Final AyED

Generated by Doxygen 1.11.0

Chapter 1

AyEdD_Routers

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

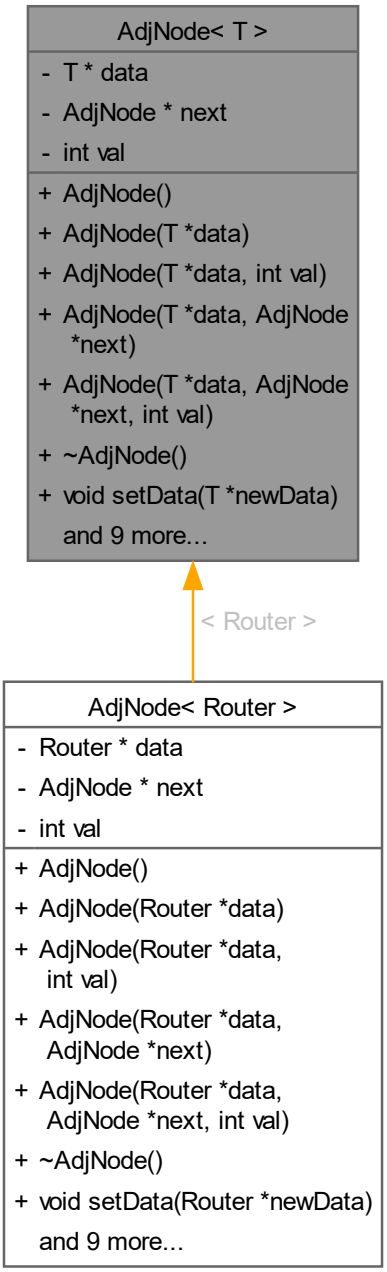
Class Documentation

4.1 AdjNode< T > Class Template Reference

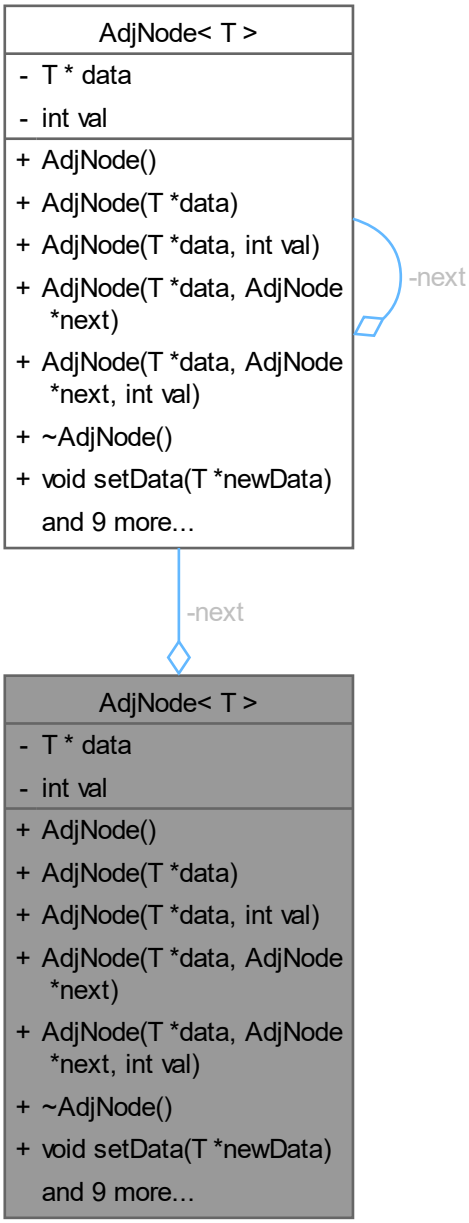
Adjacent Node class to be used in a linked list, stack or queue. Stores a pointer to the data, an integer value and a pointer to the next node in the structure.

`#include <AdjNode.hpp>`

Inheritance diagram for AdjNode< T >:



Collaboration diagram for AdjNode< T >:



Public Types

- using **valType** = T

Public Member Functions

- AdjNode ()

Default Constructor. Initializes a new instance of the Node class with ∗data and ∗next set to nullptr.

- AdjNode (T ∗data)

  Data Constructor. Initializes a new instance of the Node class with the provided data and a null next pointer.
- AdjNode (T ∗data, int val)

  Data Next Constructor. Initializes a new instance of the Node class with the provided data and int value. Sets ∗next to nullptr.
- AdjNode (T ∗data, AdjNode ∗next)

  Data Next Constructor. Initializes a new instance of the Node class with the provided data and next node.
- AdjNode (T ∗data, AdjNode ∗next, int val)

  Data Next Constructor. Initializes a new instance of the Node class with the provided data, int value and next node.
- ∼**AdjNode** ()

  Default Destructor.
- void setData (T ∗newData)

  Sets the data of the node.
- void setNext (AdjNode ∗newNext)

  Sets the next node in the structure.
- void setVal (int newA)

  Sets the integer value of the node.
- void addToVal (int a)

  Adds an integer value to the node.
- T ∗ getData () const

  Gets the data stored in the node.
- AdjNode< T > ∗ getNext () const

  Gets the next node in the structure.
- int getVal () const

  Gets the integer value stored in the node.
- bool hasNext ()

  Checks if there is a next node.
- bool operator== (const AdjNode< T > &) const

  Compares the data and integer value of two nodes.
- string toString () const

  Returns a string representation of the node's data.

Private Attributes

- T ∗ data
- AdjNode ∗ next
- int val = 1

4.1.1 Detailed Description

template<typename T>
class AdjNode< T >

Adjacent Node class to be used in a linked list, stack or queue. Stores a pointer to the data, an integer value and a pointer to the next node in the structure.

Template Parameters

T	Type of the data to be stored in the node.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 AdjNode() [1/5]

```
template<typename T >
AdjNode< T >::AdjNode ()
```

Default Constructor. Initializes a new instance of the Node class with ∗data and ∗next set to nullptr.

```
00125         : data(nullptr), next(nullptr) {}
```

4.1.2.2 AdjNode() [2/5]

```
template<typename T >
AdjNode< T >::AdjNode (
             T * data)  [explicit]
```

Data Constructor. Initializes a new instance of the Node class with the provided data and a null next pointer.

Parameters

data	Pointer to the data to be stored in the node.

```
00129         : data(data), next(nullptr) {}
```

4.1.2.3 AdjNode() [3/5]

```
template<typename T >
AdjNode< T >::AdjNode (
             T * data,
             int val)
```

Data Next Constructor. Initializes a new instance of the Node class with the provided data and int value. Sets ∗next to nullptr.

Parameters

data	Pointer to the data to be stored in the node.
val	Value to be stored in the node.

```
00133         : data(data), val(val) {}
```

4.1.2.4 AdjNode() [4/5]

```
template<typename T >
AdjNode< T >::AdjNode (
             T * data,
             AdjNode< T > * next)
```

Data Next Constructor. Initializes a new instance of the Node class with the provided data and next node.

Parameters

data	Pointer to the data to be stored in the node.
next	Pointer to the next node in the structure.

```
00137         : data(data), next(next) {}
```

4.1.2.5 AdjNode() [5/5]

```
template<typename T >
AdjNode< T >::AdjNode (
             T * data,
             AdjNode< T > * next,
             int val)
```

Data Next Constructor. Initializes a new instance of the Node class with the provided data, int value and next node.

Parameters

data	Pointer to the data to be stored in the node.
next	Pointer to the next node in the structure.
a	Integer value to be stored in the node.

```
00141         : data(data), next(next), val(val) {}
```

4.1.3 Member Function Documentation

4.1.3.1 addToVal()

```
template<typename T >
void AdjNode< T >::addToVal (
             int a)
```

Adds an integer value to the node.

Parameters

a	Integer value to add to the node.

```
00162                                {
00163     this->val += a;
00164 }
```

4.1.3.2 getData()

```
template<typename T >
T * AdjNode< T >::getData () const
```

Gets the data stored in the node.

Returns

Pointer to the node's data.

```
00167                               {
00168     return data;
00169 }
```

4.1.3.3 getNext()

```
template<typename T >
AdjNode< T > * AdjNode< T >::getNext () const
```

Gets the next node in the structure.

Returns

Pointer to the next node.

```
00172                                       {
00173     return next;
00174 }
```

4.1.3.4 getVal()

```
template<typename T >
int AdjNode< T >::getVal () const
```

Gets the integer value stored in the node.

Returns

Integer value stored in the node.

```
00177                                 {
00178     return this->val;
00179 }
```

4.1.3.5 hasNext()

```
template<typename T >
bool AdjNode< T >::hasNext ()
```

Checks if there is a next node.

Returns

True if there is a next node, false otherwise.

```
00182                             {
00183     return next != nullptr;
00184 }
```

4.1.3.6 operator==()

```
template<typename T >
bool AdjNode< T >::operator== (
            const AdjNode< T > & node) const
```

Compares the data and integer value of two nodes.

Parameters

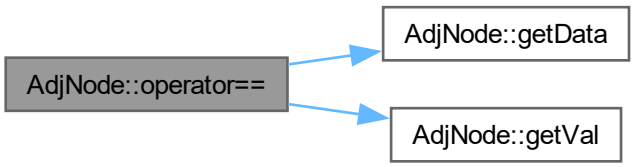
node	Node to compare data and integer value with.

Returns

True if the data and integer value are equal, false otherwise.

```
00187                                                       {
00188     return (*data == *node.getData() && val == node.getVal());
00189 }
```

Here is the call graph for this function:



4.1.3.7 setData()

```
template<typename T >
void AdjNode< T >::setData (
            T * newData)
```

Sets the data of the node.

Parameters

newData	Pointer to the new data for the node.

```
00147                                  {
00148     this->data = newData;
00149 }
```

4.1.3.8 setNext()

```
template<typename T >
void AdjNode< T >::setNext (
            AdjNode< T > * newNext)
```

Sets the next node in the structure.

Parameters

newNext	Pointer to the new next node.

```
00152                                      {
00153     this->next = newNext;
00154 }
```

4.1.3.9 setVal()

```
template<typename T >
void AdjNode< T >::setVal (
             int newA)
```

Sets the integer value of the node.

Parameters

newA	Integer value to be stored in the node.

```
00157                                {
00158     this->val = newA;
00159 }
```

4.1.3.10 toString()

```
template<typename T >
string AdjNode< T >::toString () const
```

Returns a string representation of the node's data.

Returns

String representing the node's data.

```
00192                                  {
00193     return data->toString() + "\tWeight: " + to_string(val);
00194 }
```

Here is the call graph for this function:

4.1.4 Member Data Documentation

4.1.4.1 data

```
template<typename T >
T* AdjNode< T >::data  [private]
```

Pointer to the data stored in the node.

4.1.4.2 next

```
template<typename T >
AdjNode* AdjNode< T >::next  [private]
```

Pointer to the next node in the structure.

4.1.4.3 val

```
template<typename T >
int AdjNode< T >::val = 1  [private]
```

Integer stored in the node.

The documentation for this class was generated from the following file:

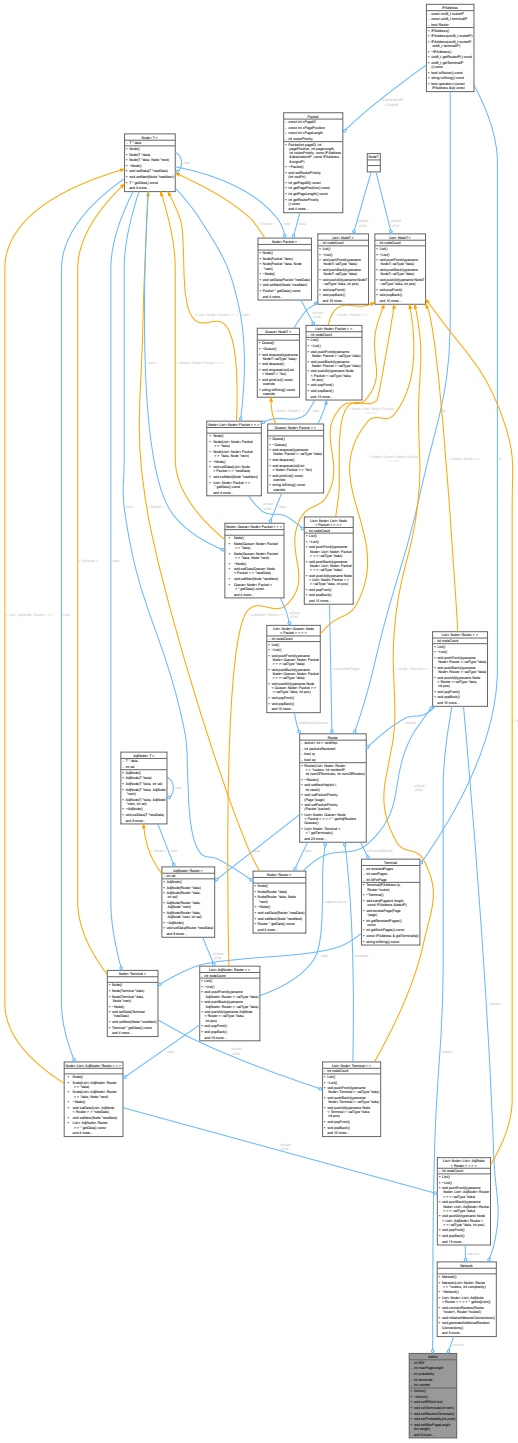
- include/AdjNode.hpp

4.2 Admin Class Reference

Manages the network and the routers.

```
#include <Admin.hpp>
```

Collaboration diagram for Admin:



Public Member Functions

- Admin ()
- ~Admin ()
- void setBW (int bw)

  Sets the bandwidth between routers.
- void setTerminals (int term)

Sets the number of terminals for each router.

- void setRoutersTerminals ()

  Creates terminals for each router, if they don't exist.

- void setProbability (int prob)

  Sets the probability of sending a page.

- void setMaxPageLength (int length)

  Sets the maximum length of a page.

- List< Node< Router > > * getRouters ()

  Gets the list of routers in the network.

- Network * getNetwork ()

  Gets the network object.

- void randomNetwork (int routersQuantity, int complexity)

  Creates a random network with a specified number of routers and complexity.

- void addUnconnectedRouter ()

  Adds an unconnected router to the network. Increases the size of every router's nextHop vector.

- void addRandomlyConnectedRouter ()

  Adds a router to the network and connects it to random routers.

- void sendPages ()

  Sends pages from terminals of each router to random destinations.

- void sendFromQueues ()

  Sends packets from the non empty queues of each router.

- bool checkCounter ()

  Checks and updates the counter, recalculating routes if necessary.

- void printRouters ()

  Prints information about all routers in the network.

Private Attributes

- List< Node< Router > > * **routers**
- Network * **network**
- int **BW** = 2
- int **maxPageLength** = 5
- int **probability** = 25
- int **terminals** = 2
- int **counter** = 0

4.2.1 Detailed Description

Manages the network and the routers.

The Admin class is responsible for managing the network and the routers within it. It provides methods to set various network parameters such as bandwidth, number of terminals, probability of sending a page, and maximum page length. The class also includes methods to create a random network, add routers (both unconnected and randomly connected), send pages and packets, it updates the routes every two iterations, and print information about the routers.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Admin()

```
Admin::Admin ()
```

Constructor. Initializes the network and the routers list as nullptr.

```
00003 : network(nullptr), routers(nullptr) {}
```

4.2.2.2 ~Admin()

```
Admin::~Admin ()
```

Destructor. Deletes the network and the routers list.

```
00005                 {
00006     delete network;
00007     delete routers;
00008 }
```

4.2.3 Member Function Documentation

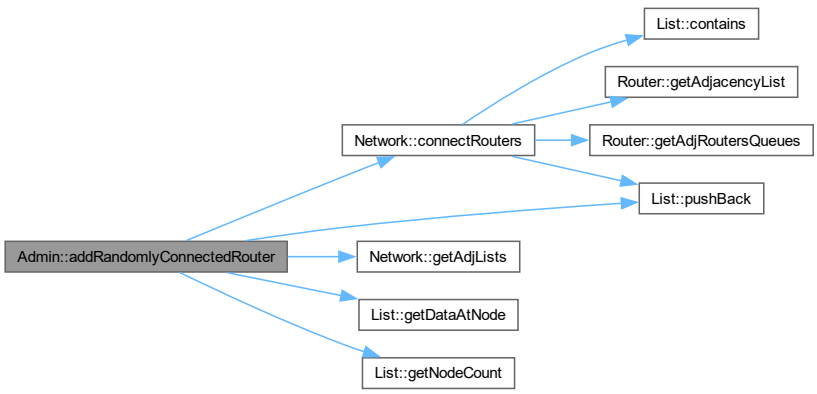
4.2.3.1 addRandomlyConnectedRouter()

```
void Admin::addRandomlyConnectedRouter ()
```

Adds a router to the network and connects it to random routers.

```
00075                                       {
00076     auto *router = new Router(routers, routers->getNodeCount(), terminals, routers->getNodeCount());
00077     routers->pushBack(router);
00078     network->getAdjLists()->pushBack(router->getAdjacencyList());
00079     for (int i = 0; i < routers->getNodeCount(); i++) {
00080         auto *router2 = routers->getDataAtNode(i);
00081         router2->addHopDest();
00082     }
00083     int routerAmount = routers->getNodeCount();
00084     for (int i = -1; i < rand() % routerAmount; i++) {
00085         auto *router2 = routers->getDataAtNode(rand() % routerAmount);
00086         network->connectRouters(router, router2);
00087     }
00088     cout«"Router added and connected to "«router->getAdjacencyList()->getNodeCount()«" random
      routers"«endl;
00089 }
```

Here is the call graph for this function:



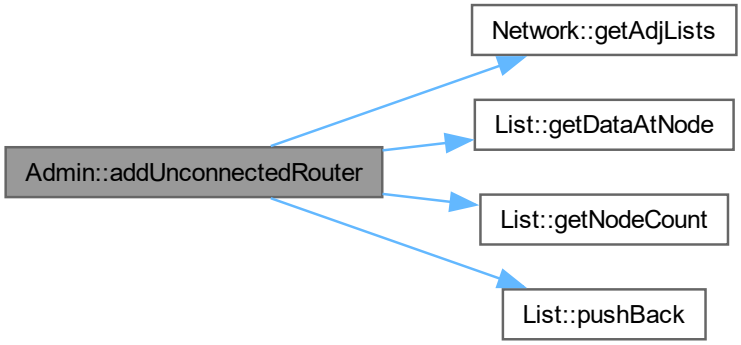
4.2.3.2 addUnconnectedRouter()

```
void Admin::addUnconnectedRouter ()
```

Adds an unconnected router to the network. Increases the size of every router's nextHop vector.

```
00065                                {
00066     auto *router = new Router(routers, routers->getNodeCount(), terminals, routers->getNodeCount());
00067     routers->pushBack(router);
00068     network->getAdjLists()->pushBack(router->getAdjacencyList());
00069     for (int i = 0; i < routers->getNodeCount(); i++) {
00070         auto *router2 = routers->getDataAtNode(i);
00071         router2->addHopDest();
00072     }
00073 }
```

Here is the call graph for this function:



4.2.3.3 checkCounter()

```
bool Admin::checkCounter ()
```

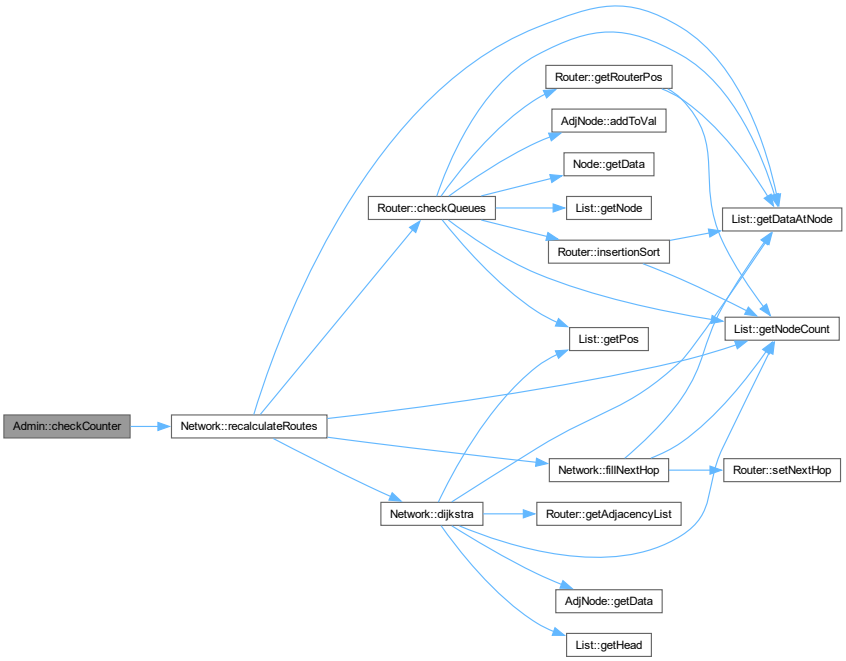
Checks and updates the counter, recalculating routes if necessary.

