LoadBuilder: a Tool for Generating and Modeling Workloads in Distributed Workstation Environments

Olivier Dalle

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

LoadBuilder: *a tool for generating and modeling workloads in distributed workstation environments*

Olivier DALLE

N° 3045

Novembre 1996

THÈME 1

ISSN 0249-6399

LoadBuilder: a tool for generating and modeling workloads in distributed workstation environments*

Olivier DALLE

Thème 1 — Réseaux et systèmes
Projet SLOOP**

Rapport de recherche n°3045 — Novembre 1996 — 23 pages

Abstract: This report presents *LoadBuilder*, a distributed environment designed to provide a portable way of empirically studying the effects of various kinds of workloads in local area networks of heterogeneous workstations. This tool is especially intended to build distributed experimentations including composite workload setting, statistics collecting and performance evaluations. The statistical analysis of the experimental results will help the dynamic load balancing designer to select the most meaningful indicators out of the plethora available on workstations, to establish behavior models of the workstations, to exhibit critical workload thresholds and finally, to establish his own set of meaningful workload indicators. In the following, we first describe the architecture of the environment. Then, we present and discuss the algorithms used to build (a) synthetic workloads, (b) statistics collectors and (c) measurement procedures.

Key-words: Performance evaluation, Workload characterization, Distributed computing, Networks, Experimental design

(Résumé : tsvp)

* A short version of this paper has been published in the proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems, held in Dijon (France) in 1996 ([11]).

** Join project I3S - CNRS / INRIA / Université de Nice Sophia Antipolis.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

LoadBuilder: un outil pour la génération et la modélisation de charges en environnement réparti de stations de travail[***]

Résumé : Ce rapport présente *LoadBuilder*, un environnement réparti dont le but est de fournir une solution portable afin d'étudier de manière empirique les effets des différents types de charges dans les réseaux locaux de stations de travail hétérogènes. Cet outil est particulièrement adapté à la construction d'expériences réparties comprenant l'établissement de niveaux de charges composites, la récupération de statistiques et l'évaluation de performance. L'analyse statistique des résultats de ces expériences permet ainsi d'aider le concepteur d'algorithmes de répartition dynamique de charge à choisir les indicateurs les plus significatifs parmi la multitude disponible sur les stations de travail, d'établir des modèles de comportement des stations de travail, d'exhiber les niveaux de charge critiques et finalement, d'établir son propre jeu d'indicateurs de charge significatifs. Dans ce qui suit, nous commençons par décrire l'architecture de l'environnement. Puis nous présentons et discutons les algorithmes utilisés pour (a) construire des charges synthétiques, (b) des collecteurs de statistiques et (c) des procédures de mesure.

Mots-clé : Évaluation de performances, Caractérisation de charge, Informatique répartie, Réseaux, Plans d'expériences.

[***] Une version courte de ce papier a été publiée dans les Actes de 9^{eme} Conference Internationale ISCA «Parallel and Distributed Computing Systems» organisée à Dijon en 1996 ([11]).

1 Introduction

Given the continuously increasing conjugate power of local area networks (LAN) and workstations with low price/performance ratio, and given the availability of distributed programming environments providing common programming interfaces on numerous platforms (such as PVM [1] or MPI [13]), network of workstations (NOW) has become the rational choice to perform heavy parallel and distributed computations.

In order to find the distributed resources that match at best the distributed processes requirements, many research efforts have been done (and are still done) on dynamic load balancing (DLB) algorithms [6, 16, 8, 12, 7, 10, 4].

Out of the many problems one is faced with the DLB algorithm design, we are especially interested in the definitions and meanings of the *load indicators* : clearly, the best our knowledge of the global system state and behavior is, the wiser our workload sharing or balancing policies should be.

The main difficulties to consider with NOWs are firstly, their multi-user and multi-programming purpose and secondly, their heterogeneity. A quite usual solution used to face the first difficulty, consist in building DLB algorithms that look for idle workstations [10, 4] : such algorithms are intended not to disturb the users/owners of the workstations and are practically quite easy to implement, given the boolean nature of the load indicator (idle or busy).

But this kind of algorithms also have noticeable drawbacks. In particular, it does not allow to fully consider NOWs' heterogeneity. This heterogeneity appears at several levels : computation power, memory configurations, operating systems, hardware components performances, workstation architecture, LAN protocols, technologies and even topologies. Preferring an idle but slow workstation to an active but powerful one may not be a good choice from the performances point of view. Furthermore, adding a few jobs to such a powerful workstation does not necessarily implies a noticeable disturbance for its users/owners.

