MetaSimGrid : Towards realistic scheduling simulation of distributed applications

Arnaud Legrand, Julien Lerouge

Laboratoire de l'Informatique du Parallélisme

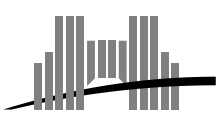
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

MetaSimGrid : Towards realistic scheduling simulation of distributed applications.

Arnaud Legrand,
Julien Lerouge

July 2002

Research Report N° 2002-28



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

MetaSimGrid : Towards realistic scheduling simulation of distributed applications.

Arnaud Legrand, Julien Lerouge

July 2002

Abstract

Most scheduling problems are already hard on homogeneous platforms, they become quite intractable in an heterogeneous framework such as a metacomputing grid. In the best cases, a guaranteed heuristic can be found, but most of the time, it is not possible. Real experiments or simulations are often involved to test or to compare heuristics. However, on a distributed heterogeneous platform, such experiments are technically difficult to drive, because of the genuine instability of the platform. It is almost impossible to guarantee that a platform which is not dedicated to the experiment, will remain exactly the same between two tests, thereby forbidding any meaningful comparison.
Simulations are then used to replace real experiments, so as to ensure the reproducibility of measured data. A key issue is the possibility to run the simulations against a realistic environment. The main idea of trace-based simulation is to record the platform parameters today, and to simulate the algorithms tomorrow, against the recorded data: even though it is not the current load of the platform, it is realistic, because it represents a fair summary of what happened previously.
A good example of a trace-based simulation tool is SimGrid, a toolkit providing a set of core abstractions and functionalities that can be used to easily build simulators for specific application domains and/or computing environment topologies. Nevertheless, SimGrid lacks a number of convenient features to craft simulations of a distributed application where scheduling decisions are not taken by a single process. Furthermore, modeling a complex platform by hand is fastidious for a few hosts and is almost impossible for a real grid. This report is a survey on simulation for scheduling evaluation purposes and present MetaSimGrid, a simulator built on top of SimGrid.

Keywords: simulation, scheduling, distributed application

Résumé

Alors que la plupart des problèmes d'ordonnancement sont déjà difficile sur une plateforme de calcul homogène, ils deviennent carrément insolubles dans un cadre hétérogène tel qu'une grille de metacomputing. Dans le meilleur des cas, on peut arriver à trouver une heuristique garantie, mais la plupart du temps, ce n'est même pas possible. On a donc souvent recours à des expériences grandeur nature ou à des simulations pour pouvoir comparer deux heuristiques. Cependant, dans le cas d'une plateforme de calcul hétérogène et distribuée, de telles expériences sont très délicates à mener en raison de l'instabilité latente de telles plateformes. Il est en effet impossible de garantir que l'état d'une plateforme de calcul qui n'est pas entièrement dédiée à l'expérimentation, va rester le même entre deux expériences, ce qui empêche donc toute comparaison rigoureuse.
On utilise donc des simulations afin d'assurer la reproductibilité des expériences. Une des difficultés majeures est d'arriver à simuler un environnement réaliste. Une approche efficace consiste à effectuer des simulations utilisant des traces, c'est-à-dire utilisant des enregistrements de différents paramètres de la plateforme pour obtenir un comportement réaliste.
SimGrid est un bon exemple d'outil de simulation utilisant des traces. Cette bibliothèque fournit un ensembles de fonctions et d'abstractions de bas niveaux permettant de construire facilement un simulateur pour un domaine particulier d'applications et/ou des topologies particulières de plateformes de calcul. Néanmoins, SimGrid souffre d'un certain nombre de lacunes, notamment en ce qui concerne la simulation d'une application distribuée où les décisions d'ordonnancement ne sont pas prises par une seule entité. De plus, si la modélisation d'une plateforme réaliste est déjà fastidieuses pour quelques machines, elle s'avère impossible quand le nombre de processeurs devient important. Ce rapport présente donc une étude sur la simulation ayant pour objectif l'évaluation d'heuristiques d'ordonnancement et présente MetaSimGrid, un simulateur construit au dessus de SimGrid.

Mots-clés: simulation, ordonnancement, application distribuée

1 Introduction

The current growth of Internet is mainly due to communication tools (mail, news) and information services [Net]. The Web can be seen, from a technical point of view, as a huge database, distributed on every server on the planet. The resources involved in the Web (and in the corresponding database) are mainly the storage space associated to hard disks and the network allowing every user to access them. The goal of *metacomputing* is to use all these resources to run applications which, a few years ago, could only be executed on dedicated parallel computers. Compute-bound applications such as SETI@home [SET] or Genome@home [Gen] have been the first metacomputing applications, but many other are currently being deployed. Internet is slowly turning into a virtual supercomputer, thanks to an efficient network connecting computing resources from research centers where parallel computers or clusters of stations are available.

However, although the physical framework already exists (VTHD [VTH], GTRN [GTR]), it still remains difficult to make an efficient use of a computing platform composed with hundreds of different processors, connected by heterogeneous, maybe not reliable, links. Most scheduling problems are already hard on homogeneous platforms, they become quite intractable in this new framework. In the best cases, a guaranteed heuristic can be found, but most of the time, it is not possible. Real experiments or simulations are often involved to test or to compare heuristics. The main advantage of real experiments is that they allow to test the behavior of an algorithm on a computer which is more complex than the model that was used to design the algorithm. However, on a distributed heterogeneous platform, such experiments are technically impossible to drive, because of the genuine instability of the platform. It is impossible to guarantee that the platform will remain exactly the same between two tests, thereby forbidding any meaningful comparison. Besides, testing the scalability of an algorithm requires a lot of computers or workstations, which, in most cases, are not available for long, periods of time.

For all these reasons, the only remaining approach to validate an heuristic is the simulation. This report presents MetaSimGrid, a simulator aiming at facilitating the design of scheduling algorithms on the grid. The strength of this simulator resides in its capacity to easily import and simulate real platforms.

2 Simulation

2.1 Framework

The future of parallel computing is best described by the key-words *distributed* and *heterogeneous*. Making use of distributed collections of heterogeneous platforms is the activity of *metacomputing*. At the low end of the field of distributed and heterogeneous computing, heterogeneous networks of workstations or PCs (HNOWs) are ubiquitous in university departments and companies, and they represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of all available resources, namely slower machines *in addition to* more recent ones. At the middle end of the field, architectures known as clusters of workstations (COWs) have also become a widely used alternative to super computers to perform distributed computing, due to an attractive price-performance ratio. A cluster is a set of processors connected with a local area network, which is often a high-performance network,

such as Myrinet or Giga-Ethernet. At the high end of the field, linking the most powerful supercomputers of the largest supercomputing centers through dedicated high-speed networks will give rise to the most powerful computational science and engineering problem-solving environment ever assembled: the so-called *computational grid*, which is nicely described in Foster and Kesselman's book [FK99].

Many environments have been developed which allow to write efficient distributed applications on each of these architectures. In order to get more processing power, it is common to connect several of those platforms with a wide-area network (in practice, the classic Internet connection). But writing and optimizing applications on such architectures is a difficult task, mainly because of the heterogeneity in communication and computation capabilities of the different components of the platform. This difficulty is increased by the fact that the wide-area links are often shared with Internet traffic from other applications, and their performance is not as constant and reliable as the one of a dedicated COW. In the same way, HNOWs are often non dedicated so the performances of the computers and of the network may vary during the execution of the application. Because of this, comparing two different algorithms or implementations cannot be done without very careful precautions. Testing and optimizing applications requires a way to run them in a reproducible way, which allows for precise scientific comparisons. But these applications also have to be tested against realistic situations, in which the network and the CPU availability can show performance loss. Therefore we need to use a simulator i) to have verifiable and reproducible results and ii) to test the algorithms in a variety of conditions.

Most currently available results on scheduling are obtained with restrictive assumptions. Computing and networking resources are often considered homogeneous and assumed to deliver constant performance throughout time. Other assumptions include infinite numbers of available processors, or particular network topologies (e.g. fully connected). Even though some scheduling algorithms specifically focus on heterogeneous environments, it is almost always assumed that a scheduling algorithm is able to obtain perfect prediction concerning the execution time of each task on each resource. These assumptions are not compatible with the actual characteristics of real systems. Network topologies in real systems are usually more complex than the ones simulated in most scheduling work. Also, grid applications use resources that exhibit dynamic performances because they are time- or space-shared. Furthermore, due to this dynamic aspect, it is not possible to obtain exact predictions for task execution times; actual implementations of scheduling algorithms will use inaccurate predictions. The impact of such inaccuracies on the efficacy of the various scheduling algorithms has not been the object of extensive studies. Hence, there is a clear need for new simulation tools that allow for more realistic resource models and for more realistic scheduling functions. Although some simulators already exist, they are not targeted to the simulation of distributed applications, for the purpose of evaluating scheduling algorithms. These tools are often very complete and sophisticated but too complex and too lowlevel for our intended purpose.

2.2 Related Work

Several software prototypes have already been developed to create environments allowing to simulate different kinds of resources, from a simple network to a whole distributed system. In this section, we present some of them and explain why they do not suit our purposes.

2.2.1 Network simulators

In order to help understanding how large networks may react to various situations, several network simulators have been developed [PF97]. These simulators, such as NS, NSE [BBE+99] or DaSSF [LN01], focus on a precise simulation of the packets traveling on the network, rather than the network behavior as it is observed by an application. They are intended to help developing faster or more adaptive network protocols. But they are not interested in the applications using these protocols, and the application is often merely seen as a packet emitter. It might be possible to simulate real applications with such simulators, as in DaSSF, but it would involve rewriting the application to fit the expected description, and the calculation times between sending or receiving calls would be neglected in the simulation (i.e., take zero simulation time).

Moreover, external load has to be simulated by creating random artificial sources of traffic. If some statistical results on the topology of a Wide Area Network are available (see Section 2.2.3), they are still difficult to exploit in order to design a realistic platform with those simulators. Modeling the external load on such a platform is even more complicated than on a Local Area Network (just think about the number of connections per hour over the internet). At last, these simulators are really slow since they simulate precisely the packets traveling on the networks. Then, increasing the number of connections to simulate the external load slows down even more the simulation.

2.2.2 Large platform simulator

LAPSE LAPSE (Large Application Parallel Simulation Environment) [DHN94] is a tool designed to simulate parallel applications. The motivation is mostly to allow researchers to run their applications on a small number of nodes, because large parallel computers are not always easily available, or in order to test the scalability of their algorithms. The emphasis is put on a precise network event simulation, which is done by other processes than the processes executing the code of the application.

In order to improve locality, the simulator and application processes are parallelized over the available nodes, and each communication request in the application code results in a communication between the application and the corresponding simulator process. The application code is instrumented so as to count the number of instructions executed between two consecutive communication calls, which allows to know how much simulated time has passed between these two calls.

The LAPSE simulator is aimed at applications written for the Intel Paragon machine, even though it allows to simulate these applications on other kinds of machines as well, such as clusters of workstations, which are more easily available. However, the network is simulated with the Paragon in mind, which leads to a single-cluster simulation where only a latency penalty is used to calculate the arrival time of a message at its destination. This time does not depend on the size of the message, and no contention is assumed. The network simulation is then too specific to a particular machine type to suit our needs and the simulation of external load has not been taken into consideration.

The MicroGrid The MicroGrid [SLJ+00] is part of the GrADS (Grid Application Development Software) project, whose goal is to simplify distributed heterogeneous computing. The main functionality of the MicroGrid is to allow Grid users (programmers of Grid

applications) to simulate their application on a virtual Grid, the behavior of which is much more controlled than a real one.

This software allows to virtualize every resource of a classical Grid, namely memory, computing and networking resources. Just as in LAPSE, this virtualization is achieved by trapping every relevant library call (socket libraries, gethostname, ...). But the simulation is not as strict than the one performed in the LAPSE software. Instead of precisely counting the executed instructions, the computing resources are simulated using a local MicroGrid scheduler, which allocates CPU time slices to each process according to a predetermined simulation rate. The network is simulated through a modified version of the real-time VINT/NSE [BBE+99] simulator, which mediates all communication; but the network configuration is part of the virtual Grid, and is not supposed to change over time.

In this simulation tool, since the distributed platform is simulated (by mapping it on a local cluster) and the application is effectively run, the ratio between the simulated time and the duration of the simulation cannot be very large. Moreover, it does not seem to be possible to simulate easily a realistic external load.

