Topgen User Guide

Juan A. Villar

Date: April 12, 2016

0.1 Introduction

TopGen is an extensible and easy-to-use library used for generating interconnection network topologies. It is written in C++ language and distributed with GPL license. The library offers a set of predefined topologies that may be used by any network simulator. New topologies can be incorporated into library without affecting the existing ones. Moreover, the library provides information that may be used to fill routing tables based on well-known deterministic routing algorithms. In the following sections, we briefly describe most of the available topologies and show real examples of usage for users who want to links their applications and simulators to the library.

0.2 Available Topologies

Basically, applications access to this library using a pointer of the class `BaseNetwork` existing in the name space `topgen`. For instance, the following lines show how to do this linkage.

```
        #include <BaseNetwork.h>

topgen::BaseNetwork * m_pTopology;
```

All the topologies have been implemented as classes that inherent from the parent class `BaseNetwork` so that applications share a common API. From the application point of view, users simply create an object of the particular class that extend `BaseNetwork`, and they assign the reference to the pointer. Next, we show various examples of topologies currently available in the library.

Single router network

This network consists of only one router that interconnects certain number of NICs. The main purpose of this topology is to debug critical components of the modeled routers. For instance, this topology can be used to check out if a router class effectively achieves the maximum throughput under certain conditions. The class `CRouter` admits one parameter: the number of NICs in the network, thus the router has as many ports as NICs. For example, the constructor of this class can be invoked as follows:

```
int m_iEndNodes = 36;
m_pTopology = new topgen::CRouter(m_iEndNodes);
```

Fully connected network

This network represents the topology in which every NIC is directly connected to the rest of NICs without intermediate routers. The main purpose of this topology is to debug critical components of the modeled NICs. The class `CFully` admits one parameter: the number of NICs in the network. For instance, the constructor of this class can be invoked as follows:

```
int m_iEndNodes = 7;
m_pTopology = new topgen::CFully(m_iEndNodes);
```

Mesh

This network builds the classic mesh often found in literature. The class `CMesh` requires the number of dimensions (only 2 and 3 dimensions are currently supported) and the width of every dimension, thus it can be established asymmetric meshes. For instance, the constructor of this class can be invoked as follows to connect 512 NICs with a 3-dimension mesh:

```
int m_iDimensions = 3;
int m_iKarity1, m_iKarity2, m_iKarity3;
m_iKarity1 = m_iKarity2 = m_iKarity3 = 8;
m_pTopology = new topgen::CMesh(m_iDimensions, m_iKarity1, m_iKarity2, ↩
    m_iKarity3);
```

Torus

This network also builds the classic torus found in literature. The constructor requires the same parameters as the constructor of the class `CMesh`. For instance, to connect 512 NICs with a 3-dimension torus do the following:

```
int m_iDimensions = 3;
int m_iKarity1, m_iKarity2, m_iKarity3;
m_iKarity1 = m_iKarity2 = m_iKarity3 = 8;
```

```
m_pTopology = new topgen::CTorus(m_iDimensions, m_iKarity1, m_iKarity2, ↩
    m_iKarity3);
```

Hypercube

The hypercube topology has been widely studied in research papers and in the academia since it was preferred to build shared-memory multiprocessor systems. There are a number of ordered dimensions where routers are assigned to. Every router connects to other routers inside its dimension and uses two ports to link with a router of the previous and next dimensions respectively. For example, an hypercube of 4 dimensions and 16 NICs can be built as follows:

```
int m_iDimensions = 4;
m_pTopology = new topgen::CCube(m_iDimensions);
```

MIN

The term MIN stands for Multistage Interconnection Network. This kind of topology is often referred as just fat-tree or k-ary n-tree because it consists of $(2k)$-port routers allocated in n stages to connect a total of k^n NICs. The constructor requires the number of NICs, the router radix (total number of ports of routers), the permutation used to interconnect stages and routing algorithm.

For instance, the following lines define a MIN network (4-ary 3-tree) interconnecting 64 NICs.

```
int m_iEndNodes = 64;
int m_iRadix = 8;
topgen::m_eConnection = topgen::butterfly;
topgen::m_eRoutingAlgorithm = topgen::destro;
m_pTopology = new topgen::CMIN(m_iEndNodes, m_iRadix, m_eConnection, ↩
    m_eRoutingAlgorithm);
```

So far, there are two permutations supported: `perfect-shuffle` and `butterfly`; and three routing algorithms: `destro`, `psdestro` (DESTRO used with perfect-shuffle permutation), and `selfrouting` (d-mod-k routing).