Thus, a single boolean idle indicator is clearly not adapted to choose the best, or even a good target workstation for a job in an heterogeneous environment.

On the other hand, dynamic load balancing of parallel applications may also gain a lot from multi-criteria load indicators [7]. For example, it is clear that the network traffic has a significant effect on the communications performances and consequently on the performances of a parallel application. Secondly, as pointed out by Pozzetti & Al. in [17] with usual communications (TCP/IP), a message transmission implies a CPU demand depending on the message length. This also means that communication performances and workstations workloads are correlated. And while this may not appear of real importance with most of the small bandwidth networks currently in use (such as 10 MBits/s CSMA/CD Ethernet),

this may become a significant factor with the new low latency, large bandwidth (over 100 MBits/s) and generally switched networks such as ATM, Myrinet or even Fast-Ethernet.

In order to build useful and meaningful workload indicators, and to model and compare the behavior of the continuously changing set of LAN and workstations of modern heterogeneous NOWs, an empirical and, as much as possible, automated approach seems to be unavoidable.

This is the general purpose of the *LoadBuilder* environment. It provides an easy way of studying the system and network workloads in networks of workstations.

In section 2, we give a general overview of the *LoadBuilder* environment. Then we describe and discuss the design of the three service categories provided by the *LoadBuilder* environment: the workload services in section 3, the statistics services in section 4 and the measurement services in section 5. Finally, we conclude in section 6, describing our current and planned future works.

2 General description

The general architecture of our environment is, as often with distributed environments, based on the client/server concept. Before we describe the general architecture of the environment, let us detail our guidelines and motivations.

2.1 Guidelines and motivations

There already exists a plethora of distributed environments such as PVM or MPI implementations. However the guidelines that leaded the development of these environments are really different of ours. For instance, providing robust and generally heavy runtimes, various and complex communications protocols, fault tolerance, distributed debugging support, or advanced features such as process communicators or task grouping is out of the scope of this study.

On the contrary, the main qualities we focus on are primarily to be not intrusive, practical, and easily extensible and portable.

Minimal intrusion is obviously the first requirement of any measurement tool. It implies that our environment needs to be as light as possible, not to disturb the behavior of the configuration under study. Concerning the communication protocol, that leaded us to use the unsafe but connection-less UDP/IP protocol. Anyway, the unsafe property of this protocol (messages may be lost) is not a problem as long as communications are limited to transmit

short control messages (requests) from one client to several servers which in turn, manage to answer immediately.

Concerning the management processes running inside the configuration under study (the servers), a special care has been taken to limit their interference with the workloads under observation: they use a minimal amount of memory and have nearly no activity during an experiment.

The most tricky part of the environment concerning this intrusion problem, is the statistics collection. As we will see in section 4, the statistics collectors have two recording policies either minimizing I/O activity or memory occupancy, depending on the experiment designer's requirements. Furthermore, these statistics collectors have a cyclic behavior (clock-driven) and thus, their interaction with the experiment may be minimized. Indeed, the experimentation design may (and should) include, in a preliminary calibration step, several experiments to modelize the effects of the monitoring tasks.

Finally, let us notice that the client front-end, from which the user defines and controls his experiments does not minimize intrusion on the workstation it is running on, since it is not intended to be run *inside* the configuration under study.

Practical This is achieved by providing the user a centralized interface to distribute and control a set of specialized tasks in the distributed environment he wishes to study. This set of specialized tasks includes all the components required to study workloads effects in distributed workstations environments: synthetic workloads generators, performance measurements tools and statistics collecting services. Thus, the *LoadBuilder* environment is a particularly useful tool in the experimental design domain ([14]). Figure 1 gives an example of such an analysis based on experimental design.

In this example, we assume we are interested in studying workloads related to the network traffic using an isolated but not stand-alone subnetwork of workstations: any traffic not concerning one of the selected workstations is supposed to be filtered. In this case, the *LoadBuilder* environment is used to control the parameters and results of each experiment from any workstation of the main network, with a very minimal intrusion on the configuration under study.