Panda The Albatross project focus on high performance applications, running in parallel on multiple clusters or MPPs that are connected by wide-area networks (WANs). The original goal of the Panda library [CWK+96] is to provide an efficient portability layer for parallel applications or runtime systems. Some recent work on simulating the variable load using this library has been performed [KBM+]. Following a different approach from the previous two projects, they do not use a simulator for the network. They use real links and slow them down artificially to simulate the behavior of WANs. The simulation takes place inside the communication library, and only in selected places, since they simulate only wide-area messages. Thus, they can enforce wide area communications to go through some dedicated gateway machine that put them into delaying queues in order to simulate slower links.

Contrarily to the previous two projects, it is then very easy to change the network behavior during the simulation. Nevertheless, that kind of tools is not designed for the purpose of evaluating scheduling algorithms since the application is not simulated but run. Furthermore, the simulated platform is very specific to their needs, i.e. a computing platform where heterogeneity occurs only in WAN links.

2.2.3 Topology Generators

Most simulations from the literature in the field have used representations of a real topology (e.g. the Arpanet or a well-known provider's backbone), simple models (e.g. a ring or a star), or randomly-generated flat topologies using a variation of Waxman's edge-probability function. Recently, more complex randomly-generated hierarchical models have been used to better approximate the Internet's hierarchical structure. Many studies on the internet topologies are available [Doa96, CDZ97, FFF99, MMB00]. Using snapshots of the Internet, some simple power-laws (outdegree of node versus rank, number of nodes versus outdegree, number of node pairs within a neighborhood versus neighborhood size (in hops), eigenvalues of the adjacency matrix versus rank,...) of the Internet topology have been discovered. Many public-domain generators create random graph following these laws [NuR].

If these generators constitutes an excellent starting point for building a realistic platform, we still lack several informations. There is no information on the traffic (i.e. the available bandwidth as time goes) and no information on the *behavior* of these links (capability, LAN/WAN, ...). Common modeling techniques involve the use of simulated sources of traffic using random laws for which no confrontation with the reality is conducted. Then, due to the variety of sources and to the continual development of new network users, such simulated sources of traffic are not adequately representative. In a test environment that uses *real* rather than *simulated* traffic sources, testing scheduling algorithms against the latest traffic and network behavior is simplified. At last, several parameters have to be fixed by hand to obtain realistic platforms. It requires a high experience or a good intuition (or both) in network modeling.

3 SimGrid

3.1 History

The Grid Research and Innovation Laboratory (GRAIL [GRA]) evolved from its predecessor, the Grid Computing Laboratory (GCL). GCL was formerly the AppLeS group(see [BW97]) which was part of the Parallel Computing Laboratory at UCSD's Computer Science and Engineering Department.

The AppLeS project was dedicated to the achievement of performance for metacomputing applications, a difficult goal due to the heterogeneity of metacomputing platforms. This is especially true for parallel applications whose performance is highly dependent upon the efficient coordination of their constituent components. Currently, to achieve a performance-efficient implementation on a distributed heterogeneous system, the application developer must select a potentially efficient configuration of resources based on load and availability, evaluate the potential performance on such configurations based on their own performance criteria, and interact with the relevant resource management systems in order to implement the application. This approach is termed *application-centric* by Berman and Wolski [BW97].

The AppLeS (Application Level Scheduler) project has been developing application-level scheduling agents to provide a mechanism for scheduling individual applications at machine speeds on production heterogeneous systems. The following statements have been used as guidelines to implement the agents :

- Both application-specific and system-specific information are required for good schedules;
- Performance depends upon the application's own performance criteria;
- The "distance" between resources is dependent upon how the application uses them;
- Dynamic information is necessary to accurately assess system state;
- Predictions are accurate only within a particular time frame;
- A schedule is only as good as its underlying prediction;

The AppLeS application-level scheduling paradigm addresses heterogeneity by explicitly assuming that all resources (even homogeneous resources) exhibit individual performance characteristics, and that these performance characteristics may vary over time. Contention is

addressed by assessing the fraction of available resources dynamically, and using this information to predict the fraction available at the time the application will be scheduled.

The GRAIL focuses on the scheduling and the deployment of distributed scientific applications in the context of the Grid. As a result, a large part of their work focuses on adaptive scheduling and they target many applications and application classes. They are also doing work in the area of simulation in order to validate and evaluate their scheduling results. Building upon this work, Henri Casanova has designed a simulation toolkit that can be used to easily build simulators for specific application domains and/or computing environment topologies : SimGrid [Cas01].

3.2 A simulation toolkit

Unlike most simulation tools, SimGrid is not a simulator: it is a simulation toolkit. It provides a set of core abstractions and functionalities that can be used to easily build simulators for specific application domains and/or computing environment topologies. SimGrid performs event-driven simulation. The most important component of the simulation process is the resource modeling. The current implementation assumes that resources have two performance characteristics: latency (time in seconds to access the resource) and service rate (number of work units performed per time unit). SimGrid provides mechanisms to model performance characteristics either as constants or from traces. This means that the latency and service rate of each resource can be modeled by a vector of time-stamped values, or trace. Traces allow the simulation of arbitrary performance fluctuations such as the ones observable for real resources. In essence, traces are used to account for potential background load on resources that are time-shared with other applications/users. This trace-based model is not sufficient to simulate all behaviors observable in real distributed systems since the actions of our schedulers might influence the decision of others, thus changing the background load. However, this first implementation was a first step in that direction and is already a great improvement over current simulation practice for the evaluation of scheduling algorithms.

SimGrid provides a C API that allows the user to manipulate two data structures: one for resources (SG_Resource) and one for tasks (SG_Task). SimGrid does not make any distinction between data transfers and computations: both are seen as tasks and it is the responsibility of the user to ensure that computations are scheduled on processors and file transfers on network links. A resource is described by a name, a set of performance related metrics and traces or constant values. For instance, a processor is described by a measure of its speed (relative to a reference processor), and a trace of its availability, i.e. the percentage of the CPU that would be allocated to a new process; a network link is described by a trace of its latency, and a trace of its available bandwidth. A task is described by a name, a cost, and a state. In the case of a data transfer the cost is the data size in bytes, for a computation it is the required processing time (in seconds) on the reference processor. Finally, the state is used to describe the task's life cycle (not scheduled, scheduled, ready, running, completed). SimGrid's API provides basic functions to operate on resources and tasks (creation, destruction, inspection, etc.), functions to describe possible task dependencies, and functions to schedule and unschedule tasks on resources. Only one function is necessary to run the simulation: SG_simulate(). This function leads tasks through their life-cycle and simulates resource usage. It is possible to run the simulation for a given number of (virtual) seconds, or until all/any/one task(s) complete(s). In all cases, SG_simulate() returns a list of tasks that have completed since the last time it was called. It is possible to perform post-mortem analysis of the application's execution by

inspecting task start and completion times, as well as the resources that were used.

An important question is that of the topology of resource interconnections. A fully-connected topology is often assumed in scheduling algorithm works. Non-fully-connected topologies can then be seen as special cases of the fully connected topology, and are addressed by some scheduling algorithms. One way to implement SimGrid would have been to enforce a fully-connected topology (e.g. one network link for each pair of computing resources). The user could then specify which connections are actually enabled. This is typically implemented with a connection matrix where some entries are zeroed out. There is one major drawback to that approach: the fully-connected model fails to model realistic distributed computing environments. In a real environment, network links are usually shared by communication tasks. Consider two distinct local area networks, say A and B, on the Internet. Two processors in A communicating with two processors in B typically share a common Internet link. The fully-connected model assumes dedicated links between communicating processors and would overestimate the achieved data transfer rate by a factor of 2 in our example. Along the same lines, the fully-connected model prevents the simulation of complex network topologies with multiple network links between any two resources (e.g. with intermediate routers). In the SimGrid approach, no interconnection topology is imposed. Instead, SimGrid treats CPUs and network links as unrelated resources and it is the responsibility of the user to ensure that his/her topology requirements are met. For instance, one can create multiple links in between hosts (or group of hosts) to simulate the behavior of simple routers. This approach is very flexible and makes it possible to use SimGrid for simulating a wide range of computing environments.

At the time this paper is being written, at least two simulators have been implemented using SimGrid. This paragraph provides brief descriptions of these two projects and draws conclusions concerning usability. The first application of SimGrid is a simulator, PSTSim, whose goal was to evaluate scheduling strategies for parameter sweep applications over the Computational Grid. The Grid model is as follows: a set of sites where hosts are available for computation; sites are interconnected among each others with network links; hosts within a site have access to a shared storage device and there is no other intrasite communication. The applications are usually longrunning and consist of large numbers of independent tasks with the added complexity that tasks may share large input files. The scheduling problem is then to place those files strategically in shared storage so as to minimize data transfers. A full description of the models and of the scheduling issues can be found in [CLZB00]. In that paper, PSTSim was used to compare 5 scheduling algorithms over a large space of Grid configurations and application specification. The results obtained led to an implementation of some of those scheduling algorithms in a userlevel Grid middleware package, APST, described and evaluated in [COBW00]. The second simulator based on SimGrid, DAGSim [JCB00], is focused on evaluating scheduling algorithms for applications structured as DAGs. The model is quite different from the previous simulator, both in terms of computing environment and application, but SimGrid's flexible approach was equally amenable to both simulators. This simulator has been used in [SCB02] to study a number of well-known algorithms (DLS, DSC, ETF, MCP, HDLFET as listed in [KA99]) and implement adaptive versions for each one of them. The current interest here is in studying the impact of resource performance prediction errors on those algorithms in order to determine which ones are practical in Computational Grid settings. Both PSTSim and DAGSim were straightforward to implement because SimGrid provides all core functionalities that address the simulation details. Therefore, it is possible to focus mostly on the implementation of the scheduling algorithms.

To conclude, the main qualities of SimGrid are its speed and its flexibility. The ratio of SimGrid CPU time over simulated time is about 10^{10} and almost any computing platform can be modeled, albeit it is the responsibility of the user to do it. Some point should nevertheless be clarified. Simulations set up using SimGrid do not aim at reproducing reality in the best possible way. It should not be expected either to use SimGrid to know the execution time of some application on a particular computing platform. SimGrid is an implementation of a more realistic model than the basic one where resources deliver constant performance. Thus, SimGrid-based simulations rather make it possible to give a serious test to a scheduling algorithm by running it in conditions that try to mimic reality. Since scheduling algorithms are generally designed with a theoretical model of the computing platform in mind, this test aims at detecting unexpected behaviors or at comparing two heuristics in a more realistic way.

3.3 Simulating a distributed application on a metacomputing grid

As explained in section 3.2, SimGrid provides an excellent framework for setting up a simulation where decisions are taken by a single scheduling process. However, in metacomputing systems, applications cannot be efficiently scheduled by a global scheduling mechanism. Centralized schedulers suffer a wide variety of problems: they are less fault-tolerant than distributed ones, they do not scale very well[CDL+02], scheduling politics may depend on computing platforms[LLM88, LR99]. Hence, being able to design a simulation where many independent decisions are taken at the same time is essential.

To cope with scalability issues, the number of schedulers may not be fixed in advance. Indeed, exactly as a centralized scheduler may become a bottleneck when it has too many tasks to process and too many resources to manage, there might be not enough processes to face the situation. Then a scheduler might decide, for example, to create some other schedulers on other machines to delegate them part of the work. Moreover, as networks are highly hierarchical, the *location* of the schedulers has a great impact on the performance of the overall architecture. One can imagine some relocation and/or forking depending upon the measured performance of the platform[CDL+02].

To achieve reasonable performance, distributed schedulers need to cooperate. For this reason, it is absolutely necessary to be able to easily simulate communications between those schedulers involved by a decision taking. We must also keep in mind that simulation aims at easing the work of scheduling algorithm designers. Thus, to avoid unrealistic simulations, the API should be lightweight and close to reality. Since most scheduling algorithms rely on a concept of task that can be either computed locally or transferred on another processor, it seems to be the right level of abstraction for our purposes. Last but not least, the wide variety of metacomputing platforms enforce the need for an easy way to set up the computing platform.

4 MetaSimGrid

As explained in Section 3.3, SimGrid lacks a number of convenient features to craft simulations of a distributed application where scheduling decisions are not taken by a single process. We decided to build a simulator using SimGrid that offers these features: MetaSimGrid. We will not fully describe the API here but rather present some general concepts and illustrate the main features with a short example. Next, we will explain more in details how

concurrency is managed and how file transfer are modeled. We will conclude this section by showing that almost any programming style can be accomodated.

4.1 General concepts

To reach the goals presented in Section 3.3, the following notions are essential.

Process. We need to simulate many independent scheduling decisions, so the concept of *process* is at the heart of the simulator. A process may be defined as a *code*, with some *private data*, executing in a *location*.

