

Atlas Level-2 Trigger Ptolemy model

Discussion and Specification of the “Network” Components: Network Interface, Link Interconnect and Switch

Contributors : The Ptolemy interest group
Editor : R. Hughes-Jones
Keywords : Network, Link, Switch, Gigabit Ethernet

Abstract

This document specifies the Network Components and Object Classes for the Ptolemy simulations of the network hardware and the High Level Model framework.

The document gives a brief discussion of the options that could be used in simulating the ATLAS Network, following various email and phone discussions. The distinction is made between Application Messages and the messages required to simulate the network. A clear aim is that the Hardware, Transport and Network Simulation components should NOT have to know any details of the Application Messages they carry. The message context, or message summary diagram, and the message descriptions are given for the Network Interface, the Network Switch, and the Network Probe. Various Object Classes required for the Hardware Transport Simulation are presented and discussed.

Please note: the present document should be taken as a discussion document.

Note Number :
Version : 0.6
Date : 20 Oct 99
Reference :

1 Introduction

1.1 Background

Within Atlas Level-2 there is interest in simulating both the low level hardware components and the overall High Level Trigger. Starting with the need to understand and simulate the 100 Mbit and Gigabit Ethernet hardware and simple test setups, initial discussions (CERN, UCL & Manchester) resulted in a simple design and some prototype code. It would clearly be advantageous to be able to use the same Network level simulation code for both hardware and High Level Trigger work. The following points emerge:

1. The Hardware, Transport and Network Simulation components should NOT have to know any details of the Application Messages they carry – this provides independent development and testing.
2. The Hardware, Transport and Network Simulation components should take care of the details of how the hardware, firmware and protocols operate. The Application e.g. "ROB" or "Processor" should not have to worry about the network timing or packet re-tries for example.
3. The boundaries or interfaces between the Network and the Application should be independent of the hardware.
4. It is advantageous to be able to use the same Network level simulation code for both hardware and High Level Trigger simulation work.

1.2 The Application and Network Context

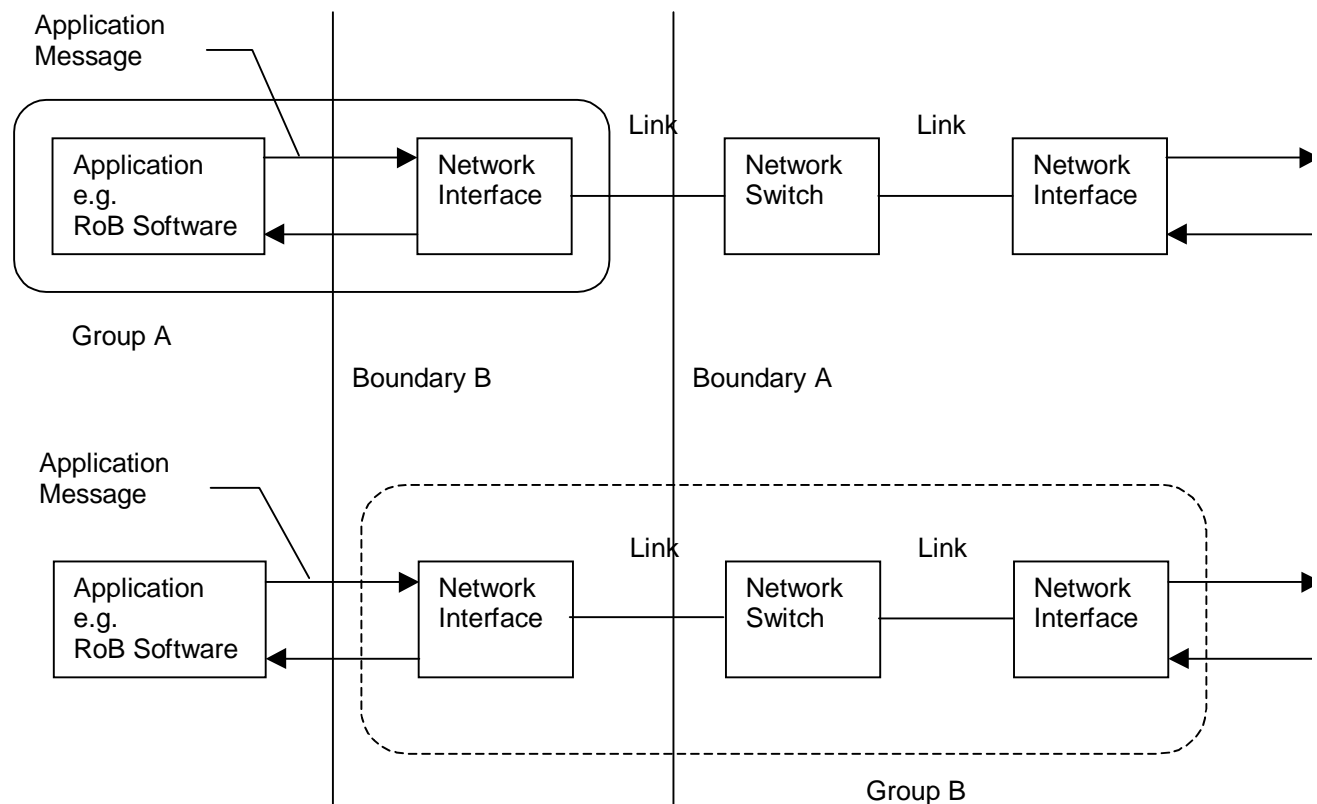


Figure 1.1 Ways of Grouping of the Application and Network Software Modules

There are several ways to map the software items shown in Figure 1.1 onto simulation nodes or in this case Ptolemy Stars.

- Group A : This integrates the Application Objects with the Network Interface Objects. It closely matches the physical world and will allow the simulation of hardware objects e.g. a PCI bus, or CPU availability, between the application and the network. However it does require close integration of Object Classes which could cause difficulties. Application Messages (i.e. derivatives of the `Amessage` class) would be carried across Boundary B by the link code.
- Group B : Implement the Network Interface and Network Switch as one big Network Object. This simplifies the Application as only derivatives of the `Amessage` class are passed over Boundary B. However, this monolithic approach is not very flexible for implementing and studying the Network.
- Implement independent Items : This seems the easiest to implement and test. The various Network Objects would be coupled together as indicated in Group B in Figure 1.1. It allows point to point network links, as required for simulation of a test bench, as well as the simulation of other technologies such as slink. It provides a modular approach to writing the Network simulation code. Various implementers can provide different items e.g. Network Switches with different architectures.