Extensible and portable Without going into implementation details, these two qualities have been considered whenever it was possible. For instance, the communication protocol uses a systematic encoding scheme (XDR, via the *Sun* `rpcgen` tool[18]) to support heterogeneity, and the client command line interpreter is generated using the `lex` and `yacc` languages which make it easily extensible.

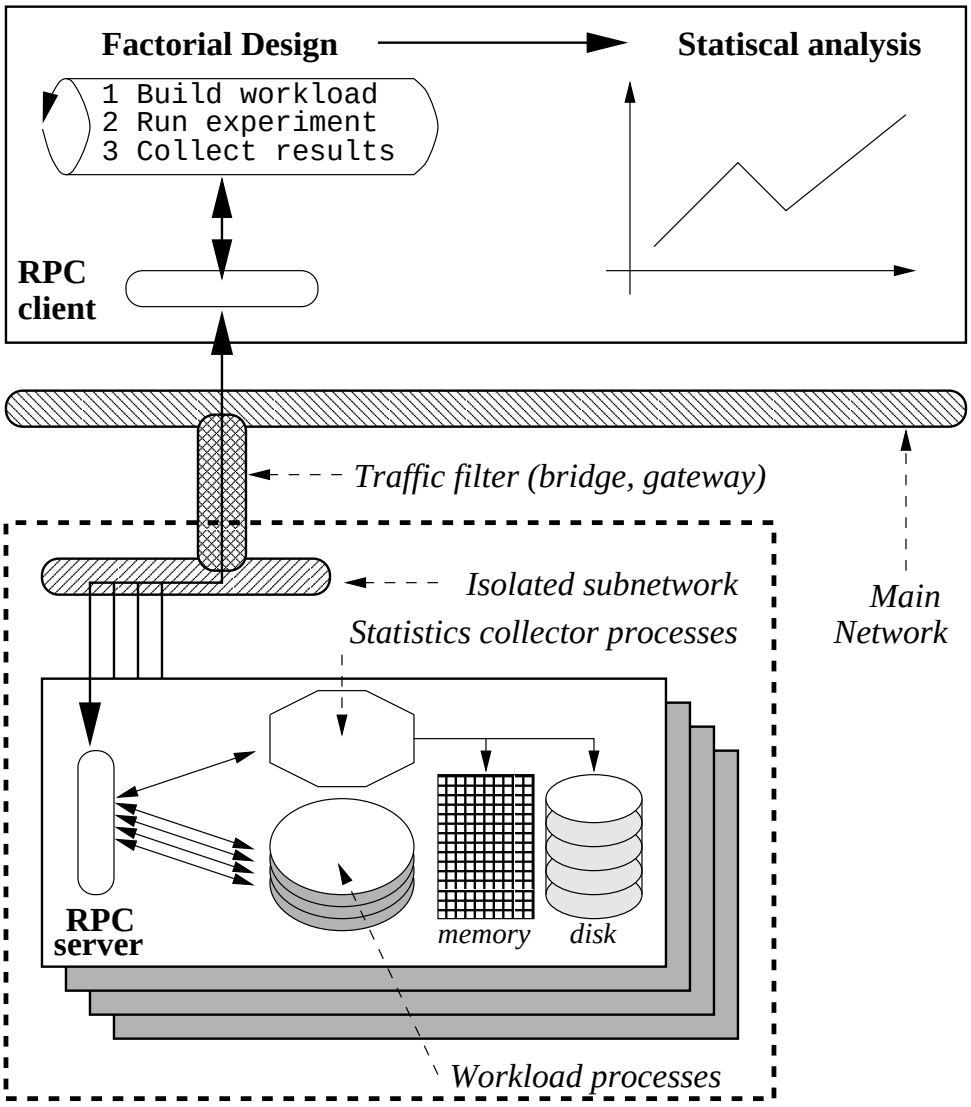

Figure 1: An example of configuration used to study workloads including network traffic: the *LoadBuilder* environment provides a remote control on the configuration under study with minimal intrusion.

2.2 Architecture

The *LoadBuilder* environment is a collection of programs organized in a classical two levels hierarchy (see figure 2): a single client process interprets the end-user's commands and dispatch the corresponding requests to several server processes, thanks to a simple, lightweight (UDP/IP connection-less based) remote procedure call (RPC) protocol. The following section that describes the servers merely presents the technical aspects of the *LoadBuilder* environment while the next section rather focus on its practical aspects by describing the client.

