Teijo Lehtonen | Ville Rantala | Pasi Liljeberg | Juha Plosila

FANSI: Fault Tolerant Network-on-Chip Simulator

Turku Centre *for* Computer Science

FANSI: Fault Tolerant Network-on-Chip Simulator

Teijo Lehtonen
Ville Rantala
Pasi Liljeberg
Juha Plosila

University of Turku, Department of Information Technology
20014 Turun yliopisto, Finland
{tetale | vttran | pakrli | juplos}@utu.fi

TUCS Technical Report
No 935, March 2009

Abstract

This report presents a simulator for simulating fault tolerance issues on Network-on-Chip (NoC) architectures. The simulator is called FANSI which stands for fault tolerant Network-on-Chip simulator. The simulator models the basic building blocks of a Network-on-Chip including a router, a link and a network interface. To simulate the NoC there are also models for a basic computational core and a packet to be transferred in network. Different NoC topologies and routing algorithms can be analyzed with the simulator. The simulator also provides features for modelling faults in the NoC and therefore makes it possible to analyze fault tolerance of NoC architectures.

The report presents two topologies, mesh and tree, and routing algorithms for them. The report includes also a simulation example demonstrating the usage of the simulator and its reporting features. The source codes for the header files are provided in the report.

Keywords: Network-on-Chip, simulator, fault tolerance

TUCS Laboratory
Distributed Systems Laboratory

Contents

1 Introduction

FANSI is a C++ based simulator environment to analyze different Network-on-Chip architectures is terms of their functionality, efficiency and fault tolerance. The FANSI stands for fault tolerant Network-on-Chip simulator.

The basics of Networks-on-Chip are presented in this section. The structure and parts of the simulator are presented in Section 2. Section 3 discusses routing and packet delivery on the simulator and Section 4 presents the reporting features of the simulator. Section 5 presents an example simulation case and the source file headers are listed in Section 6.

1.1 Network-on-Chip

Network-on-Chip (NoC) is a paradigm to implement interconnections on nanoscale integrated circuits. On complex circuits the traditional bus structure becomes inadequate and it is proposed to be replaced with the NoC structure. The NoC is a network which is composed of routers and links between them. The cores, for example processors and memories, are connected to routers through network interfaces (NI) and they communicate with each other over the network. The NoC can be described as a computer network in nanoscale.

On NoC there can be multiple alternative paths between every sender and receiver cores. The main advantage of the NoC is its scalability. Insertion of new cores, routers and links adds also new alternative routes to be used by all the cores.

On NoC based circuits the traffic between cores is mostly packet switched so that a NI on the sender side of the path packetizes the data and sends packets to the network one by one. Packets can possibly arrive to the receiver in disorder so the network interface on the receiver side should be responsible to reorder the received packets.

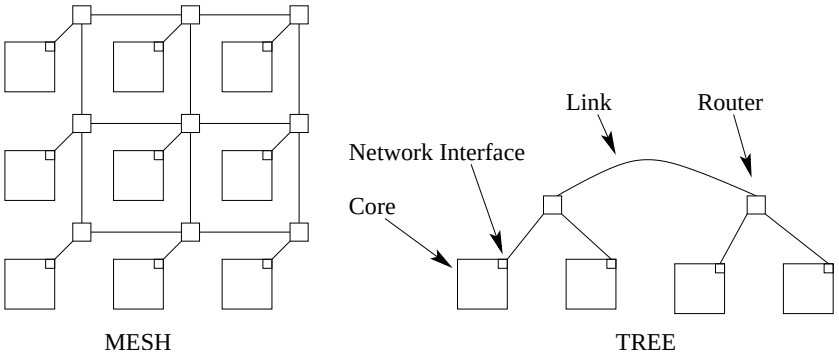

Figure 1: Mesh and tree topologies.

1.2 Topology

A topology determines the shape of the network, how the routers are connected to each other and how many input and output ports are needed on the routers. Figure 1 presents two basic regular NoC topologies, mesh and tree. The topology can also be irregular with customized links between routers.