1.3 Implementation of the Network

It is proposed to write the items as independent stars thus allowing the building blocks to be grouped into galaxies to provide the overall Network for the Applications. The "Network" will be constructed from Network Interfaces, Network Switches, and other Network Objects as required, to provide the desired paths or logical links between the Application Objects.

1.3.1 Message Class Hierarchy

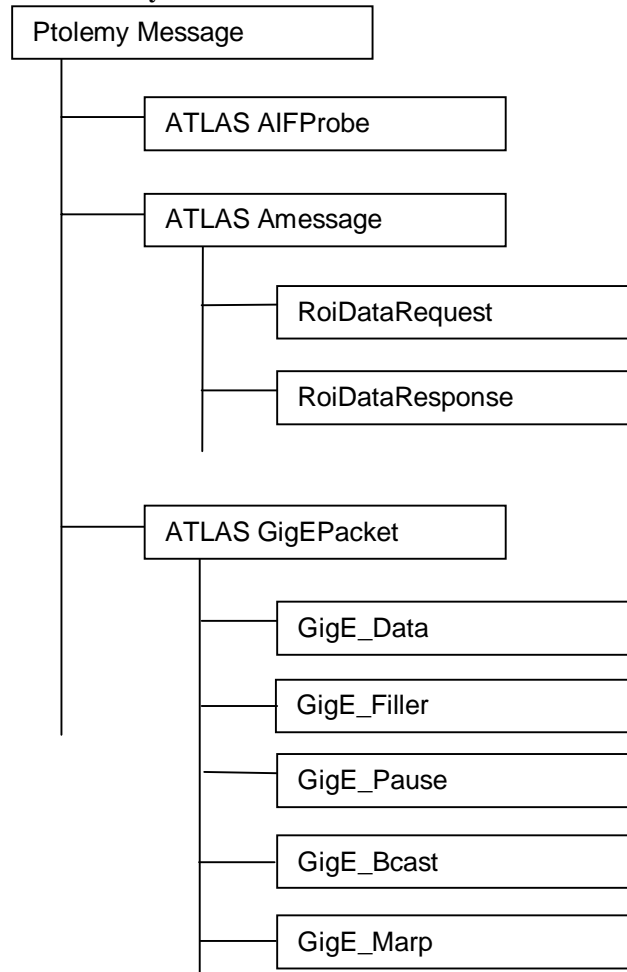


Figure 1.2 Relations of the Message Classes Passed between Applications and the Network

All messages crossing between an Application and the Network, Boundary B in Figure 1.1, using derivatives of the ATLAS message class `Amessage` (`Amessage` is never sent) will be carried by the Network. Messages of the `AIFProbe` class will be used to determine the status of the local interface. The relation between these classes is shown in Figure 1.2.

1.3.2 Adding Extra Components to the Network

Using the "Amessage" message interface between the Application and the Network provides an extensible plug and play arrangement between the simulation modules. Two Applications may be directly connected together for testing, or a specific simulation of hardware or of a network protocol may be added between the Network interface and the Application. These concepts are shown in Figure 1.3. Clearly, when two Applications are directly connected, such as in Figure 1.3 (a), the Applications will have to respond to the `AIFProbe` messages.

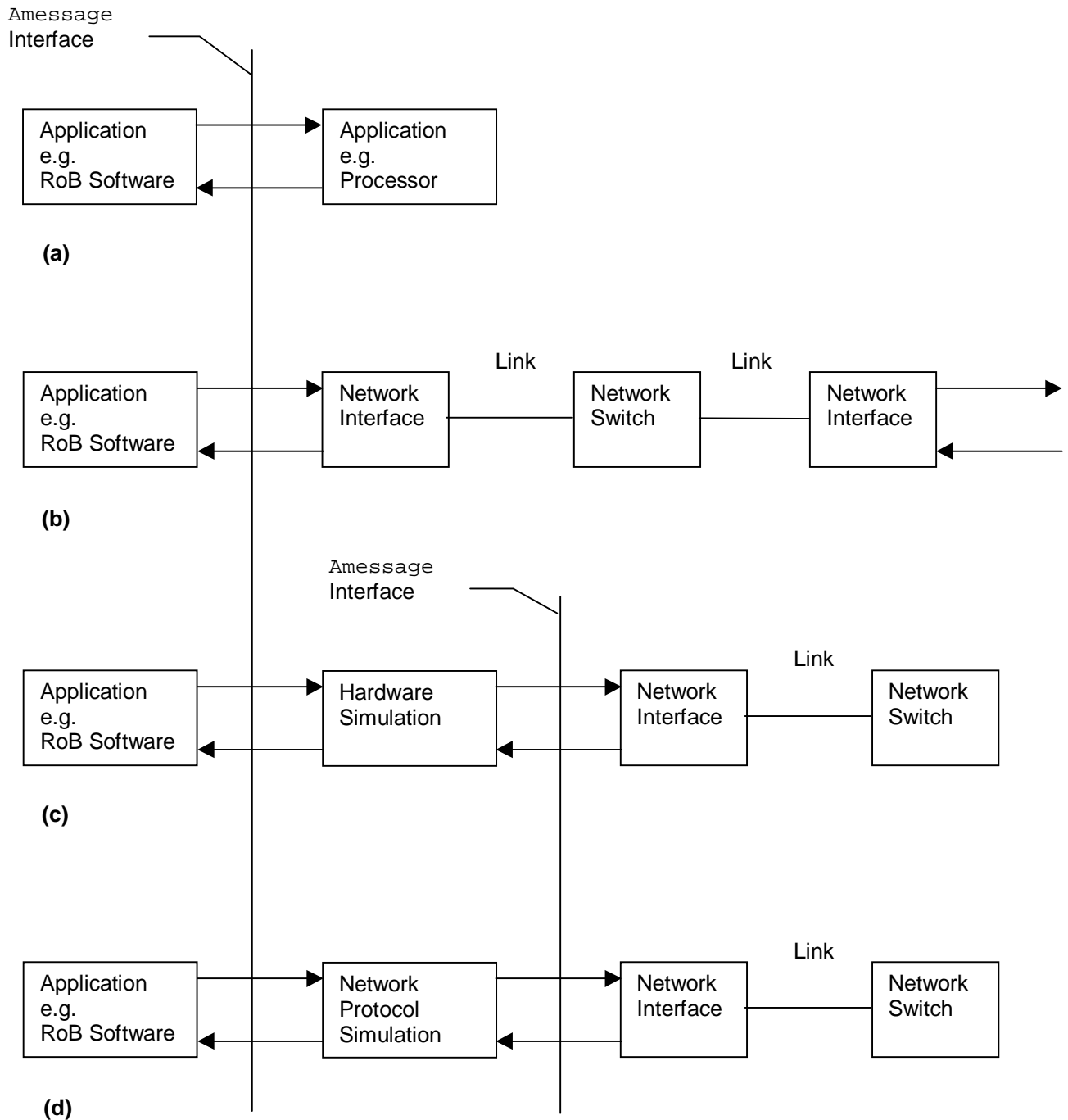


Figure 1.3 Using the Amessage Interface to allow extensions to the Application-Network Interface.