Returns

True if routes were recalculated, false otherwise.

```
00116                          {
00117     if (counter == 1) {          // If the counter reaches 1, recalculates routes.
00118         network->recalculateRoutes();
00119         counter = 0;             // Resets the counter.
00120         return true;
00121     } else {                     // Otherwise, increments the counter.
00122         counter++;
00123         return false;
00124     }
00125 }
```

Here is the call graph for this function:



4.2.3.4 getNetwork()

```
Network * Admin::getNetwork ()
```

Gets the network object.

Returns

A pointer to the network object.

```
00052                           {
00053     return network;
00054 }
```

4.2.3.5 getRouters()

```
List< Node< Router > > * Admin::getRouters ()
```

Gets the list of routers in the network.

Returns

A pointer to the list of routers.

```
00048                                   {
00049     return routers;
00050 }
```

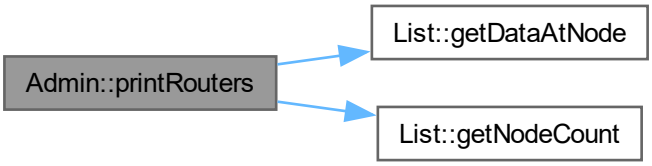
4.2.3.6 printRouters()

```
void Admin::printRouters ()
```

Prints information about all routers in the network.

```
00127                                {
00128     for (int i = 0; i < routers->getNodeCount(); i++) {
00129         auto *router = routers->getDataAtNode(i);
00130         router->printRouterInfo();
00131         cout << endl;
00132     }
00133 }
```

Here is the call graph for this function:



4.2.3.7 randomNetwork()

```
void Admin::randomNetwork (
            int routersQuantity,
            int complexity)
```

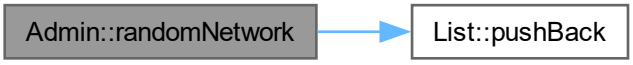
Creates a random network with a specified number of routers and complexity.

Parameters

routersQuantity	The number of routers to create.
complexity	The complexity level of the network connections.

```
00056                                                                {
00057     routers = new List<Node<Router>>();
00058     for (int i = 0; i < routersQuantity; i++) {
00059         auto *router = new Router(routers, i, terminals, routersQuantity);
00060         routers->pushBack(router);
00061     }
00062     network = new Network(routers, complexity);
00063 }
```

Here is the call graph for this function:



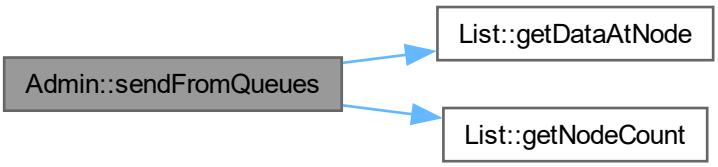
4.2.3.8 sendFromQueues()

```
void Admin::sendFromQueues ()
```

Sends packets from the non empty queues of each router.

```
00109                               {
00110     for (int i = 0; i < routers->getNodeCount(); i++) {         // Iterates through all routers.
00111         auto *router = routers->getDataAtNode(i);
00112         router->sendFromQueues(BW);                    // Sends packets from the router's queues.
00113     }
00114 }
```

Here is the call graph for this function:



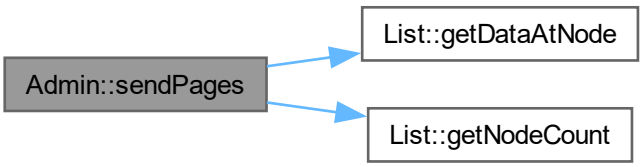
4.2.3.9 sendPages()

```
void Admin::sendPages ()
```

Sends pages from terminals of each router to random destinations.

```
00091                           {
00092     for (int i = 0; i < routers->getNodeCount(); i++) {     // Iterates through all routers.
00093         auto *router = routers->getDataAtNode(i);
00094         int rout = rand() % routers->getNodeCount();        // Random destination router.
00095         int term = rand() % terminals;                      // Random destination terminal.
00096         int length = (rand() % (maxPageLength-2)) + 2;      // Random page length.
00097         for (int j = 0; j < terminals; j++) {               // Iterates through all terminals in the
      current router.
00098             bool willSend = (rand() % 100) < probability;
00099             auto *terminal = router->getTerminals()->getDataAtNode(j);
00100             if (!willSend) {
00101                 continue;
00102             }
00103             cout«"Sending page from router "«i«" terminal "«j«" to router "«rout«" terminal "«term«"
      with length "«length«endl;
00104             terminal->sendPage(length,
      routers->getDataAtNode(rout)->getTerminals()->getDataAtNode(term)->getTerminalIp());
00105         }
00106     }
00107 }
```

Here is the call graph for this function:



4.2.3.10 setBW()

```
void Admin::setBW (
            int bw)
```

Sets the bandwidth between routers.

Parameters

bw	The bandwidth value to set.

```
00010                          {
00011     BW = bw;
00012 }
```

4.2.3.11 setMaxPageLength()

```
void Admin::setMaxPageLength (
            int length)
```

Sets the maximum length of a page.

Parameters

length	The maximum page length to set.

```
00044                                       {
00045     maxPageLength = length;
00046 }
```

4.2.3.12 setProbability()

```
void Admin::setProbability (
            int prob)
```

Sets the probability of sending a page.

Parameters

prob	The probability value to set.

```
00040                                    {
00041     probability = prob;
00042 }
```

4.2.3.13 setRoutersTerminals()

```
void Admin::setRoutersTerminals ()
```

Creates terminals for each router, if they don't exist.

```
00018                                {
00019     int routTermCount = routers->getDataAtNode(0)->getTerminals()->getNodeCount();
00020     if (routTermCount < terminals) {
00021         for (int i = 0; i < routers->getNodeCount(); i++) {
00022             auto *router = routers->getDataAtNode(i);
00023             auto *routerTerminals = routers->getDataAtNode(i)->getTerminals();
00024             for (int j = 0; j < (terminals - routTermCount); j++) {
00025                 routerTerminals->pushBack(new Terminal(IPAddress(router->getIP().getRouterIP(),
      routTermCount + j), router));
00026             }
00027         }
00028         printRouters();
00029     } else if (routTermCount > terminals) {
00030         cout « "New amount should be higher than current amount" « endl;
00031         //TODO: para implementarlo correctamente se deben recorrer las colas y eliminar los paquetes
      con destino a las terminales eliminadas
00032 //             for (int k = 0; k < (routTermCount - terminals); k++) {
00033 //                 delete routerTerminals->getTailData();
00034 //                 routerTerminals->popBack();
00035 //             }
00036     }
00037
00038 }
```

Here is the call graph for this function:

4.2.3.14 setTerminals()

```
void Admin::setTerminals (
            int term)
```

Sets the number of terminals for each router.

Parameters

term	The number of terminals to set.

```
00014                                {
00015     terminals = term;
00016 }
```

The documentation for this class was generated from the following files:

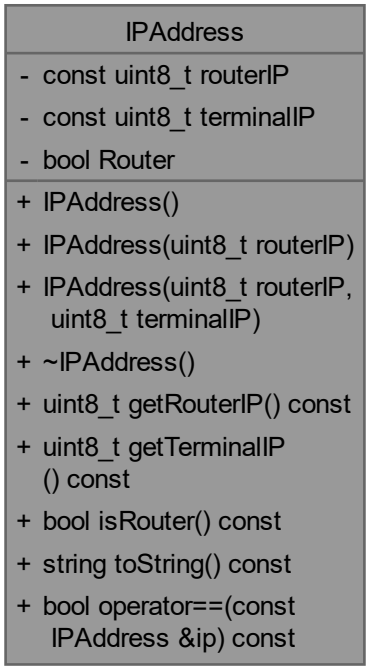
- include/Admin.hpp
- src/Admin.cpp

4.3 IPAddress Class Reference

Represents an IP address that can be used for either a router or a terminal. For a router, it uses an 8-bit address. For a terminal, it combines the router's 8-bit address with an additional 8-bit terminal address, resulting in a 16-bit address.

`#include <IPAddress.hpp>`

Collaboration diagram for IPAddress:



Public Member Functions

- IPAddress ()
  Default Constructor.
- IPAddress (uint8_t routerIP)
  Constructor for a router IP.
- IPAddress (uint8_t routerIP, uint8_t terminalIP)
  Constructor for a terminal IP.
- ~**IPAddress** ()
  Default Destructor.
- uint8_t getRouterIP () const
  Get the router IP.
- uint8_t getTerminalIP () const
  Get the terminal IP.
- bool isRouter () const

Checks if the IP is from a router.

- string toString () const

  Get a string representation of the IP.

- bool operator== (const IPAddress &ip) const

  Compare two IP addresses.

Private Attributes

- const uint8_t routerIP
- const uint8_t terminalIP
- bool Router

4.3.1 Detailed Description

Represents an IP address that can be used for either a router or a terminal. For a router, it uses an 8-bit address. For a terminal, it combines the router's 8-bit address with an additional 8-bit terminal address, resulting in a 16-bit address.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 IPAddress() [1/3]

```
IPAddress::IPAddress ()
```

Default Constructor.

```
00003 : routerIP(0), terminalIP(0), Router(true) {}
```

4.3.2.2 IPAddress() [2/3]

```
IPAddress::IPAddress (
            uint8_t routerIP)  [explicit]
```

Constructor for a router IP.

Parameters

router↩ IP	Router IP

```
00006            : routerIP(routerIP), terminalIP(0), Router(true) {}
```

4.3.2.3 IPAddress() [3/3]

```
IPAddress::IPAddress (
            uint8_t routerIP,
            uint8_t terminalIP)
```

Constructor for a terminal IP.

Parameters

routerIP	Router IP
terminal↩ IP	Terminal IP

```
00009             : routerIP(routerIP), terminalIP(terminalIP), Router(false) {}
```

4.3.3 Member Function Documentation

4.3.3.1 getRouterIP()

```
uint8_t IPAddress::getRouterIP () const
```

Get the router IP.

Returns

Router IP

```
00013                                    {
00014     return routerIP;
00015 }
```

4.3.3.2 getTerminalIP()

```
uint8_t IPAddress::getTerminalIP () const
```

Get the terminal IP.

Returns

Terminal IP

```
00017                                      {
00018     return terminalIP;
00019 }
```

4.3.3.3 isRouter()

```
bool IPAddress::isRouter () const
```

Checks if the IP is from a router.

Returns

True if the IP is from a router, false if it is from a terminal

```
00021                                {
00022     return Router;
00023 }
```

4.3.3.4 operator==()

```
bool IPAddress::operator== (
             const IPAddress & ip) const
```

Compare two IP addresses.

Parameters

ip	IP address to compare

Returns

True if the IP addresses are equal, false otherwise

```
00033                                                   {
00034     return routerIP == ip.routerIP && terminalIP == ip.terminalIP;
00035 }
```

4.3.3.5 toString()

```
string IPAddress::toString () const
```

Get a string representation of the IP.

Returns

String representation of the IP

```
00025                                   {
00026     if (Router) {
00027         return "Router IP: " + bitset<8>(routerIP).to_string();
00028     } else {
00029         return "Terminal IP: " + bitset<8>(routerIP).to_string() + "." +
      bitset<8>(terminalIP).to_string();
00030     }
00031 }
```

4.3.4 Member Data Documentation

4.3.4.1 Router

```
bool IPAddress::Router  [private]
```

True if the IP is from a router

4.3.4.2 routerIP

```
const uint8_t IPAddress::routerIP  [private]
```

Router IP.

4.3.4.3 terminalIP

```
const uint8_t IPAddress::terminalIP  [private]
```

Terminal IP

The documentation for this class was generated from the following files:

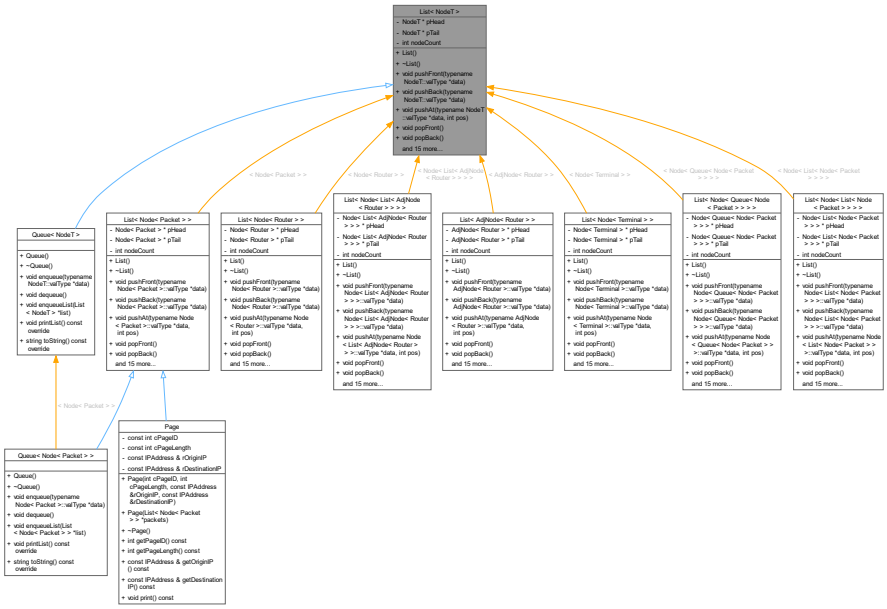
- include/IPAddress.hpp
- src/IPAddress.cpp

4.4 List< NodeT > Class Template Reference

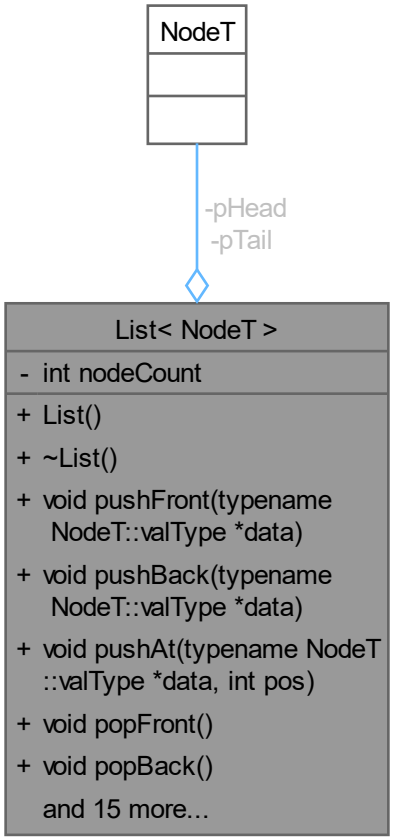
Composed of generic nodes (NodeT) that store at least a pointer to the data of the type that the node can store. This class provides a flexible structure for a singly linked list, allowing for common list operations such as insertion at the beginning, end, and at a specific position within the list, as well as removal of nodes and retrieval of data. It supports operations to check the list's emptiness, count the nodes, and search for data within the list. The class is designed to be used with any data type that can be pointed to by the nodes.

`#include <List.hpp>`

Inheritance diagram for List< NodeT >:



Collaboration diagram for List< NodeT >:



Public Member Functions

- List ()

  Default constructor.
- ~List ()

  Destructor for the List class. Iterates through the list and deletes each node to free up memory. Does not delete the data pointed to by the nodes.
- void pushFront (typename NodeT::valType *data)

  Adds a node with the specified data at the beginning of the list. The data type must match the type that NodeT can store.
- void pushBack (typename NodeT::valType *data)

  Adds a node with the specified data at the end of the list. The data type must match the type that NodeT can store.
- void pushAt (typename NodeT::valType *data, int pos)

  Adds a node with the specified data at the given position. The existing node at this position and all subsequent nodes are shifted one position to the end of the list. The data type must match the type that NodeT can store.
- void popFront ()

  Removes the first node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.
- void popBack ()

Removes the last node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.

- void popAt (int)

  Removes a node at the specified position from the list. The nodes after the removed node are shifted one position to the beginning of the list. If the position is invalid or the list is empty, it prints a message and does nothing. The NodeT object at the position is deleted, but the data pointed by the node is not deleted.
- void setDataAtNode (int pos, typename NodeT::valType *newData)

  Sets the data of the node at the specified position. If the position is invalid or the list is empty, does nothing.
- void swapNodesAt (int m, int n)

  Swaps the data of two nodes at the specified positions. If either position is invalid, it prints a message and does nothing. If the positions are the same, it does nothing.
- NodeT * getHead () const

  Gets a pointer to the first node in the list. If the list is empty, gets nullptr.
- NodeT * getTail () const

  Gets a pointer to the last node in the list. If the list is empty, gets nullptr.
- NodeT * getNode (int pos) const

  Gets a pointer to the node at the specified position. If the position is invalid or the list is empty, gets nullptr.
- NodeT::valType * getHeadData () const

  Returns a pointer to the data of the first node in the list. If the list is empty, returns nullptr.
- NodeT::valType * getTailData () const

  Returns a pointer to the data of the last node in the list. If the list is empty, returns nullptr.
- NodeT::valType * getDataAtNode (int pos) const

  Returns a pointer to the data of the node at the specified position. If the position is invalid or the list is empty, returns nullptr.
- int getPos (typename NodeT::valType *data) const

  Returns the position of the first node that contains the specified data. If the data is not found, returns -1.
- int getNodeCount () const

  Returns the number of nodes in the list.
- bool isEmpty () const

  Checks if the list is empty.
- bool contains (typename NodeT::valType *data) const

  Checks if the list contains a node with the specified data.
- virtual void printList () const

  Prints the data of all nodes in the list to the standard output.
- virtual string toString () const

  Returns a string representation of the list, containing the data of all nodes.

Private Attributes

- NodeT * pHead
- NodeT * pTail
- int nodeCount

4.4.1 Detailed Description

template<typename NodeT>
class List< NodeT >

Composed of generic nodes (NodeT) that store at least a pointer to the data of the type that the node can store. This class provides a flexible structure for a singly linked list, allowing for common list operations such as insertion at the beginning, end, and at a specific position within the list, as well as removal of nodes and retrieval of data. It supports operations to check the list's emptiness, count the nodes, and search for data within the list. The class is designed to be used with any data type that can be pointed to by the nodes.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 List()

```
template<typename NodeT >
List< NodeT >::List ()
```

Default constructor.

```
00177     : pHead(nullptr),
00178       pTail(nullptr),
00179       nodeCount(0) {}
```

4.4.2.2 ~List()

```
template<typename NodeT >
List< NodeT >::~List ()
```

Destructor for the List class. Iterates through the list and deletes each node to free up memory. Does not delete the data pointed to by the nodes.

```
00182                     {
00183     while (pHead != nullptr) {
00184         NodeT *temp = pHead;
00185         pHead = pHead->getNext();
00186         delete temp;
00187     }
00188 }
```

4.4.3 Member Function Documentation

4.4.3.1 contains()

```
template<typename NodeT >
bool List< NodeT >::contains (
            typename NodeT::valType * data) const
```

Checks if the list contains a node with the specified data.

Parameters

data	Pointer to the data to search for.

Returns

bool True if the data is found, false otherwise.

```
00464                                                       {
00465     return getPos(data) != -1;
00466 }
```

4.4.3.2 getDataAtNode()

```
template<typename NodeT >
NodeT::valType * List< NodeT >::getDataAtNode (
            int pos) const
```

Returns a pointer to the data of the node at the specified position. If the position is invalid or the list is empty, returns nullptr.

Parameters

pos	The position of the node.

Returns

The data of the node at the specified position.

```
00433                                                                 {
00434     return getNode(pos) ? getNode(pos)->getData() : nullptr;
00435 }
```

4.4.3.3 getHead()

```
template<typename NodeT >
NodeT * List< NodeT >::getHead () const
```

Gets a pointer to the first node in the list. If the list is empty, gets nullptr.

Returns

The first node.

```
00391                                 {
00392     return pHead;
00393 }
```

4.4.3.4 getHeadData()

```
template<typename NodeT >
NodeT::valType * List< NodeT >::getHeadData () const
```

Returns a pointer to the data of the first node in the list. If the list is empty, returns nullptr.