Location. A *location* (or *host*) is any possible place where a process may run. Thus it may be represented as a *physical resource with computing capabilities*, some *mailboxes* to enable running process to communicate with remote ones, and some *private data* that can be only accessed by local process.

Task. Since most scheduling algorithms rely on a concept of task that can be either *computed* locally or *transferred* on another processor, it seems to be the right level of abstraction for our purposes. A *task* may then be defined by a *computing amount*, a *message size* and some *private data*.

Link. There is no routing in SimGrid and our simulator needs a much higher level API than that of SimGrid : the scheduler needs to rely on the logical organization, not on the physical layout of the computing platform. A *link* is then an agglomeration of communicating resources representing a set of physical network links.

Channel. For convenience, the simulator provides the notion of channel that is close to the TCP port notion.

As mentioned in section 3.3, we want to keep the API as simple as possible and as close as possible to reality to avoid unrealistic simulations. This is why the scheduling should be described only in terms of *processes*, running on some *locations* and interacting by *sending*, *receiving* or *processing tasks*. A scheduler should not have direct access to *links* but rather should send a *task* to another *location* using a *channel*. In fact, a *location* may have many *mailboxes* and a *channel* is then simply a *mailbox* number. So sending a *task* to a *location* using a *channel* amounts to transfer the *task* on a particular *link*, depending on the emitter location and on the destination, and to put it in a particular *mailbox*.

4.2 Working out an example

Each of the object described above is organized in the same way. It is a represented as a C structure with a name, some simulation data, and some user data that are at the charge of the user. Simulation data are hidden and only MetaSimGrid functions can access them. A *process* is then represented by a m_process_t, a *location* by a m_host_t, a *link* by a m_link_t, and a *task* by a m_task_t. Thus m_host_t is simply defined as follows, and the user may put any information in the data field.

```
1 typedef struct s_m_host {
2   char *name;                   /* host name if any */
3   void *simdata;                /* simulator data */
```

```
  void *data;                   /* user data */
} *m_host_t;
```

4.2.1 A very simple experiment

Suppose we want to simulate the following situation. We have a master and three workers. The processors on which the master and the workers are executing are all connected with the same compound link (see Figure 1). The master will be granted 9 tasks to perform at the beginning of the simulation and will distribute them to its slaves using a simple round-robin assignment.

The choice of this experience may surprise since it could have been conducted using SimGrid only. Nevertheless it is a very simple example and it is sufficient to show the main functionalities of MetaSimGrid.

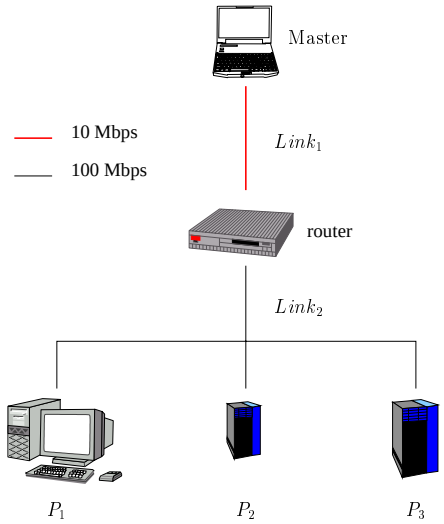


Figure 1: A very simple platform

4.2.2 End-user data

The first thing to figure out is the kind of information a process will need to execute properly. The master needs to have some work to distribute and needs to know on which host it can delegate some of its work. This information may be gathered in the following C structure.

```
typedef struct s_sched_data_process {
  m_host_t *slaves;             /* Array of other hosts you know */
  int slaves_count;

  m_task_t *todo;               /* Dynamic array of tasks you own */
  int todo_count;
  int todo_max_size;
} *sched_data_process_t;
```

We will assume this type has been defined along with the creation function:

```
sched_data_process_t sched_data_process_create()
```

A channel being basically an integer (since it is a mailbox number as mentioned in Section 4.1), we define the following type for convenience.

```
typedef enum {
  PORT_22=0,
  MAX_CHANNEL
} channel_t;
```

Note that, since these types and this function are very dependent on the simulation to set up, they are not provided by MetaSimGrid,and they have to be written separately.

4.2.3 Setting up the platform

All MetaSimGrid simulations should look like this:

```
#include "metasimgrid.h"
#include "user_definitions.h"

#define NB_SLAVE 3
#define NB_TASK 3

void master(void *t)
{...
}

void worker(void *t)
{...
}

void test_all()
{
  /* Variables */...
  {                              /*  Simulation setting */...
  }
  {                              /*  Host creation */...
  }
  {                              /*  Link and topology creation  */...
  }
  {                              /*  Process creation */...
  }
  {                              /*  Process organization */...
  }
  {                              /*  Task creation */...
  }
  if(MSG_main()!=MSG_OK) exit(1);
}

```

```
33 int main()
34 {
35   test_all();
36   return (0);
37 }
```

The functions master and worker describe the behavior of the process running on each location. The test_all function builds the platform, creates the processes, prepares their interactions and starts the simulation. Let us have a look at each step.

1. Different *locations* for the master and the three workers will be necessary. One *process* will be running on each of these *locations*. Due to the "complexity" of the communication link, we need two create two basic links (l1 and l2) and to merge them into a single one (l_merged). It drives us to the following definitions:

```
1   m_host_t h_emit, h_receive[NB_SLAVE];
2   m_process_t p_emit, p_receive[NB_SLAVE];
3   m_link_t l1, l2, l_merged;
```

2. We need to initialize MetaSimGrid internal global variables using MSG_global_init and to fix the maximum number of channels with MSG_set_channel_number. Fixing that number before creating any location is mandatory since it defines the number of mailboxes to create on each host.

```
1   {                             /*  Simulation setting */
2     MSG_global_init();
3     MSG_set_channel_number(MAX_CHANNEL);
4   }
```

3. Now, let us create each host with MSG_host_create. We use MSG_host_set_behavior to fix their relative speed and to set their cpu availability trace file if any (in this example, we use constant performance characteristics). Even if it is not useful in this simulation, MSG_host_set_data can be used to attach some private data to hosts.

```
1   {                             /*  Host creation */
2     char sprintf_buffer[64];
3     int i;
4
5     MSG_host_create(&h_emit, "Master");
6     MSG_host_set_behavior(&h_emit, NULL, 1.0);
7     MSG_host_set_data(&h_emit, NULL);
8
9     for (i = 0; i < NB_SLAVE; i++) {
10      sprintf(sprintf_buffer, "Slave %d", i);
11
12      MSG_host_create(&h_receive[i], sprintf_buffer);
13      MSG_host_set_behavior(&h_receive[i], NULL, (i+1)*2.0);
14      MSG_host_set_data(&h_receive[i], NULL);
```

```
15      }
16    }
```

4. Next, we need to create the links (with MSG_link_create and MSG_link_set_behavior) and to set up the routing table. Note that to merge l1 with l2 we have used MSG_link_merge.

```
1   {                              /*  Link and topology creation  */
2     int i,j;
3
4     MSG_link_create(&l1, "Link_1");
5     MSG_link_set_behavior(&l1, NULL, NULL, 10.0);
6     MSG_link_create(&l2, "Link_2");
7     MSG_link_set_behavior(&l2, NULL, NULL, 100.0);
8     MSG_link_merge(&l_merged, l1, l2, "Link_Merge");
9
10    MSG_routing_table_init();
11
12    for (j = 0; j < NB_SLAVE; j++)
13      MSG_routing_table_set(h_emit,_h_receive[i], l_merged);
14  }
```

5. Having created the *locations*, we can now create the *process* with the MSG_process_create function. We set the behavior, the personal data, and the *location* of the process using MSG_process_set_code, MSG_process_set_data, and MSG_process_set_host.

```
1   {                              /*  Process creation */
2     int i;
3
4     MSG_process_create(&p_emit, "Emitter");
5     MSG_process_set_code(&p_emit, emitter);
6     MSG_process_set_data(&p_emit, sched_data_process_create());
7     MSG_process_set_host(&p_emit, h_emit);
8     MSG_process_start(p_emit);
9
10    for (i = 0; i < NB_SLAVE; i++) {
11      MSG_process_create(&p_receive[i], "Emitter");
12      MSG_process_set_code(&p_receive[i], receiver);
13      MSG_process_set_data(&p_receive[i], NULL);
14      MSG_process_set_host(&p_receive[i], h_receive[i]);
15      MSG_process_start(p_receive[i]);
16    }
17  }
```

Only the master needs a real private data structure to hold the initial *tasks* and to know on which *locations* it can delegate some work. That is why p_emit is the only process for which sched_data_process_create is used.

6. Having created the private data structure of the master *process*, we can use it to store the *location* of the workers.

```
  {                             /*  Process organization */
    sched_data_process_t data_process = MSG_process_get_data(p_emit);
    int i;
    data_process->slaves_count = NB_SLAVE;
    data_process->slaves =
      calloc(data_process->slaves_count, sizeof(m_host_t));

    for (i = 0; i < NB_SLAVE; i++)
      data_process->slaves[i] = h_receive[i];
  }
```

7. At last, we can use the private data structure of the master *process* to store the initial *tasks*.

```
  {                             /*  Task creation */
    sched_data_process_t data = MSG_process_get_data(p_emit);
    char sprintf_buffer[64];
    int i;

    data->todo = calloc(NB_TASK * NB_SLAVE, sizeof(m_task_t));
    data->todo_max_size = NB_TASK * NB_SLAVE;
    data->todo_count = NB_TASK * NB_SLAVE;
    for (i = 0; i < NB_TASK * NB_SLAVE; i++) {
      sprintf(sprintf_buffer, "Task_%d", i);
      MSG_task_create(data->todo + i, sprintf_buffer);
      MSG_task_set_compute_duration(data->todo + i, 5000);
      MSG_task_set_communication_size(data->todo + i, 1000);
      MSG_task_set_data(data->todo +i, NULL);
    }
  }
```

4.2.4 Setting up the scheduler agents

A scheduler function always have the same prototype void*fun(void*t) (otherwise, the compiler may cry for foul) even if the argument is in fact the *process* whose behavior is described by fun. Thus, the behavior of the master may be encoded like this.

```
void emitter(void *t)
{
  m_process_t self = (m_process_t) t;
  sched_data_process_t data = (sched_data_process_t) self->data;
  int i;

  PRINT_MESSAGE("Got %d slave(s) :\n", data->slaves_count);
  for (i = 0; i < data->slaves_count; i++)
```

```
9     PRINT_MESSAGE("\t %s\n", data->slaves[i]->name);
10
11    PRINT_MESSAGE("Got %d task to process :\n",
12                  ((sched_data_process_t) data)->todo_count);
13    for (i = 0; i < data->todo_count; i++) {
14      PRINT_MESSAGE("\t\"%s\"\n", data->todo[i]->name);
15    }
16
17    for (i = 0; i < data->todo_count; i++) {
18      PRINT_MESSAGE("Sending \"%s\" to \"%s\"\n",
19                    data->todo[i]->name,
20                    data->slaves[i % data->slaves_count]->name);
21      MSG_put(self, data->todo[i], data->slaves[i % data->slaves_count],
22              PORT_22);
23      MSG_task_destroy(&(todo[i]));
24      PRINT_MESSAGE("Send completed\n");
25    }
26    return;
27  }
```

In the beginning, we assign the *process* associated to this function to self and we get its private data (line 3-4). These data hold the initial *tasks* and the three *locations* where they can be processed (line 7-15). The function MSG_put is then simply used to send the tasks to the workers using channel PORT_22.