Figure 1.3 (b) shows the “conventional” network connection between Applications. Detailed hardware and or network protocol simulations may be added to the model as shown in Figures 1.3 (c) and (d). Here the specified components use the "Amessage" interface between both the Application and the Network interface objects.

1.3.3 Messages on the Network Links

All data passing over the network links will use the Ptolemy Message and Envelope classes. For Ethernet this will be based on the `GigEPacket` class derived from the Ptolemy Message class. The “Network” needs to know:

The sourceAddress The originator of the application message
The destinationAddress The destination for the application message
The length The length of the application message i.e. everything in Amessage
A pointer to the object derived from Amessage . This object **is** the application message data (by definition!).

1.4 Network Addressing

Each of the interfaces will be assigned a network address. Although in reality ethernet frames use a 12 digit MAC address and UDP/IP or TCP/IP packets use an IP address as well, a simple integer will suffice for the simulation in most cases. The role of the address is to enable the Network switches to place a received frame on the correct output port. Use of the Address class methods will help to isolate the simulation code from the details of the real address being used.

Application nodes may have more than one network interface. For simplicity we may ignore dynamic load balancing by the network components at present, and assume that a message from a Source node will use just one interface on the destination node.

All Applications will need to know the network address(es) of the remote nodes that they will access. Nodes that respond with the requested data e.g. ROBs, may use the network address of the requestor node as given in the message requesting the data.

2 Network Interface

2.1 Message Summary Diagram

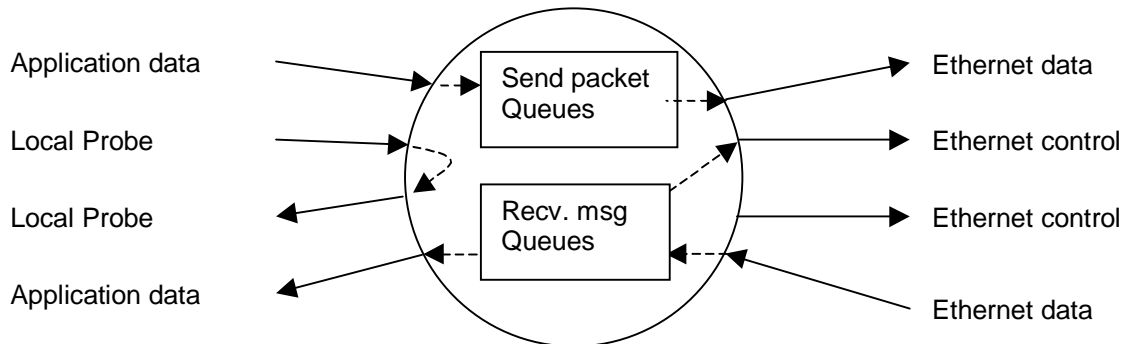


Figure 2.1 Context Diagram of the Network Interface

2.2 Network Interface Requirements

The requirements factor into two areas:

- Link level. This is concerned with :
 - sending and receiving network frames on the wire
 - flow control on Ethernet between network interfaces
 - buffering within the interface or node
- Packetisation. This allows the translation between Application messages and Network frames.
 - When sending a Message to the Network the interface will send a frame with the GigEpacket details and a number of "filler" frames so that the network is loaded with the true length of the message. Note: the object derived from the `Message` class is not split over several ethernet frames.
 - An incoming Message from the Network will be passed to the Application when all frames related to that message have been received. **[What happens if part of a message (e.g. 1 or more frames) is lost? -- needs thought -- but guess we need a timeout and then delete the object derived from `Message` !]**
 - The design should be extensible to allow the possibility of flow control between the application and the Network Interface.
 - Addressing
 - The Application "looks up" its address and informs the Network Interface at the start of simulation.
 - The Network Interface then sends a `GigE_Marp` packet on the Ethernet link to inform the Network of its address.

The Network Interface requirements are :

1. On receipt of an application message, send a set of simulated network packets to the specified destination. Real data need not be sent but the Network must be "busy" for the correct period of time.
2. Provide the message to the application on receiving the set of packets forming the message from the network.
3. Return status information about the local interface to the application in response to a probe message.
4. Provide a packet queue that can act as a FIFO, for packets being placed on the network with runtime options of: no queue, a queue of depth n, a queue of infinite length.
5. Provide a message queue that can act as a FIFO, for packets/messages arriving from the network with runtime options of: no queue, a queue of depth n, a queue of infinite length.

6. Allow specification of the bandwidth or data transfer rate of the link connecting the interface to the Network.
7. Allow specification of the bandwidth or data transfer rate of the interface between the Network interface and the Application.
8. Provide statistics / histograms / graphs for the network link and the interface to the application. These should include:
 - Bandwidth utilisation,
 - No. of data and control packets
 - No. of data and control bytes
 - Queue lengths
 - Packet transfer times over the link or network. (Note this is NOT the message transfer time)
9. Respect / implement the network standards
 - IEEE 802.3z For 100 Mbit and Gigabit Ethernet.
 - IEEE 802.3x for ethernet flow control.
10. Provide message segmentation into packets on transmission and re-assembly of packets on reception. (The assumption is that the application message object will be sent in the first packet, the others being network fillers. **[What happens if a filler packet arrives first? Do we wish/need to simulate packet ordering (may be if the switch or links are critical) ? – perhaps not at first needs discussion.]**)
11. Provide “hooks” for possible extensions to allow flow control between the application and the Network Interface using AIFProbe or AIFControl messages.

2.3 Messages In

2.3.1.1 Message name : Eth_in

Ethernet packet arriving from the network.

```
Contents {  
    class GigE_Data :  
        const int GigE_Data::msgType();  
}
```

```
Contents {  
    class GigE_Bcast :  
        const int GigE_Bcast::msgType();  
}
```

```
Contents {  
    class GigE_Filler :  
        const int GigE_Filler::msgType();  
}
```

Responsibilities :

Assemble the data packets into a message and give a AppMsg_out message to the application when it is complete.

```
Contents {  
    class GigE_Pause :  
        const int GigE_Pause::msgType();  
}
```

Responsibilities :

If the ethernet control Pause packet has a time > 0 stop sending packets for that time.
If the ethernet control Pause packet has a time = 0 start sending packets now.

```
Contents {
    class GigE_Marp :
        const int GigE_Marp::msgType();
}
```

Responsibilities :

Do nothing with the information – the Application has no need of this data.