1.3 Routing Algorithm

A routing algorithm defines how the packets are routed in the network. The routing algorithms are roughly divided to two main categories, deterministic and adaptive algorithms. Deterministic algorithms are mostly based on routing tables or fixed, previously determined routing paths. On deterministic routing the paths between every sending and receiving cores are fixed and do not change during the operation of the system.

Adaptive routing algorithms do the routing based on the circumstances on the network. The algorithm could have information for example about traffic and faults in the network. The routing is adapted to the network state to guarantee packet delivery and enable the maximum resource utilization and performance of the network.

1.4 Deadlock, Livelock and Starvation

Deadlock, livelock and starvation are events to take into account on a NoC design. The deadlock occurs when two packets are waiting for each other to be routed forward. Both of the packets are reserving some network resources and both are waiting each other to release the resources before releasing their own resources. The deadlock is typically due to a circular dependency in the network.

Livelock can occur on a system utilizing an adaptive routing algorithm. Adaptation to network faults and traffic conditions can lead to a situation where a packet keeps spinning around its destination without ever reaching it.

Starvation is a possible problem when the packets on the network have different priorities. If there is a lot of packets with high priorities on the network, a packet with a low priority may never get a possibility to use the network resources to be routed to its destination.

2 Structure

The components of the Network-on-Chip are described in separate C++ classes. The main parts are a core, a network interface, a router and a link. In addition to these there are classes for a packet (see Section 3) and a higher level description of the whole NoC to describe the size and topology of the network. All the components have constructors which reset the component variables and set the

given parameters. The components also have functions to connect them to other components and to send and receive packets. They also include report functions (see Section 4) which are used to collect simulation data to be used for further analysis.

The NoC is structured so that each core is connected to at least one network interface and each network interface is connected to exactly one core. A network interface can be connected to several routers and a router can be connected to several network interfaces. Every connection between routers and network interfaces is implemented with links. An implementation of a link is unidirectional so an implementation of a bidirectional link requires two links between the devices.

2.1 Core

The class `Core` (see Section 6.2) models computational cores in the system. The core on a Network-on-Chip can be for example a processor or memory. Here the core acts as a component which injects packets to the network and receives the packets which have been sent to it.

The core has an *address* which is given to it at the initialization. The address of a core is stored to variable *addr*. By default the address is of a type *noc_addr* (see Section 6.1) which consists of two integers. These integers can be for instance used to describe the coordinates of the core on a mesh network. The core is connected to at least one network interface. The connections are actualized by storing pointers to the networks interfaces in a vector *ni*. The core gives an ID to the network interfaces connected to it (see Section 2.2). The number of such network interfaces is stored to variable *ni_count*. While the core sends packets it stores the number of sent packets to a variable *packet_count* and the destination addresses of the sent packets to a vector *sent_packets*.

Sending and receiving of packets is done with functions *send_packet* and *receive_packet*, and is described in Section 3.

2.2 Network Interface

The network interface is modelled in the class `NetworkIF` (see Section 6.3). The network interface works as an interface between the core and the network. It is connected between a core and a router. The routers are connected to the interface with links. The network interface forwards the packets it receives from the routers through the links to the core and in the opposite direction it forwards the packets from the core towards the network through the links. The network interface is identified by the address of the core it is connected to together with an ID which it gets from the core. These IDs are required in situations where there are multiple network interfaces connected to a single core. The network interface also gives IDs to routers that are connected to it.

The network interface has functions to send and receive packets and variables to store a pointer to the core the NI is connected to, the ID of the NI given by the core, and a *route_to*-variable which stores the address of the core. There is also a vector including pointers to the links where the NI forwards packets to, and counter counting the number of *processed* packets. The *receive_packet* function increases the variable *processed* and calls the *send_packet* function, which forwards the packet towards the core or the network based on the value of the NI's *route_to*-variable and the destination address of the packet.

2.3 Link and Link_ni

The class `Link` (see Section 6.4) describes a link to connect routers to each other and NIs to routers. A link is unidirectional so bidirectional connections between routers has to be implemented with pairs of unidirectional links. The link has variables to store pointers to routers that are connected to it. There is also a counter which counts the number of processed packets and a variable which models the usability, or faultiness, of the link. The *route_to* addresses of the router, to where the link forwards packets, are stored to a vector. Functions *receive_packet* and *send_packet* handle the packet receiving and sending.

