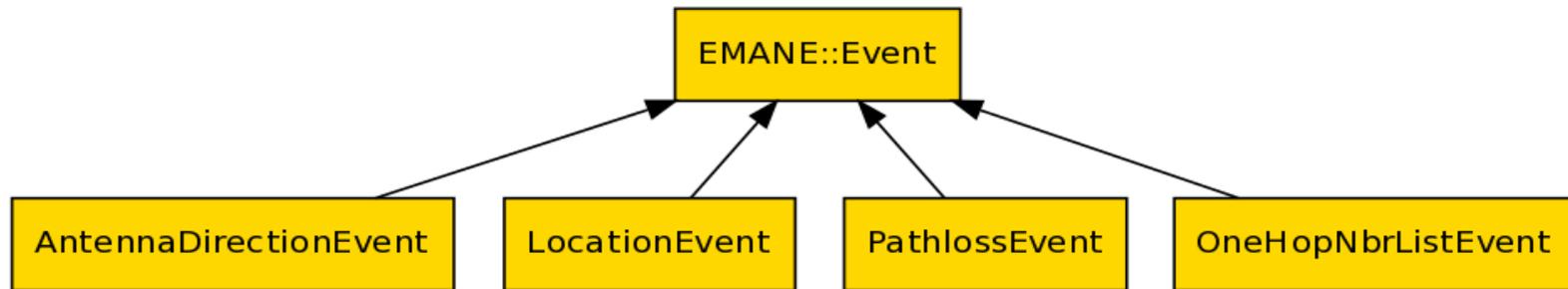


`emane/include/emane/emaneshimlayerimpl.h`

# Implementing a Shim Layer

- Create a class derived from `EMANE::ShimLayerImplementor`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the component.
- Expose the new Shim Layer to the EMANE infrastructure using the `DECLARE_SHIM_LAYER` macro.
- Create a Shim definition XML file containing the configuration parameters and library name for the new Shim Layer implementation.
- Create an NEM XML definition file using the new Shim definition.

# Events



- Events are messages sent to components containing control information

[emane/include/emane/emaneevent.h](#)

- Transmitted generically throughout the emulation deployment
- Event subsystem components:
  - Event Service, Event Generators, and Event Agents.

# Implementing an Event

- Create a class derived from `EMANE::Event`.
- Provide a constructor that takes an `EMANE::EventObjectState` constant reference argument and use the object state to reconstruct the transmitted event object.
  - The `EMANE::Event` base class takes an event id and an event name string as constructor arguments.
  - Throw an `EMANE::EventObjectStateException` if an error is detected with the object state data.
  - Event data contained in `EMANE::EventObjectState` objects are in Network Byte Order.
  - Event ids are 16 bit values. Ids with the most significant bit set are local ids.
- Provide an implementation for the `getObjectState` method. This method returns an `EMANE::EventObjectState` object containing all the data necessary to reconstruct this object when received by the targeted EMANE components.

# Event Service

- Allows the decoupling of the creation of events from their distribution
- `EMANE::EventGenerator` provides mechanism for transmitting events without imposing limitation on how the events are created

[emane/include/emane/emaneeventgenerator.h](#)

# Implementing an Event Generator

- Create a class derived from `EMANE::EventGenerator`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the component.
- Register any events that the Event Generator will produce using the `EMANE::EventGenerator::addEventId` method.
- Create a new thread to execute the backend code necessary to generate events. This might be as simple as parsing input files, building events and then sending them using the Platform Service.
- Expose the new Event Generator to the EMANE Event Service using the `DECLARE_EVENT_GENERATOR` macro.

# Implementing an Event Generator

- Create an Event Generator definition XML file containing the configuration parameters and library name for the new Event Generator implementation.
- Add the Event Generator to the EMANE Event Service XML.