Additionally, the constructor admits the type of embedded node with the method `Settings`. This advanced feature allows to define combined high-radix networks whose routers are built with single low-radix routers. Nevertheless, users who want to disable this feature and consider standard routers, they just need to set the type of node to neutral values like in the next lines.

```
((topgen::CMIN*) m_pTopology)->Settings(topgen::standard, topgen::↩
    switchSTD, topgen::Alpha, 0, "");
```

KNS

The term KNS stands for as defined in [1]. In this topology, NICs are physically distributed as in a torus, but the NICs are not connected by means of links between them forming a ring of NICs per dimension. Instead of that, the NICs of every dimension are interconnected with a single router or a indirect topology like a fat-tree. For instance, the next lines define the KNS that interconnects 1296 NICs with 2-dimension direct topology and one router in every dimension.

```
int m_iDimensions = 2;
int m_pKaries[2] = {36,36};
topgen::knsIndirectType m_eIndirectType = topgen::knsrouter
int m_pRadixes[2] = {36,36};
m_pTopology = new topgen::CKNS(m_iDimensions, m_pKaries, m_eIndirectType, ↩
    m_pRadixes, false);
```

XGFT

The term XGFT stands for *eXtended Generalized Fat Tree* as defined in [2]. This topology is a fat-tree where the number of ports upwards and downwards is not necessarily equal. Nevertheless, all the ports of the routers are interconnected. For instance, the next lines define the XGFT {3;18,36;1,18} that interconnects 648 NICs.

```
int m_iRadix = 36;
int m_iHeight = 2;
int m_pChildren[2] = {18,36};
int m_pParents[2] = {18,18};
m_pTopology = new topgen::CXGFT(m_iRadix, m_iHeight, m_pChildren, ↩
    m_pParents, false);
```

RLFT

The term RLFT stands for *Real Life Fat Trees* as defined in [3]. This topology is a fat-tree, similar to a k-ary n-tree, but the last stage has half routers than previous ones, thus all the ports are used. The interconnection pattern supported is butterfly, whereas the corresponding routing algorithm is DESTRO [4]. Next lines define a RLFT of 64 NICs built with 8-port routers.

```
int m_iEndNodes = 64;
int m_iRadix = 8;
m_pTopology = new topgen::CRLFT(m_iEndNodes, m_iRadix, topgen::butterfly, ↩
    topgen::destro);
```

0.3 Retrieving Information

The class `topgen::BaseNetwork` offers a common API through which applications can retrieve information describing topologies. The constructor of the topology defines the parameters of the topology that is really built into memory in the method `topgen::RunAll` published by every class.

After calling `topgen::RunAll`, we can read general values of the topology as the number of destinations (aka NICs) and total number of routers (switches) in the network.

```
int NumNICs = m_pTopology->NumberOfDestinations();
int NumRouters = m_pTopology->NumberOfSwitches();
printf("NICs=%d\n", NumNICs);
printf("Switches=%d\n",NumRouters);
```

If the number of NICs is known, then it would be easy to retrieve every NIC using its identifier (integer number that uniquely identifies every NIC in the network). The library returns the information of each NIC as an object of the class `topgen::CInputAdapterType_Node`, whereas the corresponding information of each router is an object of the class `topgen::CElementType_Node`.

```
for (int nic = 0; nic < NumNICs; nic++) {
    topgen::CInputAdapterType_Node * tg_ia = m_pTopology->GetInputAdapter(↩
        nic);

    int router = m_pTopology->GetElementConnectedTo(tg_ia->m_iID);
    topgen::CElementType_Node * tg_element = m_pTopology->GetElement(↩
        router);
    printf("NIC %d is connected to router %s.\n", tg_ia->m_iID, tg_element↩
        ->m_iIdentifier);
}
```