The class `Link_ni` (see Section 6.5) is a special implementation of a link to connect a router to a NI. The link stores a pointer to the NI on a variable and forwards received packets to it.

2.4 Router

The class `Router` (see Section 6.6) implements the router which is the most critical component of a NoC. The router receives a packet, does a routing decision and forwards the packet via the network interface to the core or to the another router in decided direction. The routing decision is controlled by the routing algorithm which compares packet's destination address, routers local address, addresses of the neighboring routers and cores as well as statuses of the links connected to the router's output ports.

The router has functions to send and receive packets and to connect router to other routers and to network interfaces. There is variables to store pointers to links which connect the router to a network interface and *route_to* vector to store addresses of the cores behind the network interfaces. Pointers to links between local and neighboring routers are stored to a vector. The router also includes counters to count the number of links connected to the router and the number of processed packets. The routing is discussed in more detail in Section 3.

2.5 NoC

The classes `NoC_mesh` and `Noc_tree` (see Sections 6.8 and 6.9) describe the structure of the NoC, its components and how they are connected. It includes data structures (e.g. arrays or vectors) to store the components of the system. There are also functions to generate *random traffic* patterns to the network and *link errors* to random locations in the network (see Section 3).

2.5.1 System Generation and Resource Addressing

At first the components, cores, NIs, routers and links, are created and stored to corresponding data structures. The addresses of the cores are given as parameters while generating the cores. Then the NIs are connected to the cores. After that the links from routers are connected to NIs and the links from the NIs are connected to routers. The connections have to be made exactly in this order to assure the copying of the *route_to* addresses. When the connections between routers and cores have been established, the routers are connected to each other in a way which is defined by the used topology.

2.5.2 Mesh NoC

A mesh NoC (see Figure 1 and Section 6.8) is a topology where each core has one network interface connected to one router. Each router is connected to one network interface and two to four other routers. All the connections are bidirectional thus having links into both directions. Because of this regular structure, a number of simplifications to the general model can be identified.

The core always forwards packets to the network interface found at the beginning of the vector of network interfaces. The same stands for the network interfaces forwarding packets to routers. There is only one link to choose from. In a router, the links to neighbouring routers are labeled North, East, South and West and stored in this order to the locations of the vector of links. If a link does not exist, a zero pointer is found at that location.

The *route to*-address of a router is the address of the core it is connected to and it is found at index zero in the *route to*-vector. Again the same stands for the links, they get the address from their target.

2.5.3 Tree NoC

A tree NoC (see Figure 1 and Section 6.9) consists a tree of routers and cores connected to routers on the bottom of the tree.The routers are connected to an ancestor router and two child routers, except the routers on the bottom which are connected to an ancestor router and two child cores. All the connections are bidirectional. The cores, routers and links are stored to vectors and connected to form the tree topology. A router copies the *route to* addresses of its child cores or

routers to its own *route to* vector and the ancestor router respectively copies these *route to* addresses to its own *route to* vector. The routers on the bottom of the network are connected to two cores via NIs. The routers on the top of the network are ancestors of each other.

3 Packet Delivery and Routing

The actual operation of the NoC is achieved by transmitting packets in the network generated in the previous section. Packets are described with the class `Packet` (see Section 6.7). Each packet has a *source* and a *destination address*, which are used in the routing process. Packets are identified with the source address and *identification number* given at the construction. Furthermore, the *live time* of a packet is set at the construction, and it is decreased by one at each router. The live time is needed to kill the packet in case of a livelock. Each packet also has counters for counting *NI*, *link* and *router hops*.

Packets are created at a Core, described with the class `Core` (see Section 6.2). The core copies its *address* for the packet's source address and generates a new *identification number* using its internal counter. The destination address and live time are given at the function call. Core stores each *sent packet* to a vector and forwards the packet to a *network interface* connected to the core. If there are many network interfaces connected to a core, the selection between them is done based on the routing algorithm. The packet forwarding is achieved by calling the *receive_packet* routine at the NI with a pointer to the packet as a parameter.