2.3.1.2 Message name : *AppMsg_in*

Message from the User Application.

```
Contents {
    class Amessage:

        const int Amessage::msgType();
        const Length& Amessage::length();
        const NodeAddress& Amessage::sourceAddress();
        const NodeAddress& Amessage::destinationAddress();
}
```

Responsibilities :

Split the data message into packets, put them on the output queue and send them over the network as soon as possible as *Eth_out* message(s).

```
Contents {
    class AIFProbe:

        const int AIFProbe::msgType();
}
```

Responsibilities :

Return the status of the interface in a suitable *AppMsg_out*. The status information includes the input and output Q lengths, number of packets dropped.

```
Contents {
    class AIFControl:

        const int AIFControl::msgType();
}
```

Responsibilities :

Reserved for future extensions.

2.4 Messages Out

2.4.1.1 Message name : *Eth_out*

Data Ethernet packet placed on the network:

```
Contents {
    class GigE_Data :
        const int GigE_Data::msgType();
}
```

```
Contents {  
    class GigE_Bcast :  
        const int GigE_Bcast::msgType();  
}
```

```
Contents {  
    class GigE_Filler :  
        const int GigE_Filler::msgType();  
}
```

Control Ethernet packet placed on the network:

```
Contents {  
    class GigE_Pause :  
        const int GigE_Pause::msgType();  
}
```

```
Contents {  
    class GigE_Marp :  
        const int GigE_Marp::msgType();  
}
```

MAC address resolution packet, generated by the Network Interface immediately after initialisation.

2.4.1.2 Message name : *AppMsg_out*

Message to the User Application.

```
Contents {  
    class Amessage:  
  
        const int Amessage::msgType();  
        application information;  
}
```

```
Contents {  
    class AIFProbe:  
  
        const int AIFProbe::msgType();  
}
```

```
Contents {  
    class AIFControl:  
  
        const int AIFControl::msgType();  
}
```

Reserved for future extensions.

2.5 Proposed Implementation

The Ptolemy star that simulates the network interface can be built using the following object classes:

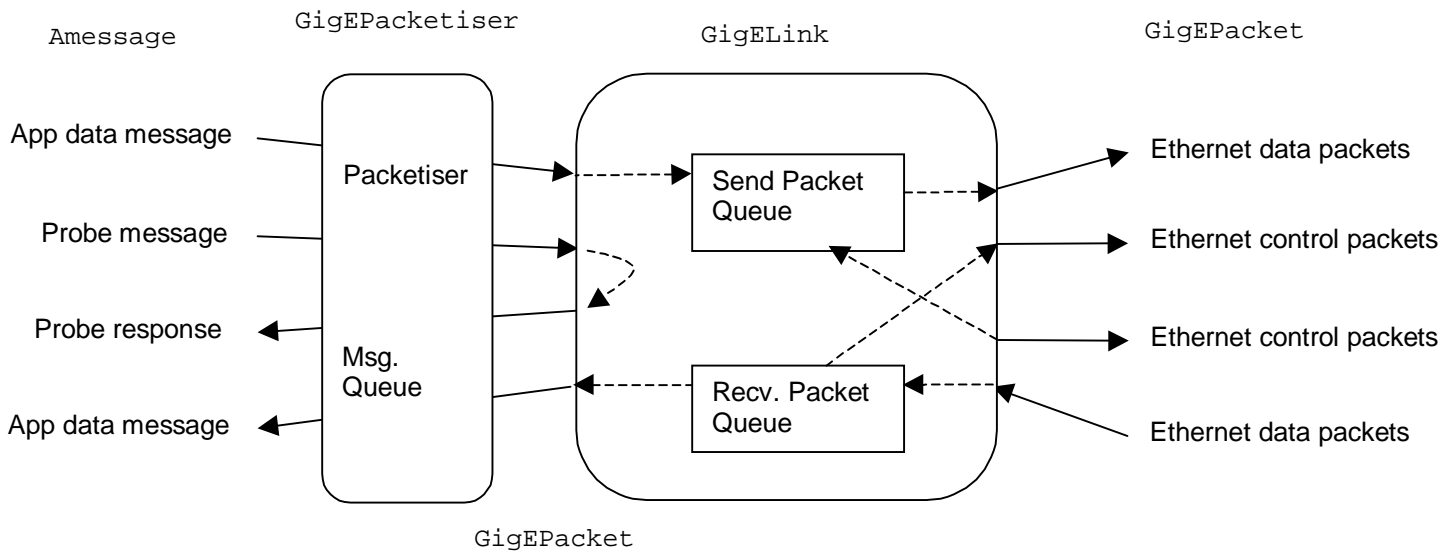


Figure 2.2 Classes used in the Network Interface

The `GigEPacket` class contains the information required to move the Ethernet frame through the system and the methods to manipulate this data.

The `GigELink` class implements the Ethernet protocol over the wire, and the packet queues for both transmission of packets to the network and storage of those that have been received from the network. When the incoming packet queue reaches the “high water mark”, a pause packet will be sent to the transmitting `GigELink` object at the other end of the Ethernet link. A pause packet with zero delay may be sent when the incoming packet queue reaches the “low water mark”.

The `GigELink` class will also report the status of its queues when it receives a Probe packet from the Application.

When a `GigELink` object receives its MAC address, it will send out a `GigE_Marp` packet to inform the network. (This usually happens only at simulation startup time.) If a `GigELink` object receives a `GigE_Marp` packet from the network it will pass it to the Network Interface. Note that the `GigELink` class works with `GigEPacket` objects.

The `GigEPacketiser` class will implement the building of packets into messages and the message queue to the Application. Messages from the Application will be split into packets and sent directly to the `GigELink` object for queuing for network access. This unit should facilitate the queuing of `Amessage` objects. The times involved in packetising (hopefully small) will be added to the transmission times of the Ptolemy messages.

The `GigELink` and `GigEPacket` classes will also be used for the ports on the switch.