Similarly to NICs, the library also allows to enumerate all the routers in the network in order to find out all the topology. The procedure basically consists in asking if every router port is sequentially connected to another router with the method `IsElementToElement`, or to a NIC with `IsElementToIA`. Nevertheless, the router port can be disconnected with the method `IsElementToAir` since this situation is also possible in some topologies.

```
for (int router = 0; router < NumRouters; router++) {
    topgen::CElementType_Node * tg_element = m_pTopology->GetElement(↩
        router);
    for (int port = 0; port < tg_element->m_iPorts; port++) {
        /* data written inside topgen library */
        unsigned int neighbour_node[1];
        unsigned int neighbour_port[1];

        printf("router %d port %d connects to ", router, port);
        if (m_pTopology->IsElementToElement(router, port, neighbour_node, ↩
            neighbour_port)) {
            printf("router %d port %d.\n", neighbour_node[0], ↩
                neighbour_port[0]);
        } else if (m_pTopology->IsElementToIA(router, port, neighbour_node↩
            , neighbour_port)) {
            printf("NIC %d port %d.\n", neighbour_node[0], neighbour_port↩
                [0]);
        } else if (m_pTopology->IsElementToAir(router, port, ↩
            neighbour_node, neighbour_port)) {
            printf("air.\n");
        } else {
            printf("Unknown error.\n");
        }
    }
}
```

On the other hand, the library provides routing information for filling routing tables used in routers. This feature is bound to a small number of topologies (e.g., hypercubes, meshes, torus, and MINs) because many topologies consider adaptive routing algorithm. The TopGen API returns the output port given a router and destination. Nevertheless, this data may be useful to enumerate the paths of any pair of NICs. For instance, given a 3D mesh with 512 NICs, we can determine the output port to reach the NIC 511 at switch 0 invoking the method `GetOutputport` with the next code:

```
int identifier = 0;
int destination = 511;
int output = m_pTopology->GetOutputport(identifier, destination);
printf("router:%d destination:%d output:%d.\n",identifier, destination, ↩
    output);
```

0.4 Creation of new topologies

TopGen allows users to easily add new topologies as simple as creating new C++ classes that inherent from `BaseNetwork`. These C++ classes encapsulate all the detailed description of the topology without affecting the rest of existing topologies. Although it is not compulsory, they are often created three new C++ classes for each topology: a main class that extends `BaseNetwork`, an two other auxiliary classes for describing the router and channels used in the new topology. For instance, let's suppose that a new topology called *CoolTopology* must be supported in TopGen, then programmers should create a main class called *CCoolTopo*, and *CCoolTopoNode* and *CCoolTopoChannel*. This is the naming policy that has been followed in the library so far.

As previously mentioned, `BaseNetwork` holds the information of NICs (aka Input Adapters), Elements (aka Switches) and Channels (aka links) in three main lists named `m_pInputAdapters`, `m_pElements` and `m_pChannels`, respectively. Moreover, `BaseNetwork` offers methods for inserting items into the lists, for instance, `_LoadIA()`, `_LoadElement()` and `_LoadChannel()`, respectively.

Although, the user is free to numerate the components of the network, but the identifiers of NICs and switches have to start in zero for optimization purposes. The user/programmer is responsible of filling these lists according to the topgen programmer's understanding of how the new topology is. This description is performed in two phases: the building phase and the loading phase.

The first one aims to provide a logical description based on objects of the classes describing the router/node and channel. This phase consists of the methods `BuildNetwork()`, `BuildInterconnection()` and `BuildChannels()`. The programmer is free to distribute the code among the methods as needed.

Later, in the second phase, the logical descriptions are completed and loaded into the lists inherited from `BaseNetwork`. Similarly, the user can freely distribute the code among the methods `LoadNetwork()` and `LoadRoutes()`. These two methods and the three Build methods will be appropriately invoked in the method `RunAll` that really builds everything.

Bibliography

[1] R. Peñaranda, C. Gómez, M. E. Gómez, P. López, and J. Duato, “The k-ary n-direct s-indirect family of topologies for large-scale interconnection networks,” *Journal on Supercomputing*, pp. 1–28, Feb. 2016.

[2] X. Yuan, S. Mahapatra, M. Lang, and S. Pakin, “Static load-balanced routing for slimmed fat-trees,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 5, pp. 2423–2432, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514000215

[3] E. Zahavi, “Fat-tree routing and node ordering providing contention free traffic for {MPI} global collectives,” *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1423 – 1432, 2012, communication Architectures for Scalable Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512000305

[4] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, and J. Duato, “Deterministic versus adaptive routing in fat-trees,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.