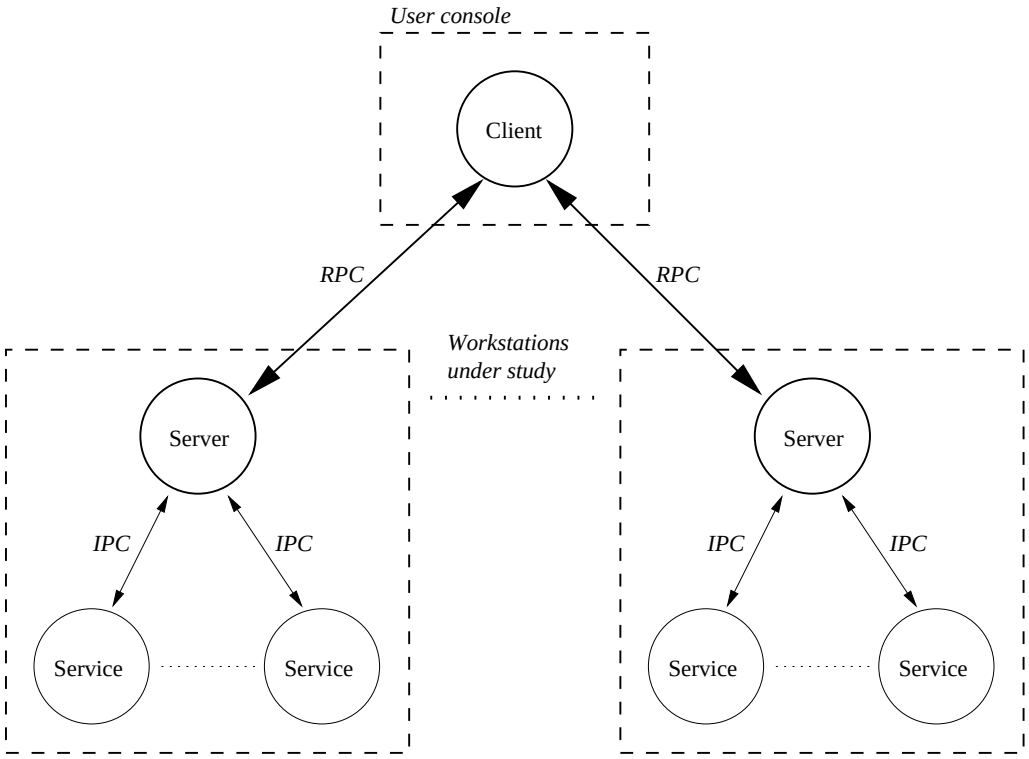

Figure 2: The *LoadBuilder* environment architecture: a two level hierarchy of processes interacting through Inter-Process Communications (IPC) schemes or Remote Procedure Calls (RPC).

2.2.1 The servers

On each workstation of the configuration under study, one of these server processes is supposed to be running. There, it has to (a) process and reply to the client requests and (b) manage several local processes.

Most of these processes are executing a *LoadBuilder* service (see table 2, page 21 for a complete list). As further explained in section 3.1, it is worth stressing that every time an experiment service is spawn, a new process is created. The remaining processes that may be run are the transient processes the server may have to spawn to answer the complex "data file encoding and transfer" request[1]; since these are not intended to be visible from the end-user, they only require a very minimal management.

The requests the servers currently recognize are the following :

- spawn a new service;
- return service status (process still running or terminated, process output if any);
- terminate service execution preserving its/their status;
- terminate service execution removing its/their status;
- wait for some service completion;
- collect/gather distributed datas.

Everywhere it makes sense to apply a request to more than one service, the server accepts the following selection policies:

- select a single service;
- select any service matching a given type;
- select any service.

The main idea behind the synthetic workload service is to provide a way to the experimental designer to build his workload by combining the execution of several specialized processes (see discussion, section 3). Each of these processes belongs to a specific class of workload : CPU, memory, network, I/O and Operating System workloads.

As we will further explain in section 5 the measurement procedures depend on the final goal of the workload study. In our case, we want to build a communication model, thus we specifically designed a measurement procedure to evaluate the performances of some interesting communication schemes. But the behavior modeling can be extended to any other part of the distributed system through different performance evaluation tools (benchmarks or specifically designed algorithms).

[1] Since this request is meant to be run at the end of the experiment, it should never interfere with an experiment.