3 Network Switch

3.1 Message Summary Diagram

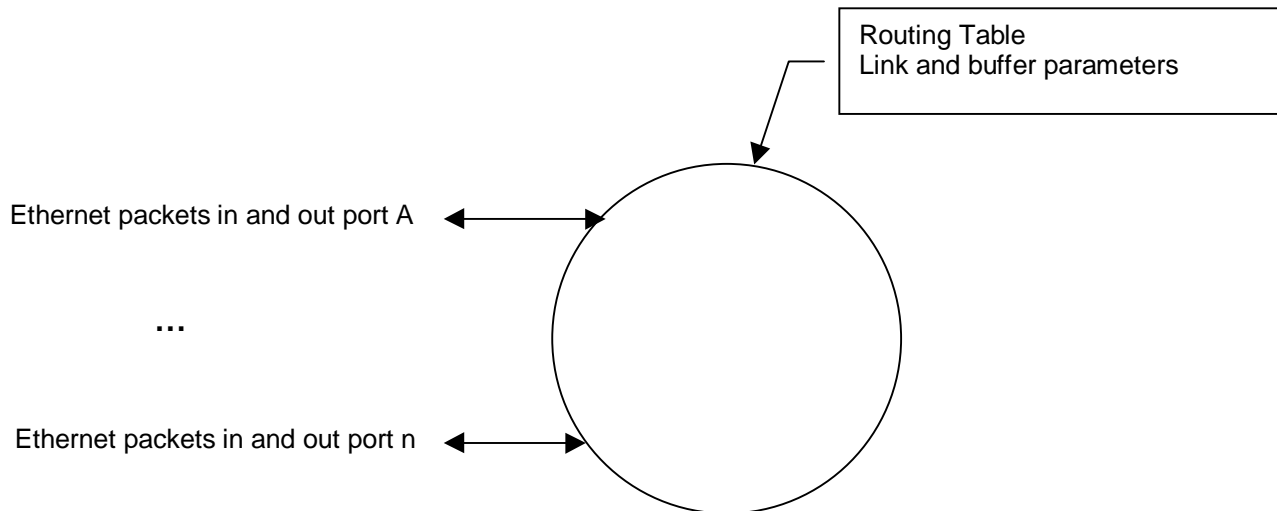


Figure 3.1 The Context Diagram of the Network Switch.

3.2 Network Switch Requirements

The Network Switch requirements include :

1. On receipt of packet, determine the output port of the switch and send the packet to the specified destination.
2. Provide input and output packet queues as defined for the Network Interface (Section 2.2). These queues may be implemented by the `GigELink` class.
3. The time of arrival of packets from the network must be given to the switch as that of the start of the packet. This allows “worm hole” routing.
4. Allow ranges of addresses on the output ports so that switches may be cascaded.
5. The switch should be able to use a static routing table.
6. The switch should be able to “learn” which network interface addresses are accessible through each switch port. The switch must inform all output ports, except the port that provided the `GigE_Marp` packet, when a new MAC address is learnt. `GigE_Marp` packets must be sent.
7. Allow for different switch Architectures, e.g. extra queues or backplanes inside the switch.
8. Allow specification of the Network connection bandwidths or data transfer rates.
9. Provide link level and switch fabric statistics / histograms / graphs including:
 - Bandwidth utilisation,
 - No. of data and control packets
 - No. of data and control bytes
 - Queue lengths
 - Packet transfer times over the switch fabric (Note this is NOT the message transfer time)
10. Respect / implement the network standards
 - IEEE 802. For 100 Mbit and Gigabit Ethernet and IEEE 802.2p for flow control.
 - Provide capability for “trunking” .e.g. using several 100Mbit links between 2 switches.

3.3 Messages In

All ports have the same messages.

3.3.1.1 Message name : *Eth_inN*

Ethernet packet arriving on channel N.

```
Contents {  
    class GigE_Data :  
        const int GigE_Data::msgType();  
}
```

```
Contents {  
    class GigE_Filler :  
        const int GigE_Filler::msgType();  
}
```

Responsibilities :

Determine the output port of the switch and route the packet to that *Eth_outM* port.

If switch queues have reached the “high water mark”, send a *GigE_Pause* packet on port *Eth_outN*.

```
Contents {  
    class GigE_Bcast :  
        const int GigE_Bcast::msgType();  
}
```

Responsibilities :

Route the packet to **all** *Eth_outM* ports **but not** port *Eth_outN*.

If the switch queues have reached the “high water mark”, send a *GigE_Pause* packet on port *Eth_outN*.

```
Contents {  
    class GigE_Pause :  
        const int GigE_Pause::msgType();  
}
```

Responsibilities :

If the ethernet control Pause packet has a time > 0 stop sending packets on port N for that time.

If the ethernet control Pause packet has a time = 0 start sending packets now on port N.

```
Contents {  
    class GigE_Marp :  
        const int GigE_Marp::msgType();  
}
```

Responsibilities :

Check that the source address is linked with the switch port number N in the switch routing table.

Route a *GigE_Marp* packet to **all** *Eth_outM* ports **but not** port *Eth_outN*.

3.4 Messages Out

All ports have the same messages.

3.4.1.1 Message name : *Eth_outN*

Control or data Ethernet packet placed on the network port N.

```
Contents {  
    class GigE_Data :  
        const int GigE_Data::msgType();  
}
```

```
Contents {  
    class GigE_Filler :  
        const int GigE_Filler::msgType();  
}
```

```
Contents {  
    class GigE_Bcast :  
        const int GigE_Bcast::msgType();  
}
```

```
Contents {  
    class GigE_Pause :  
        const int GigE_Pause::msgType();  
}
```

```
Contents {  
    class GigE_Marp :  
        const int GigE_Marp::msgType();  
}
```

3.5 Implementation

The idea is to build the Ptolemy star that simulates the Network Switch using the object classes already used for the Network Interface. The packet queues in the GigELink objects may be used or replaced by special devices when modelling a specific switch architecture.

The switch is not constructed from Network Interface components as these deal with Application messages on local ports. Switches operate at the Ethernet packet level.

