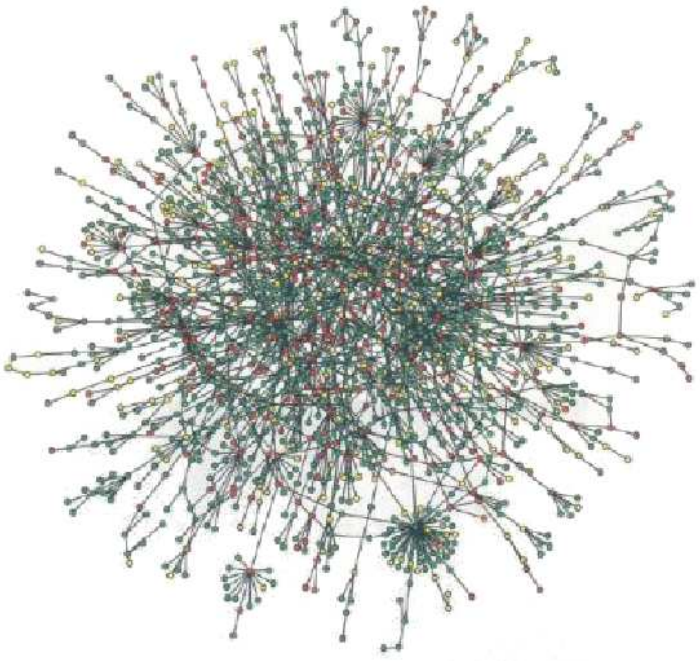
SCAR

a object-oriented framework for distributed scale-free networks simulation on top of Artís

by Andrea Sottoriva
14th December 2005



Coordinators:
Prof. Lorenzo Donatiello
Prof. Gabriele D'Angelo

Department of computer science
University of Bologna (Italy)

Contents

List of Figures

1 Introduction

Since few years ago, modelling a complex system which involves a lot of similar subjects such a cellular tissue, a set of human relationships, a bugs hierarchy or a computer network meant to work with a completely random network, that had to describe all the structure links between those subjects.
After the studies of scientists like *Albert Laszlo Barabasi* however, the entire scene of networks simulation changed definitively discovering that almost all complex systems aren't based on random or pseudo-random networks but on a particular kind of structures : *Scale-Free* networks (*SFN*).

In a *SFN* the nodes are not uniformly distributed over the space, instead they are gathered around those other nodes which already have a certain number of connections. These networks seem to represent the entire set of environment structures related to complex systems; by now in fact it is discover that, for example, the following systems are based on scale-free networks:

- Cellular metabolism
- Sexual relationships
- Research collaborations
- Protein networks
- Hollywood actors
- The world wide web

The above experimental information makes possible to suppose that scale-free networks are everywhere around us, from our daily relationships to the distribution of the mass in the universe. So a great deal of scientific interests are originated from these studies, specially those regarding the *simulation* of cited phenomena.

My idea was to develop an open, interactive and object-oriented framework which works on top of *Artís* (`http://www.jpolis.com/gda/index.html`) and that permits the distributed simulation of a scale-free network through a simple and transparent interface.

The project name is *SCAR (SCale-free ARchitecture)* and provides not only a simulation interface, extendable classes, static and dynamic network manipulation, but also a 3D scale-free network visualizer that may help the system modeler to describe the simulation environment in a interactive way.
This middleware is entirely written in C++ and it uses the *Chandy-Misra-Bryant* algorithm provided by ARTÍS to implement network synchronization, messages delivery and all the other simulation events.

2 Design

2.1 Simulation flow

SCAR is aimed to be as easy to use as possible, with a minimal amount of user-visible objects and methods to call; from the application point of view, in fact, only two objects are involved: the simulation core *Scar* (see section 3.4 on page **??**) and the virtual class *Node*.

The first step for a user is to define a class extending *Node* to specify the behavior of a single item through the *Live()* and *React()* methods implementation; together those two functions can completely describe the essence of a node: the first corresponds to its internal life (state update) and the second to its reaction to external events.
A user however is not always forced to build a node structure from scratch (following only the *Node* interface) but can be aided by the *TemplateNode* class, an extension of the node interface which adds a lot of useful features to the basic implementation and which will be discussed further on this document.

After these operations a *Scar* object must be instantiated supplying some scale-free network parameters (the node *distribution generator* and the number of *connections* per node), then a user is encourage to follow the standard usage procedure, or rather give a certain number of its nodes to the middleware and, at the end, start the simulation calling the *StartSimulation()* method.

At this step the creation of the network begins: each local processor adds one by one its nodes to the scale-free structure (*Laszlo*) generating synchronization events, so when all the LPs have completed this operation, then the real simulation starts with a global state update of all nodes.
From here everything will be event-driven.

2.2 Objects relationships

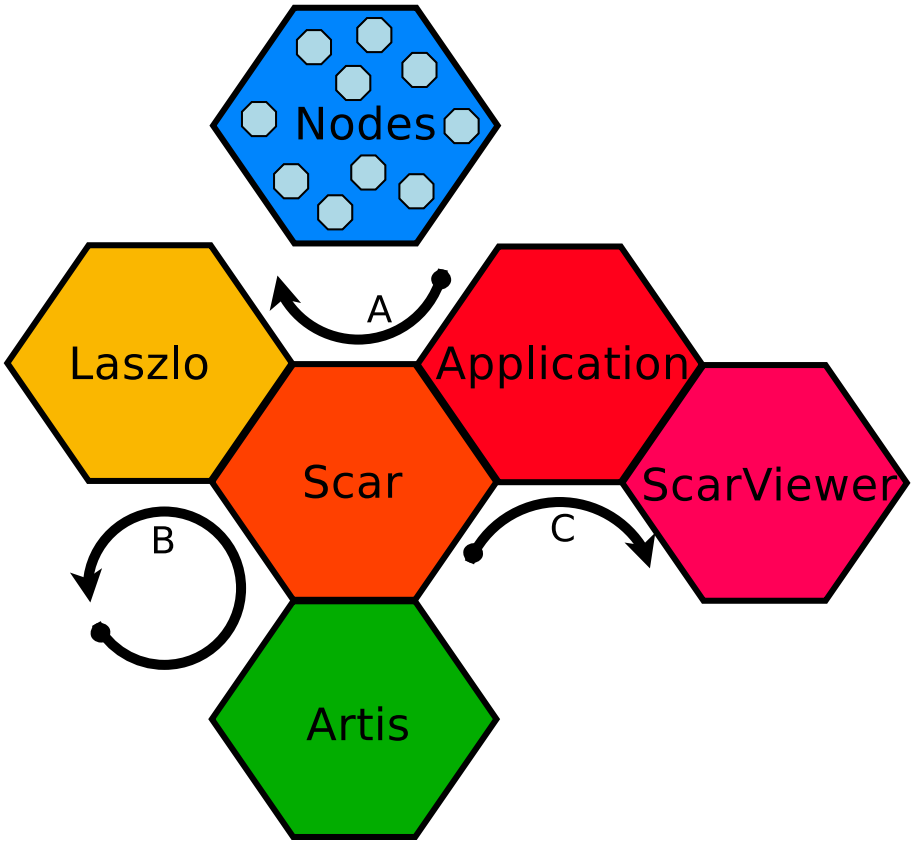


Figure 1: *SCAR's objects*

Beyond, the diagram shows the *SCAR* design and its information flow representing objects relationships: after the application has passed its nodes to the framework, as displayed by the creation loop A, the computation starts following the *simulation* loop B, at the end therefore the information flow returns to the application following the *output* loop C (for reporting and visualization).

Polygons edges represent the dynamic visibility of SCAR objects, exagons with adjacent edges can directly communicate between them.

The expressive power of the middleware in correspondence to the simulation model is encapsulated inside the nodes, it depends only from the behavior of these deployed items, from their reaction to external events and from their internal state update.

At the end of simulation the user gets back all its nodes passed through the simulation flow and the entire scale-free network structure available (from *Scar*) for reports and, if needed, for a 3D visualization using *ScarViewer*.

3 Framework structure

3.1 Middleware candies : nodes

From the modeler point of view the definition of network nodes is a critical point, after defined the *environment* of the model through the scale-free network parameters in fact the validation of the entire system depends only from the nodes behavior definition.

Just for this reason SCAR wants to provide to the modeler an as flexible and simple as possible *node-subsystem*, providing the following interface that will be discussed further:

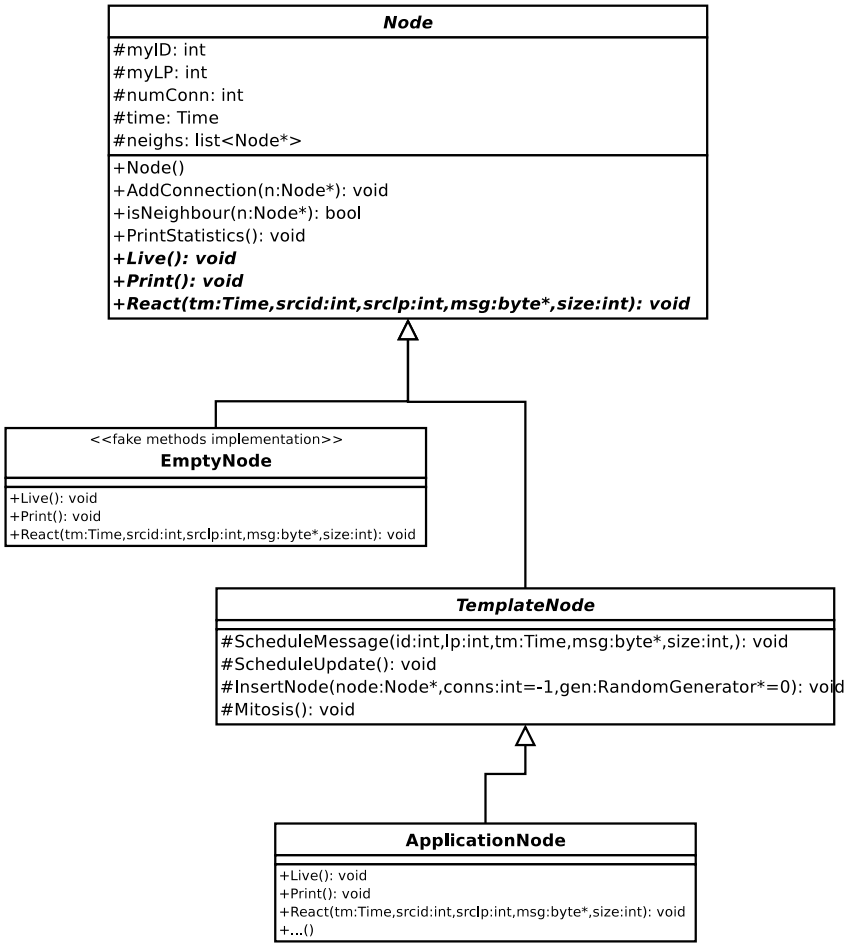


Figure 2: *Node interface [UML diagram]*

3.1.1 The *Node* class

The *Node* class is the skeleton of a scale-free node, it implements only those methods which describe its internal structure, like node links and the other information directly related to the SCAR framework (the local processor where the node resides and its identifier).

The basic idea is to leave unimplemented all those methods which define a specific behavior, so the SCAR framework constricts the application to extend this class implementing also its virtual methods.

At framework level every item managed is a *Node* object so only methods hereafter can be called from within SCAR which cannot understands anything about the node behavior.
The *Node* class therefore contains information for scale-free management purpose only, in particular the following are the main features:

- ***virtual*** *Live()* :

  Defines the internal state update function that represents the spontaneous node life. This method is called whenever an *UpdateState* event occurs and, by default, one time at the beginning of simulation.

- ***virtual*** *React()* :

  Defines reactions to external stimuli, this type of message can only be created from one of the other nodes of the network after an internal state update or another message reaction. SCAR never generates messages or any other type of event that may be directly delivered to nodes.

- ***virtual*** *Print()* :

  Applications are also forced to implement a simulation report method in order to maintain the strict reporting policy adopted by the framework. This method permits also a *middleware-level* reporting , useful for debugging purpose.

- *AddConnection()* :

  Adds a connection between this node and the one passed, creating a double side link. The supplied node is added to the local connections list and vice versa.

And the following are the (*protected*) fields inherited from extending classes:

- *myid* :

  The node identifier.

- *mylp* :

  Local LP number.

- *numConn* :

  Number of connections initially created for this node. This number doesn't refer to the actual number of links, but to the number of connections created when the node was inserted inside the network.

- *time* :

  The global time pointer. Each time a node is added to the SCAR framework it receives the current pointer to the simulation time, this field will be used for scheduling purposes.

- *neighs* :

  The list of node neighbors pointers. The entire information regarding the scale-free network shape and properties is managed through this list.

In brief *Node* is a virtual class that constricts a user to extend itself and to define its virtual methods (*Live()*, *React()* and *Print()*). After have discussed about the *Node* class, a user can think this interface for node definition is a bit simplified and reduced for a typical system modelling, however another useful structure is provided by the SCAR framework just to resolve this problem: the *TemplateNode* class, discussed in the next section.

3.1.2 The *TemplateNode* class

The *TemplateNode* class is a structure provided by the framework in order to simplify the modeler approach to the node definition. It extends the node class leaving unimplemented the methods that strictly define the node behavior (the previous discussed *Live()*, *React()* and *Print()*), but adding some of new useful features usable from within a node during the stimulation.

So much interesting are methods for dynamic network modification, a node in fact can schedule not only a message or an update, but also a new node insertion, or can clone itself (all its internal state but connections) adding this clone to the network.

The features provided by the *TemplateNode* class are the following:

- *ScheduleMessage()* :

  Schedules an event by passing the recipient ID, LP number, scheduling time, a buffer with message and its size in bytes. SCAR will generate a message event and will delivery it correctly.

- *ScheduleUpdate()* :

  Schedules an internal auto-update state in order to produce a new call to the *Live()* method at *Current + Passed* time. This method is by default called one time at the beginning to start up the event-driven simulation.

- *Mitosis()* :

  Generates a *clone* node and add it to the neighbors creating a bidirectional link, then schedules a synchronization event. This operation involves only internal state and configuration therefore connections to other nodes are not copied.

- *InsertNode()* :

  Inserts a new node inside the network, following the standard scale-free insertion policy. Obviously, generates also the synchronization event.

Examining the previous UML diagram however, we can view that if SCAR manages all its node with *Node* objects become possible for application to bypass the *TemplateNode* class and provide to the framework only *Node*-derived objects; this kind of usage is permitted but thoughtless because it reduces the expressive power of the simulation. Some of its obscure aspects are discussed below in the *EmptyNode* section.

Therefore declaring a *TemplateNode* subclass an application can create any kind on node that may represent a computer, a protein, a cell or even a person.

3.1.3 The *EmptyNode* class

Every local processor has to maintain the entire view over the scale-free network, this is necessary for node insertion and even more for messages delivery, an LP in fact is responsible on the node-to-node message delivery so when a *MsgEvent* occurs it has to discover if the destination node is inside the local network or outside it. To perform its job a *Scar* object (further discussed in the section 3.4) maintains some information about all nodes inside the network in the following way. Local nodes are present inside the structure in their original representation supplied by the user, the non-local ones instead are represented only by their fake implementation, used to retrieve the node ID and, very important, the local processor. At this point we have explained that inside each local processor there is a complete image of the network, however a considerable part of nodes are used only to maintain their absolutely necessary fields.
To keep the framework as light as possible then we have to reduce the size of those fake nodes using the simplest extension of a *Node* interface: the *EmptyNode* class that simply inherits its father resources and implements virtual methods as do-nothing functions.
Returning to the problem discussed in the previous chapter about a possible non standard usage of the framework we can now understand our trade-off:

- We have to define an interface for nodes manipulation to which SCAR refers, this interface have to be as light as possible because it is the minimum size of a single node.

** BUT **

- We have to provide a powerful and flexible template class that helps a modeler on working with this framework.

I prefer to consider at first the framework performance, so the services provided to the user have been reduced as much as possible (in order to maintain light a single node) while node functionalities have been shifted down to the *TemplateNode* class becoming an *optional* choice. Note that those features remain however highly recommended and have to be considered as the standard way of use.

3.2 *RandomGenerator* interface

The shape and the properties of a scale-free network are defined by its own links, now some words must be spent explaining the importance of the *network distribution function.*
Inserting a node into a *SFN* involves a stochastic choice between all the already present nodes, this aleatory number X has a specific probability distribution decided by the network creator. Usually scale-free networks are built using an exponential distribution function while, for example, classical random networks are generated using a uniform one.

To maintain a maximum degree of flexibility SCAR uses suitable objects called *generators* that must be supplied during the node creation and that are customizable by the user simply implementing their interface: the *RandomGenerator.*
By default our framework provide three built-in generators :

- Exponential generator $f(x) = \lambda e^{-\lambda x}$
- Uniform generator $f(x) = \begin{cases} \frac{1}{b-a} & (a < x < b) \\ 0 & otherwise \end{cases}$
- Gaussian generator $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$
- Poisson generator $P(X = i) = \frac{\lambda^i}{i!} e^{-\lambda}$

The Poisson generator however, must be handled with care, the Poisson density function codomain in fact is different from zero only between about -5 and 15 then usable values swing within the range (0, 15].
So, getting numbers in the range from 0 to the number of nodes mean stretch the output of the Poisson *CDF* inversion procedure creating a lot of numeric holes.
These holes are filled with a raw *linear interpolation* and partially compensated by the entropy introduced with network sorting (see appendix B for a further discussion).
Into these objects SCAR uses the standard *random()* function as random source. Initialization of system random structure with a seed is done by *Laszlo* once when instantiated, so it is not necessary to call a *srandom(seed)* inside a *RandomGenerator* object.

In particular a user must remember that at *every* simulation run the framework execute the following piece of code:

```
double seeds[20] = { 606112567, 2005261598, 975679233, 381883932, 1949009736,
                     457621696, 31040274, 2084138466, 63789657, 521636194,
                     670018843, 1263912019, 2072256297, 1025373591, 1693757298,
                     1115383532, 44685244, 1568381214, 2014569565, 1052870108 };

srandom((unsigned int)seeds[mylp % sizeof(seeds)]);
```

As we can see, a previous seed manipulation is always useless.

Most demanding users can also specify a different generator at every node insertion, so any kind of hybrid network is permitted and then...simulable.

3.3 The scale-free master of puppets: *Laszlo*

A *Laszlo* object describes the structure of the scale-free network containing all the nodes objects and providing all the necessary management features such as automatic node insertion and networks merging. Both *Scar* and *ScarViewer* (section 3.4 and 4) refer to nodes through this object.
The main features offered by the *Laszlo* class are:

- *SetConnXnode()* :

  Set the number of initial connections per node. This number refers to how many links are generated when a node is added to the network. This is a global (*default*) value that can be overridden when calling an *AddNode()*.

- *AddNode()* :

  Insert a new node into the network and generate its links following the specified probability distribution (or the default one). At any node insertion the entire network is reordered using a *O(n log(n))* algorithm, so this is the computational cost of this operation.

- *SetNode()* :

  Attach an already linked node to the network; in this case simply update the related links and reorder the network (used for synchronization purpose).