The packets are forwarded through the network consisting of network interfaces, links and routers, described with classes `NetworkIF` (see Section 6.3), `Link` (see Section 6.4), `Link_ni` (see Section 6.5) and `Router` (see Section 6.6), respectively. At each of them the appropriate hop count of the packet is incremented as well as the corresponding counter of the network component. When the packet reaches its destination core it is stored in a vector together with other *received packets*.

3.1 Routing

The network interfaces and routers implement the routing algorithm. The actual implementation is written into these components *receive_packet* and *send_packet* functions. The routing decisions are done on the basis of the packets destination address, the router's or network interface's *route to*-address and information of the direction the packet came from (passed as a parameter together with the pointer to packet). The router checks for the live time of the packet and if it is zero, drops the packet. Dropped packets are stored in a vector in the router. In the case of a successful routing the packet's hop count is decremented and the *processed* count of the router is increased by one.

3.1.1 XY Routing in a Mesh

In XY routing the packets are first routed to the correct column in the mesh and then along y dimension to the correct row. Therefore, the routing procedure simply compares first the x coordinates of the packet's destination and the router's *route to* address and if they do not match forwards the packet to East or West depending on which one is larger. If the x coordinates match, the comparison is done for the y coordinates. If both coordinates match, the packet is forwarded to the network interface.

3.1.2 Routing in a Tree

The tree NoC utilizes simple routing algorithm which goes through the child components of the router and compares their *route to* addresses to the packet's destination address. If the correct address has been found, the packet is forwarded to corresponding child router or core. In the case when the packet's destination address does not exist in the *route to* vectors of the child components, the packet is forwarded to the ancestor router.

3.2 Random Traffic

The simulation runs typically consist of a number of packets sent across the network. The simplest and most obvious way is to send packets to random cores in the network. For this purpose the classes `Noc_mesh` and `Noc_tree` (see Sections 6.8 and 6.9) contain functions which take the number of packets each core should send as a parameter and then create this number of random addresses in the network's address range leaving out the sending core. The functions then call the *send_packet* functions of the cores, thus initiating the whole simulation process.

3.3 Simulating errors

For simulating permanent errors in a network, a status tag *usable* has been inserted to each link. The default value given at the construction is true, but it can be changed later. Each router and network interface should check the status of the link before forwarding packets to it. The link will not forward any packets if it is not usable. Network interfaces also have a function that returns the usability of it. If no outgoing links are usable it returns false. This information can be used by the core connected to NI.

With errors possible in the network a situation may occur, where a router cannot forward packet anywhere because all routes or all routes allowed by the routing algorithm do not exist (eg. sides of mesh) or are not usable. In those cases the router drops the packets and stores it to the vector of dropped packets.

The classes `Noc_mesh` and `Noc_tree` (see Sections 6.8 and 6.9) contain functions for setting random links unusable. They get as parameters the number

of links that should be changed to unusable. If a link is already unusable a new random location is generated. In the class `Noc_mesh` only the router to router links are considered as possible locations for errors. In the `Noc_tree` also the links between routers and NIs are possible error locations.

4 Reports

The simulator stores a lot of information about the simulations and it also provides numerous ways for creating reports. The most powerful reporting feature is the *report* functions present at each class. They are used to output to the output stream given as a parameter the whole data contest together with explanation texts.

In practical simulations the information given with the report functions is too thorough. A more selective reporting gives the information in a more readable form. For these reporting purposes all the classes contain a complete set of functions for reading their data content. The selection and processing of this information is realized at the upper level, in the NoC class.

`Noc_mesh` (see Section 6.8) contains four specialized reporting functions. Two of them are used for getting the number of router and link hops in each router and link in the NoC, respectively. The output is formatted to a matrix so that it presents the structure of the NoC and is therefore easily readable. More importantly, it is in a format which can be directly read by MATLAB for further processing. Third function is used for reporting the hop counts of each packet. It outputs the frequency of each hop count number, again in a format easily readable by MATLAB. Finally the last reporting function gives the number of dropped packets in each router. The formatting is similar to the others.

`Noc_mesh` also provides functions that give the total number of dropped packets and the average of the hop count of the packets that are successfully transmitted to their destination address.

The class `Noc_tree` includes respective reporting functions than the class `Noc_mesh`.