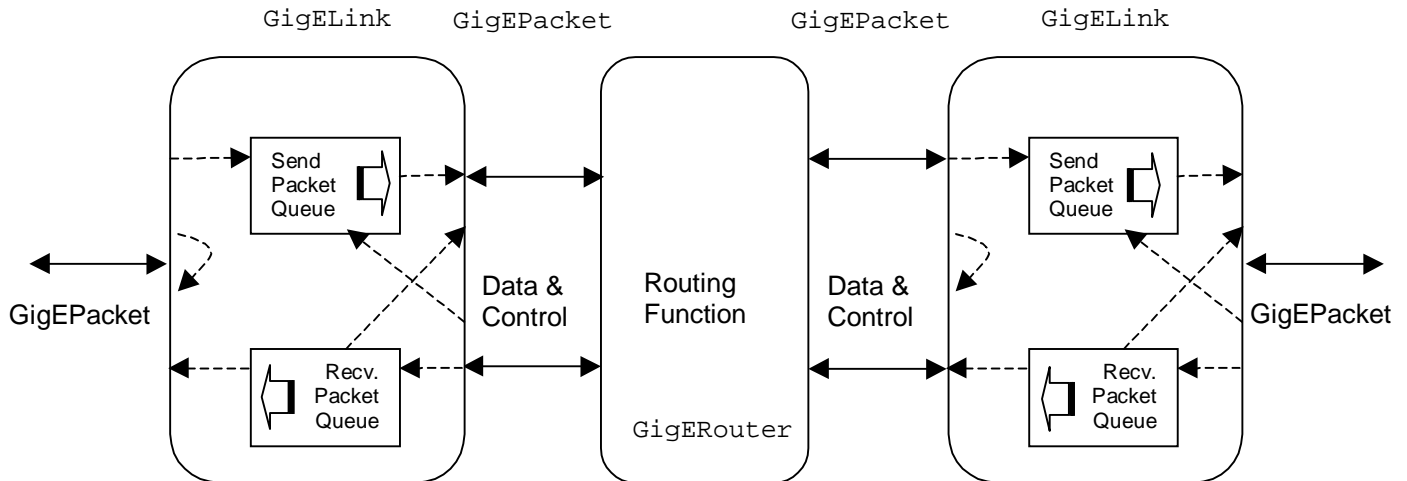


Figure 3.2 The classes that may be used in constructing a Network Switch

4 Network Probe

4.1 Message Summary Diagram

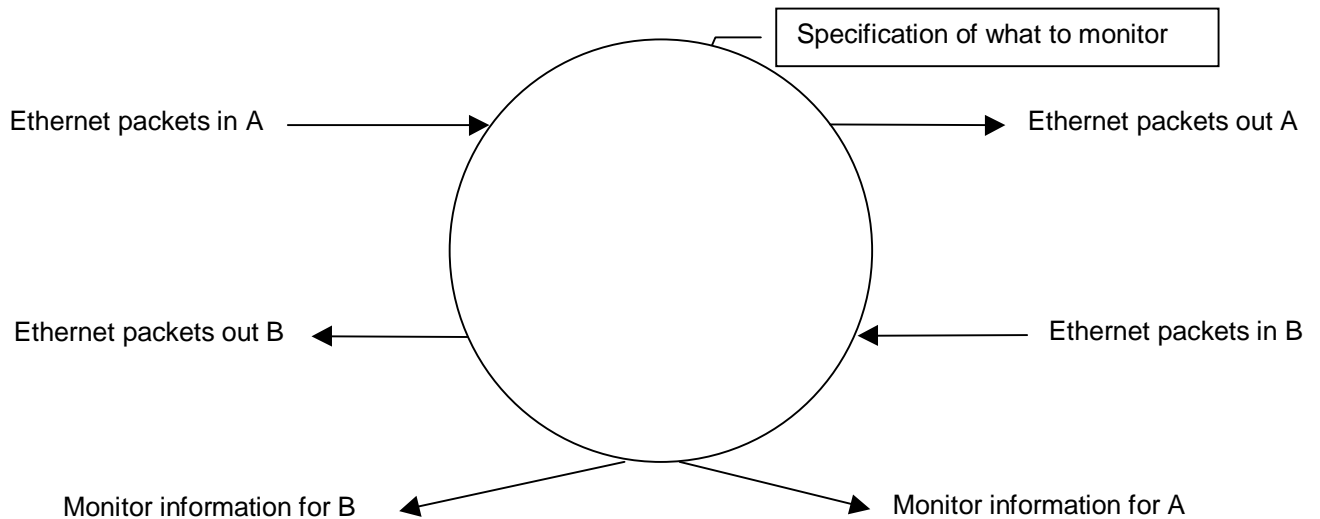


Figure 4.1 The Context Diagram of the Network Probe.

4.2 Network Probe Requirements

This is rather like a scope or logic analyser, it does NOT act as a network node, but simply looks at the link level information and sends this to a monitor output. Like 100 Mbit and Gigabit Ethernet the Probe is full duplex with 2 independent channels A and B.

4.3 Messages In

4.3.1.1 Message name : *Pkt_inA*

Ethernet packet arriving on channel A.

```
Contents {
    class GigEPacket :
}
```

Responsibilities :

Extract the required information and place on the Info_outA output. Place the ethernet packet on the PKt_outA output.

4.3.1.2 Message name : *Pkt_inB*

Ethernet packet arriving on channel B.

```
Contents {
    class GigEPacket :
}
```

Responsibilities :

Extract the required information and place on the Info_outB output. Place the ethernet packet on the PKt_outB output.

4.4 Messages Out

4.4.1.1 Message name : *Pkt_outA*

Ethernet packet that arrived on channel A.

```
Contents {  
    class GigEPacket :  
}
```

4.4.1.2 Message name : *Pkt_outB*

Ethernet packet that arrived on channel A.

```
Contents {  
    class GigEPacket :  
}
```