- *GetNode()* :

  Return, if present, the node structure with the specified ID and LP.

- *Merge()* :

  Merge this scale-free network with another, build all links correctly and reorder the network.

- *DeliverMessage()* :

  Deliver a message to a specific node inside the network, nothing happens if a delivery is requested for a non local node (fake).

- *UpdateNode()* :

  Update the internal state of a node calling its vital function, nothing happens if a update is requested for a non local node (fake).

- *UpdateAll()* :

  Update the internal state of all nodes calling their vital functions, only real nodes are updated.

Into this object the network is simply represented as an ordered list of *Node* structures linked between them.
During the simulation time inside a scale-free network of a single LP are present simultaneously two kinds of nodes: those which has been inserted by the application (we will call them *local nodes*) and all the other ones which have been generated by the others LPs and are came out from a synchronization event; all these *non-local nodes* are present inside *Laszlo* in their fake implementation in order to maintain a global view of the entire network (needed by the scale-free structure itself) reducing as much as possible those unnecessary information related to the fake nodes.
Note that these *non-local nodes* therefore are only an image of their true implementation, so the simulator knows which local processor they refers and nothing else.
Each non-local node is represented with a previous discussed *EmptyNode* object: the minimum-size structure that permits all middleware-level operations.
However, notwithstanding its flexibility, *Laszlo* cannot be considered as a total secure black box object because only the application knows how to manipulate correctly the nodes contained within, so there are some cases where becomes necessary to bring out the node list for a direct manipulation.

3.4 The simulation core : *Scar*

The simulation core is represented by the *Scar* object that is responsible on scale-free network management and on events scheduling through ARTÍS calls. When instantiated *Scar* initialize a new scale-free network (*Laszlo*) based on the parameters supplied by the user (the *RandomGenerator* object and the number of *connections* per node), then the application has to provide a certain number of nodes (at least one, extended from *Node* or, *better* from *TemplateNode* class) and start the simulation, specifying how much time has to be simulated. After this procedure *Scar* starts its loop fetching new events from below; at the beginning it usually has to be handleed all the *new-node-insertion* events and their related synchronization messages, then, when all the LPs have created their networks and everything is synchronized, the simulation can be started. At this point must be give the initial stimulus calling an update of the internal state of each node (*Laszlo::UpdateAll()*) which cause the event-driven process to start. Possible simulation events on SCAR are the following:

- *NewNode* :

  A new node has to be inserted into the network, this event can be generated only by the local application because concrete nodes can be inserted only into the local networks, for remote insertion a *NodeSync* event must be used.
  When this kind of event occurs a *Node* object has been dequeued from the new-node structure and inserted into the network through *Laszlo*.
  If the new-node queue is empty, an error is thrown.

- *NodeSync* :

  A node was inserted somewhere into a remote network, so a synchronization is required. A synchronization event provides the new node identifier, its local processor and a list of its neighbors IDs.
  No other piece of information is needed: the local processor simply create a new *EmptyNode* structure and add it to the network using the *SetNode()* method.

- *Created* :

  All initial nodes of a local processor has been added, so the specified LP is ready for simulation.
  To avoide simulation errors each local processor, after that all the nodes have been added, waits until all the other LPs have been initialized, then starts the simulation.

- *NodeMsg* :

  A message has been generated from a remote node, it has to be delivered somewhere into the local network, so this message is took and passed to a *DeliverMessage* call.

- *Update* :

  Internal state update requested. Call a *Laszlo*'s *UpdateNode()* method passing the identifiers provided by this event.

- *EndSim* :

  End of simulation: stop to handle events, begin the reporting/visualization procedure.

At the end of simulation the scale-free network inside *Laszlo* can be retrieved in order to visualize it through the 3D visualizer (discussed in the section 4) or simply for reporting and analysis. At this point we can summarize the run-time structure of a simulation as follows:

1. Define a single node structure and its behavior
2. Decide in which kind of network those nodes live
3. Add an initial number of defined nodes
4. Start the simulation
5. Retrieve simulation reports and, if needed, the entire scale-free network for 3D visualization

The SCAR project was developed with extensibility in mind, every piece of code can be overridden, every object extended, so new functionalities that a modeler can need are extremely simply to implement.

4 A 3D OpenGL visualizer : *ScarViewer*

4.1 A brief overview

ScarViewer was developed to provide a flexible and interactive network visualisation toolkit besides those analytical report methanisms like simulation tracing and the others dynamic data collections.

This visualiser permits the studying of nodes distribution over the network: it displays the scale-free using a non-deterministic algorithm which tries to spread the nodes in a as representative as possible way.

Given a *Laszlo* object in input *ScarViewer* can visualise the entire network with the following features:

- Visualisation of nodes distribution over LPs through different colours
- Interactive arcball-stype network manipulation and view
- Runtime change of the scale-free hubs gathering
- Runtime change of nodes density over the space
- Auto-adaptive density/gathering configuration
- Printing of some kind of text on window
- Possibility of make screenshots
- Link-only or sphere-node visualisation
- Rebuilding network optimization

This tool was developed to be very simple to use, every aspect regarding the used libraries (OpenGL and SDL) and I/O interaction is enclosed inside the object in a totally opaque way.

This is obviously a trade-off with flexibility but suppose that everyone with a minimal scientific knowledgement may need to use it, so the only actions involved are *instantiation* and a *display()* call.

This part of the SCAR project is intended to develop a little graphic toolkit that a user can extend, adding a lot of other functionalities like a dynamic representation of simulation, or view of some parameters of its specific model and implementation.

5 Framework analysis

I have decided to spend a certain amount of time in framework analysis, trying to verify its efficiency, its coherence and its behavior variating the number of local processors, the number of nodes and their distribution.

Those studies have produced a lot of information, useful also for some decisions that I have to be make during the simulation modelling.

These presented observations are made using an exponential distribution (an as-scale-free-as-possible network).

5.1 Scale-Free property conservation

The first step is to verify if the scale-free distribution property is conserved increasing the number of local processors.
These data represents a simulation run of an exponential distribution network with a mean of 200.0. The following graphs visualizes the distribution of links over the network, reported increasing the number of local processors:

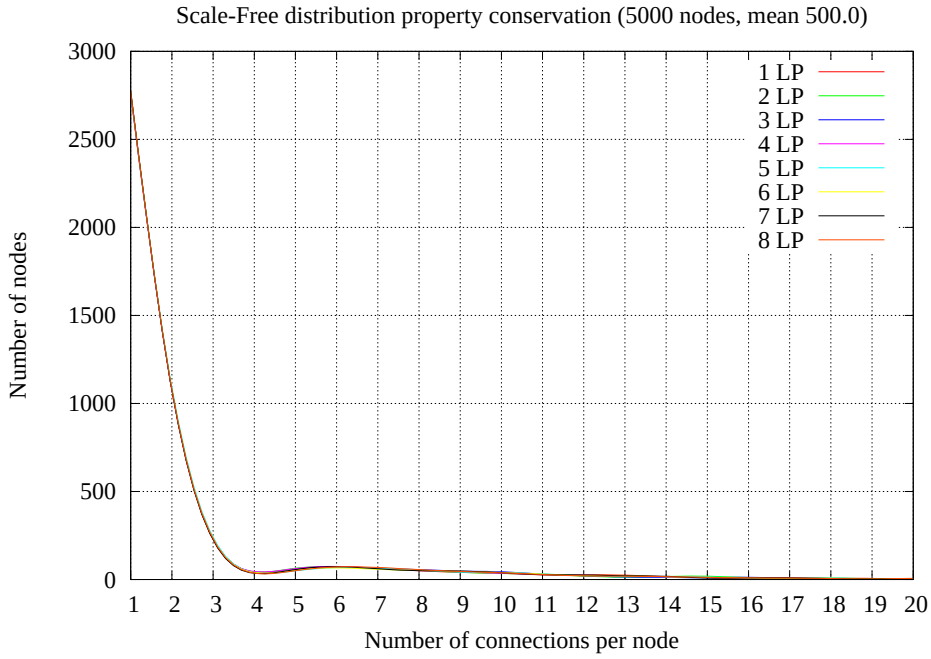


Figure 3: *Exponential node distribution*

As we can see all of these curves coincide, determinating a proof of goodness of SCAR.
Using the framework we can suppose now to have the warranty on correct network creation that will not depends on the number of local processors and then on the parallelization of calculus.

5.2 Efficiency of node distribution

Another structural analysis that may be interesting focuses on the percentage of generated events that need a remote delivery (cause the target node isn't inside the current local processor).
Through this observation we want to verity if the framework structure can hold inside the great part of nodes messages in order to use the scale-free property to minimize the inter-LP traffic:

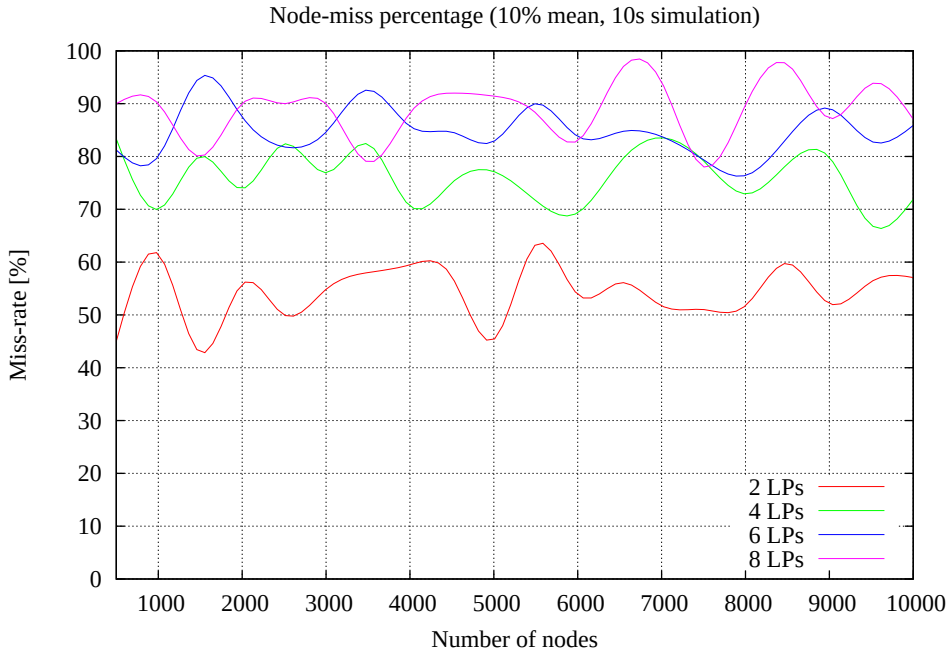


Figure 4: *Node-miss rate (variable mean)*

The above two graphs need a first step analysis:

1. The framework behavior is independent from network dimensions; this is a good thing, it means that SCAR is quite scalable and we can suppose there is no critical network dimension but the one imposed by hardware.