The receiving part is as easy to write as the previous one. The function MSG_get is used to listen on channel PORT_22 and to grab the tasks. Once a task is transferred, it is processed using MSG_execute.

```
1   void receiver(void *t)
2   {
3     m_process_t self = (m_process_t) t;
4     m_task_t todo = NULL;
5     int i;
6
7     for (i = 0; i < NB_TASK;) {
8       int a;
9       PRINT_MESSAGE("Awaiting Task %d \n", i);
10      a = MSG_get(self, &todo,PORT_22);
11      if (a == MSG_OK) {
12        PRINT_MESSAGE("Received \"%s\" \n", todo->name);
13        PRINT_MESSAGE("Processing \"%s\" \n", todo->name);
14        MSG_execute(self, todo);
15        PRINT_MESSAGE("\"%s\" done \n", todo->name);
16        sched_data_task_destroy(MSG_task_get_data(todo));
17        MSG_task_destroy(&(todo));
18        i++;
19      } else {
```

```
20        PRINT_MESSAGE("Hey ?! What's up ? \n");
21        DIE("Unexpected behavior");
22      }
23    }
24    return;
25  }
```

4.2.5 Running the simulation

```
bobur:~/metasimgrid/example $ ./masterslave
[0.00]    P1 | [emitter] Got 3 slave(s) :
[0.00]    P1 | [emitter]     Slave 0
[0.00]    P1 | [emitter]     Slave 1
[0.00]    P1 | [emitter]     Slave 2
[0.00]    P1 | [emitter] Got 9 task to process :
[0.00]    P1 | [emitter]    "Task_0"
[0.00]    P1 | [emitter]    "Task_1"
[0.00]    P1 | [emitter]    "Task_2"
[0.00]    P1 | [emitter]    "Task_3"
[0.00]    P1 | [emitter]    "Task_4"
[0.00]    P1 | [emitter]    "Task_5"
[0.00]    P1 | [emitter]    "Task_6"
[0.00]    P1 | [emitter]    "Task_7"
[0.00]    P1 | [emitter]    "Task_8"
[0.00]    P1 | [emitter] Sending "Task_0" to "Slave 0"
[0.00]    P2 | [receiver] Awaiting Task 0
[0.00]    P3 | [receiver] Awaiting Task 0
[0.00]    P4 | [receiver] Awaiting Task 0
[11.00]   P1 | [emitter] Send completed
[11.00]   P1 | [emitter] Sending "Task_1" to "Slave 1"
[11.00]   P2 | [receiver] Received "Task_0"
[11.00]   P2 | [receiver] Processing "Task_0"
[22.00]   P3 | [receiver] Received "Task_1"
[22.00]   P3 | [receiver] Processing "Task_1"
[22.00]   P1 | [emitter] Send completed
[22.00]   P1 | [emitter] Sending "Task_2" to "Slave 2"
[33.00]   P4 | [receiver] Received "Task_2"
[33.00]   P4 | [receiver] Processing "Task_2"
[33.00]   P1 | [emitter] Send completed
[33.00]   P1 | [emitter] Sending "Task_3" to "Slave 0"
...
[5055.00] P4 | [receiver] Received "Task_8"
[5055.00] P4 | [receiver] Processing "Task_8"
[5055.00] P1 | [emitter] Send completed
[5888.33] P4 | [receiver] "Task_8" done
[6294.00] P3 | [receiver] "Task_7" done
[7533.00] P2 | [receiver] "Task_6" done
```

```
[create.c , MSG_main : 105] Congratulations ! Simulation terminated.
```

4.3 Behind the invisible

4.3.1 Scheduling

When using pthreads, there is no way to control the main scheduler and to decide which thread should be running at a particular moment. Since we want to *simulate* parallel execution, we do not want our threads to run concurrently and we need to short-circuit the main scheduler to replace it by our own. To achieve easy context-switching, we use a synchronization structure (thread_synchro_t) on which our threads are always locked. The private function __MSG_synchro_yield jails the calling thread in this structure and frees the previously locked thread. Thus, each *process* has its own synchronization structure and a thread executing its code. When a process is created, its thread gets locked in the synchronization structure to ensure mutual exclusion. Then, at the beginning, all the threads (except the main one) are locked and all process are in the process_to_run list.

The main thread yields control to the first process of the process_to_run list using the __MSG_synchro_yield function. Then this thread will eventually call a MetaSimGrid function (MSG_put, MSG_get or MSG_execute). This function schedules some SimGrid tasks and wait for the termination of one of these using the __MSG_synchro_yield function, hence yielding the control to the main thread, which in turns, yield the control to the next process of the process_to_run list, and so on until there is no more process to run. Then the main thread runs the simulation, using SG_simulate, until some task is terminated. From the completed SimGrid tasks list, one determines which processes are ready to be run, and put them in the process_to_run list.

The previous operations are repeated until there is no more SimGrid task to process. If some process are still waiting for some event, then a list of pending processes along with the function in which they are stuck is displayed.

4.3.2 Modeling file transfer

We decided MSG_get, MSG_put and MSG_execute to be blocking functions. File transfers are then assumed to be synchronous, which means both the sender and the receiver have to be ready before the file transfer begins. This is simply achieved using the following SimGrid model(see Figure 2).

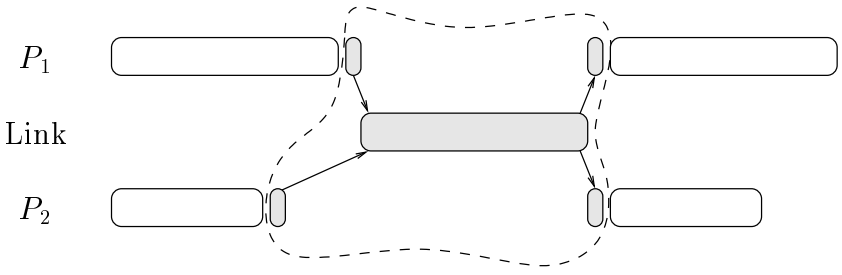


Figure 2: Synchronous communication

We could imagine plenty of other models like a non blocking call when the file to transfer is small enough, or a CPU occupation while the transfer is being done. Nevertheless, it would

require the ability to schedule a SimGrid task with a particular level of occupation on the processor and on the link. Furthermore, measuring such parameters is difficult and very dependent on the communication library to model.

4.3.3 Sufficient to use any programming style

We claim that the previous functions are sufficient to encode any programming style.

Message Passing It is rather easy to implement basic MPI-like functions. Blocking message passing is simply provided by the core functions MSG_get and MSG_put. Non blocking message passing functions can easily be simulated by creating a process on both sending and receiving part. It is then the responsibility of the user to set up a mechanism to implement a MPI_Test like function but it is rather straightforward.

Remote Procedure Call The easiest way to do this is to reserve a channel for each type of service. A dedicated process keeps listening on this channel at any location. When it receives a *task*, it might just execute it or create another process to handle it and then continue listening on its channel.

Process Migration As mentioned in Section 4.1, a *process* is simply a *code* executing on a *location*. Process migration turns out to be a simple modification of the process location. It is then straightforward to design a function that performs a task transfer (using a RPC style transfer to cope with synchronization) to simulate the migration and then change the value of the location.

5 Network Replay

5.1 What MetaSimGrid needs

Since MetaSimGrid is built on top of SimGrid, it can simulate a wide variety of platforms. Nevertheless, modeling a complex platform by hand with the base API is still fastidious for a few hosts. For a real grid, it is almost impossible: hosts and links have to be created one by one, by calling `MSG_host_create` and `MSG_link_create`. Then, routes have to be taken into account .... It seemed judicious to introduce new functionalities to help "importing" an existing platform. For this, we needed a tool to discover an *approximately* realistic topology of a given platform, including switches and routers (which are layer 2 informations, see 5.2.1), and to gather data on its available resources. Besides, this tool should store its information in a simple, and human readable way, thus allowing to easily modify the imported structure by hand whenever needed.

In order to reach this goal, we used two tools, that we slightly modified to fit our needs: Effective Network View and Network Weather Service. ENV allowed us to discover the effective topology of a network from a given host, whereas NWS was used to gather all the traces we needed (link latency, bandwidth, CPU load, ...) to simulate external load.

The problems we encountered while making such a tool were almost all purely technical and are discussed in this section. Section 5.3 describes the way we used ENV and problems we encountered, Section 5.4 is about NWS and finally, Section 5.5 explains how we integrated both with MetaSimGrid.

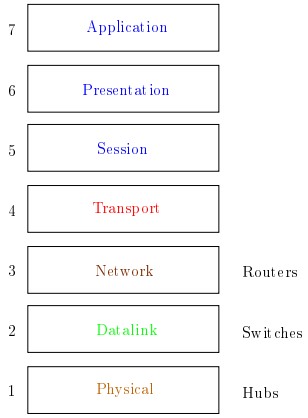


Figure 3: OSI Reference Model

5.2 Topology discovery, related work

There are a lot of tools designed for network discovery. In this section we explain why we have chosen ENV for MetaSimGrid.

5.2.1 Layer 2 versus Layer 3 topology

When speaking of network topology, there are many kinds of possible meanings due to ISO/OSI Network Model (see http://www.iso.org, Figure 3).To each layer of this model, corresponds one possible topology, the most used being the layer 2 and layer 3 topologies. Layer 2 is the *Medium Access Control* layer or *Datalink* layer, whereas layer 3 is the *Network* layer. Protocols as *Internet Protocol* are in layer 3, whereas *Ethernet* resides in layer 2. Topology may be very different considering one of these layer or the other one. Layer 2 is closer to the physical links than layer 3. So, layer 2 topology may be able to give useful informations on routers and switches, that are not directly available on layer 3. Nevertheless, we have chosen the layer 3 approach, for several reasons:

- Layer 2 topology is more difficult to discover. It is closer to physical resources, so, for some routers/switches, proprietary informations are sometimes needed, which are not always documented. Tools to discover this topology are then often proprietary, for example, Cisco's Discovery Protocol (www.cisco.com) and Bay Networks' Optivity Enterprise (www.baynetworks.com).

- Tools to discover layer 2 topology are mainly based on SNMP protocol [BGM+00], which is not always available on old or dumb routers.

- Whether a given host is a router (layer 3) or a switch (layer 2), can be discovered without using low level protocols as SNMP. We will explain this in Section 5.3.

- VLAN [MD01] can dramatically change the layer 2 topology [LOG], so special mechanisms have to be involved to take them into account. For example, within ENS Lyon, most of the networks are VLAN, so we should adopt a tool that is "VLAN insensitive".

- For most distributed applications, layer 2 topology is completely hidden, so, for our simulator, layer 2 information is not totally needed. In order to be placed in the most possible realistic environment, only layer 3 is needed, because it is the only viewpoint that a running application has on a network. Introducing layer 2 in the simulator would not make it more efficient, nor more realistic.

We focused on layer 3 discovery tools. These tools are presented in the following sections.

5.2.2 Pathchar derived tools

Among all tools to discover layer 3 network topology, a wide range is based on `pathchar`.

As `traceroute` (used by ENV), `pathchar` takes advantage of the time-to-live (ttl) field in IP packets. The ttl determines how many hops a packet can go through before it expires. If a router receives a packet that has expired, it drops the packet and sends an ICMP error packet to the sender. `patchchar` works by sending out a series of probes with varying values of ttl and packet sizes. Analyzing the time before the error packet is received, it infers the latency and bandwidth of each link in the path, the distribution of queue times, and the probability that a packet is dropped [Dow99].

The problem with `pathchar` is that it needs to make the assumption that for each link, the probe that it sends will have negligible queue delays. In order for such an assumption to be real, it needs to send a lot of probes, because the probability of such an event is low. In current implementations, more than 1500 probes are usually used, so tests can last for hours on routes where many hops are encountered.

So, tools based on `pathchar` were not the ones we needed for our tool. We wanted to have a fast discovery tool. For this, `traceroute` based tools are preferred. `traceroute` uses the same mechanisms as `pathchar` but it only sends a few packets, so measures only take a few seconds, the drawback being that only the information on encountered routers is available (see 5.3.2).

5.2.3 Other layer 3 tools

We found in the available literature, 4 main projects, that we present in this section. For each of these tools, we explain why we did not use it for MetaSimGrid.

SPAND: Shared Passive Network Performance Discovery [SSK97] is a tool to determine available bandwidth and packet loss probability on routes aimed to a given server. This tool is passive, which means that the server has a modified network stack to make live statistics. We cannot use this tool for our simulation, because most of the time, we only have normal user access to a server. Besides, this tool does not provide information on switched or shared nature of links involved in a communication.

IDMaps, GNP: IDMaps and Global Network Positioning [FJJ+01, NZ01] were designed to determine the distance between two hosts on a network. This is not sufficient for our needs. We need more information on links, such as bandwidth (which IDMaps may sometimes provide), but also on routers (see above).

Mercator: Mercator [GT00] is a program that only uses hop-limited probes (the same primitive used in `traceroute`), from a single, arbitrary location to infer an Internet map. This approach does not fit our needs because we need a map of a small, restricted

and well defined space, compared to the Internet. Besides, as described above, we need more information on layer 2 switches.

5.3 Effective Network View

5.3.1 Presentation

ENV (Effective Network View) [SBW99] was developed by Gary Shao at University of California (San Diego). It was made in Python, based on PEG (Python Extensions for the Grid) which is part of the AppLeS Project [BW97]. PEG comprises a set of Python extension modules which allow the use of various tools and utilities that have been developed for Grid application development and scheduling to be accessed directly from the Python programming language.

ENV was made to discover the effective topology of a network, from the point of view of a given host (the master in case of master/slave computing). Data acquired are then dependent from the master's choice. We do not discuss this here. It has already been done in [SBW99]. ENV uses GridML to store its data. GridML is a specialized form of XML representation which has been designed to be a flexible format for describing the physical and observable characteristics of resources and networks constituting a Grid computing. The relevant DTD and schema file can be found at http://apples.ucsd.edu/env/.