Returns

The data of the first node.

```
00423                                                 {
00424     return pHead ? pHead->getData() : nullptr;
00425 }
```

4.4.3.5 getNode()

```
template<typename NodeT >
NodeT * List< NodeT >::getNode (
            int pos) const
```

Gets a pointer to the node at the specified position. If the position is invalid or the list is empty, gets nullptr.

Parameters

pos	The position of the node.

Returns

The node at the specified position.

```
00401                                       {
00402     if (pHead == nullptr || pos < 0 || pos >= nodeCount)    // Return nullptr for invalid position or
      empty list
00403         return nullptr;
00404
00405     if (pos == 0)                             // If the position is 0, call getHeadData
00406         return this->getHead();
00407
00408     if (pos == nodeCount - 1)                 // If the position is the last one, call getTailData
00409         return this->getTail();
00410
00411     NodeT *aux = pHead->getNext();
00412     int curr_pos = 1;
00413
00414     while (curr_pos != pos) {                 // Traverse the list until the position is reached
00415         aux = aux->getNext();
00416         curr_pos++;
00417     }
00418
00419     return aux;
00420 }
```

4.4.3.6 getNodeCount()

```
template<typename NodeT >
int List< NodeT >::getNodeCount () const
```

Returns the number of nodes in the list.

Returns

The number of nodes.

```
00454                                   {
00455     return nodeCount;
00456 }
```

4.4.3.7 getPos()

```
template<typename NodeT >
int List< NodeT >::getPos (
            typename NodeT::valType * data) const
```

Returns the position of the first node that contains the specified data. If the data is not found, returns -1.

Parameters

data	Pointer to the data to search for.

Returns

The position of the node, or -1 if not found.

```
00438                                                   {
00439
00440     NodeT *current = pHead;
00441     int pos = 0;
00442
00443     while (current) {                         // Traverse the list until the node with the specified
      data is found
00444         if (*current->getData() == *data)
00445             return pos;
00446         current = current->getNext();
00447         pos++;
00448     }
00449
00450     return -1;                                // Return -1 if the data is not found
00451 }
```

4.4.3.8 getTail()

```
template<typename NodeT >
NodeT * List< NodeT >::getTail () const
```

Gets a pointer to the last node in the list. If the list is empty, gets nullptr.

Returns

The last node.

```
00396                                   {
00397     return pTail;
00398 }
```

4.4.3.9 getTailData()

```
template<typename NodeT >
NodeT::valType * List< NodeT >::getTailData () const
```

Returns a pointer to the data of the last node in the list. If the list is empty, returns nullptr.

Returns

The data of the last node.

```
00428                                                   {
00429     return pHead ? pTail->getData() : nullptr;
00430 }
```

4.4.3.10 isEmpty()

```
template<typename NodeT >
bool List< NodeT >::isEmpty () const
```

Checks if the list is empty.

Returns

True if the list is empty, false otherwise.

```
00459                                   {
00460     return (nodeCount == 0);
00461 }
```

4.4.3.11 popAt()

```
template<typename NodeT >
void List< NodeT >::popAt (
            int pos)
```

Removes a node at the specified position from the list. The nodes after the removed node are shifted one position to the beginning of the list. If the position is invalid or the list is empty, it prints a message and does nothing. The NodeT object at the position is deleted, but the data pointed by the node is not deleted.

Parameters

pos	The position of the node to be removed.

```
00304                                {
00305     if (pos < 0 || pos >= nodeCount) {      // Validate the position
00306         cout « "Can't delete node, invalid position." « endl;
00307         return;
00308     }
00309
00310     if (pos == 0) {                         // If the position is 0, calls popFront
00311         this->popFront();
00312         return;
00313     }
00314
00315     if (pos == (this->getNodeCount() - 1)) {// If the position is the last one, calls popBack
00316         this->popBack();
00317         return;
00318     }
00319
00320     NodeT* aux_prev = pHead;                // Will point to the node before the position
00321     NodeT* aux = pHead->getNext();          // Will point to the node to delete
00322     int curr_pos = 1;
00323
00324     while (curr_pos != pos) {               // Traverse the list until the position is reached
00325         aux_prev = aux;
00326         aux = aux->getNext();
00327         curr_pos++;
00328     }
00329
00330     aux_prev->setNext(aux->getNext());      // The node before the position points to the node after
      the position
00331     delete aux;                             // Delete the node at the position
00332     nodeCount--;                            // Decrease the node count
00333 }
```

4.4.3.12 popBack()

```
template<typename NodeT >
void List< NodeT >::popBack ()
```

Removes the last node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.

```
00277                           {
00278     if (pTail == nullptr) {                 // If the list is empty
00279         cout « "Can't delete anything, the list is empty." « endl;
00280         return;
00281     }
00282
00283     NodeT* old_tail = pTail;
00284
00285     if (this->getNodeCount() == 1) {        // If there is only one node the list becomes empty
00286         pHead = nullptr;
00287         pTail = nullptr;
00288     }
00289     else {
00290         NodeT* aux = pHead;
00291         while (aux->getNext() != pTail) {
00292             aux = aux->getNext();
00293         }
00294
00295         aux->setNext(nullptr);              // The node before the last one becomes the last one
00296         pTail = aux;                        // Also becomes the pTail
00297     }
00298
00299     delete old_tail;                        // Delete the old pTail
00300     nodeCount--;                            // Decrease the node count
00301 }
```

4.4.3.13 popFront()

```
template<typename NodeT >
void List< NodeT >::popFront ()
```

Removes the first node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.

```
00256                                {
00257     if (pHead == nullptr) {                 // If the list is empty
00258         cout « "Can't delete anything, the list is empty." « endl;
00259         return;
00260     }
00261
00262     NodeT* old_head = pHead;
00263
00264     if (this->getNodeCount() == 1) {        // If there is only one node the list becomes empty
00265         pHead = nullptr;
00266         pTail = nullptr;
00267     }
00268     else {
00269         pHead = pHead->getNext();           // The second node becomes the pHead
00270     }
00271
00272     delete old_head;                        // Delete the old pHead
00273     nodeCount--;                            // Decrease the node count
00274 }
```

4.4.3.14 printList()

```
template<typename NodeT >
void List< NodeT >::printList () const  [virtual]
```

Prints the data of all nodes in the list to the standard output.

Reimplemented in Queue< NodeT >, and Queue< Node< Packet > >.

```
00469                                  {
00470     cout « "Start of list" « endl;
00471     NodeT *aux = pHead;
00472
00473     while (aux) {                           // Traverse the list and print the data of each node while
      the pointer is not null
00474         if (aux->getData() != nullptr) {
00475             cout « aux->getData()->toString() « endl; // Dereference the pointer to the data, needs to
      overload the « operator
00476         }
00477         aux = aux->getNext();
00478     }
00479
00480     cout « "End of list" « endl;
00481 }
```

4.4.3.15 pushAt()

```
template<typename NodeT >
void List< NodeT >::pushAt (
             typename NodeT::valType * data,
             int pos)
```

Adds a node with the specified data at the given position. The existing node at this position and all subsequent nodes are shifted one position to the end of the list. The data type must match the type that NodeT can store.

Parameters

data	Pointer to the data to be stored in the new node.
pos	The position at which the new node will be inserted.

```
00222                                                              {
00223
00224     if (pos < 0 || pos > nodeCount) {       // validate the position
00225         cout « "Can't insert node, invalid position." « endl;
00226         return;
00227     }
00228
00229     if (pos == 0) {                         // If the position is 0, calls pushFront
```

```
00230         this->pushFront(data);
00231         return;
00232     }
00233
00234     if (pos == nodeCount) {                 // If the position is the last one, calls pushBack
00235         this->pushBack(data);
00236         return;
00237     }
00238
00239     NodeT *aux_prev = pHead;                // Will point to the node before the position
00240     NodeT *aux = pHead->getNext();          // Will point to the node at the position
00241     int curr_pos = 1;
00242
00243     while (curr_pos != pos) {               // Traverse the list until the position is reached
00244         aux_prev = aux;
00245         aux = aux->getNext();
00246         curr_pos++;
00247     }
00248
00249     auto* new_node_at = new NodeT(data, aux);  // The node at the position becomes the next node of
      the new node
00250     aux_prev->setNext(new_node_at);         // The node before the position points to the new node
00251
00252     nodeCount++;                            // Increase the node count
00253 }
```

4.4.3.16 pushBack()

```
template<typename NodeT >
void List< NodeT >::pushBack (
             typename NodeT::valType * data)
```

Adds a node with the specified data at the end of the list. The data type must match the type that NodeT can store.

Parameters

data	Pointer to the data to be stored in the new node

```
00206                                                   {
00207
00208     auto *new_tail = new NodeT(data, nullptr);
00209
00210     if (this->isEmpty()) {                  // If the list is empty, the new node is also the first
      node.
00211         pHead = new_tail;
00212     }
00213     else {
00214         pTail->setNext(new_tail);           // If the list is not empty, the old pTail points to the
      new node
00215     }
00216
00217     pTail = new_tail;                       // The new node is now pTail
00218     nodeCount++;                            // Increase the node count
00219 }
```

4.4.3.17 pushFront()

```
template<typename NodeT >
void List< NodeT >::pushFront (
             typename NodeT::valType * data)
```

Adds a node with the specified data at the beginning of the list. The data type must match the type that NodeT can store.

Parameters

data	pointer to the data of the type that can be stored in the new node

```
00191                                                       {
00192
00193     auto *new_head = new NodeT(data, pHead);
00194
00195     if (this->isEmpty()) {                  // If the list is empty, the new node is also the last
      node.
00196         pHead = new_head;
00197         pTail = new_head;
00198     }
00199     else {                                  // If the list is not empty, pHead points to the new node.
00200         pHead = new_head;
00201     }
00202     nodeCount++;                            // Increase the node count.
00203 }
```

4.4.3.18 setDataAtNode()

```
template<typename NodeT >
void List< NodeT >::setDataAtNode (
             int pos,
             typename NodeT::valType * newData)
```

Sets the data of the node at the specified position. If the position is invalid or the list is empty, does nothing.

Parameters

pos	The position of the node.
newData	Pointer to the new data to be set.

```
00336                                                                   {
00337     if (pHead == nullptr || pos < 0 || pos >= nodeCount)    // Return nullptr for invalid position or
      empty list
00338         return;
00339
00340     if (pos == 0) {                         // If the position is 0, calls setData on the head
00341         pHead->setData(newData);
00342         return;
00343     }
00344
00345     if (pos == nodeCount - 1) {             // If the position is the last one, calls setData on the
      tail
00346         pTail->setData(newData);
00347         return;
00348     }
00349
00350     NodeT *aux = pHead->getNext();
00351     int curr_pos = 1;
00352
00353     while (curr_pos != pos) {               // Traverse the list until the position is reached
00354         aux = aux->getNext();
00355         curr_pos++;
00356     }
00357
00358     aux->setData(newData);                  // Sets the new data
00359 }
```

4.4.3.19 swapNodesAt()

```
template<typename NodeT >
void List< NodeT >::swapNodesAt (
             int m,
             int n)
```

Swaps the data of two nodes at the specified positions. If either position is invalid, it prints a message and does nothing. If the positions are the same, it does nothing.

Parameters

m	The position of the first node to swap.
n	The position of the second node to swap.

```
00362                                                {
00363     if (m < 0 || n < 0 || m >= nodeCount || n >= nodeCount) {     //validate the positions
00364         cout « "Invalid value." « endl;
00365         return;
00366     }
00367
00368     if (m == n) {                           // If the positions are the same, there is nothing to do
00369         return;
00370     }
00371
00372     NodeT *node_m = pHead;
00373     NodeT *node_n = pHead;
00374
00375     for (int i = 0; i <= max(m, n); ++i) {  // Traverse the list to find the nodes at M and N
00376         if (i < m) {
00377             node_m = node_m->getNext();
00378         }
00379
00380         if (i < n) {
00381             node_n = node_n->getNext();
00382         }
00383     }
00384
00385     typename NodeT::valType* aux = node_m->getData();   // Swap the data of the nodes
00386     node_m->setData(node_n->getData());
00387     node_n->setData(aux);
00388 }
```

4.4.3.20 toString()

```
template<typename NodeT >
string List< NodeT >::toString () const  [virtual]
```

Returns a string representation of the list, containing the data of all nodes.

Returns

string A string representation of the list.

Reimplemented in Queue< NodeT >, and Queue< Node< Packet > >.

```
00484                                       {
00485     return "holi, soy una lista";
00486     // TODO: implementar bien el toString y el printList, testearlos
00487 }
```

4.4.4 Member Data Documentation

4.4.4.1 nodeCount

```
template<typename NodeT >
int List< NodeT >::nodeCount  [private]
```

Number of nodes in the list

4.4.4.2 pHead

```
template<typename NodeT >
NodeT* List< NodeT >::pHead  [private]
```

Pointer to the first node in the list

4.4.4.3 pTail

```
template<typename NodeT >
NodeT* List< NodeT >::pTail  [private]
```

Pointer to the last node in the list

The documentation for this class was generated from the following file:

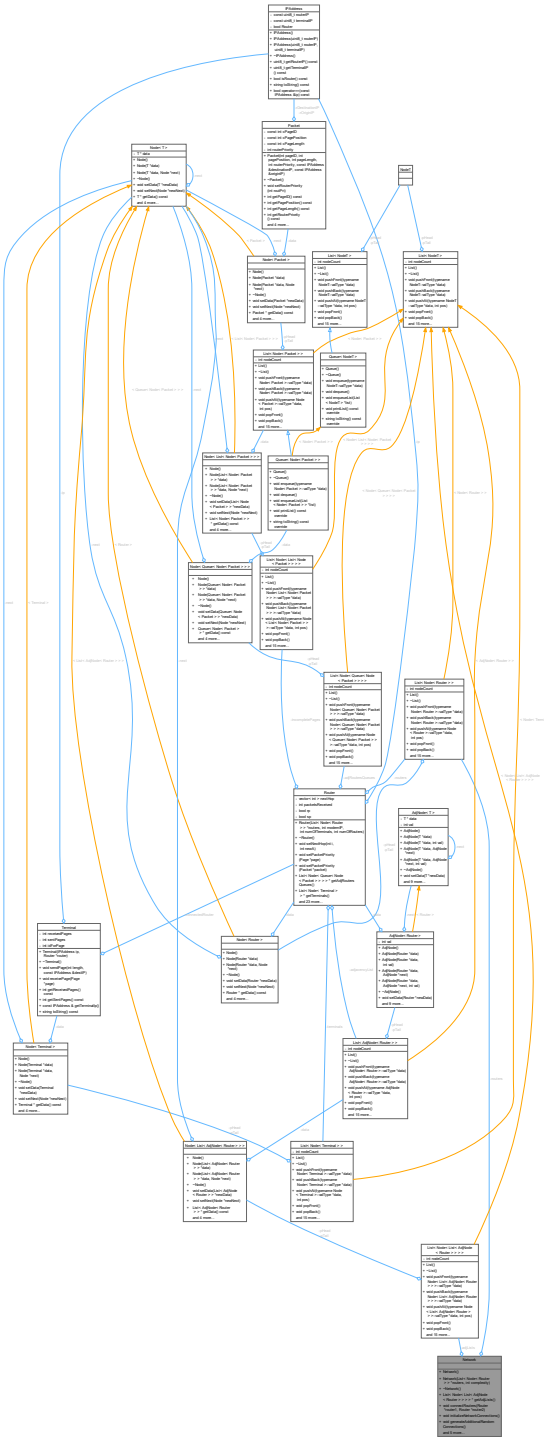
- include/List.hpp

4.5 Network Class Reference

Manages a network of routers, providing functionalities for network topology and routing. The Network class encapsulates a collection of routers and their connections, simulating a network environment. It allows for the initialization of the network with a set of routers, the establishment of connections between routers, and the calculation of routing paths using Dijkstra's algorithm. The class supports dynamic network configurations, enabling the addition of new connections and the recalibration of routing paths as the network evolves.

```
#include <Network.hpp>
```

Collaboration diagram for Network:



Public Member Functions

- Network ()

  Default constructor. Initializes an empty list of routers and an empty list of adjacency lists.

- Network (List< Node< Router > > *routers, int complexity)

  Constructor, initializes the list of routers with the routers received as a parameter. For each router, gets the adjacency list and adds it to the list of adjacency lists. Then generates a random network.

- ~Network ()

  Destructor. Traverses the list of routers, deleting each router. Deletes all the nodes in routers, but does not delete the list itself.
- List< Node< List< AdjNode< Router > > > > * getAdjLists ()

  Gets the list every router's adjacency list.
- void connectRouters (Router *router1, Router *router2)

  Connects two routers bidirectionally with a default weight of 1. Adjacency list in both routers is updated. If both routers are the same, or they are already connected, the method does nothing.
- void initializeNetworkConnections ()

  This method constructs a random network where each existing router is randomly connected to one of the already connected routers, ensuring there are no isolated routers.
- void generateAdditionalRandomConnections ()

  Traverse the list of routers, generating a potential new connection for each router.
- void generateRandomNetwork (int iter)

  Generates a random network with a specified level of connectivity (0-20). The higher the level, the more connected the network will be. If the parameter is out of bounds, 1 is used as the default value.
- vector< int > dijkstra (int start)

  Applies Dijkstra's algorithm to calculate the shortest path from a given router to all other routers, generating a vector of parents for each router in the network.
- void fillNextHop (int posRouter, vector< int > parent)

  For a given router, calculates the next hop to reach each router in the network and stores it in the nextHop vector of the router.
- void recalculateRoutes ()

  Recalculates the shortest path from each router to all other routers in the network, updating the nextHop vector of each router accordingly.
- bool isConnected (Router *router1, Router *router2)

  Checks if two routers are connected.

Private Attributes

- List< Node< Router > > * routers
- List< Node< List< AdjNode< Router > > > > adjLists

4.5.1 Detailed Description

Manages a network of routers, providing functionalities for network topology and routing. The Network class encapsulates a collection of routers and their connections, simulating a network environment. It allows for the initialization of the network with a set of routers, the establishment of connections between routers, and the calculation of routing paths using Dijkstra's algorithm. The class supports dynamic network configurations, enabling the addition of new connections and the recalibration of routing paths as the network evolves.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 Network() [1/2]

```
Network::Network ()
```

Default constructor. Initializes an empty list of routers and an empty list of adjacency lists.

```
00003 : routers(), adjLists() {}
```

4.5.2.2 Network() [2/2]

```
Network::Network (
            List< Node< Router > > * routers,
            int complexity)  [explicit]
```

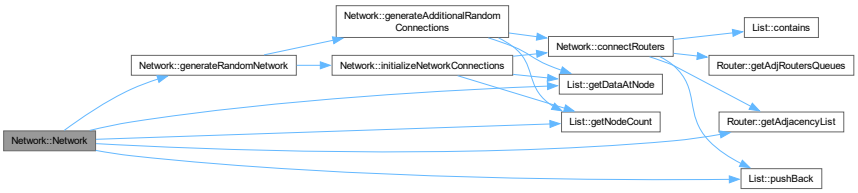
Constructor, initializes the list of routers with the routers received as a parameter. For each router, gets the adjacency list and adds it to the list of adjacency lists. Then generates a random network.