Finally, the statistics recording services are provided by processes collecting on the fly any interesting indicators concerning the system states (traces, logs, usual kernels internal statistics).

Figure 3 shows a schematic point of view of such a server node configuration.

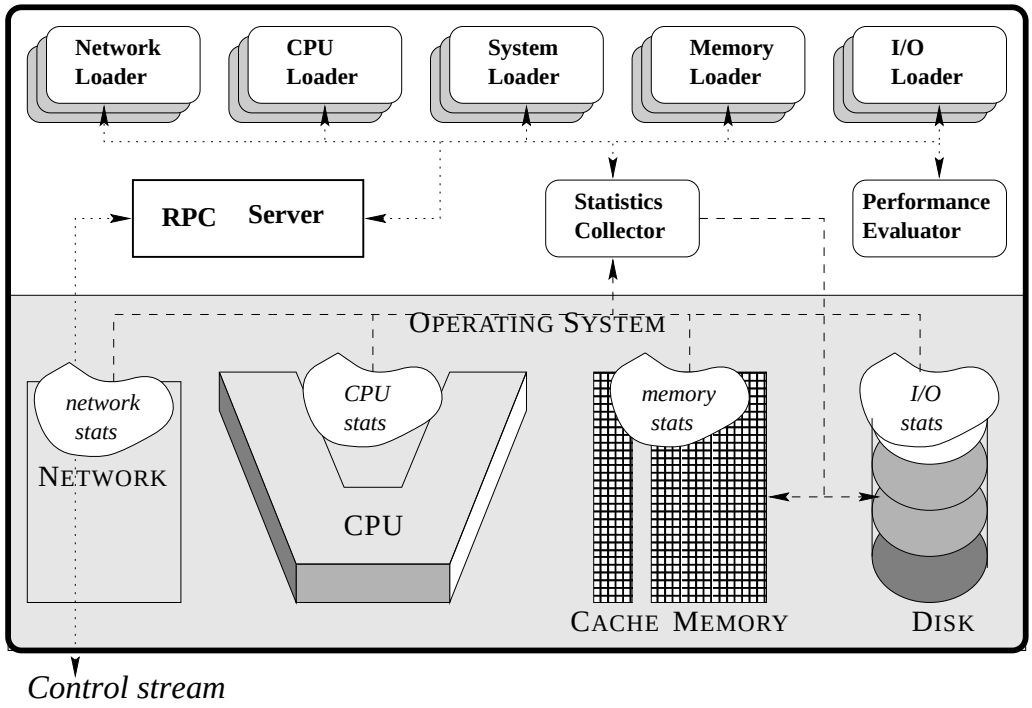

Figure 3: During an experiment, a *LoadBuilder* environment server node usually run a few workload services, a statistics recording service and some measurement procedure.

2.2.2 The client

The user interface of the client is a simple command-line interpreter proposing two running modes: an “on-line” one, producing customized outputs and an “off-line” (batch) one for an automated use. In fact, this client is primarily meant to be run in second mode, that is coupled to an external program automically generating instructions according to a given experiment design. To ease such a coupling, the client only accepts single-line instructions composed in the following syntax:

1. **keyword**
2. **keyword host** [**host** ...]
3. **keyword service** [**service** ...]
4. **keyword global_svc_id** [**global_svc_id** ...]
5. **keyword host service** [**service** ...]
6. **keyword host service parameters**
7. **keyword host process_id** [**process_id** ...]
8. **keyword parameters**

The table 1 page 20 lists and describes available keywords while the table 2 page 21 lists and describes the available services.

Figure 4 shows a typical interactive session during which the user run an experiment on three workstations (`cheers, sante` and `prosit`) from a fourth one, so the client will not disturb the system under study. This example also reflects a typical experiment sequence:

- **Start the local and network statistics recording services** `lstat` and `nstat`. Without any parameters, their default policy is to use the "memory only" stategy that minimizes I/O activity (all the data collected are bufferized until the process is stopped).
- **Start some synthetic workload services**, such as the `lcpu` one that runs an infinite loop to load the CPU.
- **Start a measurement**: in this example we start an UDP/IP "ping-pong" measurement between two workstations, using the `uping` and `upong` services. The `uping` process will record the completion times of several hundred message exchanges with the `upong process` for various message sizes.
- **Wait for the measurement completion**. For experiments not involving finite time measurements such as the previous "ping-pong" test, another scheme is to wait here until a given duration has elapsed, depending on the requirements of the later statistical analysis of the data being collected.
- **Stop services**. Without any parameter this command apply to any service still running. This `term` command notify the services that the experiment is over and that they should save their data and possibly perform some final computations. In this example, this command is mainly destinated to the statistics recording services: whenever they receive this request, the `lstat` and `nstat` services save the contents of their buffers

to disk and write the corresponding number of records and the data file name on their standard output.