The main advantage of ENV is that it only uses user-level observations of network performance to create an effective profile of network configuration and thus, ENV allows to get information on layer 2 and 3 routers, without using low-level features.

Here is an example of ENV running on the ENS Lyon network. Figure 4 shows the physical network topology. This topology is simplified here, because some routes are not symmetric, and VLAN are used (see RFC 3069 [MD01] for info on VLAN, this problem will be explained in the following section). Figure 5 shows the effective network view from *the-doors*, which was the master in our experiments, derived from the obtained GridML. We can see on these pictures that not all routers are seen (only *routlhpc* and *routeur_backbone* are on the ENV topology, this is due to asymmetrical routes). Besides, ENV reports the fact that *popc0*, *myri0* and *sci0* are on a 100 Mbps hub, whereas links to reach *popc0* and *myri0* from *the-doors* must go trough a bottleneck at 10 Mbps. This kind of information is crucial for scheduling algorithms where a lot of data transfers are involved [CLZB00, SBW99] since the latter bottleneck can lead to poor decisions, if not known a priori.

5.3.2 How does it work?

One simple way to discover the topology and characteristics of a network of n machines would be to drive $n*(n-1)/2$ bandwidth tests, between each couple of hosts first, and then for each pair, drive concurrent tests to determine if hosts are on independent links or not. But, this would not be efficient because the bandwidth and time consumed for this operation would not allow such a program to scale to real platforms, where hundreds of machines are available. To prevent too much bandwidth loss, ENV only makes tests to discover the network, given a viewpoint.

The acquisition of ENV data can be split in two parts: the first is independent from the choice of the master, the other is not.

- The first part is itself split in 3 parts:

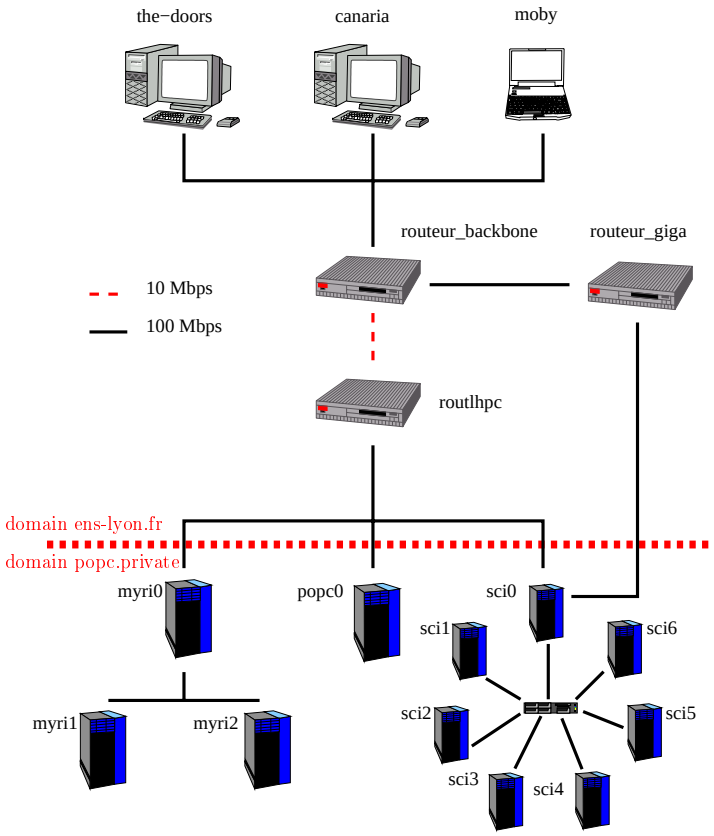


Figure 4: Physical topology of our test network

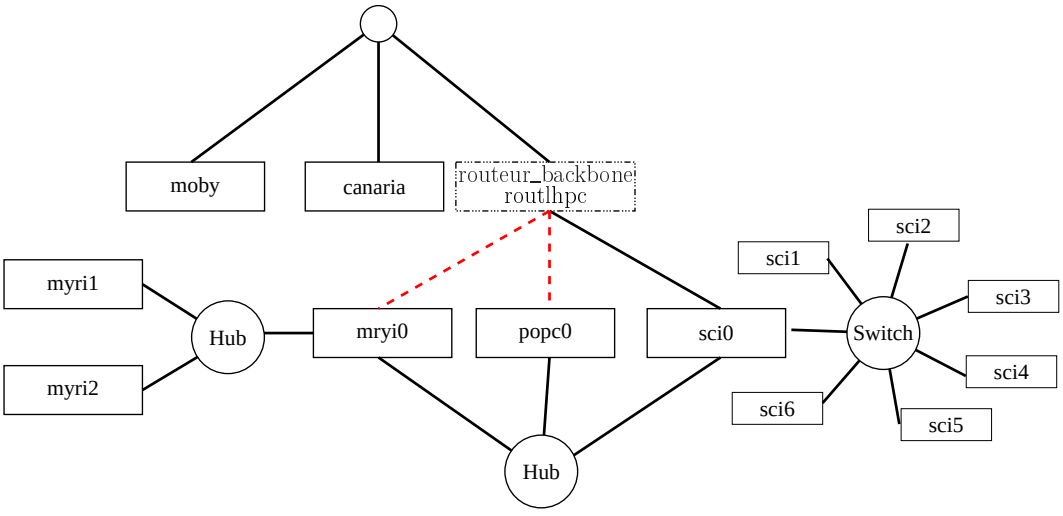


Figure 5: Effective topology of our network test from the-doors

– **Lookup:** during this phase, the core GridML is created, using host-names provided. For example, on *canaria* and *moby* we would have the following GridML

```
<?xml version="1.0" encoding="iso-8859-1"?>
<GRID>
    <SITE domain="ens-lyon.fr">
        <LABEL name="ENS-LYON-FR" />
        <MACHINE>
            <LABEL ip="140.77.13.229" name="canaria.ens-lyon.fr">
                <ALIAS name="canaria" />
            </LABEL>
        </MACHINE>
        <MACHINE>
            <LABEL ip="140.77.13.82" name="moby.cri2000.ens-lyon.fr">
                <ALIAS name="moby" />
            </LABEL>
        </MACHINE>
    </SITE>
</GRID>
```

The `GRID` is divided in `SITE` nodes, for which each machine is represented by a `MACHINE` node.

– **Info:** informations on hosts such as Operating System or processor type are acquired. We used tests provided with ENV, but any test can be implemented instead. For example, we added a benchmark taken from LINPACK [Bre91] to suit our simulation purposes. LINPACK was designed for supercomputers in use in the 1970s and early 1980s. It has been largely superseded by LAPACK, which has been designed to run efficiently on shared-memory, vector supercomputers. But the benchmark is still widely used, for example on the "TOP500 Supercomputer Sites" [DMS93], the best LINPACK benchmark performance achieved is used as a performance measure in ranking the computers.

All these informations can be used in MetaSimGrid to create a realistic platform if no other data are available (such as NWS traces, see 5.4).

```
<MACHINE>
   <LABEL ip="140.77.13.92" name="pikaki.cri2000.ens-lyon.fr">
       <ALIAS name="pikaki" />
   </LABEL>
   <PROPERTY name="CPU_clock" value="198.951" units="MHz" />
   <PROPERTY name="CPU_model" value="Pentium Pro" />
   <PROPERTY name="CPU_num" value="1" />
   <PROPERTY name="Machine_type" value="i686" />
   <PROPERTY name="OS_version" value="Linux 2.4.19-pre7-act" />
   <PROPERTY name="kflops" value="17607" />
</MACHINE>
```

– **Structural topology:** During this phase, `traceroute` is intensively used. For each hosts in the Grid, a `traceroute` from this host to a given target (the same for

the whole site) is made. The results are used to build a first tree of the network. In our example, a test on *canaria*, *moby*, *the-doors*, *myri*, *popc* and *sci* gives the following GridML :

```
<NETWORK type="Structural">
   <LABEL ip="192.168.254.1" name="192.168.254.1" />
   <NETWORK>
       <LABEL ip="140.77.13.1" name="140.77.13.1" />
       <MACHINE name="canaria.ens-lyon.fr" />
       <MACHINE name="moby.cri2000.ens-lyon.fr" />
       <MACHINE name="the-doors.ens-lyon.fr" />
   </NETWORK>
   <NETWORK>
       <LABEL ip="140.77.161.1" name="routeur-backbone" />
       <NETWORK>
           <LABEL ip="140.77.12.1" name="routlhpc" />
           <MACHINE name="myri.ens-lyon.fr" />
           <MACHINE name="popc.ens-lyon.fr" />
           <MACHINE name="sci.ens-lyon.fr" />
       </NETWORK>
   </NETWORK>
</NETWORK>
```

Here, `traceroute` was targeted at a router, outside the ENS Lyon network. All hosts given by `traceroute` which belong to the ENS network are kept to build this tree. For example, here, the farthest host in our network is `192.168.254.1` since this address is not routable outside our domain.

With such data, we can build the tree shown on Figure 6. Leaves on the same branch are clustered together to give what we call "Structural Networks". This tree is not sufficient, because it does not show clearly which links are shared and which are switched. For this, a few more tests will be done, that depend on the master's viewpoint.

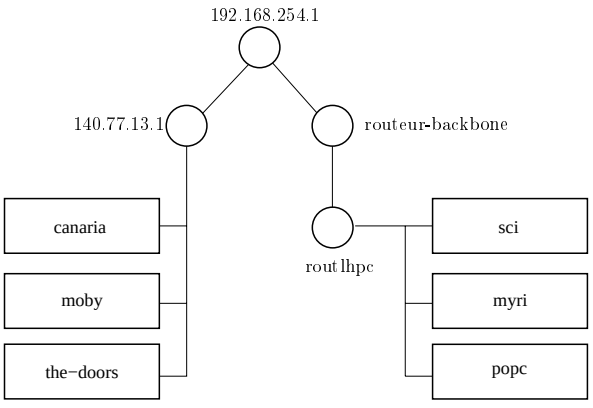


Figure 6: Initial Tree in ENV

- Effective topology: the second part of the tests is dependent from the chosen master. These measures are filters that are applied, one after another to the core GridML. All these filters allow us to generate a new tree, in which we find ENV networks. These networks contain informations on layer 2 topology which are critical in scheduling algorithms.

  host to host bandwidth: bandwidth is measured between the master and each machine listed in the GridML file. For each site, clusters previously made are divided into groups with similar bandwidth. Threshold used to determine if clusters shall be divided or not is determined experimentally. A value of 3 was used: if the bandwidth ratio between two hosts exceeds this value, the corresponding cluster is split.

Figure 7: Host to host bandwidth test

  pairwise host bandwidth: for each cluster discovered in the first stage, bandwidth is measured between the master and any pair of hosts in this cluster, the two transfers being made concurrently. Clusters are then divided into groups of hosts that have similar sharing characteristics: for each couple of measures, the bandwidth between the master and a given host is compared to the bandwidth on a pair containing this host. If the ratio $Bandwidth/Bandwidth_{paired}$ is below a given value (1.25 in our tests), then the given host is declared independent.

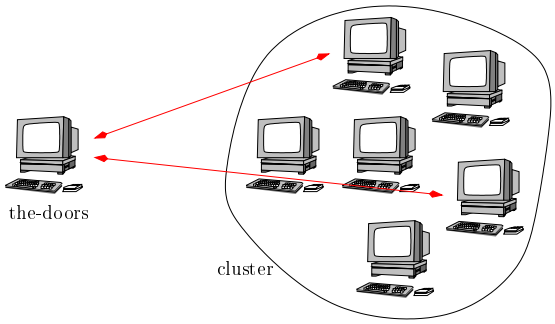


Figure 8: Pairwise host bandwidth test

  internal host bandwidth: for each cluster, bandwidth is measured between any pair of machine, within this cluster. This allows to set the local bandwidth parameters for a given cluster. This may be useful for clusters where local bandwidth is different from bandwidth achieved between the cluster and the master. For example, the *popc* cluster in ENS Lyon: route from *the-doors* go trough a 10 Mbps bottleneck, whereas *popc* is on a local hub, which speed is 100 Mbps (see Figure 4).

  jammed bandwidth: for each host in a given cluster, bandwidth to the master is measured while a transfer between two hosts in that cluster occurs. This measure

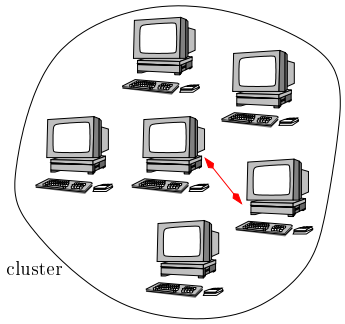


Figure 9: Internal host bandwidth test

is repeated 5 times, and the ratio $Bandwidth/Bandwidth_{jammed}$ is computed for each measure. The average is computed and compared to `SHARED_THRESHOLD` and `INDEP_THRESHOLD`:

- if the average is below `SHARED_THRESHOLD` (0.7 in our experiments) then, the cluster may be on a shared link.
- if it is above `INDEP_THRESHOLD` (0.9 in our experiments), the cluster may be switched.
- in any other case, there is no more reported information on this cluster, because values are not significant enough.