Parameters

routers	List of routers to work with

```
00005                                                   : routers(routers), adjLists() {
00006     for (int i = 0; i < routers->getNodeCount(); i++) {
00007         Router *router = routers->getDataAtNode(i);
00008         adjLists.pushBack(router->getAdjacencyList());
00009         generateRandomNetwork(complexity);
00010     }
00011 }
```

Here is the call graph for this function:



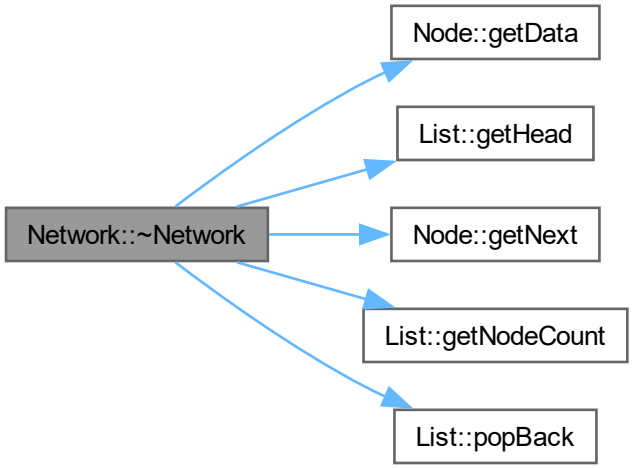
4.5.2.3 ~Network()

```
Network::~Network ()
```

Destructor. Traverses the list of routers, deleting each router. Deletes all the nodes in routers, but does not delete the list itself.

```
00013                   {
00014     Node<Router> *curr = routers->getHead();
00015     while(curr){
00016         delete curr->getData();
00017         curr = curr->getNext();
00018     }
00019     for (int i = 0; i < routers->getNodeCount(); i++) {
00020         routers->popBack();
00021     }
00022 }
```

Here is the call graph for this function:



4.5.3 Member Function Documentation

4.5.3.1 connectRouters()

```
void Network::connectRouters (
            Router * router1,
            Router * router2)
```

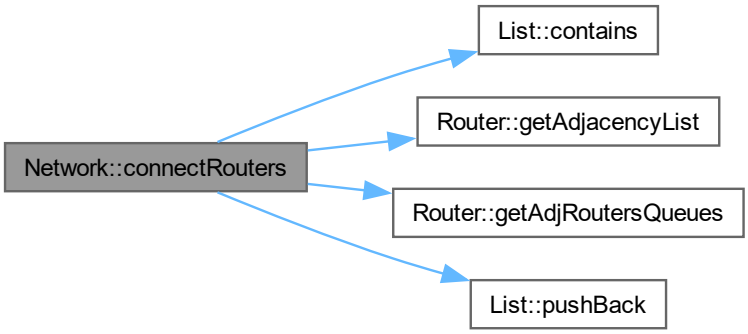
Connects two routers bidirectionally with a default weight of 1. Adjacency list in both routers is updated. If both routers are the same, or they are already connected, the method does nothing.

Parameters

router1	Pointer to the first router
router2	Pointer to the second router

```
00028                                                                {
00029     if (*router1 == *router2) {
00030         return;
00031     }
00032     List<AdjNode<Router>> *list1 = router1->getAdjacencyList();
00033     List<AdjNode<Router>> *list2 = router2->getAdjacencyList();
00034     if (!list1->contains(router2)) {
00035         list1->pushBack(router2);
00036         list2->pushBack(router1);
00037         router1->getAdjRoutersQueues()->pushBack(new Queue<Node<Packet>>());
00038         router2->getAdjRoutersQueues()->pushBack(new Queue<Node<Packet>>());
00039     }
00040 }
```

Here is the call graph for this function:



4.5.3.2 dijkstra()

```
vector< int > Network::dijkstra (
            int start)
```

Applies Dijkstra's algorithm to calculate the shortest path from a given router to all other routers, generating a vector of parents for each router in the network.

Parameters

start	Position of the starting router in the list of routers

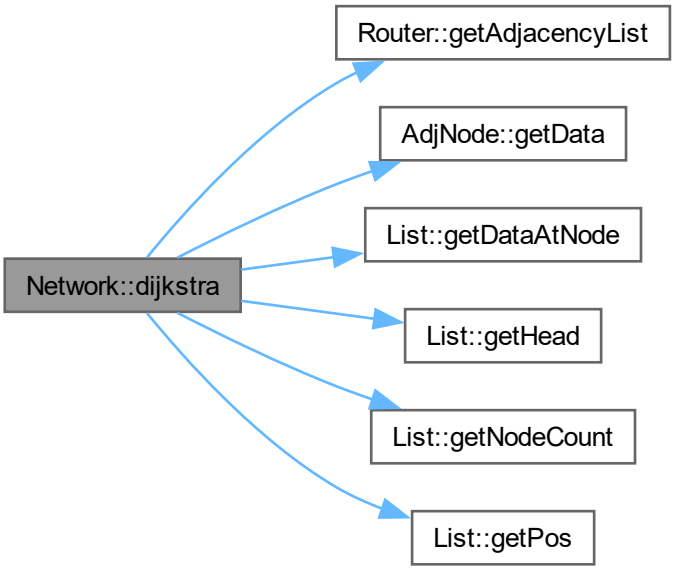
Returns

Vector of parents of each router in the path

```
00084                                       {
00085     int nodeCount = routers->getNodeCount();
00086     vector<bool> visited(nodeCount, false);
00087     vector<int> dist(nodeCount, INF);
00088     vector<int> parents(nodeCount, 0);
00089     dist[start] = 0;                          // Set the Weight from the starting router to itself to 0
00090     parents[start] = start;                   // Set the parent of the starting router to itself
00091     Router *nearest;                        // Pointer to the nearest router in the actual iteration
00092     int adjNode;                           // Position of the current adjacent node
00093     int adjWeight;                         // Weight of the current adjacent node
00094 // Iteration for each non-visited router with the minimum distance
00095     for(int i = 0; i < adjLists.getNodeCount(); i++){
00096         int locDist = INF;
00097         int distNearest = 0;                                                  // termina siendo la pos del
      router más cercano
00098         for(int j = 0; j < adjLists.getNodeCount(); j++) {  // este for es el getMinDist()
00099             if (!visited[j] && dist[j] < locDist) {
00100                 locDist = dist[j];
00101                 distNearest = j;
00102             }
00103         }
00104         visited[distNearest] = true;         // Mark the nearest router as visited
00105         nearest = routers->getDataAtNode(distNearest);    // Get the pointer to the nearest router
00106         //distNearest = locDist;              // Get the weight to the nearest router
00107         auto *currNode = nearest->getAdjacencyList()->getHead();
00108         while(currNode){
00109             auto *adjRouter = currNode->getData();
00110             adjWeight = currNode->getVal();
```

```
00111             adjNode = routers->getPos(adjRouter);
00112             if(!visited[adjNode] && locDist + adjWeight < dist[adjNode]){
00113                 dist[adjNode] = locDist + adjWeight;
00114                 parents[adjNode] = distNearest;
00115             }
00116             currNode = currNode->getNext();
00117         }
00118     }
00119     return parents;
00120 }
```

Here is the call graph for this function:



4.5.3.3 fillNextHop()

```
void Network::fillNextHop (
             int posRouter,
             vector< int > parent)
```

For a given router, calculates the next hop to reach each router in the network and stores it in the nextHop vector of the router.

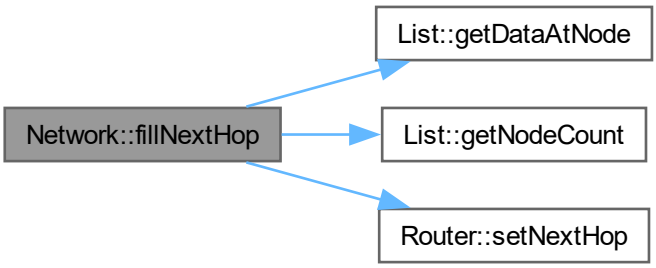
Parameters

posRouter	Position of the router in the list of routers
parent	Vector of parents of each router in the path

```
00122                                                                {
00123     Router *startRouter = routers->getDataAtNode(posRouter);
00124     for (int i = 0; i < routers->getNodeCount(); i++) {
00125         if (i == posRouter) {
00126             startRouter->setNextHop(i, posRouter);
00127         } else {
00128             int current = i;
00129             while (parent[current] != posRouter) {
```

```
00130                 current = parent[current];
00131             }
00132             startRouter->setNextHop(i, current);
00133         }
00134     }
00135 }
```

Here is the call graph for this function:



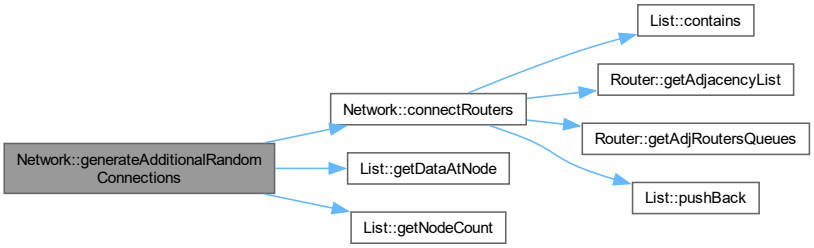
4.5.3.4 generateAdditionalRandomConnections()

```
void Network::generateAdditionalRandomConnections ()
```

Traverse the list of routers, generating a potential new connection for each router.

```
00057                                                 {
00058     int nodeCount = routers->getNodeCount();
00059     if (nodeCount < 3) {
00060         return;
00061     }
00062     srand(time(nullptr));
00063     for (int i = 0; i < nodeCount; i++) {
00064         Router *router = routers->getDataAtNode(i);
00065         int randPos;
00066         do {
00067             randPos = rand() % nodeCount;
00068         } while (randPos == i);
00069         connectRouters(routers->getDataAtNode(randPos), router);
00070     }
00071 }
```

Here is the call graph for this function:



4.5.3.5 generateRandomNetwork()

```
void Network::generateRandomNetwork (
            int iter)
```

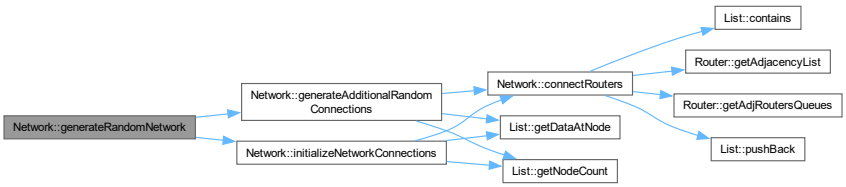
Generates a random network with a specified level of connectivity (0-20). The higher the level, the more connected the network will be. If the parameter is out of bounds, 1 is used as the default value.

Parameters

iter	Number of iterations to generate the network, must be in range [0, 20]

```
00073                                              {
00074     int defaultIter = iter;
00075     if(iter > 20 || iter < 0){
00076         defaultIter = 1;
00077     }
00078     initializeNetworkConnections();
00079     for (int i = 0; i < defaultIter; ++i) {
00080         generateAdditionalRandomConnections();
00081     }
00082 }
```

Here is the call graph for this function:



4.5.3.6 getAdjLists()

```
List< Node< List< AdjNode< Router > > > > * Network::getAdjLists ()
```

Gets the list every router's adjacency list.

Returns

List of adjacency lists.

```
00024                                                  {
00025     return &adjLists;
00026 }
```

4.5.3.7 initializeNetworkConnections()

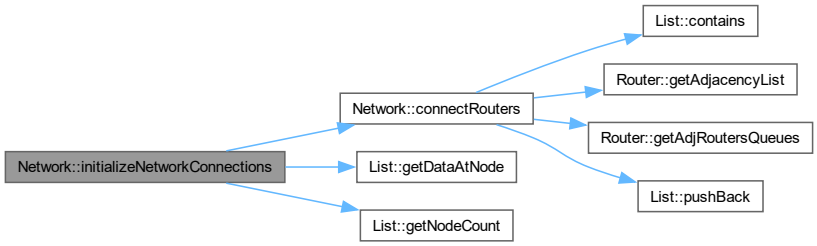
```
void Network::initializeNetworkConnections ()
```

This method constructs a random network where each existing router is randomly connected to one of the already connected routers, ensuring there are no isolated routers.

```
00042                                              {
00043     if (routers->getNodeCount() < 2) {
00044         return;
00045     }
```

```
00046     srand(time(nullptr));
00047     int routersAlreadyConnected = 1;
00048     connectRouters(routers->getDataAtNode(0), routers->getDataAtNode(1));
00049     for (int i = 2; i < routers->getNodeCount(); i++) {
00050         Router *router = routers->getDataAtNode(i);
00051         int randPos = rand() % routersAlreadyConnected;
00052         connectRouters(routers->getDataAtNode(randPos), router);
00053         routersAlreadyConnected++;
00054     }
00055 }
```

Here is the call graph for this function:



4.5.3.8 isConnected()

```
bool Network::isConnected (
            Router * router1,
            Router * router2)
```

Checks if two routers are connected.

Parameters

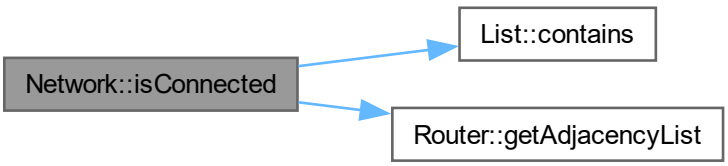
router1	Pointer to the first router
router2	Pointer to the second router

Returns

True if the routers are connected, false otherwise

```
00153                                                    {
00154     return router1->getAdjacencyList()->contains(router2);
00155 }
```

Here is the call graph for this function:



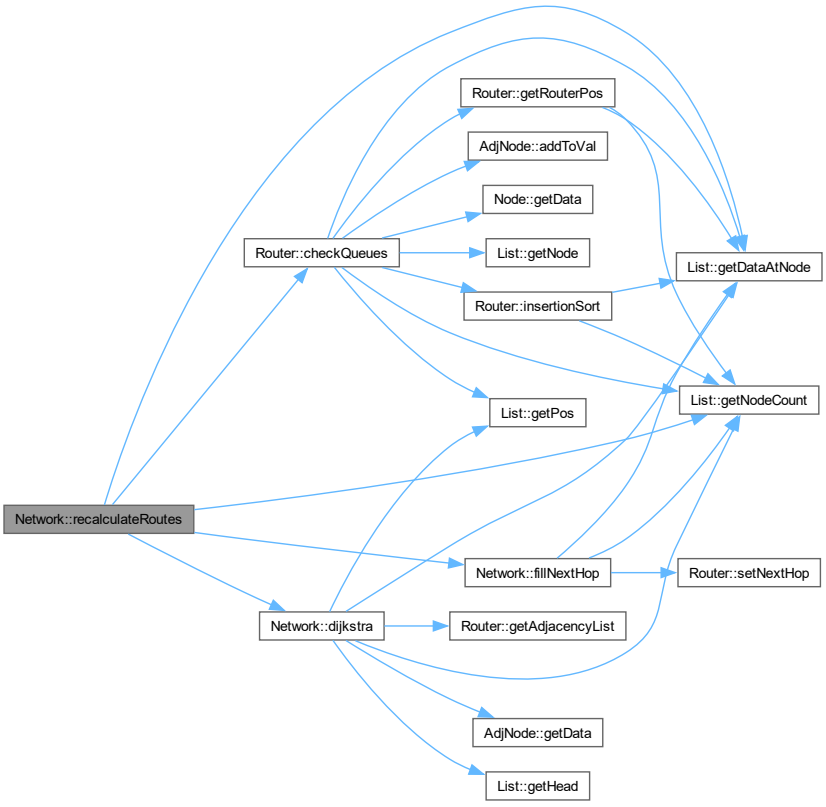
4.5.3.9 recalculateRoutes()

```
void Network::recalculateRoutes ()
```

Recalculates the shortest path from each router to all other routers in the network, updating the nextHop vector of each router accordingly.

```
00137                                {
00138     cout « "Recalculating routes..." « endl « endl;
00139     for (int i = 0; i < routers->getNodeCount(); i++) {
00140         vector<int> parents = dijkstra(i);
00141         for (int j = 0; j < parents.size(); j++) {
00142             cout « parents[j] « " ";
00143         }
00144         cout « endl;
00145         fillNextHop(i, parents);
00146     }
00147     for (int i = 0; i < routers->getNodeCount(); i++) {
00148         Router *router = routers->getDataAtNode(i);
00149         router->checkQueues();
00150     }
00151 }
```

Here is the call graph for this function:



4.5.4 Member Data Documentation

4.5.4.1 adjLists

```
List<Node<List<AdjNode<Router> > > > Network::adjLists  [private]
```

List of every router's adjacency list

4.5.4.2 routers

```
List<Node<Router> >* Network::routers  [private]
```

List of routers in the network

The documentation for this class was generated from the following files:

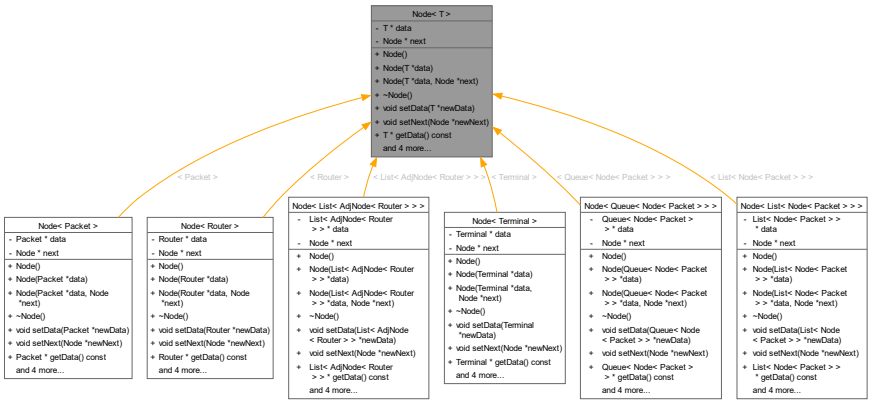
- include/Network.hpp
- src/Network.cpp

4.6 Node< T > Class Template Reference

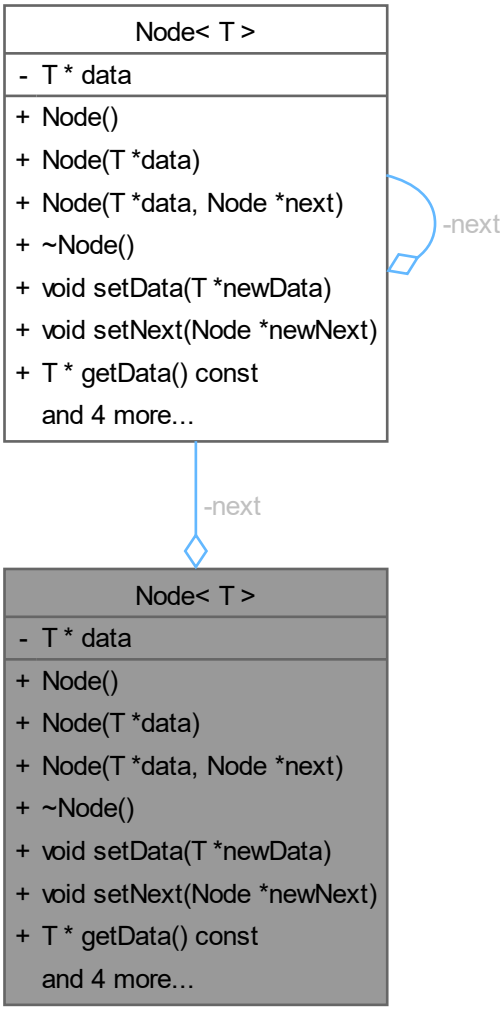
Node class to be used in a linked list, stack or queue. Stores a pointer to the data and a pointer to the next node in the structure.

```
#include <Node.hpp>
```

Inheritance diagram for Node< T >:



Collaboration diagram for Node< T >:



Public Types

- using **valType** = T

Public Member Functions

- Node ()

  Default Constructor. Initializes a new instance of the Node class with ∗data and ∗next set to nullptr.
- Node (T ∗data)

  Data Constructor. Initializes a new instance of the Node class with the provided data and a null next pointer.
- Node (T ∗data, Node ∗next)

  Data Next Constructor. Initializes a new instance of the Node class with the provided data and next node.
- ∼**Node** ()

Default Destructor.

- void setData (T *newData)

  Sets the data of the node.
- void setNext (Node *newNext)

  Sets the next node in the structure.
- T * getData () const

  Gets a pointer to the data of the node.
- Node< T > * getNext () const

  Gets the next node in the structure.
- bool hasNext ()

  Checks if there is a next node.
- bool operator== (const Node< T > &node) const

  Compares the data of two nodes.
- string toString () const

  Returns a string representation of the node's data.

Private Attributes

- T * data
- Node * next

4.6.1 Detailed Description

template<typename T>
class Node< T >

Node class to be used in a linked list, stack or queue. Stores a pointer to the data and a pointer to the next node in the structure.

Template Parameters

T	Type of the data to be stored in the node.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Node() [1/3]

```
template<typename T >
Node< T >::Node ()
```

Default Constructor. Initializes a new instance of the Node class with *data and *next set to nullptr.

```
00090         : data(nullptr), next(nullptr) {}
```

4.6.2.2 Node() [2/3]

```
template<typename T >
Node< T >::Node (
            T * data)  [explicit]
```

Data Constructor. Initializes a new instance of the Node class with the provided data and a null next pointer.

Parameters

data	Pointer to the data to be stored in the node.

```
00094         : data(data), next(nullptr) {}
```

4.6.2.3 Node() [3/3]

```
template<typename T >
Node< T >::Node (
            T * data,
            Node< T > * next)
```

Data Next Constructor. Initializes a new instance of the Node class with the provided data and next node.