2. SCAR is also independent on the mean of network distribution, in the first graph we have simulate a network with a variable mean value (10% of total nodes) instead in the second one we have fixed its value to 200.0. In both cases the efficiency doesn't change.

So we can assume that SCAR is (network) shape-invariant and we can by now consider it as a proof of goodness of our middleware; however thinks are not so simple.

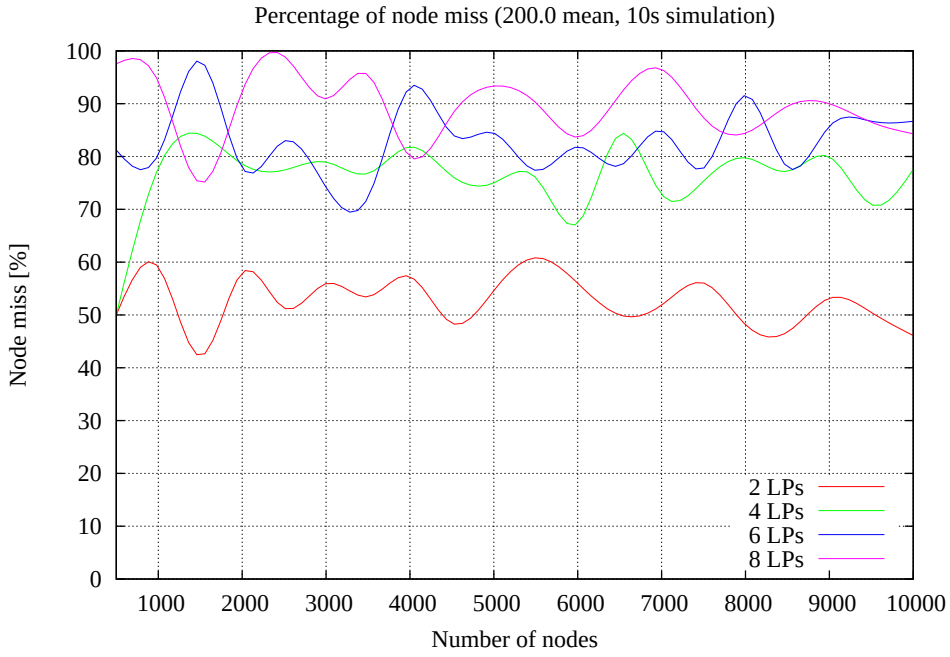


Figure 5: *Node-miss rate (fixed mean)*

As the graph shows we have also some not so good results, the framework seems to act like a random-network simulator instead of what we hoped:
with two LPs we have a about 50% of probability of delivery a message outside the local processor, with four LPs this probability become 25%, with six 16% and so on.
So we have expected to find lower probabilities of node-miss due to the structure of a scale-free network but we didn't keep in mind that by now the scale-free network exists only inside our framework as a data-structure, and SCAR doesn't guarantee that the scale-free property regards also the LPs distribution. The reason of these results therefore is that during the network creation SCAR doesn't have any preference about node links, it chooses connections only using the distribution function and it doesn't care about the node LP placement. This behavior can give a warranty on a correct network creation but introduce a trade-off for performance:

- We have to create an as coherent as possible scale-free network respecting the probability distribution provided by the user.

** BUT **

- We have to optimized the performance of the framework trying to distribute the nodes paying attention on parallel computation (we want to gather all the nodes around an hub on a single local processor).

By now I have decided to maintain the scale-free properties, without implement any kind of preferential policy on links creation, however I'm examining some ideas that may reduce this problem.

5.3 Network creation performance

Now it's time to talk about performance and then...let's start with benchmarks. A first interesting test regards the network creation, having defined a network synchronization algorithm a careful analysis is in fact necessary.
The simulation benchmark that has to be done is aimed to measure the goodness of the synchronization algorithm when varying the number of nodes and the number of local processor.
This first graph represents the behavior of the algorithm varying the number of nodes:

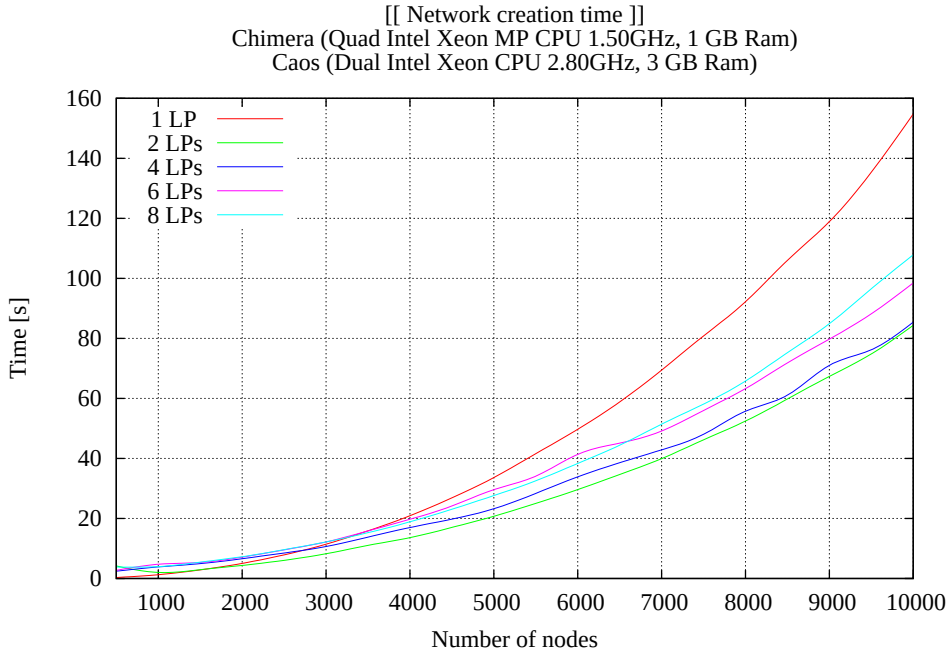


Figure 6: *Network creation performance (nodes-time)*

According to our previsions all curves have the same shape with the same polynomial slope caused principally by the computational cost of the sort algorithm, O(nlog(n)).
An entire network creation therefore has a computational cost of $O(n^2 log(n))$.
Increasing effectively performance, from this point of view, means to work only on the sort algorithm that, by now, is implemented via the *stable_sort()* of STL.

Comparing curves between them we can also observe that we have a great performance increase on passing from a single LP simulation to a dual processor simulation, however increasing the number of local processors even more cause a little but constant performance decay.

It must also be considered that increasing the number of LPs means to generate an enormous amount of synchronization traffic which decrease the performance so, by now, even though are possible a lot of optimizations, we can

consider this a fairly good result.

This second graph contains the same data, but highlighting the trend of creation performance varying the number of LPs:

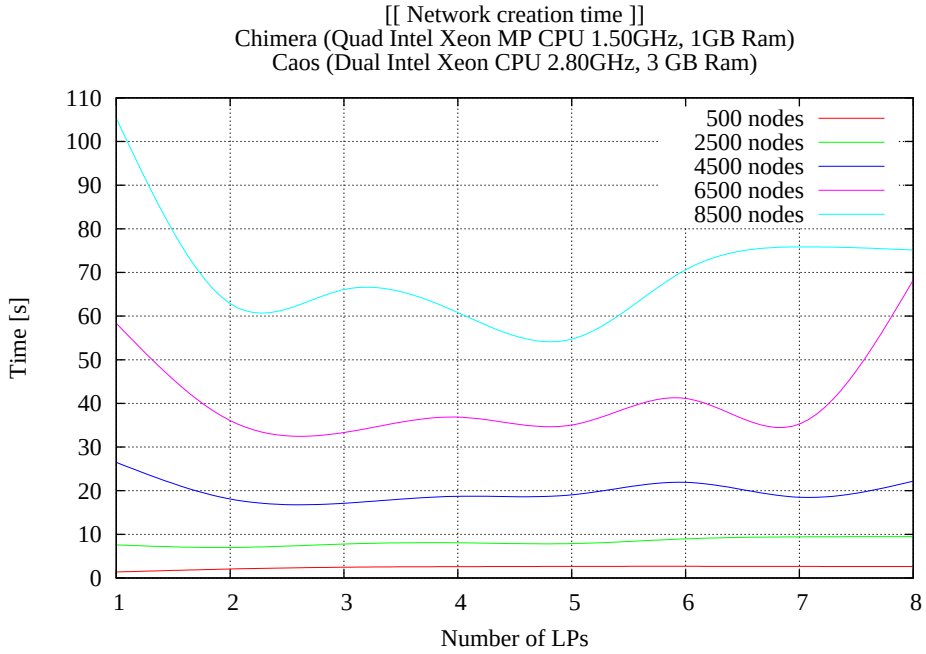


Figure 7: *Network creation performance (LPs-time)*

In this graph the considerable performance increase passing from one to two local processors is more evident, then adding even more LPs to the simulation we can note that performance are stable until 7/8 processors when the results became quite bad.
This behavior is easy to explain, in fact we are simulating over two machines: one dual processor and one quad processor; the total number of *real* parallel tasks is six, so its expectable a performance decrease when the computation is not so well distributed.