For example, here is the description of a switched network, created after this test (*sci* cluster):

```
...
<NETWORK type="ENV_Switched">
  <LABEL name="sci0" />
  <PROPERTY name="ENV_base_BW" value="32.65" units="Mbps" />
  <PROPERTY name="ENV_base_local_BW" value="32.29" units="Mbps" />
  <MACHINE name="sci1.popc.private" />
  <MACHINE name="sci2.popc.private" />
  <MACHINE name="sci3.popc.private" />
  <MACHINE name="sci4.popc.private" />
  <MACHINE name="sci5.popc.private" />
  <MACHINE name="sci6.popc.private" />
</NETWORK>
...
```

All these filters allow us to determine whether structural networks are switched or shared based upon bandwidths observed between hosts. This information is really important because a shared network is often a bottleneck when traffic is high, whereas a switched network can handle a heavier load. Besides, this information will be used for deploying NWS (see Section 5.4).

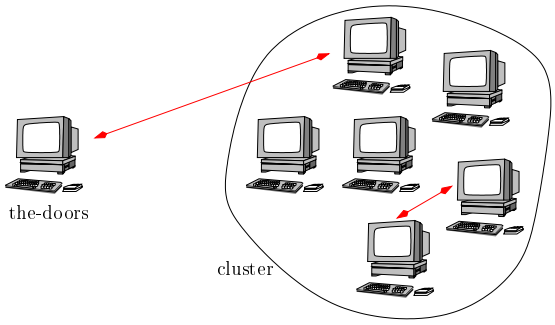


Figure 10: Internal host bandwidth test

5.3.3 Problems

In this section, we describe problems that we encountered with ENV, and solutions that we tried to implement:

- Firewalls: there is no automatic way to handle them. We solved this problem by running ENV on each side of a firewall, and merging the results. For example, at ENS Lyon, the network is split in two parts, as shown on Figure 4. Hosts *sci1*, *sci2*, ... cannot communicate with the outside world, but they are connected with *sci0*, *popc0* and *myri0*, which are able to contact outside computers. So, we launched ENV on both sides of *popc0*, and we merged the results.

  The merge operation is quite simple: we create a new GridML structure containing both sites (here ens-lyon.fr and popc.private), and we give aliases to hosts which belong to in both. So this operation is in fact almost the concatenation of several files, which is a simple operation to perform. The only thing a user has to know is which hosts can be gateways between both sides. This information is used to create aliases in the merged file. For example, for our tests, aliases were:

```
popc.ens-lyon.fr popc0.popc.private
myri.ens-lyon.fr myri0.popc.private
sci.ens-lyon.fr sci0.popc.private
```

  Which gives the following in our GridML :

```
<GRID>
    <LABEL name="Grid1" />
    <SITE domain="ens-lyon.fr">
        <LABEL name="ENS-LYON-FR" />
        <MACHINE>
            <LABEL ip="140.77.12.52" name="myri.ens-lyon.fr">
                <ALIAS name="myri" />
                <ALIAS name="myri0.popc.private" />
            </LABEL>
            ...
```

```
        </MACHINE>
        ...
    </SITE>
    <SITE domain="popc.private">
        <LABEL name="POPC-PRIVATE" />
        <MACHINE>
            <LABEL ip="192.168.81.50" name="myri0.popc.private">
                <ALIAS name="myri0" />
                <ALIAS name="myri.ens-lyon.fr" />
            </LABEL>
            ...
        </MACHINE>
        ...
    </SITE>
</GRID>
```

This final GridML will then be used by another program, written in C, that implement the resulting platform with the MetaSimGrid API (see Section 5.5 for details)

- Machine without host-name: in the first stage of ENV run, when `traceroute` are performed, hosts are said to be in the same domain if they have the same domain name. But, some hosts do not have names. It was the case on our network, so we modified ENV in order to do this with IP address if resolution fails. For this, we used IP class, as explained in RFC 1166 [KSR90]. Most networks where ENV is run are old networks, based on those classes, so we did not really need CIDR (see [IH93]) for our experiments, but a full implementation should take this into account.

  Besides, we also added non routable IP, because those addresses are always part of the domain where they are seen. Using figure 6 as an example, if those IP are not taken into account, we may lose an information on the topology of our network: the root of the tree is a non routable IP, it is the last interface seen inside the ENS Lyon network, so we must not skip it.

- Asymmetric routes: in ENS Lyon, the physical network is more complex than on Figure 4. Indeed the route between *the-doors* and *popc* goes trough a 10 Mbps link, whereas in the other direction it is on 100 Mbps links only. Since ENV tests bandwidth in only one way, this situation cannot be modeled yet.

  We did not solve this problem yet, because it would have involved an almost complete rewrite of ENV tests. In the same way, networks are usually packet driven, so we cannot guarantee that the same route is always used between two hosts, but we assume that this is the case for our study, or at least, that global measures are not too sensitive to this.

  All these problems are really common, as explained in [PF97], and they have no simple solution yet.

- Reliability, effectiveness: the question whether ENV results are accurate or not is not really important. ENV is not used here for its accuracy, which is enough for our purpose. It is used because it allows a user to easily determine a network topology that

is effective. The only problem resides in the length of measures: there are many tests to run, and their results are not independent, so if the platform evolves between two tests (important increase or decrease of the network load) the results given by ENV may be corrupted. There is no solution yet to this problem, except rapidity: tests only take five minutes to run on our platform, so we assume the environment is stable enough to provide viable results. This may not be the case on bigger platforms, where the number of hosts is above 200. But, for such platforms, the GridML could be provided by the administrators of the cluster, who exactly know the topology of their network. In this case, the use of ENV would not be necessary.

Besides, the use of experimental thresholds tends to be problematic, because they were determined using specific media types. These thresholds may not be able to detect topology on specific media types, as VTHD or Terabit links. So ENV may not be useful on such networks. Nevertheless, GridML can still be used to model such links by hand. Our layered structure makes it possible for MetaSimGrid to use any resources discovered by ENV.

- Master/Slave paradigm: ENV was designed for master/slave computing. Nevertheless, MetaSimGrid allows to build decentralized schedulers where two processes located on slaves may have to communicate. For this, in the current implementation, they use data discovered by their master, which may not be the same as the one they are experiencing. For example, Figure 11 depicts a *Master* and two clusters, *Cluster A* and *Cluster B*. Whereas *Link A* and *Link B* are discovered by the preceding tests, *Link C* does not appear because ENV does not make "inter-cluster" tests, since it was designed for centralized master/slave scheduling[1]. So, in the situation of the Figure 11, if *Cluster A* was to communicate with *Cluster B*, it would use *Link A*, go through the *Platform* and then to *Link B*.

- Dropped `traceroute`: as said in [LOG], many modern routers no longer respond to `traceroute` packets. This may not be a great problem for ENV, because clusters are still split based on bandwidth measures made in preceding tests. So, structural networks may be affected by this, but not the final result. Nevertheless, this would make measures longer, given that for clusters discovered in first stage, all possible pairs of hosts are tested. Bigger clusters means more measures in the second stage, hence more execution time.

- Bandwidth loss: tests used by ENV are still introducing a significant bandwidth loss. We cannot prevent it, since active probes are needed to infer layer 2 informations. The point is that on a given platform ENV needs to be run only once. Results can then be used by a lot of different people to test their own scheduling algorithms. For example, an administrator could provide the result of ENV tests on his/her network and make it available to all users, so that any user can test his/her algorithm on this network, without even touching it.

[1] Changes can still be made manually to the GridML produced by ENV, but as explained in Section 6, ENV is aimed at evolving and at producing more accurate information.

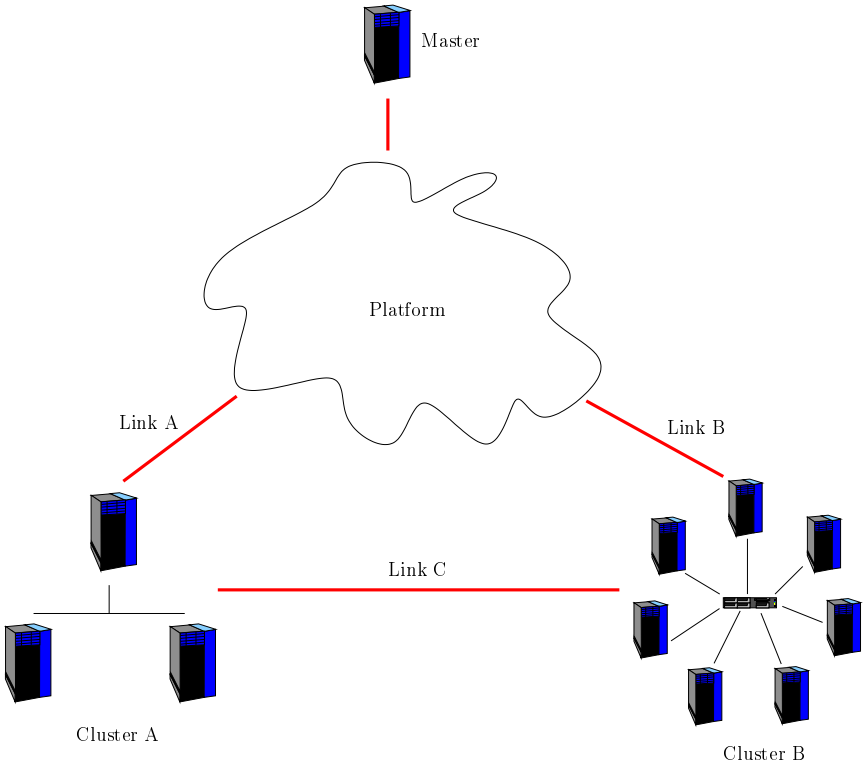


Figure 11: ENV decentralization drawback

5.4 Network Weather Service

We used NWS [WSH99] to collect traffic data on our network. NWS is now developed at the University of California (Santa Barbara) and the University of Tennessee (Knoxville). We used a slightly modified version of this software, to better fit our needs. The goal of this project is to provide accurate data on dynamically changing performance characteristics from a distributed set of metacomputing resources. These informations are intensively used in many scheduling algorithms for the grid.

Besides, NWS not only collects data, it also allows to predict performance/load of a resource in the near future, based on an adaptive approach (see [Wol98] for details). In our experiments, we only used NWS to collect traces on resources: links and processors. In this section, we discuss both link and CPU monitoring.

5.4.1 NWS organization

NWS is organized in four different component processes:

- Persistent State : process used to retrieve and store data
- Name Server : used to list processes available on a given host
- Sensor : used to conduct measures
- Forecaster : process used to predict performance

For our example, at ENS Lyon, we launched two instances of NWS, because of the two domains we had to deal with. We could have launched only one Name Server and Persistent State on *popc0* or *myri0* or *sci0*, given the fact that these 3 hosts are the only ones that can be seen from everywhere, but we preferred to launch one instance of NWS per domain, because we may have to deal with platforms where several firewalls (more than one) exist. So, there is only a matter of scalability here.

Our organization was the following (see Figure 4):

- Domain ens-lyon.fr:

Process	Hosts
Name Server	*the-doors*
Persistent State	*the-doors*
Sensor	*the-doors, canaria, moby, myri, sci, popc*

- Domain popc.private:

Process	Hosts
Name Server	*popc0*
Persistent State	*popc0*
Sensor	*popc0, myri0, myri1, sci0, sci1, sci2, sci3, sci4, sci5, sci6*

Launching any of these processes on a given host is not yet automatic, so deploying NWS on a given platform is generally quite a long and fastidious task. Since ENV is already deployed at the time we try to gather data with NWS, our topological informations can be used to do it automatically. This will be explained in Section 5.5.