Parameters

data	Pointer to the data to be stored in the node.
next	Pointer to the next node in the structure.

```
00098         : data(data), next(next) {}
```

4.6.3 Member Function Documentation

4.6.3.1 getData()

```
template<typename T >
T * Node< T >::getData () const
```

Gets a pointer to the data of the node.

Returns

Node's data.

```
00114                           {
00115     return data;
00116 }
```

4.6.3.2 getNext()

```
template<typename T >
Node< T > * Node< T >::getNext () const
```

Gets the next node in the structure.

Returns

Next node.

```
00119                                {
00120     return next;
00121 }
```

4.6.3.3 hasNext()

```
template<typename T >
bool Node< T >::hasNext ()
```

Checks if there is a next node.

Returns

True if there is a next node, false otherwise.

```
00124                       {
00125     return next != nullptr;
00126 }
```

4.6.3.4 operator==()

```
template<typename T >
bool Node< T >::operator== (
             const Node< T > & node) const
```

Compares the data of two nodes.

Parameters

node	Node to compare data with.

Returns

True if the data are equal, false otherwise.

```
00129                                                {
00130     return *data == *node.getData();
00131 }
```

Here is the call graph for this function:

4.6.3.5 setData()

```
template<typename T >
void Node< T >::setData (
             T * newData)
```

Sets the data of the node.

Parameters

newData	Pointer to the new data for the node.

```
00104                                {
00105     this->data = newData;
00106 }
```

4.6.3.6 setNext()

```
template<typename T >
void Node< T >::setNext (
             Node< T > * newNext)
```

Sets the next node in the structure.

Parameters

newNext	Pointer to the new next node.

```
00109                                    {
00110     this->next = newNext;
00111 }
```

4.6.3.7 toString()

```
template<typename T >
string Node< T >::toString () const
```

Returns a string representation of the node's data.

Returns

String representing the node's data.

```
00134                                {
00135     return data->toString();
00136 }
```

4.6.4 Member Data Documentation

4.6.4.1 data

```
template<typename T >
T* Node< T >::data  [private]
```

Pointer to the data stored in the node.

4.6.4.2 next

```
template<typename T >
Node* Node< T >::next  [private]
```

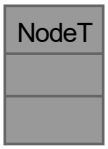
Pointer to the next node in the structure.

The documentation for this class was generated from the following file:

- include/Node.hpp

4.7 NodeT Class Reference

Collaboration diagram for NodeT:



4.7.1 Detailed Description

of the nodes that compose the list.

The documentation for this class was generated from the following file:

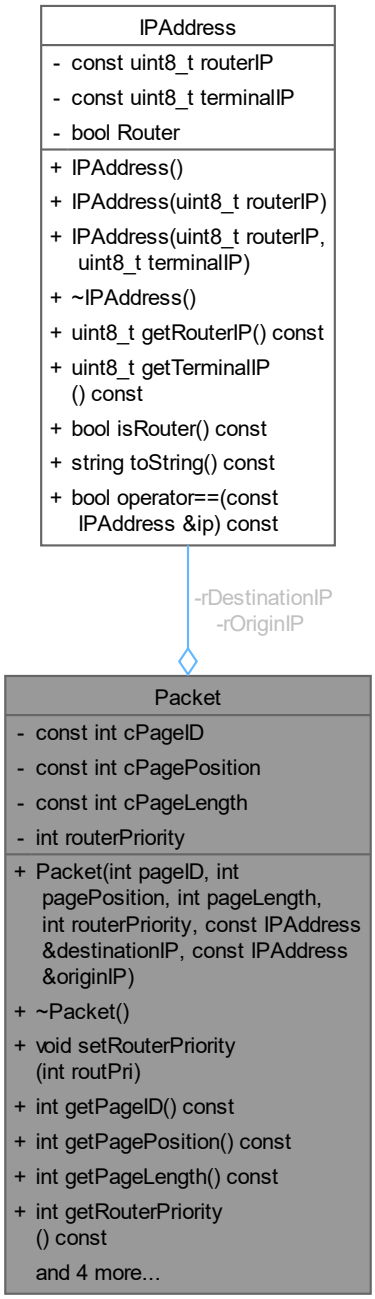
- include/List.hpp

4.8 Packet Class Reference

Represents a packet that forms part of a page. Stores all the information from the page it belongs to, including its position within the page, and a priority assigned by the router. This priority is used to determine the packet's transmission priority. Packets are the fundamental units used by routers for data transmission.

```
#include <Packet.hpp>
```

Collaboration diagram for Packet:



Public Member Functions

- Packet (int pageID, int pagePosition, int pageLength, int routerPriority, const IPAddress &destinationIP, const IPAddress &originIP)

  Constructor, called by the page constructor.
- ~**Packet** ()

  Default Destructor.

- void setRouterPriority (int routPri)

  Sets the router priority.
- int getPageID () const

  Gets the page ID.
- int getPagePosition () const

  Gets the page position.
- int getPageLength () const

  Gets the length of the page the packet belongs to.
- int getRouterPriority () const

  Gets the router priority it got assigned the last time it was received by a router.
- const IPAddress & getDestinationIP () const

  Gets the IP of it's destination terminal.
- const IPAddress & getOriginIP () const

  Gets the IP of the terminal that originated the packet.
- string toString () const

  Gets a string with it's position in the page.
- bool operator== (const Packet &) const

  Compares two packets.

Private Attributes

- const int cPageID
- const int cPagePosition
- const int cPageLength
- int routerPriority
- const IPAddress & rDestinationIP
- const IPAddress & rOriginIP

4.8.1 Detailed Description

Represents a packet that forms part of a page. Stores all the information from the page it belongs to, including its position within the page, and a priority assigned by the router. This priority is used to determine the packet's transmission priority. Packets are the fundamental units used by routers for data transmission.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Packet()

```
Packet::Packet (
            int pageID,
            int pagePosition,
            int pageLength,
            int routerPriority,
            const IPAddress & destinationIP,
            const IPAddress & originIP)
```

Constructor, called by the page constructor.

Parameters

pageID	ID of the page that the packet belongs to.
pagePosition	Position of the packet in the page.
routerPriority	Priority of the packet in the router.
destination↩ IP	Reference to the destination terminal IP.
originIP	Reference to the origin terminal IP.

```
00004         : cPageID(pageID),
00005           cPagePosition(pagePosition),
00006           routerPriority(routerPriority),
00007           rDestinationIP(destinationIP),
00008           rOriginIP(originIP),
00009           cPageLength(pageLength) {}
```

4.8.3 Member Function Documentation

4.8.3.1 getDestinationIP()

```
const IPAddress & Packet::getDestinationIP () const
```

Gets the IP of it's destination terminal.

Returns

Destination terminal IP.

```
00033                                                {
00034     return rDestinationIP;
00035 }
```

4.8.3.2 getOriginIP()

```
const IPAddress & Packet::getOriginIP () const
```

Gets the IP of the terminal that originated the packet.

Returns

Origin terminal IP.

```
00037                                           {
00038     return rOriginIP;
00039 }
```

4.8.3.3 getPageID()

```
int Packet::getPageID () const
```

Gets the page ID.

Returns

Page ID.

```
00017                           {
00018     return cPageID;
00019 }
```

4.8.3.4 getPageLength()

```
int Packet::getPageLength () const
```

Gets the length of the page the packet belongs to.

Returns

Page length.

```
00025                                {
00026     return cPageLength;
00027 }
```

4.8.3.5 getPagePosition()

```
int Packet::getPagePosition () const
```

Gets the page position.

Returns

Page position.

```
00021                                  {
00022     return cPagePosition;
00023 }
```

4.8.3.6 getRouterPriority()

```
int Packet::getRouterPriority () const
```

Gets the router priority it got assigned the last time it was received by a router.

Returns

Router priority

```
00029                                    {
00030     return routerPriority;
00031 }
```

4.8.3.7 operator==()

```
bool Packet::operator== (
            const Packet & packet) const
```

Compares two packets.

Parameters

packet	Packet to compare with.

Returns

True if both packets have the same cPageID and cPagePosition, false otherwise.

```
00048                                                 {
00049     return cPageID == packet.cPageID && cPagePosition == packet.cPagePosition;
00050 }
```

4.8.3.8 setRouterPriority()

```
void Packet::setRouterPriority (
            int routPri)
```

Sets the router priority.

Parameters

routPri	Router priority.

```
00013                                           {
00014     routerPriority = routPri;
00015 }
```

4.8.3.9 toString()

```
string Packet::toString () const
```

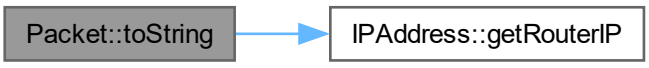
Gets a string with it's position in the page.

Returns

"Packet xx".

```
00041                                 {
00042     ostringstream oss;
00043     int a = rDestinationIP.getRouterIP();
00044     oss « a « std::setw(9) « std::setfill('0') « cPageID;
00045     return oss.str() + "-" + to_string(cPagePosition);
00046 }
```

Here is the call graph for this function:



4.8.4 Member Data Documentation

4.8.4.1 cPageID

```
const int Packet::cPageID  [private]
```

ID of the page that the packet belongs to.

4.8.4.2 cPageLength

```
const int Packet::cPageLength  [private]
```

Length of the page.

4.8.4.3 cPagePosition

```
const int Packet::cPagePosition  [private]
```

Position of the packet in the page.

4.8.4.4 rDestinationIP

```
const IPAddress& Packet::rDestinationIP  [private]
```

Reference to the destination terminal IP.

4.8.4.5 rOriginIP

```
const IPAddress& Packet::rOriginIP  [private]
```

Reference to the origin terminal IP.

4.8.4.6 routerPriority

```
int Packet::routerPriority  [private]
```

Priority of the packet in the router.

The documentation for this class was generated from the following files:

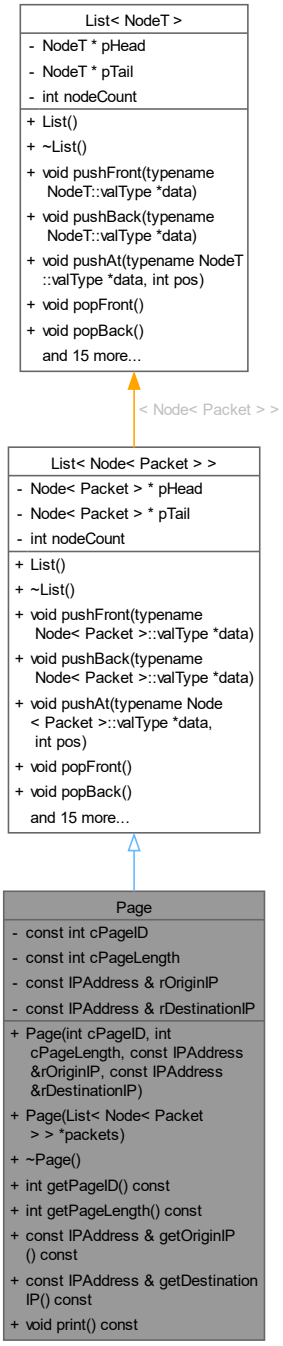
- include/Packet.hpp
- src/Packet.cpp

4.9 Page Class Reference

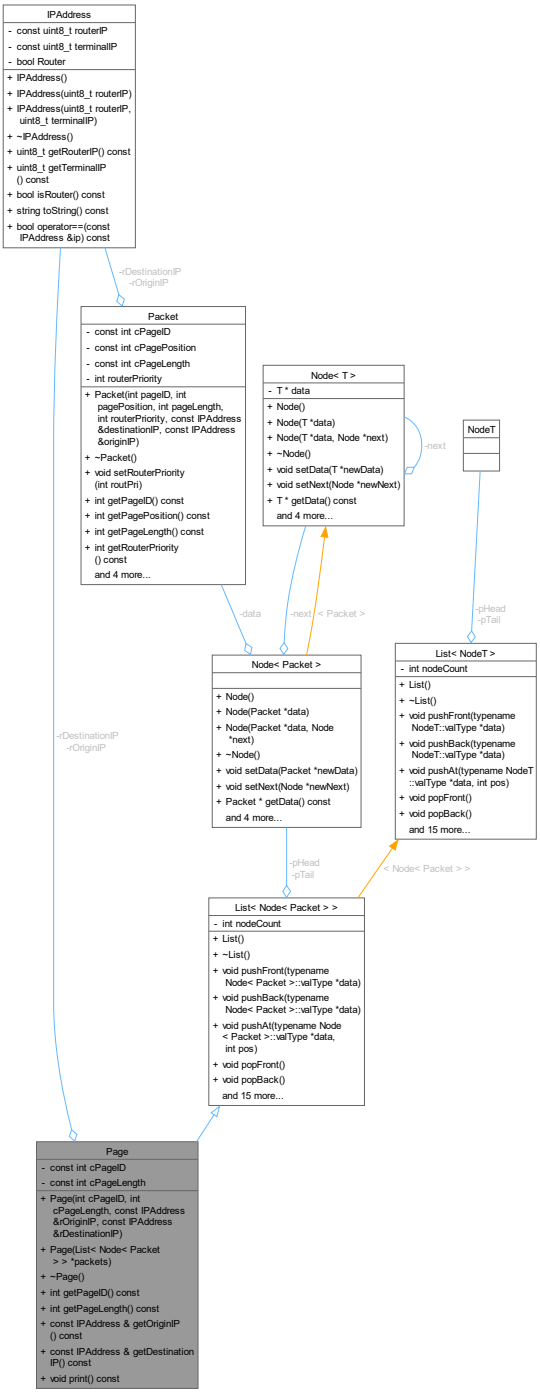
Class to represent a page made of packets, which are created when it's instantiated. Inherits from List<Node<Packet>>. Stores a reference to origin and destination IPs. Also has a page ID for its packets to be identified and page length for the amount of packets it holds.

```
#include <Page.hpp>
```

Inheritance diagram for Page:



Collaboration diagram for Page:



Public Member Functions

- Page (int cPageID, int cPageLength, const IPAddress &rOriginIP, const IPAddress &rDestinationIP)

  Constructor for the Page class. Initializes a new Page instance with specified parameters and populates it with Packet objects.
- Page (List< Node< Packet > > *packets)

  Constructor. Initializes a new Page instance with a list of packets.

- ~**Page** ()

  Default destructor.
- int getPageID () const

  Retrieves the page ID.
- int getPageLength () const

  Retrieves the length of the page in terms of the number of packets.
- const IPAddress & getOriginIP () const

  Retrieves the origin IP address of the page.
- const IPAddress & getDestinationIP () const

  Retrieves the destination IP address of the page.
- void print () const

  Prints the details of the page, including its ID, packets, and length.

Public Member Functions inherited from List< Node< Packet > >

- List ()

  Default constructor.
- ~List ()

  Destructor for the List class. Iterates through the list and deletes each node to free up memory. Does not delete the data pointed to by the nodes.
- void pushFront (typename Node< Packet >::valType *data)

  Adds a node with the specified data at the beginning of the list. The data type must match the type that NodeT can store.
- void pushBack (typename Node< Packet >::valType *data)

  Adds a node with the specified data at the end of the list. The data type must match the type that NodeT can store.
- void pushAt (typename Node< Packet >::valType *data, int pos)

  Adds a node with the specified data at the given position. The existing node at this position and all subsequent nodes are shifted one position to the end of the list. The data type must match the type that NodeT can store.
- void popFront ()

  Removes the first node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.
- void popBack ()

  Removes the last node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.
- void popAt (int)

  Removes a node at the specified position from the list. The nodes after the removed node are shifted one position to the beginning of the list. If the position is invalid or the list is empty, it prints a message and does nothing. The NodeT object at the position is deleted, but the data pointed by the node is not deleted.
- void setDataAtNode (int pos, typename Node< Packet >::valType *newData)

  Sets the data of the node at the specified position. If the position is invalid or the list is empty, does nothing.
- void swapNodesAt (int m, int n)

  Swaps the data of two nodes at the specified positions. If either position is invalid, it prints a message and does nothing. If the positions are the same, it does nothing.
- Node< Packet > * getHead () const

  Gets a pointer to the first node in the list. If the list is empty, gets nullptr.
- Node< Packet > * getTail () const

  Gets a pointer to the last node in the list. If the list is empty, gets nullptr.
- Node< Packet > * getNode (int pos) const

  Gets a pointer to the node at the specified position. If the position is invalid or the list is empty, gets nullptr.
- Node< Packet >::valType * getHeadData () const

  Returns a pointer to the data of the first node in the list. If the list is empty, returns nullptr.
- Node< Packet >::valType * getTailData () const

Returns a pointer to the data of the last node in the list. If the list is empty, returns nullptr.

- Node< Packet >::valType * getDataAtNode (int pos) const

  Returns a pointer to the data of the node at the specified position. If the position is invalid or the list is empty, returns nullptr.

- int getPos (typename Node< Packet >::valType *data) const

  Returns the position of the first node that contains the specified data. If the data is not found, returns -1.

- int getNodeCount () const

  Returns the number of nodes in the list.

- bool isEmpty () const

  Checks if the list is empty.

- bool contains (typename Node< Packet >::valType *data) const

  Checks if the list contains a node with the specified data.

- virtual void printList () const

  Prints the data of all nodes in the list to the standard output.

- virtual string toString () const

  Returns a string representation of the list, containing the data of all nodes.

Private Attributes

- const int cPageID
- const int cPageLength
- const IPAddress & rOriginIP
- const IPAddress & rDestinationIP

4.9.1 Detailed Description

Class to represent a page made of packets, which are created when it's instantiated. Inherits from List<Node<Packet>>. Stores a reference to origin and destination IPs. Also has a page ID for its packets to be identified and page length for the amount of packets it holds.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 Page() [1/2]

```
Page::Page (
            int cPageID,
            int cPageLength,
            const IPAddress & rOriginIP,
            const IPAddress & rDestinationIP)
```

Constructor for the Page class. Initializes a new Page instance with specified parameters and populates it with Packet objects.

Parameters

cPageID	Unique identifier for the page.
cPageLength	The number of packets the page will contain.
rOriginIP	The origin IP address for the page.
r↩ DestinationIP	The destination IP address for the page.

```
00004         : cPageID(cPageID), cPageLength(cPageLength), rOriginIP(rOriginIP),
      rDestinationIP(rDestinationIP) {
00005     for (int i = 0; i < cPageLength; i++){          // Instantiates a new packet for each position in
      the page.
00006         auto *new_packet = new Packet(cPageID, i, cPageLength, 0, rDestinationIP, rOriginIP);
00007         this->pushBack(new_packet);
00008     }
00009 }
```

Here is the call graph for this function:

4.9.2.2 Page() [2/2]

```
Page::Page (
            List< Node< Packet > > * packets)  [explicit]
```

Constructor. Initializes a new Page instance with a list of packets.

Parameters

packets	Pointer to a list of packets to be used to build the page.

```
00012         : cPageID(packets->getHeadData()->getPageID()),
00013           cPageLength(packets->getNodeCount()),
00014           rOriginIP(packets->getHeadData()->getOriginIP()),
00015           rDestinationIP(packets->getHeadData()->getDestinationIP()) {
00016     Packet *packet;
00017     for (int i = 0; i < packets->getNodeCount(); i++){      // Adds a node for each of the original
      packets.
00018         packet = packets->getDataAtNode(i);
00019         this->pushBack(packet);
00020     }
00021     delete packets;     // Deletes the original list of packets, but not the packets themselves.
00022 }
```

Here is the call graph for this function:

4.9.3 Member Function Documentation

4.9.3.1 getDestinationIP()

```
const IPAddress & Page::getDestinationIP () const
```

Retrieves the destination IP address of the page.

Returns

A constant reference to the destination IP address.

```
00038                                              {
00039     return rDestinationIP;
00040 }
```

4.9.3.2 getOriginIP()

```
const IPAddress & Page::getOriginIP () const
```

Retrieves the origin IP address of the page.

Returns

A constant reference to the origin IP address.

```
00034                                          {
00035     return rOriginIP;
00036 }
```

4.9.3.3 getPageID()

```
int Page::getPageID () const
```

Retrieves the page ID.

Returns

The unique identifier of the page.

```
00026                          {
00027     return cPageID;
00028 }
```

4.9.3.4 getPageLength()

```
int Page::getPageLength () const
```

Retrieves the length of the page in terms of the number of packets.