On the whole we can confirm a fairly good computational advantage on parallel calculus also regarding the network creation.

5.4 Simulation performance

To make some real simulation benchmarks I have written a simple example of framework usage which represents a model of epidemic, simulated over a network of sexual human relationships.
The results are the following:

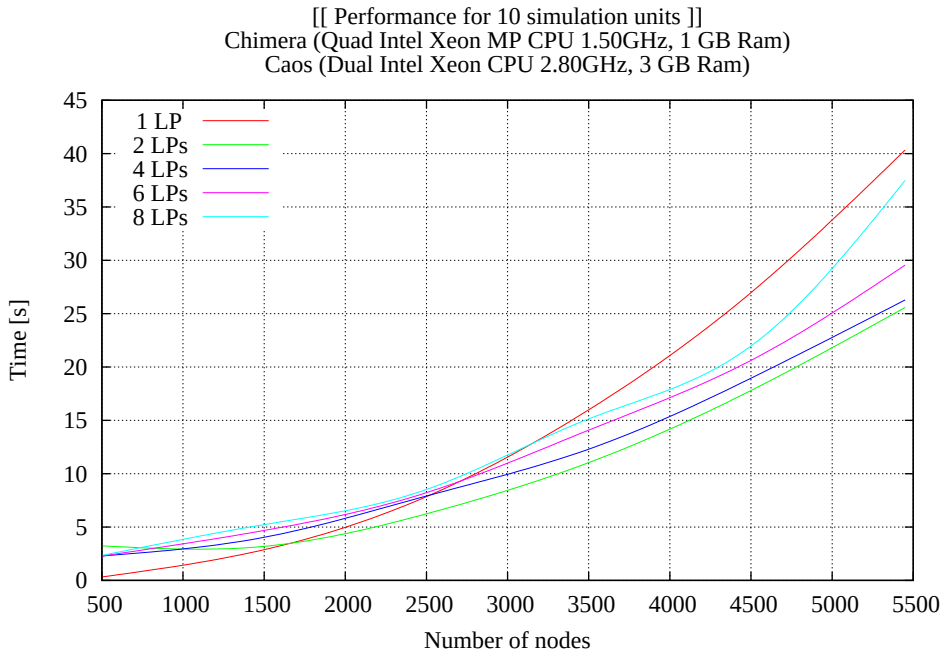


Figure 8: *Simulation performance (nodes-time)*

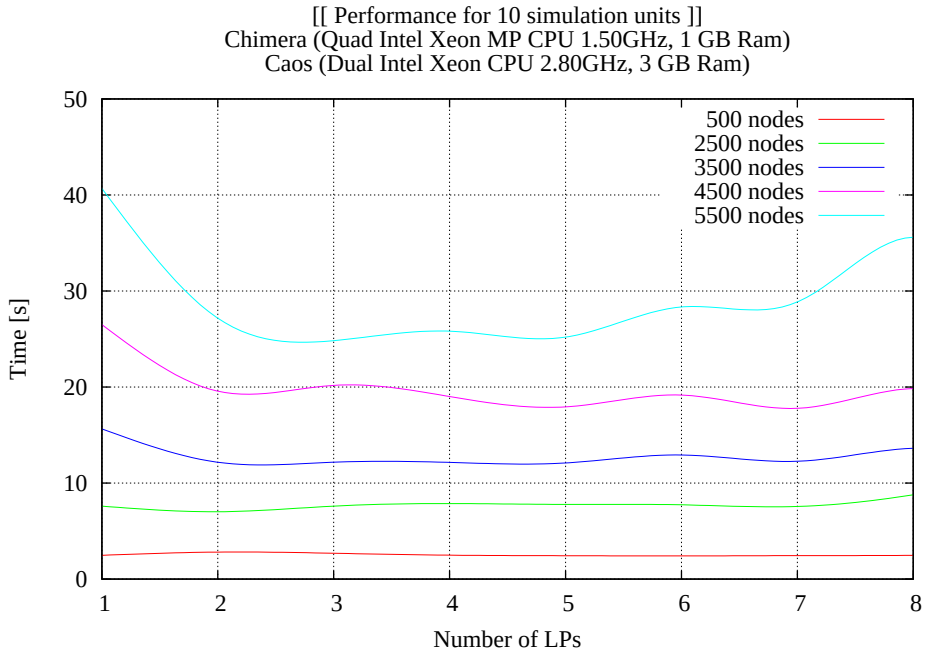


Figure 9: *Simulation performance (LPs-time)*

Performance directly related to the simulation are not so different from those regarding the network creation.
We can point out a considerable increase of performance passing from one to two LPs, then observe a little but constant decreasing.

So the same conclusion of above can be made examining the bad results with 7/8 local processor.
Results are the same as for creation: performance increases polynomially related to the number of nodes increment and also passing from one to two local processors (but decrease if we increasing LPs even more).

6 Conclusions

Even if a lot of work have to be done, I'm sactisfied of the actual results; it is proved that the correctness and coherence of the framework is fairly good, however expected performance increasing over two local processors was not happen. So the next work have to be focused mainly on performance increasing adding some creational policies in order to reduce the node-miss rate.
Possible other framework extensions regards provided probability distributions (random generators) and a dynamic visualization of simulated models.

Actually the *SCAR* framework have to be considered still in development, even if a lot of tests and benchmarks are made in fact, it probably contains bugs related to some not so used feature.

I hope that the work I have done may be useful for simulation purpouse in order to continue on its development and bugfix.
Concluding, I would thank to Prof. Lorenzo Donatiello and Prof. Luciano Bonini for the *Simulation* course and Prof. Gabriele D'Angelo for support and advice.

A Appendix A : discrete graphs

Data displayed with spline approximation in the previous chapters are presented here as discrete lines-points graphs for examination porpouse.

All my benchmarks and reports are generated from simulation runs over the PADS cluster available at the department of computer science and that includes the following machines running GNU/Linux 2.6:

- ***Cerbero*** : Dual Intel Xeon CPU 2.80GHz, HT, 1 GB Ram
- ***Chimera*** : Quad Intel XEON MP CPU 1.50GHz, HT, 1 GB Ram
- ***Caos*** : Dual Intel Xeon CPU 2.80GHz, 3 GB Ram
- ***Cassandra*** : Dual Intel Xeon CPU 2.80GHz, 3 GB Ram