- **Collect data**. Here, we only collect information written on standard outputs using the `get` command. We could also request a data file transfer using the `coll` command.
- **Reset servers**. This `kill` command kills the services still running (in fact, none in this example) and remove their entries from the servers tables. Any information about a service is lost after it has been killed.

3 Loading modules

In the following, we first discuss the reasons that leaded us to the concept of the specialized workload service. Then we describe the algorithms we used in order to build CPU, memory, network and system synthetic workloads.

3.1 Discussion

In [9], Calzarossa & Al. give a hierarchical classification of workloads: a system is running applications which may be decomposed into sequences of algorithms which in turn may be decomposed into sequences of basic routines.

Using this classification, we can mainly derive two methods to build workloads. Either we use a high-level one, by running typical standard applications (such as compilers, text processors, data bases, interpreters, and so on) or algorithms (matrix products, data sorting or scatter/gather communications scheme for instance), or we use a low level one by running synthetic programs.

Obviously, the higher the workload is built in this hierarchy, the more realistic it is. But while the high level approach may be of interest in a more comparison-oriented performance evaluation domain (as with benchmark suites, for instance), its use in a more behavior-oriented modeling domain leads to a few drawbacks. First, there is a high risk for workload situations built that way to cover only a subset of all the possible states of system, depending on the initial pool of test applications or algorithms. In order to minimize this risk, one will have to oversize this pool, which on the other hand will also lengthen the measurement procedures and globally the whole modeling. Let us notice also that the behavior of such applications may vary from one architecture to another, depending on implementation details. Consequently, it is clear that a behavior modeling based on such an analysis method is heavy, complex and hardly portable.

At a lower level, we can easily make an inventory of the basic routines onto which any algorithm is built : it uses some CPU time and memory space for computations, accesses the

Commands, parameters, *outputs*	[Comments]
LB>**start** cheers lstat	[Start a local statistics recorder on cheers]
#1 <cheers:5172>(running) LSTAT	[Return global id, host name, pid and state]
LB>**start** prosit nstat	[Start a network statistics recorder on prosit]
#2 <prosit:8423>(running) NSTAT	[Start a CPU workload service on cheers]
LB>**start** cheers lcpu	
#3 <cheers:5173>(running) LCPU	
... Start a few other workload and statistics services ...	
LB>**start** cheers upong	[Start a slave UDP/IP measurement process on
#10 <cheers:5180>(running) UPONG	cheers]
LB>**start** sante uping	[Start the master UDP/IP measurement process
#11 <sante:25283>(running) UPING	on sante]
LB>**wait** uping	[Wait for the measurement completion]
Waiting for 1 services	[Only 1 service matches the wait request]
The console hangs until the expected completion happens	
Notification from sante: Ok	[The expected completion happens]
LB>**term**	[Terminate every service still running]
#1 <cheers:5172>(running) LSTAT	[List of all the services to be terminated and their
#2 <prosit:8423>(running) NSTAT	current status. Their entries that are still needed
#3 <cheers:5173>(running) LCPU	for some commands (such as **get**) are kept by
#10 <cheers:5180>(end: 0) UPONG	the RPC servers as long as they are not explicitly
#11 <sante:25283>(end: 0) UPING	deleted (**kill**).]
... list of all other workload and statistics services that are to be terminated ...	
LB>**get**	[Collect every service output]
#1 LSTAT:'258 lstat.cheers.5172'	[Statistics services return a filename and its
#2 NSTAT:'250 nstat.prosit.8423'	number of records]
#3 LCPU:"	[Some services do not output results.]
... list of all the others service outputs ...	
LB>**kill**	[Terminate services still running and delete their
#1 <cheers:5172>(sig: 15) LSTAT	entry. There are two kinds of service
#2 <prosit:8423>(sig: 15) NSTAT	terminations: services that are terminated through
#3 <cheers:5173>(sig: 15) LCPU	a demon request (signal), and services that
#11 <sante:25283>(end: 0) UPING	terminate by their own (exit).]
... list of all other workload and statistics services that are to be cleared ...	