Returns

The number of packets in the page.

```
00030                               {
00031     return cPageLength;
00032 }
```

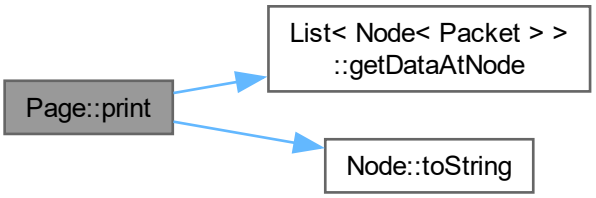
4.9.3.5 print()

```
void Page::print () const
```

Prints the details of the page, including its ID, packets, and length.

```
00042                       {
00043     cout « "Page ID: " « cPageID « endl;
00044     for (int i = 0; i < cPageLength; i++) {
00045         cout « this->getDataAtNode(i)->toString() « endl;
00046     }
00047     cout « "Page length: " « cPageLength « endl;
00048     cout « endl;
00049 }
```

Here is the call graph for this function:



4.9.4 Member Data Documentation

4.9.4.1 cPageID

```
const int Page::cPageID  [private]
```

originIP+numberOfPageSent

4.9.4.2 cPageLength

```
const int Page::cPageLength  [private]
```

Page length in packets

4.9.4.3 rDestinationIP

```
const IPAddress& Page::rDestinationIP  [private]
```

Reference to the destination Terminal IP

4.9.4.4 rOriginIP

```
const IPAddress& Page::rOriginIP  [private]
```

Reference to the sender Terminal IP

The documentation for this class was generated from the following files:

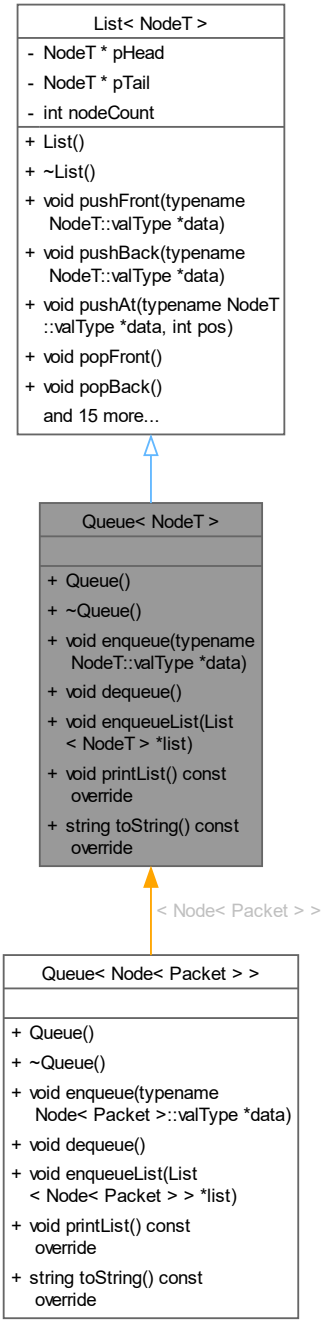
- include/Page.hpp
- src/Page.cpp

4.10 Queue< NodeT > Class Template Reference

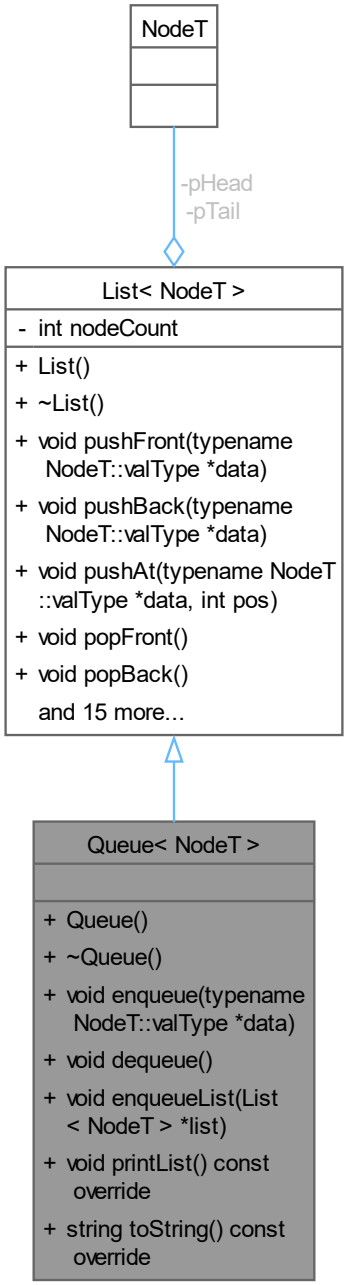
Composed of generic nodes that store a pointer to the data of the type that the node can store. Individual nodes can be enqueued as well as lists of nodes. Only individual nodes can be dequeued.

```
#include <Queue.hpp>
```

Inheritance diagram for Queue< NodeT >:



Collaboration diagram for Queue< NodeT >:



Public Member Functions

- **Queue** ()

  Default constructor.
- ~**Queue** ()

  Default destructor.
- void enqueue (typename NodeT::valType ∗data)

Enqueues a new node with the specified data The data type must match the type that NodeT can store.

- void dequeue ()

  Dequeues the next element from the queue. If the queue is empty, it prints a message and does nothing.

- void enqueueList (List< NodeT > *list)

  Enqueues all the data from a list into the queue, maintaining the order. If the list is empty, it does nothing. Does not modify the original list.

- void printList () const override

  Prints the contents of the queue from front to back. This method overrides the printList method from the List class.

- string toString () const override

  Returns a string representation of the queue.

Public Member Functions inherited from List< NodeT >

- List ()

  Default constructor.

- ~List ()

  Destructor for the List class. Iterates through the list and deletes each node to free up memory. Does not delete the data pointed to by the nodes.

- void pushFront (typename NodeT::valType *data)

  Adds a node with the specified data at the beginning of the list. The data type must match the type that NodeT can store.

- void pushBack (typename NodeT::valType *data)

  Adds a node with the specified data at the end of the list. The data type must match the type that NodeT can store.

- void pushAt (typename NodeT::valType *data, int pos)

  Adds a node with the specified data at the given position. The existing node at this position and all subsequent nodes are shifted one position to the end of the list. The data type must match the type that NodeT can store.

- void popFront ()

  Removes the first node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.

- void popBack ()

  Removes the last node from the list. If the list is empty, it prints a message and does nothing. The NodeT object is deleted, but the data pointed to by the node is not deleted.

- void popAt (int)

  Removes a node at the specified position from the list. The nodes after the removed node are shifted one position to the beginning of the list. If the position is invalid or the list is empty, it prints a message and does nothing. The NodeT object at the position is deleted, but the data pointed by the node is not deleted.

- void setDataAtNode (int pos, typename NodeT::valType *newData)

  Sets the data of the node at the specified position. If the position is invalid or the list is empty, does nothing.

- void swapNodesAt (int m, int n)

  Swaps the data of two nodes at the specified positions. If either position is invalid, it prints a message and does nothing. If the positions are the same, it does nothing.

- NodeT * getHead () const

  Gets a pointer to the first node in the list. If the list is empty, gets nullptr.

- NodeT * getTail () const

  Gets a pointer to the last node in the list. If the list is empty, gets nullptr.

- NodeT * getNode (int pos) const

  Gets a pointer to the node at the specified position. If the position is invalid or the list is empty, gets nullptr.

- NodeT::valType * getHeadData () const

  Returns a pointer to the data of the first node in the list. If the list is empty, returns nullptr.

- NodeT::valType * getTailData () const

  Returns a pointer to the data of the last node in the list. If the list is empty, returns nullptr.

- NodeT::valType * getDataAtNode (int pos) const

Returns a pointer to the data of the node at the specified position. If the position is invalid or the list is empty, returns nullptr.

- int getPos (typename NodeT::valType *data) const

  Returns the position of the first node that contains the specified data. If the data is not found, returns -1.
- int getNodeCount () const

  Returns the number of nodes in the list.
- bool isEmpty () const

  Checks if the list is empty.
- bool contains (typename NodeT::valType *data) const

  Checks if the list contains a node with the specified data.

4.10.1 Detailed Description

template<typename NodeT>
class Queue< NodeT >

Composed of generic nodes that store a pointer to the data of the type that the node can store. Individual nodes can be enqueued as well as lists of nodes. Only individual nodes can be dequeued.

Template Parameters

NodeT	Type of the nodes to compose the queue.

4.10.2 Member Function Documentation

4.10.2.1 dequeue()

```
template<class NodeT >
void Queue< NodeT >::dequeue ()
```

Dequeues the next element from the queue. If the queue is empty, it prints a message and does nothing.

```
00072                                {
00073     if (this->getNodeCount() == 0) {
00074         cout « "Can't dequeue, queue is empty" « endl;
00075         return;
00076     }
00077     this->popFront();
00078 }
```

4.10.2.2 enqueue()

```
template<class NodeT >
void Queue< NodeT >::enqueue (
            typename NodeT::valType * data)
```

Enqueues a new node with the specified data The data type must match the type that NodeT can store.

Parameters

data	Pointer to the data to be stored in the new node.

```
00067                                                    {
00068     this->pushBack(data);
00069 }
```

4.10.2.3 enqueueList()

```
template<typename NodeT >
void Queue< NodeT >::enqueueList (
             List< NodeT > * list)
```

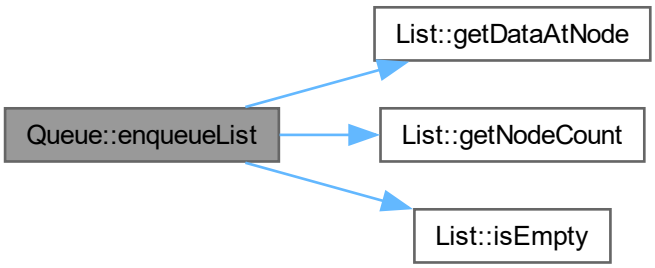
Enqueues all the data from a list into the queue, maintaining the order. If the list is empty, it does nothing. Does not modify the original list.

Parameters

list	Pointer to the list to be enqueued.

```
00081                                                {
00082     if (list->isEmpty()) {
00083         return;
00084     }
00085     for (int i = 0; i < list->getNodeCount(); i++) {
00086         this->pushBack(list->getDataAtNode(i));
00087     }
00088 }
```

Here is the call graph for this function:



4.10.2.4 printList()

```
template<typename NodeT >
void Queue< NodeT >::printList () const  [override], [virtual]
```

Prints the contents of the queue from front to back. This method overrides the printList method from the List class.

Reimplemented from List< NodeT >.

```
00091                                    {
00092     cout « "Inicio de cola" « endl;
00093     NodeT *aux = this->getHead();
00094
00095     while (aux) {
00096         if (aux->getData() != nullptr) {
00097             cout « aux->getData()->toString() « endl;
00098         }
00099         aux = aux->getNext();
00100     }
00101
00102     cout « "Fin de cola" « endl;
00103 }
```

4.10.2.5 toString()

```
template<typename NodeT >
string Queue< NodeT >::toString () const  [override], [virtual]
```

Returns a string representation of the queue.

Returns

A string representation of the queue.

Reimplemented from List< NodeT >.

```
00106                                   {
00107     return "holi soy una cola";
00108     // TODO: implementar bien el toString y el printList, testearlos
00109 }
```

The documentation for this class was generated from the following file:

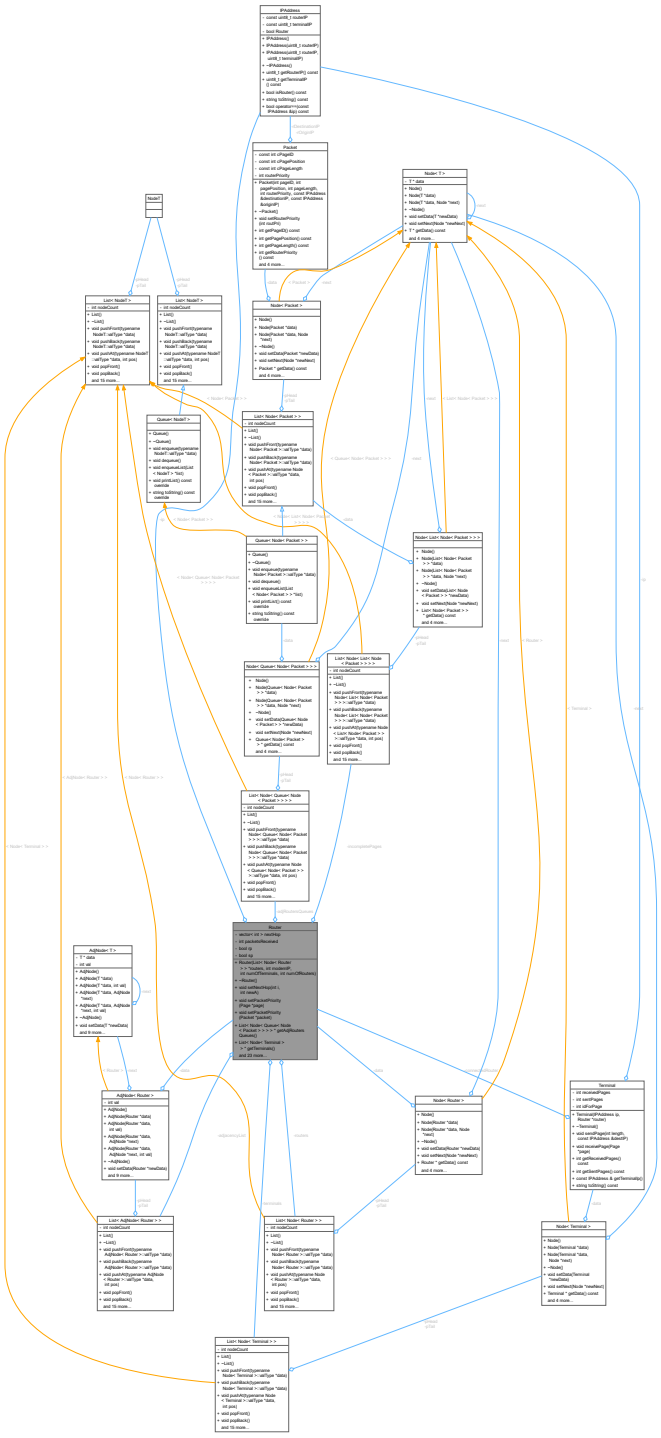
- include/Queue.hpp

4.11 Router Class Reference

Represents a router in a network. The Router class manages the routing of packets and pages between terminals and other routers. It maintains lists of connected terminals, adjacent routers, and queues for packet transmission. The router can receive and send pages, disassemble pages into packets, and manage packet priorities. It also provides various methods to print router information and manage routing paths.

```
#include <Router.hpp>
```

Collaboration diagram for Router:



Public Member Functions

- Router (List< Node< Router > > *routers, int modemIP, int numOfTerminals, int numOfRouters)

  Constructor. Creates a Router object with a specified IP address, initializes all the lists as empty, creates a number of terminals.

- ~Router ()

  Destructor. Deletes all the terminals, queues and incomplete pages with their respective packets.

- void setNextHop (int i, int newA)

  Sets the position of the next router to send a packet to in the position of the destination router.
- void setPacketPriority (Page *page)

  Sets the priority of the packets in a page, based on the number of packets received.
- void setPacketPriority (Packet *packet)

  Sets the priority of the packet based on the number of packets received.
- List< Node< Queue< Node< Packet > > > > * getAdjRoutersQueues ()

  Gets the list of queues of packets to send to neighbor routers.
- List< Node< Terminal > > * getTerminals ()

  Gets the list of terminals connected to the router.
- List< AdjNode< Router > > * getAdjacencyList ()

  Gets the adjacency list of the router.
- int getPacketsReceived () const

  Gets the number of packets received by the router.
- int getRouterPos (uint8_t destRouterIP)

  Gets the position of the router in the list of routers using the IP address.
- void addHopDest ()

  Increments the size of the nextHop vector.
- void receivePage (Page *page)

  Receives a page, if the destination is a terminal connected to the router, it sends the page to the terminal. If the destination is another router, disassembles the page into packets, and enqueues them in the corresponding queue.
- void sendPage (int termPos, Page *page)

  Sends a page to a terminal connected to the router.
- void receivePacket (Packet *packet)

  Receives a packet, sets the router priority, and enqueues the packet in the corresponding queue. If the packet is for a terminal connected to the router, it adds the packet to the waiting list.
- void packetForTerminal (Packet *packet)

  Stores the package in a waiting list with the same page ID. If it does not exist, a new list is created. If the page is completed with this package, the page is sent.
- Page * buildPage (List< Node< Packet > > *packets)

  Builds a page with the packets in the list. The list gets deleted.
- bool isPageComplete (int i)

  Checks if a page is complete.
- void sendFromQueues (int bandWith)

  Sends a number of packets equal to the bandwidth from each queue to its respective router.
- void checkQueues ()

  Checks each queue and packet to ensure they are in the correct queue after the path has been recalculated. If a packet is not in the correct queue, it moves the packet to the end of the appropriate queue. Finally, performs an insertion sort on each queue to maintain order.
- void insertionSort ()

  Sorts the packets in all the queues using insertion sort.
- const IPAddress & getIP ()

  Gets the IP address of the router.
- void printRouterName ()

  Prints the name of the router in a "Router + binary IP" format.
- void printActivity ()

  Prints if the router received or sent a page in the last iteration.
- void printAdjacencyList ()

  Prints the adjacency list of the router.
- void printTerminals ()

  Prints the terminals connected to the router.
- void printQueues ()

Prints the queues with their packets, if there are no packets, it prints "Empty".

- void printIncompletePages ()

  Prints the incomplete pages with their packets, if any.

- void printRouterInfo ()

  Prints the name, the activity, the adjacency list, the terminals, the queues, and the incomplete pages of the router.

- string toString ()

  Generates a string representation of the router, including its IP address in decimal format.

- bool operator== (const Router &router)

  Overloads the equality operator to compare two routers.

Private Attributes

- const IPAddress ip
- List< Node< Router > > * routers
- List< Node< Terminal > > terminals
- List< Node< Queue< Node< Packet > > > > adjRoutersQueues
- List< AdjNode< Router > > adjacencyList
- List< Node< List< Node< Packet > > > > incompletePages
- vector< int > nextHop
- int **packetsReceived** = 0
- bool rp = false
- bool sp = false

4.11.1 Detailed Description

Represents a router in a network. The Router class manages the routing of packets and pages between terminals and other routers. It maintains lists of connected terminals, adjacent routers, and queues for packet transmission. The router can receive and send pages, disassemble pages into packets, and manage packet priorities. It also provides various methods to print router information and manage routing paths.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 Router()

```
Router::Router (
            List< Node< Router > > * routers,
            int modemIP,
            int numOfTerminals,
            int numOfRouters)
```

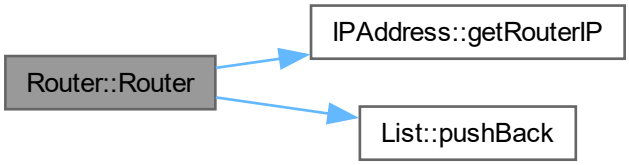
Constructor. Creates a Router object with a specified IP address, initializes all the lists as empty, creates a number of terminals.

Parameters

routers	Pointer to the list of routers.
modemIP	IP address of the router.
numOfTerminals	Number of terminals connected to the router.
numOfRouters	Number of routers in the network.

```
00004     : ip(modemIP),
00005       routers(routers),
00006       terminals(),
00007       adjRoutersQueues(),
00008       adjacencyList(),
00009       incompletePages(),
00010       packetsReceived(0){
00011     for (int i = 0; i < numOfTerminals; i++) {
00012         terminals.pushBack(new Terminal(IPAddress(ip.getRouterIP(), i), this));
00013     }
00014     for (int i = 0; i < numOfRouters; i++) {
00015         nextHop.push_back(i);
00016     }
00017 }
```

Here is the call graph for this function:



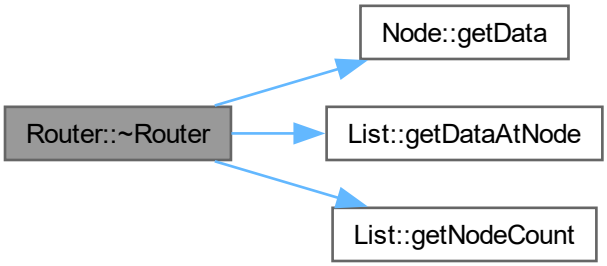
4.11.2.2 ~Router()

```
Router::~Router ()
```

Destructor. Deletes all the terminals, queues and incomplete pages with their respective packets.

```
00019                 {
00020     for (int i = 0; i < terminals.getNodeCount(); ++i) {    //Delete terminals
00021         delete terminals.getDataAtNode(i);
00022     }
00023     for (int i = 0; i < adjRoutersQueues.getNodeCount(); ++i) {     //Delete queues and every packet
      in them
00024         auto *queue = adjRoutersQueues.getDataAtNode(i);
00025         auto *node = queue->getHead();
00026         while (node) {
00027             delete node->getData();
00028             node = node->getNext();
00029         }
00030         delete queue;
00031     }
00032     for (int i = 0; i < incompletePages.getNodeCount(); ++i) {      //Delete lists and every packet in
      them
00033         auto *list = incompletePages.getDataAtNode(i);
00034         auto *node = list->getHead();
00035         while (node) {
00036             delete node->getData();
00037             node = node->getNext();
00038         }
00039         delete list;
00040     }
00041 }
```

Here is the call graph for this function:



4.11.3 Member Function Documentation

4.11.3.1 addHopDest()

```
void Router::addHopDest ()
```

Increments the size of the nextHop vector.

```
00086                          {
00087     nextHop.push_back(0);
00088 }
```

4.11.3.2 buildPage()

```
Page * Router::buildPage (
            List< Node< Packet > > * packets)
```

Builds a page with the packets in the list. The list gets deleted.