4.4.1.3 Message name : *Info_outA*

Data from channel A.

```
Contents {  
    Int param;  
}
```

4.4.1.4 Message name : *Info_outB*

Data from channel B.

```
Contents {  
    Int param;  
}
```

5 Implementaion Details of the Classes

5.1 Address::

```
class NodeAddress
{
    NodeType node_type ;
    int node_id ;
...
}
```

5.2 GigEPacket::

The packet sent on the 100 Mbit or Gigabit Ethernet network will be based on the class :

```
class GigEPacket : public Message {

private:
//..... class data .....

    NodeAddress dest_address; // destination node ID
    NodeAddress source_address; // source node ID
    int broadcast_flag; // =1 if a broadcast packet
    int header_length; // length of header + CRC in bytes
    int data_length; // length of data in bytes
    double birth_time; // time packet created in Ethernet
Interface / Chip
    double allrecv_time; // time packet fully received from
the Ethernet
    int packet_ID; // Packet identification tag - eg the number
of packets sent by this node

protected:
//..... derived classes' data.....

    int Pkt_Type; // Packet type - eg GigE_Data

public:
//..... public constructors .....

// There should be no default constructor as we don't ever want to
create instances
// of GigEPacket (only subclasses), therefore it should return an
error
    GigEPacket( )
    {

        Error::abortRun ("Default GigEPacket Constructor
called","0","0");
        return ;
    }

//Destructor
    virtual ~GigEPacket() { return;}
// ~GigEPacket(); // destructor - use the default
one as all int etc.

//..... public services .....
```

```

NodeAddress get_dest_nodeid() const {return dest_address;}
NodeAddress get_source_nodeid() const {return source_address;}
int get_broadcast() const {return broadcast_flag;}
int get_data_len() const {return data_length;}
int get_total_len() const {return (data_length + header_length);}
double get_birth_time() const {return (birth_time);}
void set_birth_time(double time) {birth_time = time;}
double get_allrecv_time() const {return (allrecv_time);}
void set_allrecv_time(double time) {allrecv_time = time;}
int packet_id() {return packet_ID;}
void packet_id(int pkt_id) {packet_ID = pkt_id;}
StringList print_packet_id() const;
int get_Pkt_Type() const {return Pkt_Type;}
//..... Used by Ptolemy but not mandatory .....
    virtual StringList print () const ;

//..... Required by Ptolemy (see Ptolemy manual)
.....
    virtual const char* dataType() const    { return "GigEPacket" ; }
    ISA_INLINE( GigEPacket, Message ) // redefines isa() returns the
Message="GigEPacket"
    virtual GigEPacket* clone() const { LOG_NEW; return new GigEPacket(
*this ) ; }

//..... public services for subclasses
.....

    protected:
// Basic constructor required by subclass constructors
    GigEPacket(const NodeAddress& dest, const NodeAddress& source, int
bcast, int data_len)
    {
        dest_address    = dest ;
        source_address  = source ;
        broadcast_flag   = bcast ;
        data_length     = data_len ;
        header_length    = 18 ;
    }

// Copy constructor required by subclass constructors
    GigEPacket(const GigEPacket& src) : Message( src )
    {
        dest_address    = src.dest_address ;
        source_address  = src.source_address ;
        broadcast_flag   = src.broadcast_flag ;
        data_length     = src.data_length ;
        header_length    = src.header_length ;
        birth_time      = src.birth_time;
        allrecv_time    = src.allrecv_time;
        packet_ID       = src.packet_ID;
    }
};

// ===== class GigE_Data =====
//
// Message derived from GigEPacket base message class.
//
// Represents a packet that carries the applications data

class GigE_Data: public GigEPacket {

```

```

private:
//..... class data .....
    int message_id;          // An id to link all packets for 1
Amessage
    int num_packets;        // number of packets in a message
    int packet_no;         // the sequence number of this packet
(from 0)
    Amessage* user_data;    // pointer to the users data ie the ATLAS
message
public:
//..... public constructors .....

//Main constructor for creating message in first place
    GigE_Data(const NodeAddress& dest_address,
              const NodeAddress& source_address,
              int broadcast,
              int data_length,
              int message,
              int num_pkts,
              int pkt_no,
              Amessage* User_Data )
        :GigEPacket( dest_address, source_address, broadcast,
data_length)
    {
// Set information specific to this subclass
        message_id = message;
        num_packets = num_pkts;
        packet_no = pkt_no;
        Pkt_Type = 1;
        user_data = User_Data;
    }
//Copy constructor for creating message in first place
    GigE_Data( const GigE_Data& pkt )
        :GigEPacket( pkt )
    {
// Set information specific to this subclass
        message_id = pkt.message_id;
        num_packets = pkt.num_packets;
        packet_no = pkt.packet_no;
        Pkt_Type = 1;
        user_data = pkt.user_data;
    }

//..... public services .....
    int get_message_id() const {return message_id;}
    int get_num_packets() const {return num_packets;}
    int get_packet_no() const {return packet_no;}
    Amessage* get_user_data() const {return user_data;}

//..... Used by Ptolemy but not mandatory .....
// StringList print () const ;

//..... Required by Ptolemy (see Ptolemy manual)
.....
    const char* dataType() const { return "GigE_Data" ; }
    ISA_INLINE( GigE_Data, Message ) // redefines isA() returns the
MessageType="GigE_"
    GigEPacket* clone() const { LOG_NEW; return new GigE_Data( *this )
; }
};
};

```

```

// ===== class GigE_Filler =====
//
// Message derived from GigEPacket base message class.
//
// Represents a packet that pads out the network usage to the length
// of the applications data

class GigE_Filler: public GigEPacket {

private:
//..... class data .....
    int message_id;          // An id to link all packets for 1
Amessage
    int num_packets;        // number of packets in a message
    int packet_no;         // the sequence number of this packet
(from 0)

public:
//..... public constructors .....

//Main constructor for creating message in first place
    GigE_Filler(const NodeAddress& dest_address,
                const NodeAddress& source_address,
                int broadcast,
                int data_length,
                int message,
                int num_pkts,
                int pkt_no )
        :GigEPacket( dest_address, source_address, broadcast,
data_length)
    {
// Set information specific to this subclass
        message_id = message;
        num_packets = num_pkts;
        packet_no = pkt_no;
        Pkt_Type = 2;
    }
//Copy constructor for creating message in first place
    GigE_Filler( const GigE_Filler& pkt )
        :GigEPacket( pkt )
    {
// Set information specific to this subclass
        message_id = pkt.message_id;
        num_packets = pkt.num_packets;
        packet_no = pkt.packet_no;
        Pkt_Type = 2;
    }

//..... public services .....
    int get_message_id() const {return message_id;}
    int get_num_packets() const {return num_packets;}
    int get_packet_no() const {return packet_no;}

//..... Used by Ptolemy but not mandatory .....
// StringList print () const ;

//..... Required by Ptolemy (see Ptolemy manual)
.....
    const char* dataType() const { return "GigE_Filler" ; }

```

```

    ISA_INLINE( GigE_Filler, Message ) // redefines isA() returns the
    MessageType="GigE_"
    GigEPacket* clone() const { LOG_NEW; return new GigE_Filler( *this
    ) ; }
};

// ===== class GigE_Pause =====
//
// Message derived from GigEPacket base message class.
//
// Represents a Ethernet pause packet -
// indicates how long the transmitter must wait prior to sending the
// next packet

class GigE_Pause: public GigEPacket {

private:
//..... class data .....
    double pause_time;    // time to wait before sending next packet -
    0 == send now

public:
//..... public constructors .....

//Main constructor for creating message in first place
    GigE_Pause(const NodeAddress& dest_address,
               const NodeAddress& source_address,
               int broadcast,
               int data_length,
               double pause )
        :GigEPacket( dest_address, source_address, broadcast,
data_length)
    {
// Set information specific to this subclass
        pause_time = pause;
        Pkt_Type   = 3;
    }

//Copy constructor for creating message in first place
    GigE_Pause( const GigE_Pause& pkt )
        :GigEPacket( pkt )
    {
// Set information specific to this subclass
        pause_time = pkt.pause_time;
        Pkt_Type   = 3;
    }

//..... public services .....

    double get_pause_time() const    {return (pause_time);}
    void set_pause_time(double time) {pause_time = time;}

//..... Used by Ptolemy but not mandatory .....
// StringList print () const ;

//..... Required by Ptolemy (see Ptolemy manual)
.....
    const char* dataType() const    { return "GigE_Pause" ; }
    ISA_INLINE( GigE_Pause, Message ) // redefines isA() returns the
    MessageType="GigE_"
    GigEPacket* clone() const { LOG_NEW; return new GigE_Pause( *this )
; }
}