5.4.2 Link monitoring

Testing link bandwidth and latency may be quite simple at first: in a set of n computers, the simplest solution is to make $n * (n-1)/2$ tests, assuming links are symmetric. But, if two hosts are on the same physical link, tests must not be run concurrently because measures for the first one would include bandwidth consumption of the second one. Besides, conducting so many tests would generate a high bandwidth loss. So tests have to be run in the right order, and at the right places. NWS handles this by introducing *cliques*. *Cliques* are subsets of k ($k < n$) computers for which the $k * (k-1)/2$ tests are made. If two hosts belong to the same *clique*, then the bandwidth and latency between those hosts is effectively measured. But, if they are not, these measures have to be computed using existing cliques (for example, the bandwidth on a route can be approximated by the minimum bandwidth achieved on that route, involving as many cliques as necessary). A single host can be in as many cliques as necessary.

3 different types of measures are conducted:

- small-message round-trip time: a 4 byte TCP socket transfer is timed from one host to another one and back (TCP connection already established).
- large-message throughput: 64 K bytes messages are sent and timed (based on acknowledgement time received from destination Sensor, TCP connection already established)
- TCP socket connect-disconnect time.

Measurements are not made concurrently. A token is passed from host to host in each clique, granting permission to launch tests. Each clique has a clique leader which begins tests and then sends the token to another host among the clique and so on. So, like all token-based protocols, several mechanisms have to be defined, in case of network error or network split [WSH99].

To build our NWS structures, we used data already gathered by ENV : thanks to ENV, we know which hosts are on switched links, and which are not. So, we made a program that builds the cliques, based on the GridML resulting from ENV, deploys NWS, and collects the traces.

For each network/subnetwork discovered in ENV, we build at least two cliques:

- If the network is shared, it is assumed that hosts in this network are on the same physical link (see Figure 4), so measuring latency and bandwidth between one couple of hosts on this link is enough to know for every couple. We pick two hosts in that network to build a local clique, and we pick another host to build a clique between this local network and the parent network (if any).
- If the network is switched, measuring the bandwidth between one couple is not sufficient to know the available bandwidth for other hosts, so we build the fully connected graph as our clique, and one more clique between the gateway and the parent network.

For our example, we built cliques as described on Figure 12. The *sci* cluster is switched, so we pick all its machines to form a new clique, whereas the *myri* cluster is shared, so we pick only two hosts for the local clique (*myri1* and *myri2*) and another host for the clique with parent network (*myri0* and *popc0*).

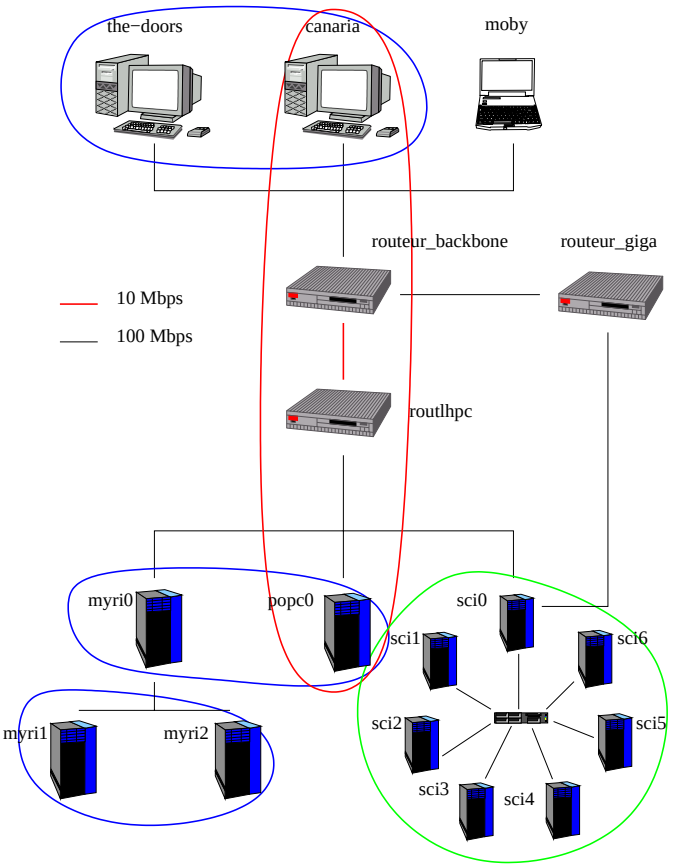


Figure 12: Cliques definitions in ENS Lyon

5.4.3 CPU monitoring

CPU monitoring on UNIX-like systems is a difficult task because variations are more important than for networks links, in which mechanisms such as TCP slow-start, may contribute to a relatively smooth variation on measures. With CPUs, the load can vary very quickly, depending on users logged in, daemons running,... It can appear to be quite chaotic at first sight. So monitoring such variations is hard because tests have to be made frequently, but should not be too intrusive. The NWS CPU Monitor achieves this goal with a pretty good accuracy (at most 10 % as explained in [WSH00]).

For this, NWS uses in fact three different tests:

- the first one uses the `uptime` command, which gives the average load of the system (usually, this value is a one minute smooth-average of the run length queue, but it can vary according to the Operating System used).
- the second one is based on the vmstat utility, which provides periodically updated readings of CPU idle time, consumed user time and consumed system time.
- the last one is a probe process that occupies the CPU for a short period of time and reports the ratio of the CPU time it used to the wall-clock time that passed as a measure of the availability it experienced.

A probe process is necessary, for many reasons:

- a measurement based on load average is insensitive to short term load variability.
- load average and vmstat do not take nice processes into account: a process launched on a host where a nice process run could hence get more resources than what was indicated by load average or vmstat bound measures.
- system time in not shared fairly and so user level processes may be denied CPU time as kernel services interrupt.

In NWS, a CPU load is measured every ten seconds by default. The active probe is launched once per minute, and for the subsequent tests the most accurate passive probe (compared to the active one) is chosen.

As mentioned in [WSH00], the accuracy of NWS is sufficient to get valid results on scheduling algorithms. Besides, obtaining more accurate values would involve more intrusive tests, due to the process priority mechanisms employed by Unix, so we cannot expect an arbitrary precision without consuming more and more resources for the test, which may be even worse than having a less accurate value with a simpler test.

We do not expect to get truly accurate traces. But, as said in [Cas01], the main purpose of SimGrid (and hence of MetaSimGrid) is the comparison of scheduling algorithms, so we do not need perfect traces. All we need is coherent values: a link ten times more loaded than another one should be reported the same in our traces, as for CPU loads.

5.5 MetaSimGrid integration

In this section, we explain how we integrated ENV results in MetaSimGrid, and the models we used for this. Concerning NWS, there is not so much to add, given that NWS traces are natively supported by SimGrid, and so by MetaSimGrid.

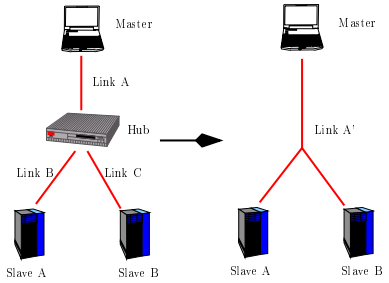


Figure 13: Hub model in MetaSimGrid

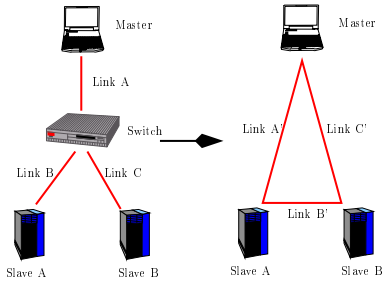


Figure 14: Switch model in MetaSimGrid

5.5.1 Modeling hubs, routers and switches

We call hubs, routers and switches, the routers found respectively in layer 1, 2 and 3 of the OSI model (see Figure 3). These routers are detected by ENV :

Hub: shared networks are described in GridML with a type `ENV_Shared`. This kind of network is used when a hub has been detected on a network. In this case, we model the hub by a single link, shared by every hosts (see Figure 13). This model is perfectly valid since *Link A*, *Link B* and *Link C* only make a single link (a hub works at layer 1, so directly on physical links).

Switch: switched networks are described in GridML with a type `ENV_Switched`. This kind of network is used when a switch has been detected on a network. In this case, we model the switch by a fully connected network (see Figure 14). This model is not perfectly exact. For example, if *Link A* is 100 Mbps, then in the model, we can achieve a 200 Mbps bandwidth if *Link A'* and *Link C'* are 100 Mbps. This model is the only one we can use, without introducing new features.

Routers: routers are detected by ENV, as root of a `ENV_Undetermined` network. We have two models for them, depending whether the router is an host where ENV is running or not:

- In the following GridML, we have a description of the first case:

...

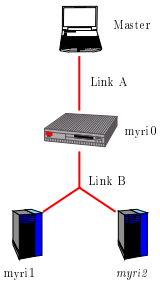


Figure 15: Router model in MetaSimGrid

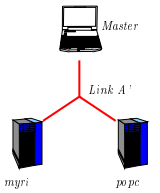


Figure 16: Router model in MetaSimGrid

```
<NETWORK type="ENV_Undetermined">
  <LABEL name="myri0-Misc" />
  <PROPERTY name="ENV_base_BW" value="32.10" units="Mbps" />
  <PROPERTY name="ENV_base_local_BW" value="32.22" units="Mbps" />
  <MACHINE name="myri1.popc.private" />
  <MACHINE name="myri2.popc.private" />
</NETWORK>
...
```

Here, *myri0* acts as a router for its cluster, so we simply walk the tree to model it. As depicted on Figure 15, we assume that links between hosts in `ENV_Undetermined` networks are shared.

- If the router is not running ENV, we do not have any value for its upstream bandwidth. This case is described by the following GridML :

```
...
<NETWORK type="ENV_Undetermined">
  <LABEL name="routlhpc_1-Misc" />
  <PROPERTY name="ENV_base_BW" value="4.70" units="Mbps" />
  <PROPERTY name="ENV_base_local_BW" value="32.31" units="Mbps" />
  <MACHINE name="myri.ens-lyon.fr" />
  <MACHINE name="popc.ens-lyon.fr" />
</NETWORK>
...
```

Since `routlhpc` is not a host running ENV, we simply delete it. The model is as shown on Figure 16. *Link A'* is given a bandwidth of 4.70 Mbps. We could also have introduced a special host in the MetaSimGrid model to simulate the router

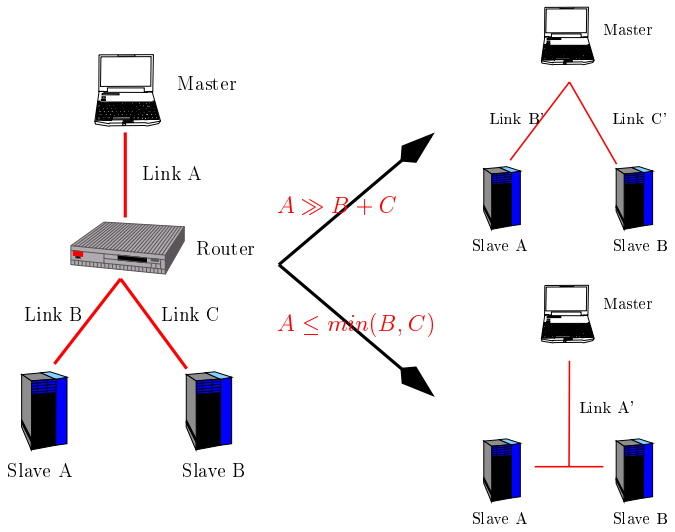


Figure 17: Router bandwidth

(for example a host that would only forward packets), but we do not know the upstream bandwidth of the router (ENV is not running on it). Besides, this would not have improved the simulation.

To summarize the situation about layer 3 routers not running ENV, consider the Figure 17, the *Master* is connected to a *Router* (layer 3), which leads to two slaves, *Slave A* and *Slave B*. In current implementation of ENV, we do not run any test on the router (we must have a remote shell access to the machine for this). If bandwidth available on *Link A* is greater than the sum of available bandwidth on *Link B* and *Link C* then ENV tests will conclude that *Slave A* and *Slave B* are independent and will split their cluster, thus creating two independent links, *Link B'* and *Link C'*, with characteristics similar to *B* and *C*. In this situation, *Link A* is completely hidden. In the other situation, where bandwidth is not sufficient, ENV test conclude *Slave A* and *Slave B* are on a shared network, and only one link, *A'*, is created. In intermediate situations, we assume the link is shared too, as shown above.

5.5.2 Network topology

In this section, we explain how we managed to produce a realistic network, based on a GridML description, written with the MetaSimGrid API.