5 Simulation Example

Figure 2 presents simulation results of previously presented tree and mesh NoCs each consisting 16 cores. The implementations were simulated injecting 1000 packets with random destination addresses to each core and repeating the simulations with 0 to 100 percent of faulty, or unusable links in the network. The results were collected using the report functions and analyzed with MATLAB.

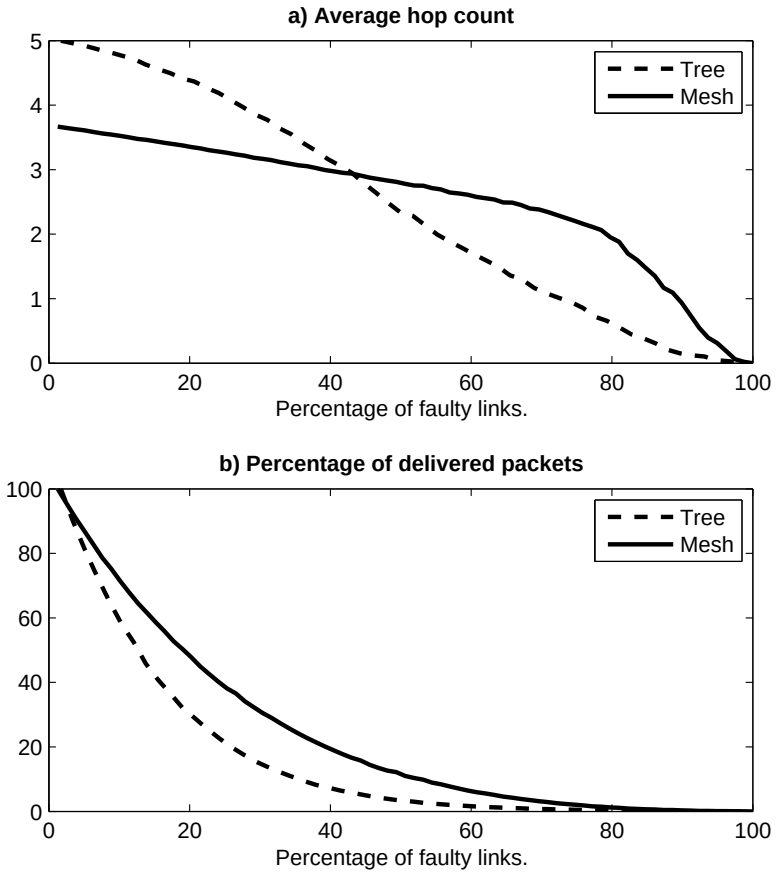

Figure 2: Simulation results.

6 Source Files

6.1 noc_addr.h

```
#ifndef NOC_ADDR_DEF
#define NOC_ADDR_DEF

#include <iostream>

/* Struct presenting a NoC address consisting of two coordinates: x and y.
 */
struct noc_addr{
  int x;
  int y;
  inline noc_addr( ):x( 0 ), y( 0 ){ }
  inline noc_addr( int x, int y ):x( x ), y( y ){ }
};

inline bool operator==( noc_addr n1, noc_addr n2 ){
  return ( ( n1.x == n2.x ) && ( n1.y == n2.y ) );
}

inline bool operator!=( noc_addr n1, noc_addr n2 ){
  return ( ( n1.x != n2.x ) || ( n1.y != n2.y ) );
}

inline std::ostream& operator<<( std::ostream& o, noc_addr na ){
  o << "(" << na.x << ", " << na.y << ")";
  return o;
}

#endif
```