A.1 Scale-Free property conservation

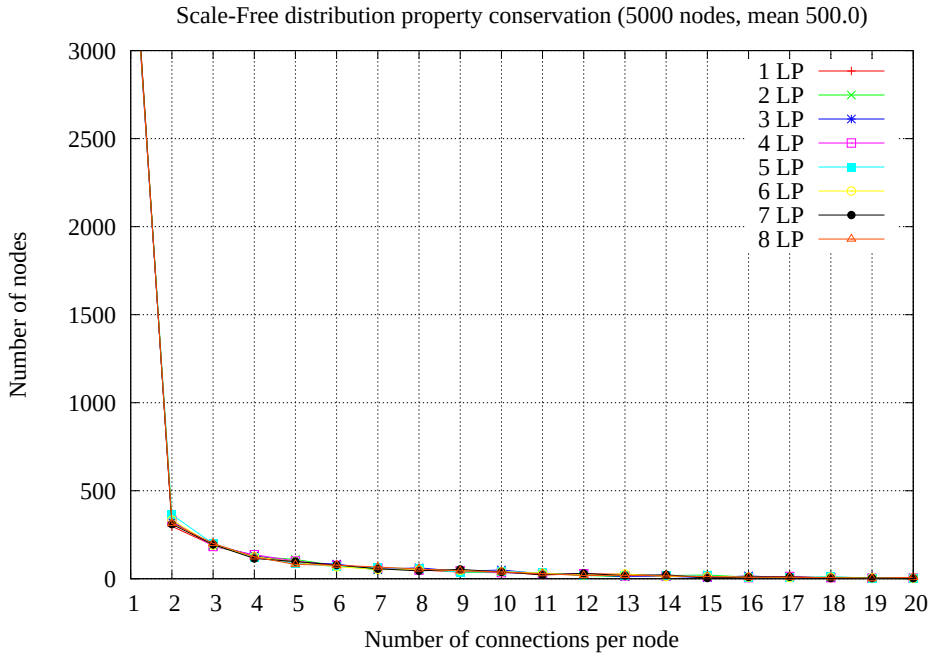


Figure 10: *Discrete node distribution*

A.2 Efficency of node distribution

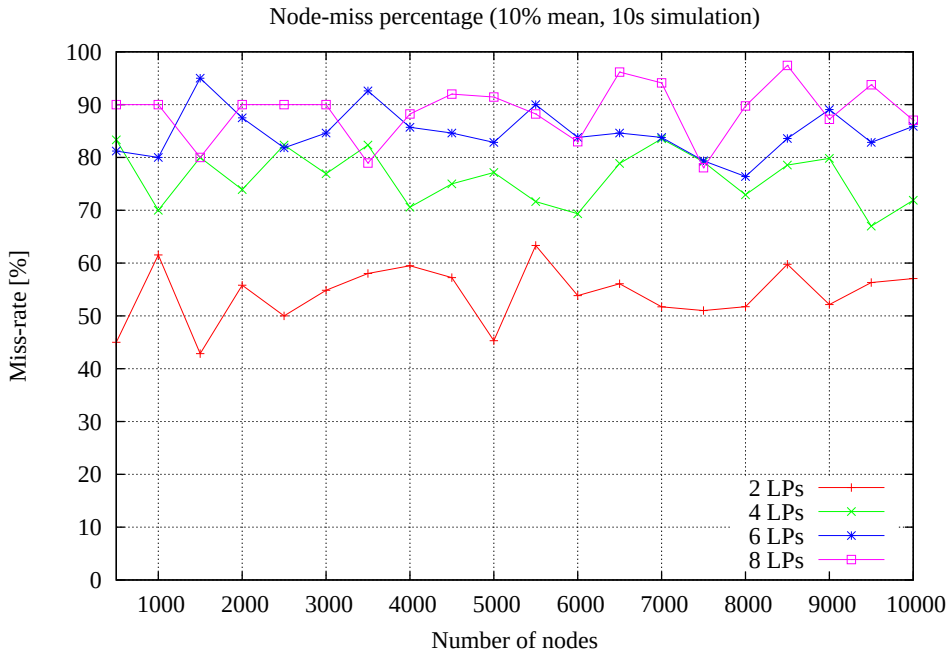


Figure 11: *Discrete node-miss rate (variable mean)*

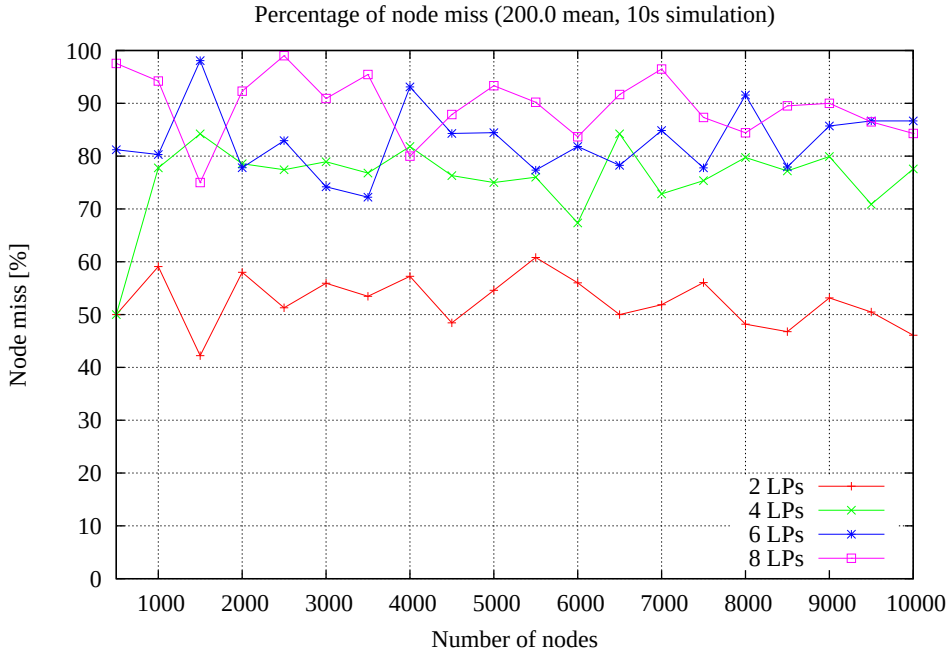


Figure 12: *Discrete node-miss rate (fixed mean)*

A.3 Network creation performance

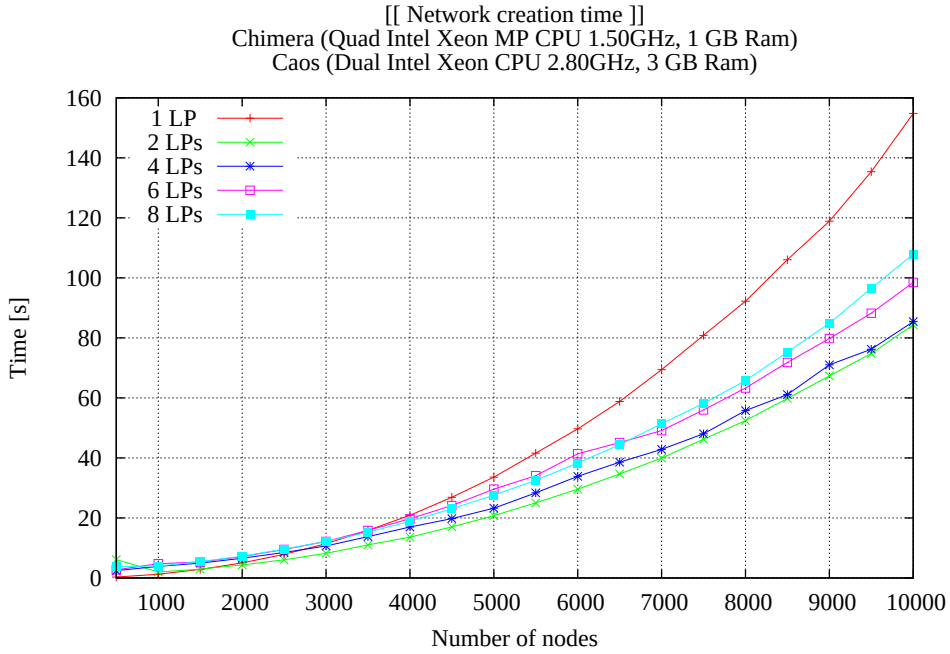


Figure 13: *Discrete creation performance (nodes-time)*

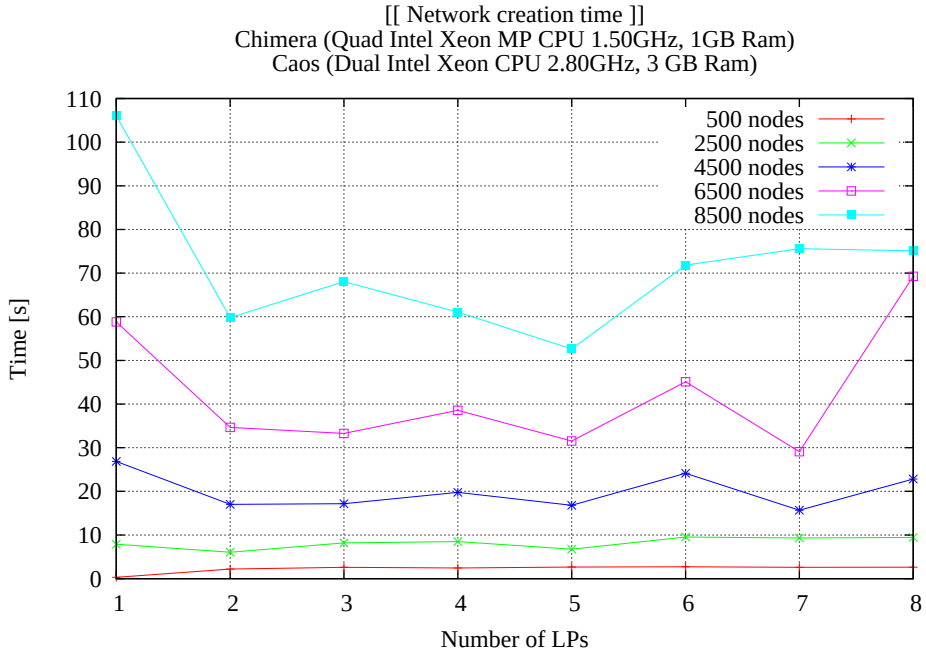


Figure 14: *Discrete creation performance (LPs-time)*

A.4 Simulation performance

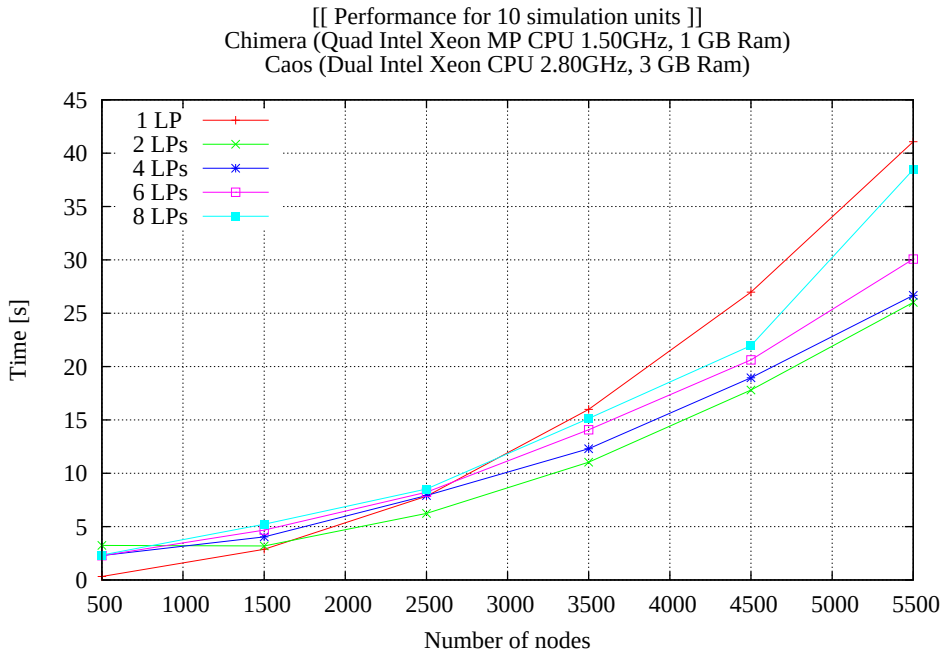


Figure 15: *Discrete simulation performance (nodes-time)*

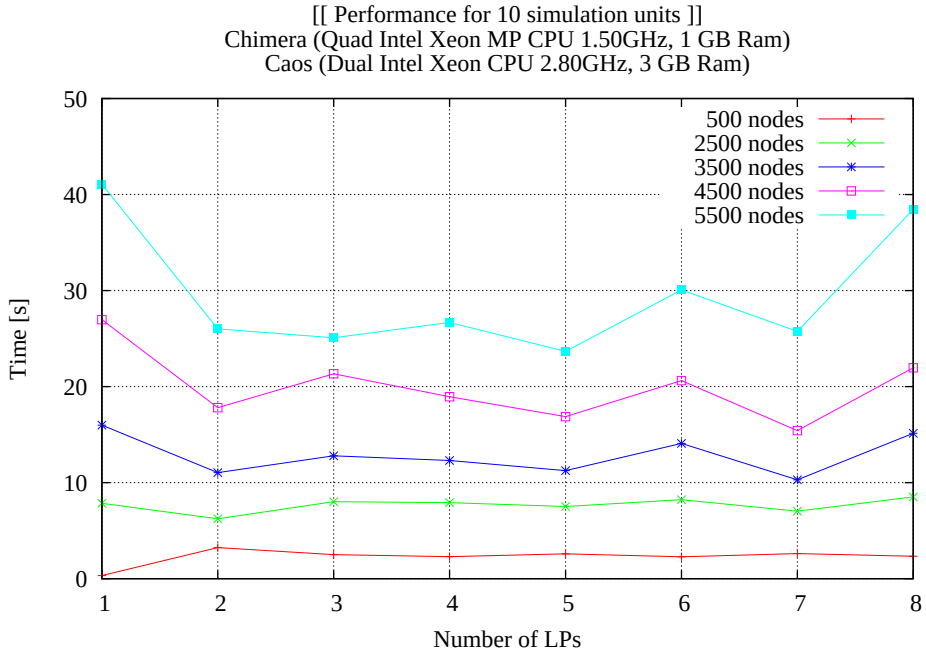


Figure 16: *Descrete simulation performance (LPs-time)*

B Appendix B : built-in generators

In this appendix I have reported some benchmarks regarding the random generators goodness. For each one of them I have produced 1,000,000 of numbers within some notable ranges and with various means and I have reported the significative ones.

B.1 Exponential generator

The exponential distribution is produced with the inverse transformation method, in fact supposing to have the following distribution function:

$$F(x) = \begin{cases} 1 - e^{-x/\beta} & if\ x \geq 0 \\ 0 & otherwise \end{cases}$$

it's easy, to find its inverse (let $u = F(x)$):

$$F^{-1}(u) = -\beta\ ln(1-u)$$

this permits to transform a uniform generator into an exponential one.
The following are the made benchmarks that confirm the correctness of the generator:

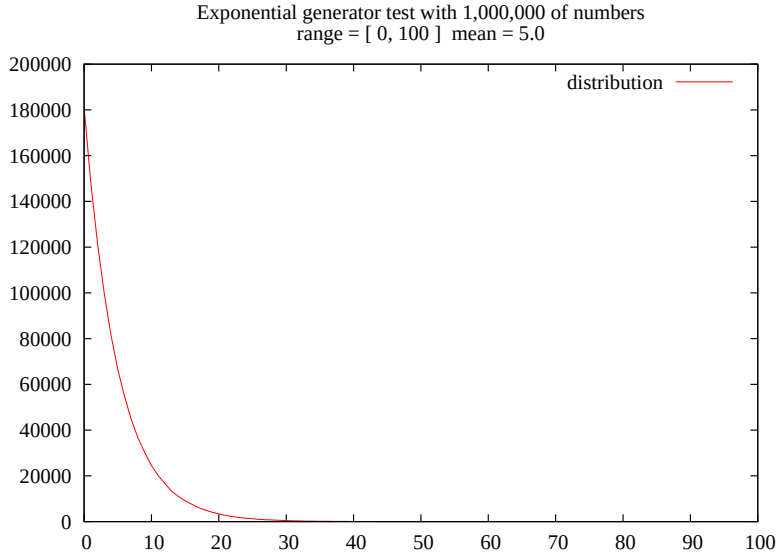


Figure 17: *Exponential distribution density*

B.2 Uniform generator

In this case I have simply adapted the system generator using a *modulo(n)* operation:

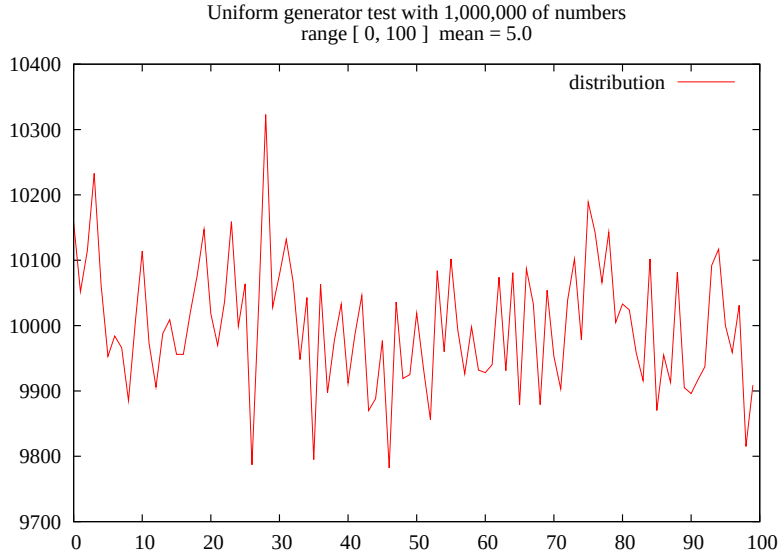


Figure 18: *Uniform distribution density*

B.3 Gaussian generator

In order to produce an as correct as possible normal (*Gauss*) distribution I have used the polar form of the *Box-Mueller* algorithm.
This method suppose to get two numbers x and y out from a uniform generator in the range [-1, 1], then compute the value $R = x^2 + y^2$ and if $R = 0$ or $R > 1$ throw them away and try another pair *(x, y)* else generate the following Gauss-distributed values:

$$z_0 = x\sqrt{\frac{-2\ ln(R)}{R}}$$
$$z_1 = y\sqrt{\frac{-2\ ln(R)}{R}}$$

Here's some benchmarks results;

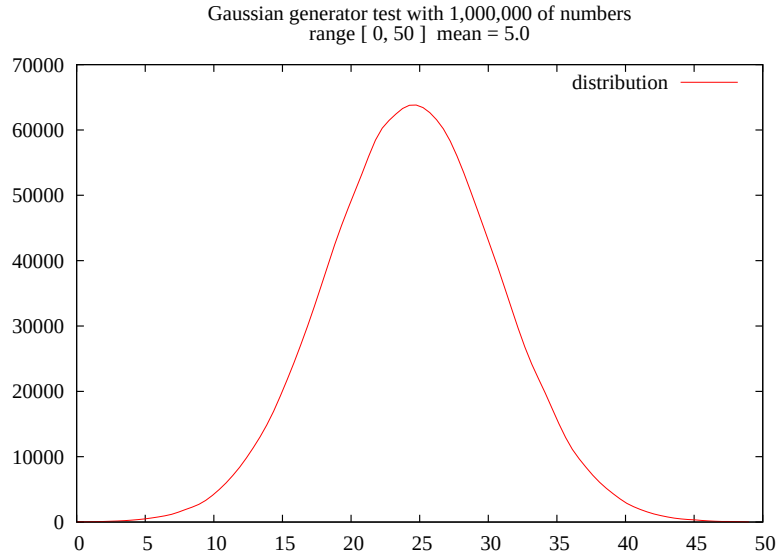


Figure 19: *Gaussian distribution density*

B.4 Poisson generator

A Poisson distribution generator is obtained through a boor algorithm using a graphical way. The main problem for the Poisson distribution is that it is a discrete distribution with usable positive values in the range [0, 10], so we have generated all the distribution function values at first, then we have stretched them as needed and used a raw linear interpolation to fill the numeric holes.
At this point we have use an inverse *numeric* method to retrieve the Poisson distributed number starting from the uniform one.
This way to proceed is so much slow and boor, it's also numerically unstable and, as we can see in the following graphs has a lot of problems with ranges great then its native ones ([0, 10]). For these reasons the Poisson generator provided by *SCAR* may considered a didactic experiment and its usage is highly thoughtless. The following is the Poisson generator behavior in the range [0, 10]:

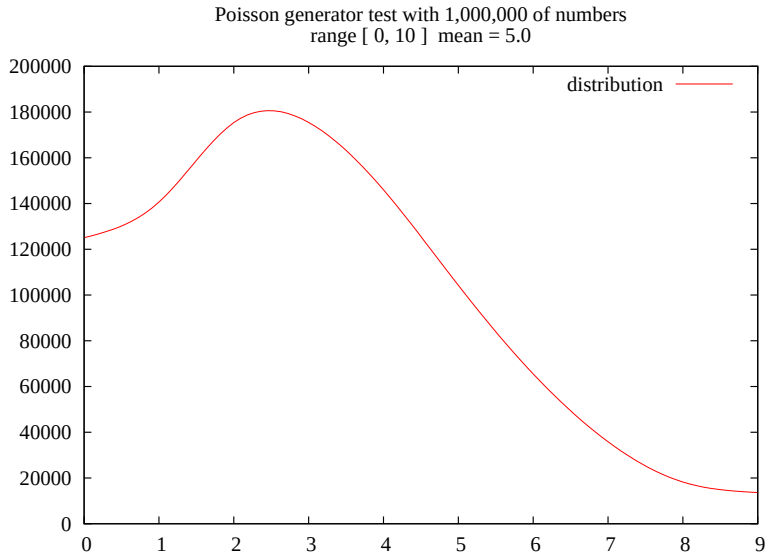


Figure 20: *Poisson distribution density with native range [0, 10]*

But increasing the range generates a lot of bad results:

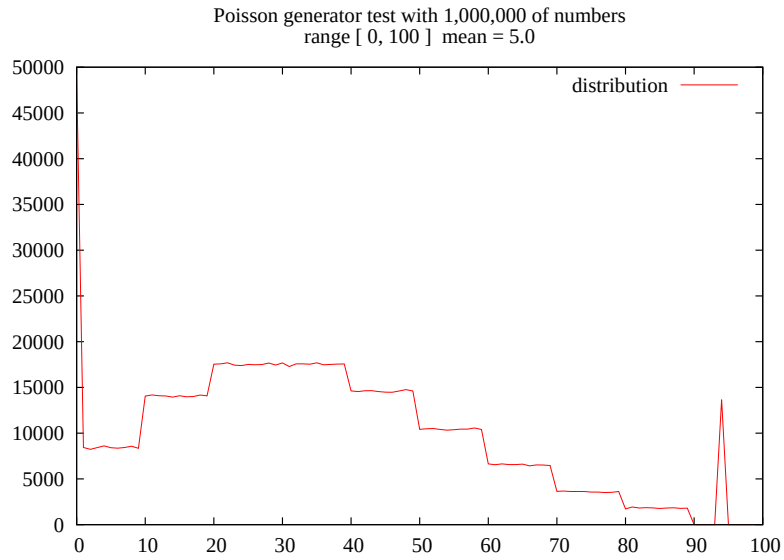


Figure 21: *Poisson distribution density with range [0, 100]*

... even though, with some immagination, it still seems a Poisson density:

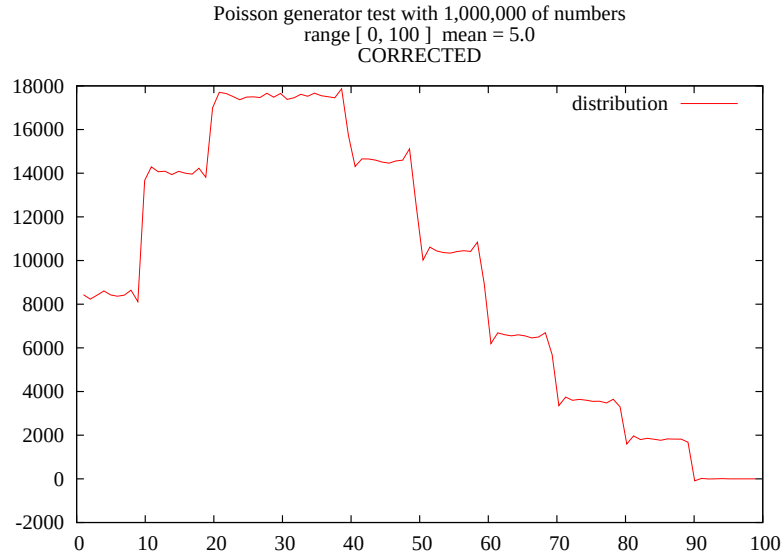


Figure 22: *Poisson distribution density with a raw numeric errors correction*

C Appendix C : some *ScarViewer* screenshots

C.1 An exponential distribution network

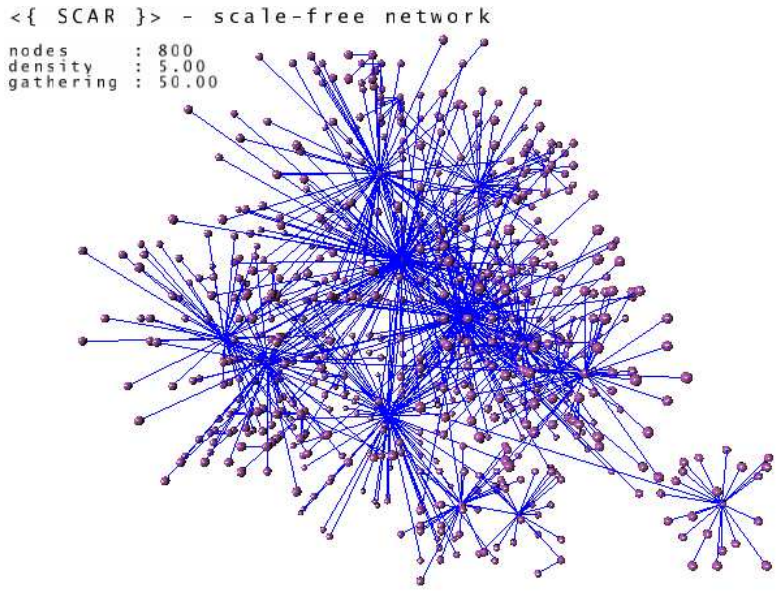


Figure 23: *Exponential network*

C.2 A uniform distribution network

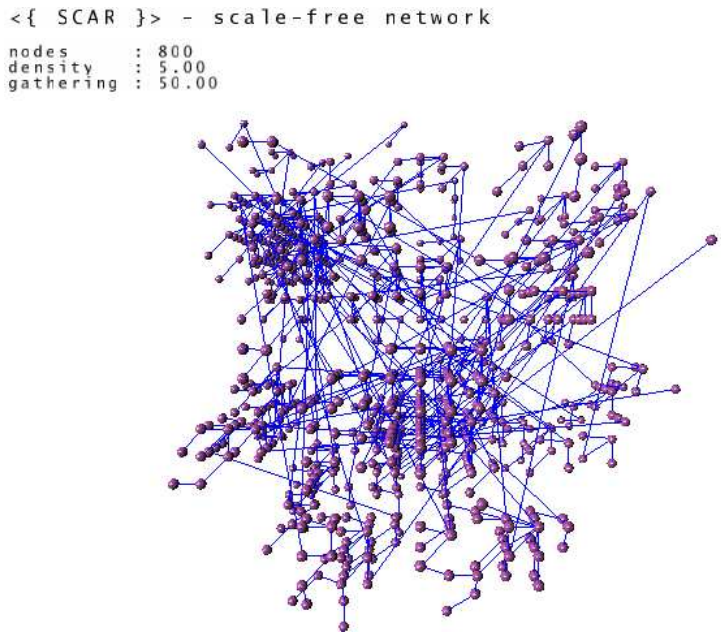


Figure 24: *Uniform network*

C.3 A Poisson distribution network

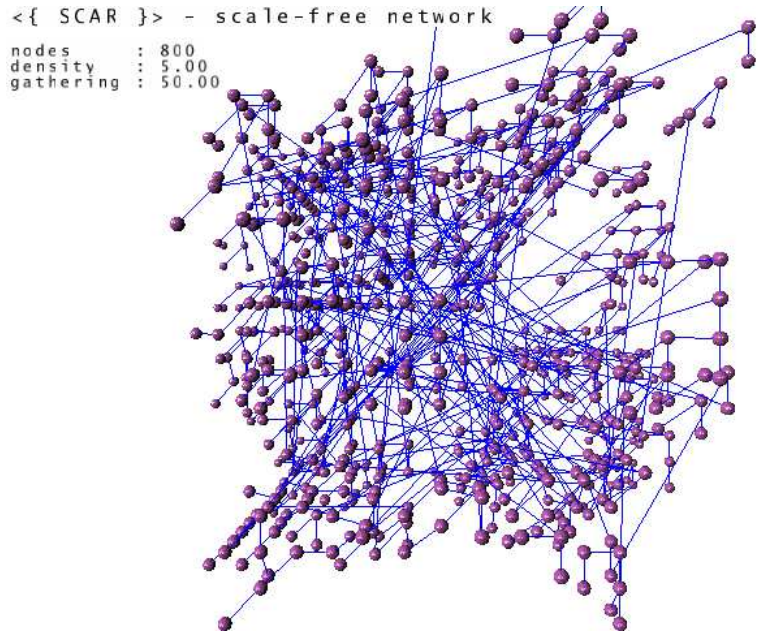


Figure 25: *Poisson network*

C.4 A gaussian distribution network

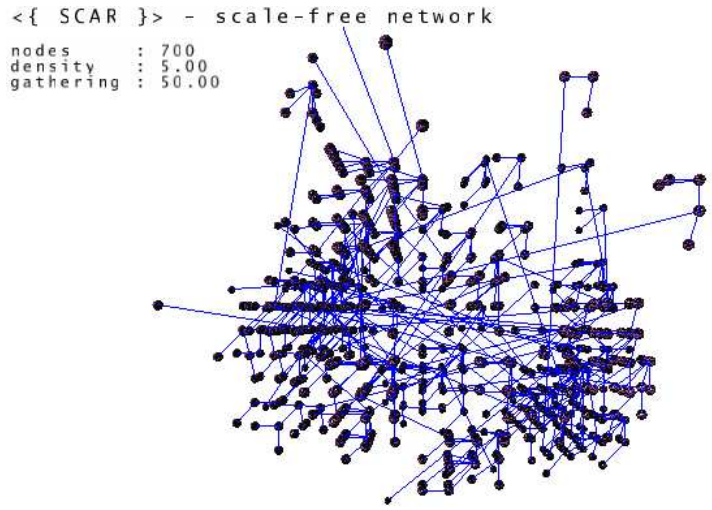


Figure 26: *Gaussian network*

MeTALAbS - building information subways >>

[www.metalabs.org]
[www.metalabs.org/hifi]
[hifi@metalabs.org]