# Event Daemon

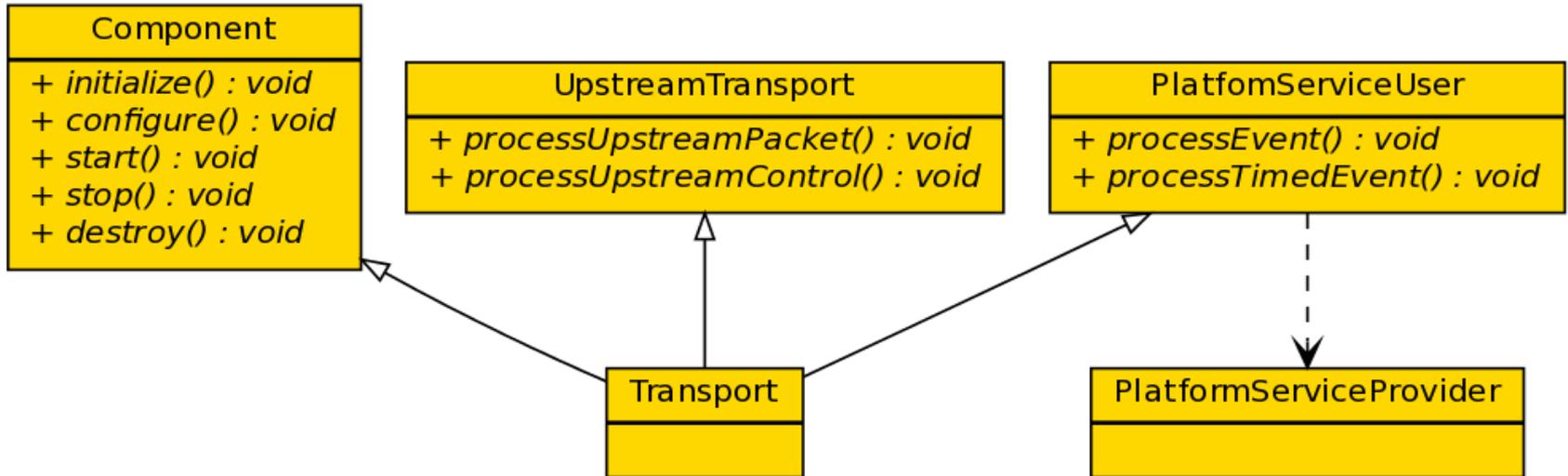
- Event Daemon provides a mechanism to transport event data from the emulation domain to other application domains
  - `EMANE::EventAgent` provides the ability to register to received events in order to communicate the data to other applications

[emane/include/emane/emaneagent.h](#)

# Implementing an Event Agent

- Create a class derived from `EMANE::EventAgent`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the agent.
- Expose the new Event Agent to the Event Daemon using the `DECLARE_EVENT_AGENT` macro.
- Create an Event Agent definition XML file containing the configuration parameters and library name for the new Event Agent implementation.
- Add the Event Agent to the EMANE Event Daemon XML.

# Transport



- Transports are the emulation boundary interfaces that provide the entry and exit points for all data routed through the emulation.
- A Transport component implementation is a realization of the `EMANE::Transport` interface.

[emane/include/emane/emanetransport.h](http://emane/include/emane/emanetransport.h)

# Implementing a Transport

- Create a class derived from `EMANE::Transport`.
- Fill in the implementation for all virtual methods.
- Define and load the `EMANE::Component` configuration requirements for the component.
- Create a new thread to execute the backend code necessary to receive and process the input appropriate for the transport implementation. Use the `sendDownstreamPacket` and `sendDownstreamControl` methods to send packets and control messages to the transport's respective NEM.
- Expose the new transport to the EMANE infrastructure using the `DECLARE_TRANSPORT` macro.
- Create an NEM XML definition file using the new transport definition.



# libmaneeventservice

- libmaneeventservice – C Language API for interfacing with the Event Service
- libmaneeventpathloss– C Language API for building and parsing Pathloss Events
- libmaneeventlocation – C Language API for building and parsing Location Events
- libmaneeventantennadirection – C Language API for building and parsing Antenna Direction Events