6.2 Core.h

```
#ifndef CORE_DEF
#define CORE_DEF

#include <vector>
#include <iostream>

#include "noc_addr.h"
#include "Packet.h"

class NetworkIF;

/* The class represents a core in a NoC. It has an address given at the
   initialization. The core is connected to a number of  network interfaces,
   pointers to which are stored in a vector. The core gives an id to NIs
   connecting to it (sending packets to it). The number of such NIs is counted.
   Core sends packets using send_packet. The number of sent packets is stored
   as well as the addresses of the sent packets. Pointers to the received
   packets are stored in a vector as well as the pointers to dropped packets.
*/
class Core{
 public:
  Core( const noc_addr addr );
  ~Core( );
  void connect( NetworkIF* n );
  void send_packet( noc_addr dest, int live_time );
  void receive_packet( Packet* p );
  inline noc_addr get_addr( ) const { return addr; }
  inline int get_new_id( ) { return ++ni_count; }
  std::vector< Packet* >* get_received_packets( );
  void report( std::ostream& o );
  void report_sent( std::ostream& o );
  void report_received( std::ostream& o );
  void clear_traffic( );
  inline int get_dropped_count( ) const { return dropped.size( ); }

 private:
  // address of the core
  const noc_addr addr;

  // pointers to NIs the core is connected to (core forwards packets to)
  std::vector< NetworkIF* > ni;

  // number of NIs connected to core (sends packets to core)
  int ni_count;

  // number of packets sent
  int packet_count;

  // addresses of the sent packets
  std::vector< noc_addr > sent_packets;

  // pointers to the received packets
  std::vector< Packet* > received_packets;

  // pointers to the dropped packets
  std::vector< Packet* > dropped;

};

#endif
```

6.3 NetworkIF.h

```
#ifndef NETWORK_IF_DEF
#define NETWORK_IF_DEF

#include <iostream>
#include <vector>
#include "Packet.h"

class Core;
class Link;

/* The class represents a network interface (NI) which is connected to a core
   in a NoC. The NI forwards the packets it receives from the link(s) to the
   core. The NI is also connected to a number of links into which it forwards
   the packets it receives from the core. The NI is identified by the address
   of the core it is connected to together with an id it gets from the core.
   This means that there can be more than one NIs connected to a single core.
   The NI also counts the number of links that are connected to it and gives
   ids to them.
 */
class NetworkIF{

 public:
  NetworkIF( );
  ~NetworkIF( );
  void connect( Core* co);
  void connect( Link* li);
  void receive_packet( Packet* p );
  void report( std::ostream& o );
  inline noc_addr get_route_to( ) const { return route_to; }
  inline int get_new_id( ) { return ++link_count; }
  void clear_traffic( );
  bool is_usable( );

 private:

  // pointer to the core the NI is connected to (NI forwards packets to)
  Core* c;

  // address of the core the NI is connected to
  noc_addr route_to;

  // id number given by the core the NI is connected to
  int id;

  // vector of pointers to the links the NI forwards packets to
  std::vector< Link* > l;

  // number of links connected to NI (sending packets to NI)
  int link_count;

  // number of packets processed
  int processed;

  void send_packet( Packet* p );
};

#endif
```

6.4 Link.h

```
#ifndef LINK_DEF
#define LINK_DEF

#include <iostream>
#include <vector>
#include "Packet.h"

class Router;

/* The class represents an unidirectional link in a NoC. It is connected to
   a router from which it gets the route to address and id number (there can
   be several addresses). Link forwards packets to this router. The link is
   also connected to the router it receives packet from. The link has usability
   status for indicating if it can be used or not.
 */
class Link{
 public:
  Link( );
  void connect( Router* ro );
  void connect_back( Router* ro );
  void receive_packet( Packet* p );
  void report( std::ostream& o );
  inline const std::vector< noc_addr >& get_route_to( ) const { return route_to; }
  inline int get_id( ) const { return id; }
  inline int get_processed( ) const { return processed; }
  inline bool is_usable( ) const { return usable; }
  inline void set_usable( bool u ) { usable = u; }
  void clear_traffic( );

 private:

  // pointer to the router the link is connected to (forwards packets to)
  Router* r;

  // pointer to the router the link is connected to (receives packets from)
  Router* r_b;

  // route to addresses of the router the link is connected to
  std::vector< noc_addr > route_to;

  // an id given by the component the link is connected to
  int id;

  // number of packets processed
  int processed;

  // status of the link
  bool usable;

  void send_packet( Packet* p );

};

#endif
```