Parameters

packets	Pointer to the list of packets to build the page.

Returns

Pointer to the page built.

```
00152                                                  {
00153     Page *page = new Page(packets);
00154         return page;
00155 }
```

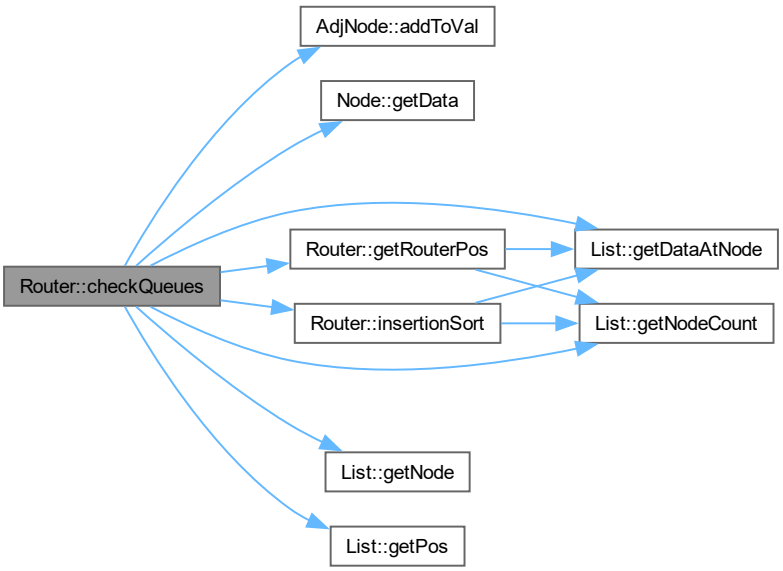
4.11.3.3 checkQueues()

```
void Router::checkQueues ()
```

Checks each queue and packet to ensure they are in the correct queue after the path has been recalculated. If a packet is not in the correct queue, it moves the packet to the end of the appropriate queue. Finally, performs an insertion sort on each queue to maintain order.

```
00179                           {
00180     for (int i = 0; i < adjRoutersQueues.getNodeCount(); ++i) {             // Iterates through all
    queues.
00181         auto *currQueue = adjRoutersQueues.getDataAtNode(i);
00182         auto *node = currQueue->getHead();
00183         while (node) {
00184             auto *packet = node->getData();
00185             int routerPos = getRouterPos(packet->getDestinationIP().getRouterIP());
00186             auto *nextRouter = routers->getDataAtNode(nextHop[routerPos]);
00187             int adjPos = adjacencyList.getPos(nextRouter);
00188             if (adjPos == i) {
00189                 node = node->getNext();
00190                 continue;
00191             } else {
00192                 adjRoutersQueues.getDataAtNode(adjPos)->enqueue(packet);
00193                 adjacencyList.getNode(adjPos)->addToVal(1);
00194                 auto *auxNode = node->getNext();
00195                 currQueue->popAt(currQueue->getPos(packet));
00196                 adjacencyList.getNode(i)->addToVal(-1);
00197                 node = auxNode;
00198             }
00199         }
00200     }
00201     insertionSort();
00202 }
```

Here is the call graph for this function:



4.11.3.4 getAdjacencyList()

```
List< AdjNode< Router > > * Router::getAdjacencyList ()
```

Gets the adjacency list of the router.

Returns

Adjacency list of the router.

```
00067                                           {
00068     return &adjacencyList;
00069 }
```

4.11.3.5 getAdjRoutersQueues()

```
List< Node< Queue< Node< Packet > > > > * Router::getAdjRoutersQueues ()
```

Gets the list of queues of packets to send to neighbor routers.

Returns

Pointer to the list of queues of packets.

```
00059                                                       {
00060     return &adjRoutersQueues;
00061 }
```

4.11.3.6 getIP()

```
const IPAddress & Router::getIP ()
```

Gets the IP address of the router.

Returns

IP address of the router.

```
00225                                {
00226     return ip;
00227 }
```

4.11.3.7 getPacketsReceived()

```
int Router::getPacketsReceived () const
```

Gets the number of packets received by the router.

Returns

Number of packets received.

```
00071                                      {
00072     return packetsReceived;
00073 }
```

4.11.3.8 getRouterPos()

```
int Router::getRouterPos (
            uint8_t destRouterIP)
```

Gets the position of the router in the list of routers using the IP address.

Parameters

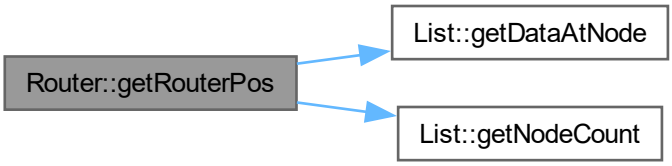
destRouterIP	IP of the destination router.

Returns

Position of the router in the list of routers.

```
00075                                                {
00076     int routerPos;
00077     for (routerPos = 0; routerPos < routers->getNodeCount(); routerPos++) {
00078         uint8_t b = routers->getDataAtNode(routerPos)->getIP().getRouterIP();
00079         if(b == destRouterIP) {
00080             break;
00081         }
00082     }
00083     return routerPos;
00084 }
```

Here is the call graph for this function:



4.11.3.9 getTerminals()

```
List< Node< Terminal > > * Router::getTerminals ()
```

Gets the list of terminals connected to the router.

Returns

List of terminals.

```
00063                                               {
00064     return &terminals;
00065 }
```

4.11.3.10 insertionSort()

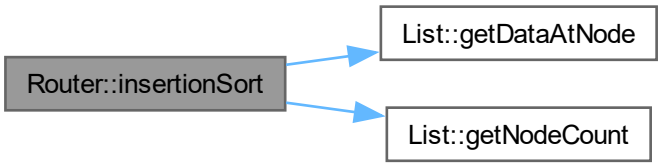
```
void Router::insertionSort ()
```

Sorts the packets in all the queues using insertion sort.

```
00204                                {
00205     for (int h = 0; h < adjRoutersQueues.getNodeCount(); ++h) {
00206         auto *queue = adjRoutersQueues.getDataAtNode(h);
00207         if (queue->getNodeCount() < 2) {
00208             continue;
```

```
00209         }
00210         for (int i = 1; i < queue->getNodeCount(); ++i) {     // Iterates through all queues.
00211             auto packet = queue->getDataAtNode(i);
00212             int currPriority = packet->getRouterPriority();
00213             int j = i - 1;
00214             while (j >= 0 && queue->getDataAtNode(j)->getRouterPriority() > currPriority) {
00215                 j--;
00216             }
00217             if (j != i - 1) {
00218                 queue->popAt(i);
00219                 queue->pushAt(packet, j + 1);
00220             }
00221         }
00222     }
00223 }
```

Here is the call graph for this function:



4.11.3.11 isPageComplete()

```
bool Router::isPageComplete (
             int i)
```

Checks if a page is complete.

Parameters

i	Position of the page in the list of incomplete pages.

Returns

True if the page is complete, false otherwise.

```
00157                                {
00158     int packetsInList = incompletePages.getDataAtNode(i)->getNodeCount();
00159     int pageLength = incompletePages.getDataAtNode(i)->getHeadData()->getPageLength();
00160     return packetsInList == pageLength;
00161 }
```

Here is the call graph for this function:

4.11.3.12 operator==()

```
bool Router::operator== (
            const Router & router)
```

Overloads the equality operator to compare two routers.

Parameters

router	Router to compare with.

Returns

True if the routers are equal, false otherwise.

```
00306                                           {
00307     return ip == router.ip;
00308 }
```

4.11.3.13 packetForTerminal()

```
void Router::packetForTerminal (
            Packet * packet)
```

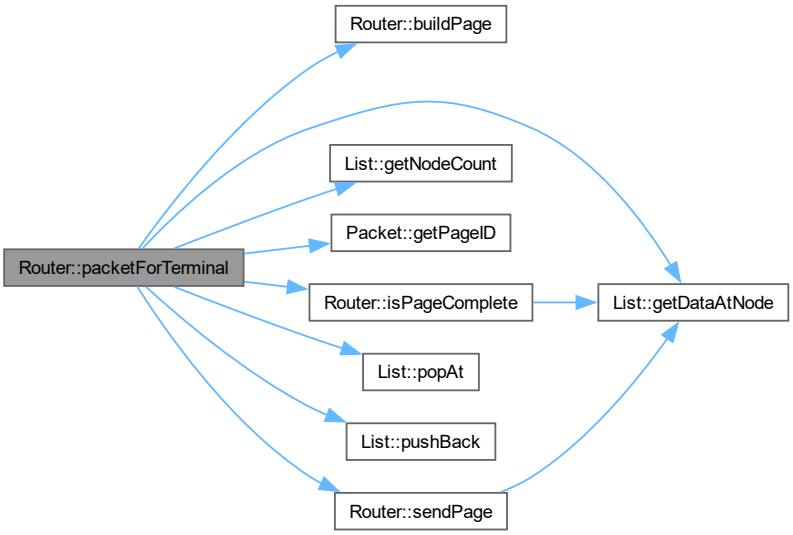
Stores the package in a waiting list with the same page ID. If it does not exist, a new list is created. If the page is completed with this package, the page is sent.

Parameters

packet	Pointer to the packet being sent.

```
00134                                               {
00135     for (int i = 0; i < incompletePages.getNodeCount(); ++i) {
00136         auto *auxPacket = incompletePages.getDataAtNode(i)->getHeadData();
00137         if (auxPacket->getPageID() == packet->getPageID()) {
00138             incompletePages.getDataAtNode(i)->pushBack(packet);
00139             if (isPageComplete(i)) {
00140                 auto *page = buildPage(incompletePages.getDataAtNode(i));
00141                 sendPage(auxPacket->getDestinationIP().getTerminalIP(), page);
00142                 incompletePages.popAt(i);
00143             }
00144             return;
00145         }
00146     }
00147     auto *newList = new List<Node<Packet>>();
00148     newList->pushBack(packet);
00149     incompletePages.pushBack(newList);
00150 }
```

Here is the call graph for this function:



4.11.3.14 printActivity()

```
void Router::printActivity ()
```

Prints if the router received or sent a page in the last iteration.

```
00233                               {
00234     if (rp) {
00235         cout«"Has received a page from a terminal"«endl;
00236     }
00237     if (sp) {
00238         cout«"Has sent a page to a terminal"«endl;
00239     }
00240     rp = false;
00241     sp = false;
00242 }
```

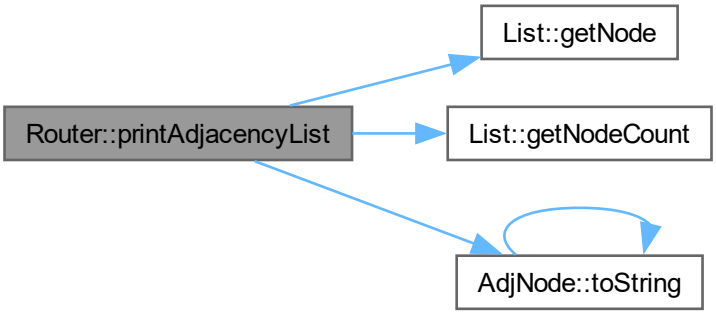
4.11.3.15 printAdjacencyList()

```
void Router::printAdjacencyList ()
```

Prints the adjacency list of the router.

```
00244                                    {
00245     cout«"Adjacency List: "«endl;
00246     for (int i = 0; i < adjacencyList.getNodeCount(); ++i) {
00247         cout«adjacencyList.getNode(i)->toString()«endl;
00248     }
00249 }
```

Here is the call graph for this function:



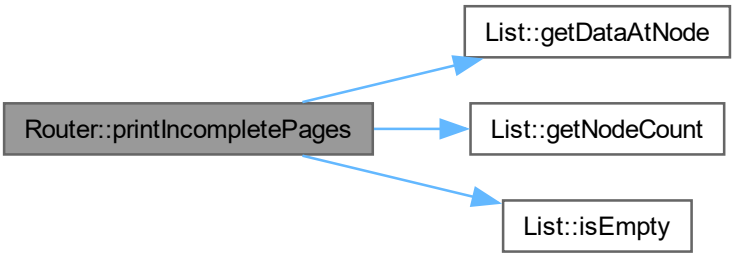
4.11.3.16 printIncompletePages()

```
void Router::printIncompletePages ()
```

Prints the incomplete pages with their packets, if any.

```
00276                                   {
00277     cout«"Incomplete Pages: "«endl;
00278     if (incompletePages.isEmpty()) {
00279         cout«"Empty"«endl;
00280         return;
00281     }
00282     for (int i = 0; i < incompletePages.getNodeCount(); ++i) {
00283         auto *list = incompletePages.getDataAtNode(i);
00284         cout«"Page "«list->getHeadData()->getPageID()«" Length
      "«list->getHeadData()->getPageLength()«":\t";
00285         for (int j = 0; j < list->getNodeCount(); ++j) {
00286             cout«to_string(list->getDataAtNode(j)->getPagePosition())«"\t";
00287         }
00288         cout«endl;
00289     }
00290 }
```

Here is the call graph for this function:



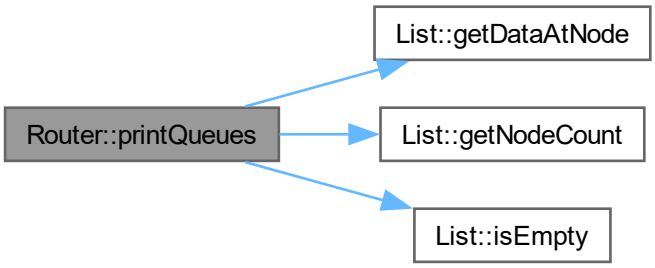
4.11.3.17 printQueues()

```
void Router::printQueues ()
```

Prints the queues with their packets, if there are no packets, it prints "Empty".

```
00258                                {
00259     cout«"Queues: "«endl;
00260     for (int i = 0; i < adjRoutersQueues.getNodeCount(); ++i) {
00261         cout « "To ";
00262         adjacencyList.getDataAtNode(i)->printRouterName();
00263         cout«endl;
00264         Queue<Node<Packet>> *queue = adjRoutersQueues.getDataAtNode(i);
00265         if (queue->isEmpty()) {
00266             cout«"Empty"«endl;
00267             continue;
00268         }
00269         for (int j = 0; j < queue->getNodeCount(); ++j) {
00270             cout«queue->getDataAtNode(j)->toString()«"\t";
00271         }
00272         cout«endl;
00273     }
00274 }
```

Here is the call graph for this function:



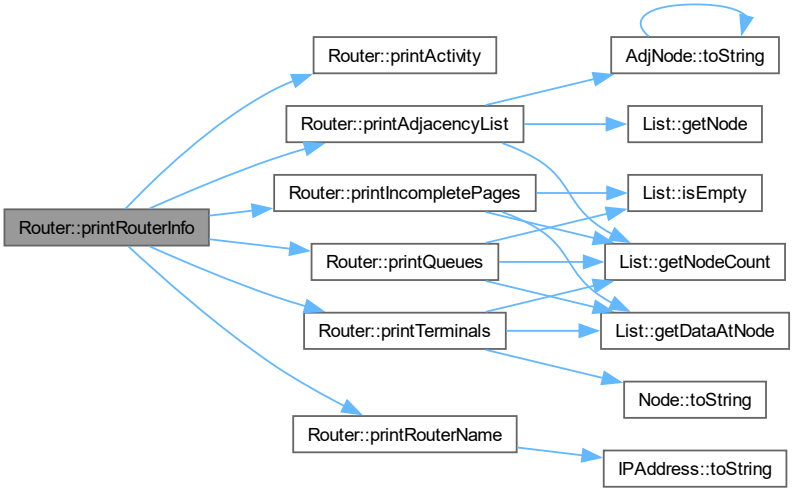
4.11.3.18 printRouterInfo()

```
void Router::printRouterInfo ()
```

Prints the name, the activity, the adjacency list, the terminals, the queues, and the incomplete pages of the router.

```
00292                                    {
00293     printRouterName();
00294     cout«endl;
00295     printActivity();
00296     printAdjacencyList();
00297     printTerminals();
00298     printQueues();
00299     printIncompletePages();
00300 }
```

Here is the call graph for this function:



4.11.3.19 printRouterName()

```
void Router::printRouterName ()
```

Prints the name of the router in a "Router + binary IP" format.

```
00229                                {
00230     cout<<ip.toString();
00231 }
```

Here is the call graph for this function:

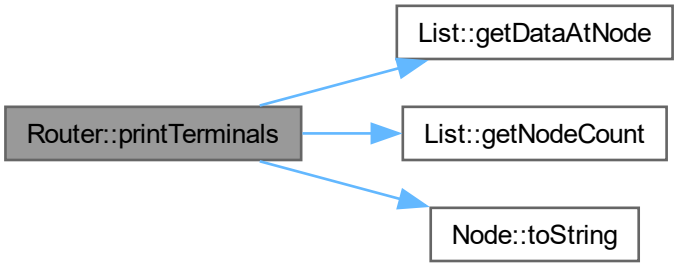
4.11.3.20 printTerminals()

```
void Router::printTerminals ()
```

Prints the terminals connected to the router.

```
00251                               {
00252     cout<<"Terminals: "<<endl;
00253     for (int i = 0; i < terminals.getNodeCount(); ++i) {
00254         cout<<terminals.getDataAtNode(i)->toString()<<endl;
00255     }
00256 }
```

Here is the call graph for this function:



4.11.3.21 receivePacket()

```
void Router::receivePacket (
            Packet * packet)
```

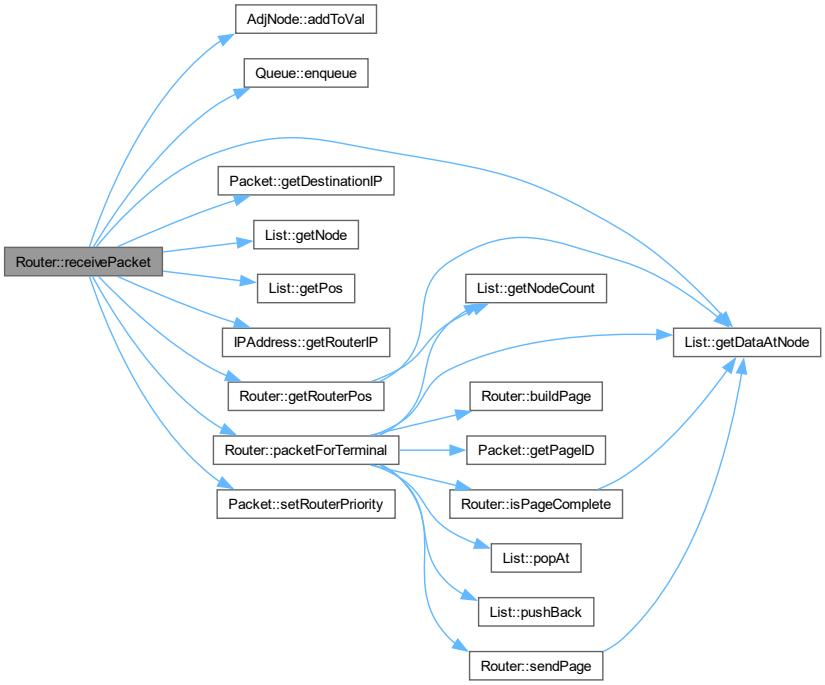
Receives a packet, sets the router priority, and enqueues the packet in the corresponding queue. If the packet is for a terminal connected to the router, it adds the packet to the waiting list.