There are in fact two problems involved in building the virtual platform from ENV data:

- If there are several sites in the GridML, due to firewalls, the merge operation must not duplicate links that have been reported in both sites. For example, on our test network, the link between *myri0*, *popc0* and *sci0* is part of *ens-lyon.fr* and *popc.private* domain (see 4). To handle this case, we simply do not duplicate links which have same characteristics (i.e. bandwidth and hosts).

- Among one site, some information contained in the `Structural` network is not repeated

in the ENV network. For example, in the following GridML, we have a Structural network containing *popc*, *myri* and *sci*:

```
<NETWORK>
  <LABEL ip="140.77.161.1" name="routeur-backbone" />
  <NETWORK>
    <LABEL ip="140.77.12.1" name="routlhpc" />
    <MACHINE name="myri.ens-lyon.fr" />
    <MACHINE name="popc.ens-lyon.fr" />
    <MACHINE name="sci.ens-lyon.fr" />
  </NETWORK>
</NETWORK>
```

So, we assume that these three hosts are linked. This information is not repeated in the corresponding effective network, because this cluster is split during bandwidth measures:

```
<NETWORK type="ENV">
  <LABEL name="ENV-192.168.254.1" />
  <NETWORK type="ENV_Undetermined">
    <LABEL name="routlhpc-Misc" />
    <PROPERTY name="ENV_base_BW" value="30.29" units="Mbps" />
    <MACHINE name="sci.ens-lyon.fr" />
  </NETWORK>
  <NETWORK type="ENV_Undetermined">
    <LABEL name="routlhpc_1-Misc" />
    <PROPERTY name="ENV_base_BW" value="4.80" units="Mbps" />
    <PROPERTY name="ENV_base_local_BW" value="32.40" units="Mbps" />
    <MACHINE name="myri.ens-lyon.fr" />
    <MACHINE name="popc.ens-lyon.fr" />
  </NETWORK>
</NETWORK>
```

So, we have to take both kinds of networks into account, in order to create appropriate links.

The algorithm to build the network is rather simple:
For each SITE in the GRID:

- First stage: hosts are parsed.
- Second stage: walk the Structural tree. Networks in this tree are the first clusters made by ENV, based on traceroute. In this stage we create a link for each network, assuming that links are all shared.
- Last stage: walk the ENV tree. This tree contains the informations on switches, hubs, ... If we encounter a switch, the link created in the previous stage is deleted and the fully connected graph between switched hosts is built. This tree is used to refine the preceding one.

Then all sites are merged among one new site, deleting duplicated links. For example, on our test network (see Figure 4), we get the following links created:

```
Link 0: myri0/popc0/sci0
Link 1: the-doors/canaria/moby
Link 2: sci0/the-doors
Link 3: myri0/popc0/the-doors
Link 4: myri0/myri1/myri2
Link 5: sci0/sci1
Link 6: sci0/sci2
Link 7: sci0/sci3
...
Link 25: sci5/sci6
```

For each link, we are able to give a link speed, which almost corresponds to reality. As shown above, a special link is made between *sci0* and *the-doors* (link 2), as this link is on 100 Mbps, whereas with *myri0* and *popc0*, speed is 10 Mbps (see Figure 4).

5.5.3 Route table

Once we have created all links and all hosts, the problem of the route table remains: MetaSimGrid does not handle routing between hosts, so the user has to explicitly specify which links should be involved in any transfer. To simplify this, we build a route table with hosts and links discovered in the previous section. For each host, we compute the shortest path (hop count) to any host.

We tested our algorithm on the platform described on Figure 18. This topology is an "ideal" topology, where all clusters are on fully switched links. This topology involves creating 268 hosts and 20740 links, which may not be easy to deal with by hand .... The time elapsed building the whole topology, route table included, is less than 15 seconds on an Athlon 800 MHz.

The corresponding GridML was generated by hand, with copy/paste operations, in less than 5 minutes, but we are currently working on a graphical tool to generate such GridML easily, without requiring the user any knowledge GridML.

6 Conclusion

We have described MetaSimGrid in this paper, a new simulator designed for decentralized master/slave computing. This tool is currently available at http://www.carva.org/julien.lerouge/. This simulator aims at simplicity and speed. It allows fast comparisons of scheduling algorithms in realistic conditions, which can be created by a user, or imported from reality, thanks to ENV and NWS. The flexibility of use of GridML makes it possible to extend ENV in many ways, to handle new kinds of networks, asymmetrical routes, or any possible benchmark.

References

[BBE+99] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu,

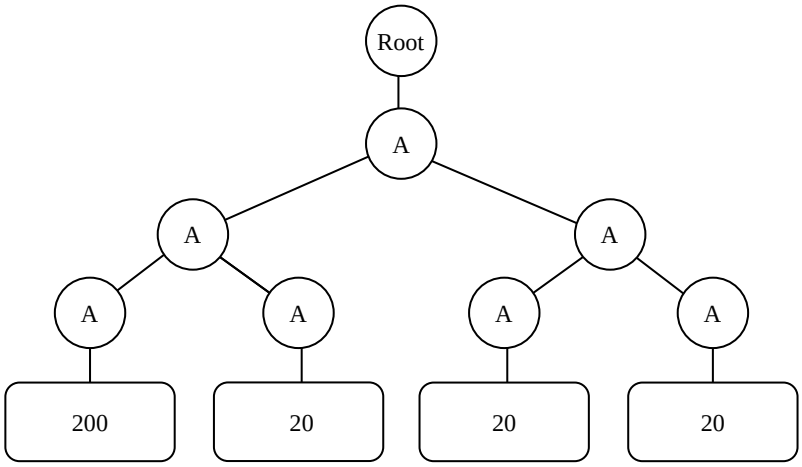


Figure 18: Test on a real platform

Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702, University of Southern California, 1999. Available at http://citeseer.nj.nec.com/bajaj99improving.html.

[BGM+00] Yuri Breitbart, Minos N. Garofalakis, Cliff Martin, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Topology discovery in heterogeneous IP networks. In *INFOCOM (1)*, pages 265–274, 2000. Available at ttp://citeseer.nj.nec.com/breitbart00topology.html.

[Bre91] R. P. Brent. The LINPACK Benchmark on the AP1000: Preliminary Report. In *CAP Workshop 91*. Australian National University, 1991. Website http://www.netlib.org/linpack/.

[BW97] F. Berman and R. Wolski. The appleS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, May 1997. Available at http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html#AppLeS. Website http://apples.ucsd.edu/.

[Cas01] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'01)*. IEEE Computer Society, May 2001. Available at http://grail.sdsc.edu/papers/simgrid_ccgrid01.ps.gz. Further information on Simgrid is available at http://gcl.ucsd.edu/simgrid.

[CDL+02] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In *Proceedings of EuroPar 2002*, Paderborn, Germany, 2002. (short paper). Available at http://www.ens-lyon.fr/~fsuter/pages/metacomputing.html.

[CDZ97] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997. Available at http://citeseer.nj.nec.com/calvert97modeling.html.

[CLZB00] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 349–363, 2000. Available at http://grail.sdsc.edu/papers/hcw00_pst.ps.gz.

[COBW00] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the grid. In *Proceedings of Super Computing*, pages 75–76, 2000. Available at http://grail.sdsc.edu/papers/apst_sc00.ps.gz.

[CWK+96] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K. E. Seamons. Performance modeling for the Panda array I/O library. In *Proceedings of Supercomputing '96*. ACM Press and IEEE Computer Society Press, 1996. Available at http://cdr.cs.uiuc.edu/pubs/super96.ps.

[DHN94] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. A distributed memory LAPSE: Parallel simulation of message-passing programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, 1994. Available at http://www.csam.iit.edu/~pmd/pubs/Distributed.ps.

[DMS93] J. J. Dongarra, H. W. Meuer, and E. Strohmaier. TOP500 Supercomputers. Technical Report RUM 33/93, Computing Center - university of Mannheim - Germany, July 1993. Website http://www.top500.org/.

[Doa96] M. Doar. A better model for generating test networks. In *Proceedings of Globecom '96*, November 1996. Available at http://citeseer.nj.nec.com/doar96better.html.

[Dow99] Allen B. Downey. Using pathchar to estimate internet link characteristics. In *Measurement and Modeling of Computer Systems*, pages 222–223, 1999. Available at http://citeseer.nj.nec.com/downey99using.html.

[FFF99] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999. Available at http://citeseer.nj.nec.com/faloutsos99powerlaw.html.

[FJJ+01] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. Idmaps: A global internet host distance estimation service. *IEEE/ACM Transactions on Networking*, oct 2001. Available at http://citeseer.nj.nec.com/francis01idmaps.html.

[FK99] I. Foster and C. Kesselman, editors. *High-performance schedulers*. Morgan-Kaufmann, 1999.

[Gen] Genome@HOME project. Website http://genomeathome.stanford.edu/.

[GRA] Grid Research And Innovation Laboratory (GRAIL). Website http://grail.sdsc.edu/.

[GT00] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM 2000*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE. Available at citeseer.nj.nec.com/govindan00heuristics.html.

[GTR] Global terabit research network project. Website http://www.gtrn.net/.

[IH93] Internet Engineering Steering Group and R. Hinden. RFC 1517: Applicability statement for the implementation of classless inter-domain routing (CIDR), September 1993. Status: PROPOSED STANDARD. Available at ftp://ftp.math.utah.edu/pub/rfc/rfc1517.txt.

[JCB00] Aubin Jarry, Henri Casanova, and Francine Berman. DAGSim: A simulator for DAG scheduling algorithms. Technical Report RR-2001-46, LIP, ENS Lyon, December 2000. Available at www.ens-lyon.fr/LIP/.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999. Available at http://www.eee.hku.hk/~ykwok/papers/compare-jpdc.pdf.

[KBM+] Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Lionel Eyraud, Rutger Hofman, and Kees Verstoep. Programming environments for high-performance grid computing: the albatross project. Available at http://www.cs.vu.nl/~kielmann/papers/fgcs01.ps.gz.

[KSR90] S. Kirkpatrick, M. K. Stahl, and M. Recker. RFC 1166: Internet numbers, July 1990. Obsoletes RFC1117, RFC1062, RFC1020 Status: INFORMATIONAL. Available at ftp://ftp.math.utah.edu/pub/rfc/rfc1166.txt.

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111. IEEE Computer Society Press, 1988.

[LN01] Jason Liu and David M. Nicol. *DaSSF 3.1 User's Manual*, April 2001. Available at http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-3.1.ps.

[LOG] Bruce Lowekamp, David R. O'Hallaron, and Thomas Gross. Topology discovery for large ethernet networks. pages 237–248. Available at http://citeseer.nj.nec.com/500129.html.

[LR99] M. Livny and Rajesh Raman. *High-performance schedulers*, pages 311–337. Morgan-Kaufmann, 1999.

[MD01] D. McPherson and B. Dykes. RFC 3069: VLAN aggregation for efficient IP address allocation numbers, February 2001. Status: INFORMATIONAL. Available at ftp://ftp.math.utah.edu/pub/rfc/rfc3069.txt.

[MMB00] Alberto Medina, Ibrahim Matta, and John Byers. On the origin of power laws in internet topologies. Technical Report 2000-004, Computer Science Departement, Boston University, 20, 2000. Available at http://citeseer.nj.nec.com/medina00origin.html.

[Net] Netcraft web server survey. Website http://www.netcraft.com/survey/.

[NuR] Network research group : Topology generation. Website http://www.nrg.cs.uoregon.edu/topology_generation/.

[NZ01] E. Ng and H. Zhang. Predicting internet network distance with coordiantes-based approaches, 2001. Available at citeseer.nj.nec.com/article/ng01predicting.html.

[PF97] V. Paxon and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the Winder Communication Conference*, December 1997. Available at http://citeseer.nj.nec.com/article/floyd99why.html.

[SBW99] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999. Available at http://apples.ucsd.edu/pubs/pdpta99.ps.

[SCB02] A. Su, H. Casanova, and F. Berman. Utilizing DAG scheduling algorithms for entity-level simulations. In *Proceedings of the High Performance Computing*, April 2002. Available at http://grail.sdsc.edu/papers/dag_hpc02.ps.gz.

[SET] SETI@home project. Website http://setiathome.ssl.berkeley.edu/.

[SLJ+00] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, Kenjiro Taura, and Andrew A. Chien. The microgrid: a scientific tool for modeling computational grids. In *Supercomputing*, 2000. Available at http://www.sc2000.org/techpapr/papers/pap.pap286.pdf.

[SSK97] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, 1997. Available at http://citeseer.nj.nec.com/seshan97spand.html.

[VTH] VTHD project. Website http://www.vthd.org/.

[Wol98] Richard Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-tr.ps.gz.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-arch.ps.gz.

[WSH00] Richard Wolski, Neil T. Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-cpu.ps.gz.