6.5 Link_ni.h

```
#ifndef LINK_NI_DEF
#define LINK_NI_DEF

#include <iostream>
#include "Packet.h"

class NetworkIF;

/* The class represents an unidirectional link in a NoC connected to a NI
   (forwarding packets to NI). It gets the route to address and id number
   from the NI. Link forwards packets to this NI. The link has usability
   status for indicating if it can be used or not.
 */
class Link_ni{

 public:
  Link_ni( );
  void connect( NetworkIF* n );
  void receive_packet( Packet* p );
  void report( std::ostream& o );
  inline noc_addr get_route_to( ) const { return route_to; }
  inline int get_id( ) const { return id; }
  inline int get_processed( ) const { return processed; }
  inline bool is_usable( ) const { return usable; }
  inline void set_usable( bool u ) { usable = u; }
  void clear_traffic( );

 private:

  // pointer to the NI the link is connected to (forwards packets to)
  NetworkIF* ni;

  // route to address of the NI the link is connected to
  noc_addr route_to;

  // an id given by the component the link is connected to
  int id;

  // number of packets processed
  int processed;

  // status of the link
  bool usable;

  void send_packet( Packet* p );
};

#endif
```

6.6 Router.h

```
#ifndef ROUTER_DEF
#define ROUTER_DEF

#include <iostream>
#include <vector>
#include "noc_addr.h"
#include "Packet.h"

class Link;
class Link_ni;

/* The class represents a router in a NoC. The router is connected via links
   to network interfaces and other routers. It forwards packets to the NIs
   if the packet destination address matches the route to address of a NI.
   Otherwise the packet is forwarded to one of the other routers according to
   the routing algorithm. The router is identified by the addresses of the
   cores it is connected to (via NIs and links) together with ids which it gets
   from the NIs (via link). The router also counts the number of links that are
   connected to it and gives ids to them.
 */
class Router{

 public:
  Router( );
  ~Router( );
  void connect( Link* li );
  void connect( Link_ni* li );
  void connect_copy( Link* li); // Special connect-function for tree networks
  void receive_packet( Packet* p, Router* last_router );
  void report( std::ostream& o );
  inline const std::vector< noc_addr >& get_route_to( ) const { return route_to; }
  inline int get_dropped_count( ) const { return dropped.size( ); }
  inline int get_new_id( ) { return ++link_count; }
  inline int get_processed( ) { return processed; }
  void clear_traffic( );

 private:
  // vector of pointers to the links connected to a NI the router
  // forwards packets to
  std::vector< Link_ni* > l_ni;

  // addresses of the cores the router is connected to (via a link and NI)
  std::vector< noc_addr > route_to;

  // id numbers given by the NIs the router is connected to (via links)
  std::vector< int > id;

  // vector of pointers to the links connected to other routers the router
  // forwards packets to
  std::vector< Link* > l_router; // Vector for child routers
  std::vector< Link* > l_router_a; // Vector for ancestor routers

  // number of links connected to router (sending packets to router)
  int link_count;

  // number of packets processed
  int processed;

  // vector of pointers to the packets dropped by the router
  std::vector< Packet* > dropped;

  void send_packet( Packet* p, Router* last_router );
};
#endif
```

6.7 Packet.h

```
#ifndef PACKET_DEF
#define PACKET_DEF

#include <iostream>
#include "noc_addr.h"

/* The class represents a packet in a NoC. It has a source and destination
   addresses as well as a number given to it by the source. At construction a
   live time is given for the packet, which is then decreased at each router.
   The packet counts hops in NIs, links and routers.
 */
class Packet{

 public:
  Packet( noc_addr dest, noc_addr source, int nr, int live_time );
  inline noc_addr get_dest( ) const { return dest; }
  inline noc_addr get_source( ) const { return source; }
  inline int get_nr( ) const { return nr; }
  inline int get_router_hops( ) const { return router_hop; }
  inline int get_ni_hops( ) const { return ni_hop; }
  inline int get_link_hops( ) const { return link_hop; }
  inline void add_ni_hop( ) { ni_hop++; }
  inline void add_link_hop( ) { link_hop++; }
  inline void add_router_hop( ) { router_hop++; }
  inline int decrease_live_time( ) { return --live_time; }
  void report( std::ostream& o );

 private:

  // source address
  noc_addr source;

  // a number given by the source
  int nr;

  // destination address
  noc_addr dest;

  // number of NI hops
  int ni_hop;

  // number of link hops
  int link_hop;

  // number of router hops
  int router_hop;

  // number of router hops before the packet should be dropped
  int live_time;
};

#endif
```