Figure 4: An sample interactive session with the *LoadBuilder* environment

network and I/O subsystems via system calls and asks for some system kernel services. The main difficulty with this approach is then to exhibit the typical generic routines that belong to each of these class of services.

Hereafter, we describe some of these routines we have implemented in our environment.

3.2 CPU Load

Only loading a CPU is quite easy, especially to reach a constantly busy state, since a CPU knows only two states : working or not working. Thus a single infinite loop is usually enough to place a CPU in a constantly busy state. Nevertheless, on a time-sharing system, "constantly busy" does not necessarily mean "very loaded", but only "never idle" (i.e. a "very loaded" state dramatically slows down the execution performances of the running application while a "never idle" state may have no consequence at all). So, in order to bypass the priority rules of the system and reach a "very loaded" state, one should launch several of these greedy processes. Something more difficult to obtain is to set the CPU in a "sometime busy" state which is the most frequent state of a CPU. The easiest way to do so is by alternating active periods (running a loop for instance) and sleep periods (via a system call).

In the current implementation, the service has two starting parameters : the length of the sleep period and the length of the active period (both given in microseconds). With no parameter the service executes an infinite loop. However, one should consider that the mechanism used to swap the process state (signal/interrupt handler) increases noticeably the system activity. In particular, we experienced that giving an identical value to the two parameters does not produce a "half time busy" state of the CPU (the busy time increases monotonically as the period decreases).

3.3 Memory Load

In fact, there are two kinds of memory loads : an allocation one and an utilization one. Indeed, nowadays, any Unix system has at least a three level memory architecture : at least one level of cache memory between the CPU (and/or DMA) and the main memory boards, the main core memory (RAM) and some swapping/paging space (disk). Schematically, the allocation policy of the algorithm essentially has effects on the high levels (RAM and disk), while the utilization policy has effects on the low level one (cache).

The way our algorithm manages to give a control over these two dimensions of the load is by defining the following parameters :

- the global amount of memory it uses ;
- the size of the memory chunks it self-allocates to reach this global amount ;

- the size of the data vectors it translates from one of the previous chunk to another (to generate cache faults) ;
- the time ratio it spends in each of the previous allocation and translation activity ;
- the time ratio between its active and sleep periods ;

Let us also notice that this memory load, and especially the utilization oriented one, has a significant side-effect on the CPU activity.

3.4 Network Load

Internet local traffic has a clearly random behavior, especially when the underlying network is Ethernet ([2]). Nevertheless, as pointed out by Jain and Routhier in [15], this randomness does not follow a Poisson process model. It rather follows a packet train model. The main characteristic of this model is not only the "train" one (i.e. messages larger than a packet are transmitted using several successive packets) but also the fact that several of these trains may be exchanged (thus, in the two directions) between the peer hosts.

Out of the many factors that explain this behavior, we could for instance observe that because of the unsafe nature of the underlying transmissions (Ethernet is the currently most widely used) and for performance considerations, most of the communication protocols use windowing algorithms to transmit the messages across the network (TCP/IP, and usually end-users ones, when they are built over UDP/IP) .

So, to render this quite complex behavior, we first have decided to introduce some randomness in our algorithm and secondly, to provide the two communications classes of the Internet Protocol: TCP/IP, which is the safe and connected one (stream), and UDP/IP, which is the unsafe and connection-less one (datagram).

The TCP/IP workload service follows a "ping-pong" scheme between the two workstations involved : the initiator sends a TPC/IP message to the receiver which immediately returns the message. This service accepts the following parameters : the range limits of the message lengths (randomly drawn in that range) and the lengths of the active and sleep periods of the process.

On the other hand, the UDP/IP workload service is a bit more sophisticated : it allows the definitions of multiple packet trains that are to be randomly chosen (according to the relative weight they are given) and sent between the initiator and a greedy process running on the target workstation (this process does not return the message). The characteristics that may be given for each train are the following :

- the packet size;
- the inter-packet delay;

- the train packet count;
- the weight of the train;

At last, the UDP/IP service also accepts a global parameter to fix the sleep period between each train sending.