Parameters

packet	Pointer to the packet being received.

```
00118                                       {
00119     packet->setRouterPriority(packetsReceived);
00120     packetsReceived++;
00121     uint8_t destIP = packet->getDestinationIP().getRouterIP();
00122     if(destIP == ip.getRouterIP()) {
00123         packetForTerminal(packet);
00124         return;
00125     }
00126     int routerPos = getRouterPos(destIP);
00127     Router *nextRouter = routers->getDataAtNode(nextHop[routerPos]);
00128     int listPos = adjacencyList.getPos(nextRouter);
00129     adjacencyList.getNode(listPos)->addToVal(1);
00130     Queue<Node<Packet>> *queue = adjRoutersQueues.getDataAtNode(listPos);
00131     queue->enqueue(packet);
00132 }
```

Here is the call graph for this function:



4.11.3.22 receivePage()

```
void Router::receivePage (
            Page * page)
```

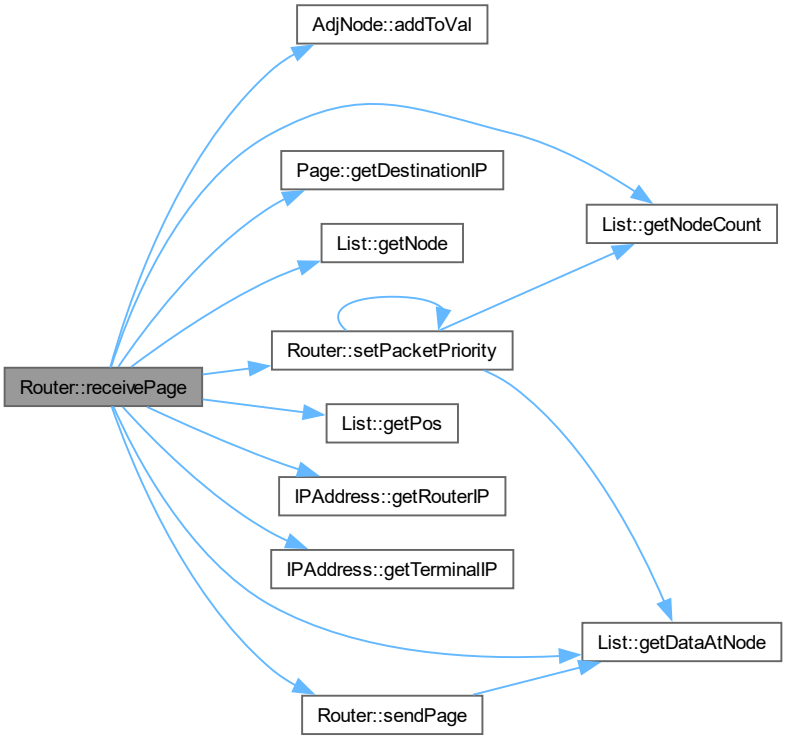
Receives a page, if the destination is a terminal connected to the router, it sends the page to the terminal. If the destination is another router, disassembles the page into packets, and enqueues them in the corresponding queue.

Parameters

page	Pointer to the page being received.

```
00090                                   {
00091     uint8_t destIP = page->getDestinationIP().getRouterIP();
00092     if(destIP == ip.getRouterIP()) {
00093         int termPos = page->getDestinationIP().getTerminalIP();
00094         sendPage(termPos, page);
00095         return;
00096     }
00097     int a = 0;
00098     for(a = 0; a < routers->getNodeCount(); a++) {
00099         uint8_t b = routers->getDataAtNode(a)->getIP().getRouterIP();
00100         if(b == destIP) {
00101             break;
00102         }
00103     }
00104     Router *nextRouter = routers->getDataAtNode(nextHop[a]);
00105     int qPos = adjacencyList.getPos(nextRouter);
00106     setPacketPriority(page);
00107     adjacencyList.getNode(qPos)->addToVal(page->getNodeCount());
00108     adjRoutersQueues.getDataAtNode(qPos)->enqueueList(page);
00109     delete page;
00110     rp = true;
00111 }
```

Here is the call graph for this function:



4.11.3.23 sendFromQueues()

```
void Router::sendFromQueues (
            int bandWith)
```

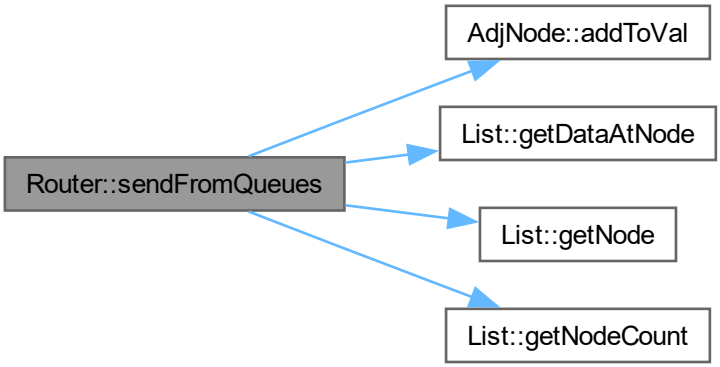
Sends a number of packets equal to the bandwidth from each queue to its respective router.

Parameters

bandWith	Bandwidth of the connection between routers.

```
00163                                       {
00164     for (int i = 0; i < adjacencyList.getNodeCount(); ++i) {
00165         auto *router = adjacencyList.getDataAtNode(i);
00166         auto *queue = adjRoutersQueues.getDataAtNode(i);
00167         for (int j = 0; j < bandWith; ++j) {
00168             if (queue->isEmpty()) {
00169                 break;
00170             }
00171             auto *packet = queue->getHeadData();
00172             router->receivePacket(packet);
00173             adjacencyList.getNode(i)->addToVal(-1);
00174             queue->dequeue();
00175         }
00176     }
00177 }
```

Here is the call graph for this function:



4.11.3.24 sendPage()

```
void Router::sendPage (
            int termPos,
            Page * page)
```

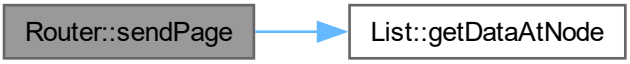
Sends a page to a terminal connected to the router.

Parameters

termPos	Position of the terminal in the list of terminals.
page	Pointer to the page being sent.

```
00113                                           {
00114     terminals.getDataAtNode(termPos)->receivePage(page);
00115     sp = true;
00116 }
```

Here is the call graph for this function:



4.11.3.25 setNextHop()

```
void Router::setNextHop (
            int i,
            int newA)
```

Sets the position of the next router to send a packet to in the position of the destination router.

Parameters

i	Position of the destination router in the list of routers.
newA	Position of the next router in the path to reach the destination.

```
00043                                       {
00044     nextHop[i] = newA;
00045 }
```

4.11.3.26 setPacketPriority() [1/2]

```
void Router::setPacketPriority (
            Packet * packet)
```

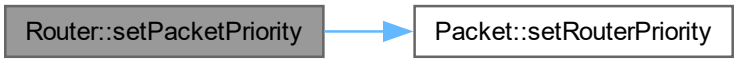
Sets the priority of the packet based on the number of packets received.

Parameters

packet	Pointer to the packet to set the priority.

```
00054                                              {
00055     packet->setRouterPriority(packetsReceived);
00056     packetsReceived++;
00057 }
```

Here is the call graph for this function:



4.11.3.27 setPacketPriority() [2/2]

```
void Router::setPacketPriority (
            Page * page)
```

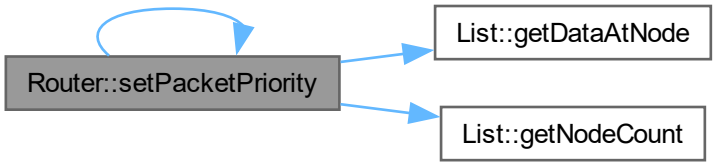
Sets the priority of the packets in a page, based on the number of packets received.

Parameters

page	Pointer to the page that contains the packets to set the priority.

```
00047                                          {
00048     for (int i = 0; i < page->getNodeCount(); ++i) {
00049         auto *packet = page->getDataAtNode(i);
00050         setPacketPriority(packet);
00051     }
00052 }
```

Here is the call graph for this function:



4.11.3.28 toString()

```
string Router::toString ()
```

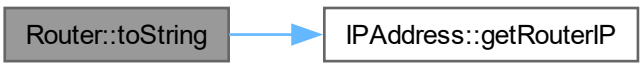
Generates a string representation of the router, including its IP address in decimal format.

Returns

A string representation of the router.

```
00302                       {
00303     return "Router: " + to_string(ip.getRouterIP());
00304 }
```

Here is the call graph for this function:



4.11.4 Member Data Documentation

4.11.4.1 adjacencyList

```
List<AdjNode<Router> > Router::adjacencyList  [private]
```

List of adjacent routers

4.11.4.2 adjRoutersQueues

```
List<Node<Queue<Node<Packet> > > > Router::adjRoutersQueues  [private]
```

List of queues of packets to send to neighbor routers

4.11.4.3 incompletePages

```
List<Node<List<Node<Packet> > > > Router::incompletePages  [private]
```

List of Lists of packets of an incomplete page

4.11.4.4 ip

```
const IPAddress Router::ip  [private]
```

IP address of the router

4.11.4.5 nextHop

```
vector<int> Router::nextHop  [private]
```

Vector of next hops to reach each router

4.11.4.6 routers

```
List<Node<Router> >* Router::routers  [private]
```

List of routers

4.11.4.7 rp

```
bool Router::rp = false  [private]
```

Flag to check if the router received a page from a terminal in the last iteration

4.11.4.8 sp

```
bool Router::sp = false  [private]
```

Flag to check if the router sent a page to a terminal in the last iteration

4.11.4.9 terminals

```
List<Node<Terminal> > Router::terminals  [private]
```

List of terminals connected to the router

The documentation for this class was generated from the following files:

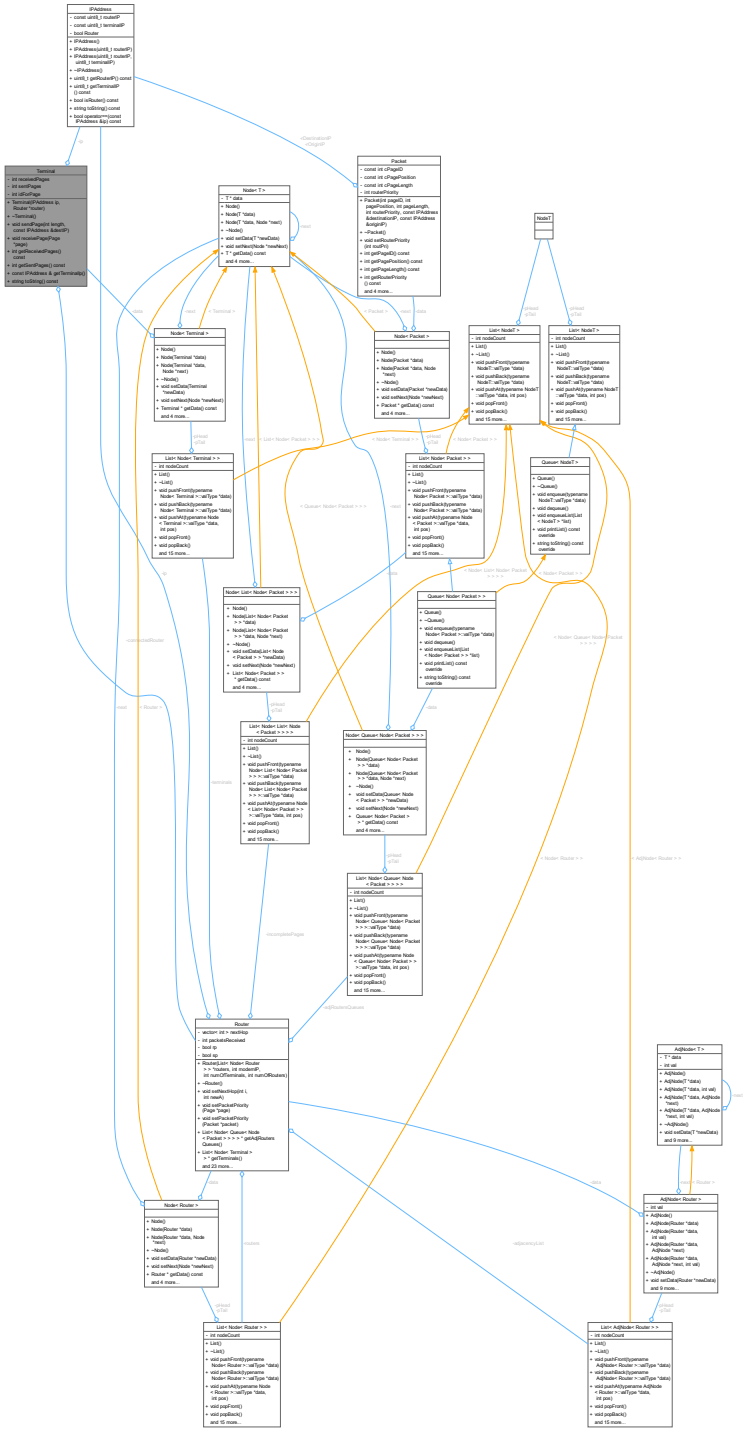
- include/Router.hpp
- src/Router.cpp

4.12 Terminal Class Reference

Represents a computer with an IP address. It tracks the number of pages sent and received, and can send and receive pages through a connected router.

`#include <Terminal.hpp>`

Collaboration diagram for Terminal:



Public Member Functions

- Terminal (IPAddress ip, Router *router)

  Constructs a Terminal object with a specified IP address and a connection to a router.
- ~**Terminal** ()

  Default Destructor.
- void sendPage (int length, const IPAddress &destIP)

  Sends a page with specified length to a destination IP through the connected router.
- void receivePage (Page *page)

  Receives a page from a router, iterates through its nodes deleting each packet, then deletes the page, and increments the count of received pages.
- int getReceivedPages () const

  Retrieves the total number of pages received by the terminal.
- int getSentPages () const

  Retrieves the total number of pages sent by the terminal.
- const IPAddress & getTerminalIp ()

  Retrieves the IP address of the terminal.
- string toString () const

  Generates a string representation of the terminal, including its IP address.

Private Attributes

- const IPAddress ip
- int receivedPages = 0
- int sentPages = 0
- int idForPage = 0
- Router * connectedRouter

Friends

- class **TerminalTest**

4.12.1 Detailed Description

Represents a computer with an IP address. It tracks the number of pages sent and received, and can send and receive pages through a connected router.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 Terminal()

```
Terminal::Terminal (
            IPAddress ip,
            Router * router)
```

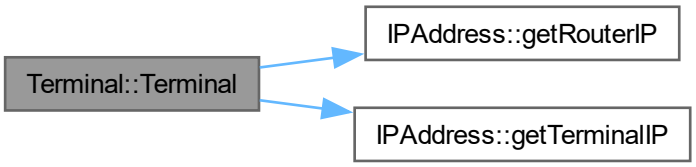
Constructs a Terminal object with a specified IP address and a connection to a router.

Parameters

ip	The IP address of the terminal.
router	Pointer to the router the terminal is connected to.

```
00008         : ip(ip), connectedRouter(router) {
00009     idForPage = ip.getRouterIP()*1000000+ip.getTerminalIP()*1000;
00010 }
```

Here is the call graph for this function:



4.12.3 Member Function Documentation

4.12.3.1 getReceivedPages()

```
int Terminal::getReceivedPages () const
```

Retrieves the total number of pages received by the terminal.

Returns

Integer with the number of received pages.

```
00029                                   {
00030     return receivedPages;
00031 }
```

4.12.3.2 getSentPages()

```
int Terminal::getSentPages () const
```

Retrieves the total number of pages sent by the terminal.

Returns

Integer with the number of sent pages.

```
00033                                {
00034     return sentPages;
00035 }
```

4.12.3.3 getTerminalIp()

```
const IPAddress & Terminal::getTerminalIp ()
```

Retrieves the IP address of the terminal.

Returns

A constant reference to the terminal's IP address.

```
00037                                       {
00038     return ip;
00039 }
```

4.12.3.4 receivePage()

```
void Terminal::receivePage (
            Page * page)
```

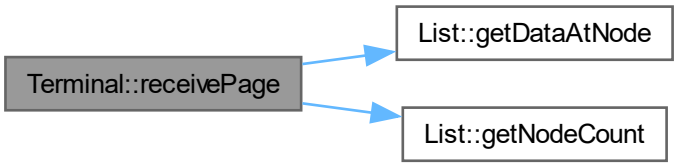
Receives a page from a router, iterates through its nodes deleting each packet, then deletes the page, and increments the count of received pages.

Parameters

page	Pointer to the page being received.

```
00021                                       {
00022     for (int i = 0; i < page->getNodeCount(); i++) {
00023         delete page->getDataAtNode(i);
00024     }
00025     delete page;
00026     receivedPages++;
00027 }
```

Here is the call graph for this function:



4.12.3.5 sendPage()

```
void Terminal::sendPage (
            int length,
            const IPAddress & destIP)
```

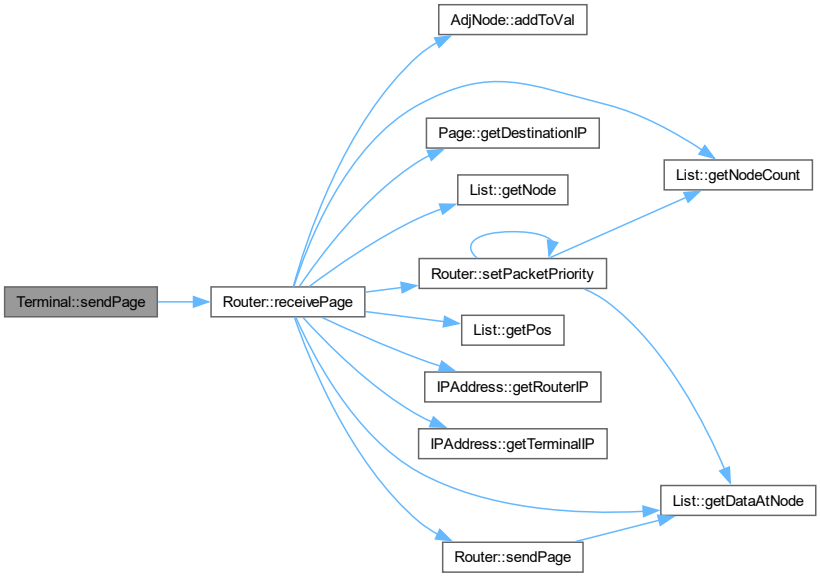
Sends a page with specified length to a destination IP through the connected router.

Parameters

length	The length of the page to be sent.
dest↩ IP	The destination IP address for the page.

```
00014                                                                   {
00015     Page *page = new Page(idForPage, pageLength, ip, destIP);
00016     connectedRouter->receivePage(page);
00017     sentPages++;
00018     idForPage++;
00019 }
```

Here is the call graph for this function:



4.12.3.6 toString()

```
string Terminal::toString () const
```

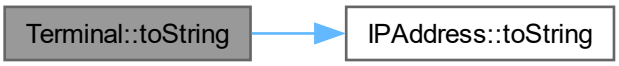
Generates a string representation of the terminal, including its IP address.

Returns

A string representation of the terminal

```
00041                                {
00042     string a;
00043     a += ip.toString();
00044     a += "\tRP: " + to_string(receivedPages) + "\tSP: " + to_string(sentPages);
00045     return a;
00046 }
```

Here is the call graph for this function:



4.12.4 Member Data Documentation

4.12.4.1 connectedRouter

```
Router* Terminal::connectedRouter  [private]
```

Pointer to the router connected to the terminal

4.12.4.2 idForPage

```
int Terminal::idForPage = 0  [private]
```

ID for the next page to be sent

4.12.4.3 ip

```
const IPAddress Terminal::ip  [private]
```

IP address of the terminal

4.12.4.4 receivedPages

```
int Terminal::receivedPages = 0  [private]
```

Number of total received pages

4.12.4.5 sentPages

```
int Terminal::sentPages = 0  [private]
```

Number of total sent pages

The documentation for this class was generated from the following files:

- include/Terminal.hpp
- src/Terminal.cpp

Index