6.8 Noc_mesh.h

```
#ifndef NOC_MESH_DEF
#define NOC_MESH_DEF

#include <iostream>
#include <boost/multi_array.hpp>
#include "noc_addr.h"
#include "Core.h"
#include "NetworkIF.h"
#include "Link.h"
#include "Link_ni.h"
#include "Router.h"

/* The class represents a mesh-shaped Network on Chip (NoC). It gets the
   maximum address as a paramter, minimum address is (0, 0).
 */
class Noc_mesh{
 public:
  Noc_mesh( const noc_addr max_addr );
  ~Noc_mesh( );
  void random_link_error( int n );
  void random_traffic( int n );
  void report( std::ostream& o );
  void report_router( std::ostream& o );
  void report_link( std::ostream& o );
  void report_hop_count( std::ostream& o );
  void report_dropped( std::ostream& o );
  int number_of_dropped( );
  double average_of_hop_count( );
  void clear_traffic_and_state( );

 private:

  // the maximum address in the NoC
  const noc_addr max_addr;

  // array [max_addr.x + 1][max_addr.y + 1] of cores
  boost::multi_array< Core*, 2 >* c;

  // array [max_addr.x + 1][max_addr.y + 1] of network interfaces
  boost::multi_array< NetworkIF*, 2 >* n;

  // array [max_addr.x + 1][max_addr.y + 1] of routers
  boost::multi_array< Router*, 2 >* r;

  // array [max_addr.x + 1][max_addr.y + 1] of links from routers to NIs
  boost::multi_array< Link_ni*, 2 >* l_ni;

  // array [max_addr.x + 1][max_addr.y + 1] of links from NIs to routers
  boost::multi_array< Link*, 2 >* l_r;

  // array [max_addr.x][max_addr.y + 1][2] of horizontal links between routers,
  // 3rd dimenision: 0 forward direction, 1 reverse
  boost::multi_array< Link*, 3 >* l_h;

  // array [max_addr.x + 1][max_addr.y][2] of verticalal links between routers,
  // 3rd dimension: 0 forward direction, 1 reverse
  boost::multi_array< Link*, 3 >* l_v;

  void connect( Core* c, NetworkIF* n);
  void connect( NetworkIF* n, Link_ni* l, Router* r );
  void connect( Router* r, Link* l, NetworkIF* n );
  void connect( Router* r1, Router* r2, Link* l12, Link* l21 );
};
#endif
```

6.9 Noc_tree.h

```
#ifndef NOC_TREE_DEF
#define NOC_TREE_DEF

#include <iostream>
#include <cmath>
#include <boost/multi_array.hpp>
#include "noc_addr.h"
#include "Core.h"
#include "NetworkIF.h"
#include "Link.h"
#include "Link_ni.h"
#include "Router.h"

/* The class represents a mesh-shaped Network on Chip (NoC). It gets the
   maximum address as a paramter, minimum address is (0, 0).
 */

class Noc_tree{
 public:
  Noc_tree( const noc_addr max_addr );
  ~Noc_tree();
  void random_link_error( int n );
  void random_traffic( int n );
  void report( std::ostream& o );
  void report_router( std::ostream& o);
  void report_link( std::ostream& o);
  void report_hop_count( std::ostream& o);
  void report_dropped( std::ostream& o);
  int number_of_dropped( );
  double average_of_hop_count( );
  void clear_traffic_and_state( );
 private:
  const noc_addr max_addr;
  std::vector< Core* > c;
  std::vector< NetworkIF* > n;
  std::vector< Link_ni* > l_ni;
  std::vector< Link* > l_r;
  std::vector< Link* > l;
  std::vector< Router* > r;
  std::vector< Link* > l_t;

  void connect( Core* c, NetworkIF* n);
  void connect( NetworkIF* n, Link_ni* l, Router* r );
  void connect( Router* r, Link* l, NetworkIF* n );
  void connect( Router* r1, Router* r2, Link* l12, Link* l21 );
  void connect2( Router* rh, Router* rl, Link* lhl, Link* llh );
};

#endif
```

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2272-6
ISSN 1239-1891