```

```

};

// ===== class GigE_Marp =====
//
// Message derived from GigEPacket base message class.
//
// Used to inform other interfaces on the network of the
// Ethernet MAC address of this node

class GigE_Marp: public GigEPacket {

private:
//..... class data .....
    int known_MAC;

public:
//..... public constructors .....

//Main constructor for creating message in first place
    GigE_Marp(const NodeAddress& dest_address,
              const NodeAddress& source_address,
              int broadcast,
              int data_length,
              int MAC_address)

        :GigEPacket( dest_address, source_address, broadcast,
data_length)
    {
// Set information specific to this subclass
        Pkt_Type    = 4;
        known_MAC = MAC_address;
    }
//Copy constructor for creating message in first place
    GigE_Marp( const GigE_Marp& pkt )
        :GigEPacket( pkt )
    {
// Set information specific to this subclass
        Pkt_Type    = 4;
        known_MAC = pkt.known_MAC;
    }

//..... public services .....
    int get_known_MAC() { return known_MAC;}

//..... Used by Ptolemy but not mandatory .....
// StringList print () const ;

//..... Required by Ptolemy (see Ptolemy manual)
.....
    const char* dataType() const    { return "GigE_Marp" ; }
    ISA_INLINE( GigE_Marp, Message ) // redefines isA() returns the
Message="GigE_"
    GigEPacket* clone() const { LOG_NEW; return new GigE_Marp( *this )
; }
};

```

5.3 GigELink::

This class sends and receives Gigabit Ethernet packets full duplex, IEEE 802.3z and implements flow control IEEE 802.3x. VLANs, IEEE 802.1Q and multicast support IEEE 802.1p are not considered. The class contains buffers for packets being sent to, and received from the Ethernet link. The action and depth of these buffers may be configured when the object is created. The speed of the Ethernet link and the application bus may also be set.

The class is as follows:

```
class GigELink {
private:
// Ptolemy specific
// InDEPort* link_in;
  OutDEPort* link_out;
  OutDEPort* FeedBack_out;

  double   LinkBitTime;           // GigE link speed in ns/bit
  double   LinkIPG;              // Time in ns for the 96 bit IPG

  int   inQlen;                  // length of input Q ethernet ->
local 0 == none
  int   inQspeed;                // ns. per byte
  int   inQhigh;                 // high water mark for Q
  int   outQlen;                 // length of output Q local ->
ethernet 0 == none
  int   outQspeed;               // ns. per byte
  int   outQhigh;               // high water mark for Q

  int LinkNo;                    // Link no. allowing identification of
simulation link
  int OutputControl;             // 0== no stats/histograms  oe the OR of:
                                  // 1= Stats printout
                                  // 2= Histos of Send and Recv Q lengths
                                  // 4= Graph vs time of Send and Recv Q lengths
  int MACaddress;                // MACaddress

  int packet_id_count;           // For identification of the packets
  int LinkState;                 // Status of Ethernet link : RUNNING, PAUSED
  int SendQState;                // Status of send Q re pause packet: NORMAL,
PKT_WAITING
  double pause_time;             // time to wait before sending next
packet - 0 == send now
  double packet_sent_time;       // time current packet will have
finished being transmitted on the ethernet
  double packet_output_time;     // time current packet will have
finished being sent to local system

  int Control_pkt_type; // save info here if already sending a packet
  double Control_PauseTime; // save info here if already sending a
packet

  SequentialList SendQ;          // list or Q of packets to send
  SequentialList RecvQ;          // list or Q of packets recieved

  GigEMacStats* SendStats;       // stats for sending part of the
ethernet link
  GigEMacStats* RecvStats;       // stats for receiving part of the
ethernet link
  char SendStatsMsg[80];
  char RecvStatsMsg[80];
};
```

```

    int inPktdata;          // Total data packets received from ethernet
    int inPktcontrol;      // Total control packets received from
ethernet
    int inPktdrop;        // Total data packets dropped
    int inPktdeliver;     // Total packets received from ethernet
and delivered to local
    int outPktreq;        // Total packets requested to be sent to
ethernet
    int outPktdata;       // Total data packets sent to ethernet
    int outPktcontrol;    // Total control packets sent to
ethernet
    int outPktdrop;       // Total data packets dropped

    XHistogram* SendQ_size_Hist;
    XHistogram* RecvQ_size_Hist;
    XGraph* SendQ_size_graph;
    XGraph* RecvQ_size_graph;
    char SendHistMsg[80];
    char RecvHistMsg[80];

    pt_ofstream* prt_out;    // for debug printout
    char  ERRmsg[100];

    NodeAddress paused_MAC;
    NodeAddress mynode_MAC;

protected:
    void ReFireAtTime(double when, int value);

public:
// methods defined in this section will be inline
    void Initialise(int LinkNo, OutDEPort* link_out_port, OutDEPort*
FeedBackOUT,
                    int inQlength, int inQrate, int inQ_high,
                    int outQlength, int outQrate, int outQ_high,
                    double LinkSpeed, int Output_control, int MyNodeID );
    GigEPacket* HaveFeedback(double TimeTag, int action);
    GigEPacket* HaveInput(double TimeTag, GigEPacket* Pkt);
    void SendData(double TimeTag, GigEPacket* Pkt);
    void SendControl(double TimeTag, double pause_time);

    int link_no() {return LinkNo;}

    GigEPacket* GigELink::PeekInQ(int position);
    int GigELink::RemoveInQ(GigEPacket* pointer);
    int GigELink::IsOutQFree();
    void GigELink::terminate();

    GigELink();          // constructor
    ~GigELink();         // destructor

//..... Required by Ptolemy (see Ptolemy manual)
.....
    const char* dataType() const    { return "GigELink" ; }

};

```