3.5 System Load

The system workload service is very similar to the CPU one, except that its infinite loop contains an operating system call (time of day). The calling parameters are the same as the CPU service one : the active and sleeping periods lengths.

4 Statistics modules

Except for a very few of them (such as CPU use ratio over the last second), these indicators are still very hardware and operating system interdependent. As an example, one could compare on some different configurations the behavior of the *standard* load indicators given by the `uptime` command (a *standard* combination of some CPU, I/O, memory and wait-queues indicators averaged over the last one, five, and fifteen minutes) with the effective response time an end-user gets, to be convinced they cannot be used as is to compare different workstations workloads.

We distinguish two statistics classes: the local and the network statistics. The local one refers to every indicator of the system activity that is specific to a given workstation, while the network one refers to every indicator that is shared among every workstation (in particular the network global traffic indicators).

4.1 Local statistics

Whatever the architecture and the Operating System we choose, these statistics are very numerous. Let us try to categorize them.

Firstly, we can distinguish two levels of specialization: a general one concerning the global state of the system (the load indicators returned by the `uptime` command), and a very specialized one concerning very particular components (subsystems) of the workstation (such as the number of memory pages currently swapped out or the current cumulative count of incoming IP packets).

Secondly, we can distinguish three levels of accessibility : the normal user commands, which refer to the standard commands a normal user can execute (`uptime`, `who`, `ps`,...), the super user reserved commands, which refer to commands only available for the super

user (depending on the Operating System configuration, this may include the `top`, `vmstat` or `iostat` standard commands) and at the lowest level, we can directly access the kernel internal statistics. The lower the informations are collected, the better is their quality but also the potentially less portable and easy to use they are.

Thirdly, we can distinguish the different kinds of data collected : these may be unbounded cumulative totals, ratio indicators or just amounts of resource usage.

Finally, we can distinguish two kinds of accuracies, depending on the time policy of the statistic source : the real time one gives better results while the delayed one (periodic updates) gives less accurate results.

Currently our local statistics service records the statistics returned by the `rstat` *Sun* RPC service which is widely available on different architectures and operating systems (SunOS, Solaris, OSF/1 at least). It is a low accuracy (one second update period) but user-level, portable, and complete service that provides 23 indicators, covering CPU, I/O, memory, system and network subsystems. We are currently writing another service to record the output of the `iostat` and `vmstat` commands and plan to provide soon a very low level service on the SunOS system, accessing directly its internal indicators (the most accurate, but neither portable nor user-level).

As the amount of the data we need to record is quite important, a special care has been taken to minimize the impact of this monitoring process on the system behavior. This is achieved by implementing two buffering strategies : a “memory only” one that keeps the whole recorded data in the memory until the end of the experiment and a “memory and disk” one that periodically flushes the record buffer to the disk (every minute, for instance).

Consequently, this service has just one parameter, to designate which of these two strategies to use.

4.2 Network statistics

Unfortunately, the network statistics we are interested in (especially the traffic level and a few distribution characteristics) are not easy and very expensive to collect. Not easy because it generally requires the super user privileges (for security considerations), and very expensive because it usually requires to toggle the network interface of the workstation into a special mode named “promiscuous mode[2]”, which disturbs a lot the normal behavior of the selected workstation. This special mode deactivate the automatic low level filtering feature of the network interface that prevent it to catch useless packets. On the other hand, we only need one of these sonde workstation per (sub)network link since the traffic is common to each of the workstations it connects. Furthermore, local network administrators often have

[2] On Ethernet networks.

such sounding dedicated workstations for their own purpose. Indeed, sounding a network does not necessarily require CPU power (especially if the only requested work is to count the packets and possibly apply a minimal sort), so any deprecated workstation may be good enough.

A few powerful standard tools, such as `tcpdump` or `etherfind`, for instance, may be used to collect these statistics. But as we previously said, these all require the super user privileges to be run.

There also exists a quite flexible *Sun* RPC service that provides a useful enough set of network statistics : the `etherstat` service. Its main drawback is that it do not seem to be available on others platforms.

Nevertheless, we have currently implemented our network statistics service using this RPC `etherstat` service. The statistics we currently get with our service are the following: the distribution of the packets according to main protocols (broadcasts, UDP/IP, TCP/IP, ICMP, …), the distribution of the communications volume according to the packet sizes (16 size ranges), a global volume indicator and a global packet counter. Each of these values are updated every second.

As previously, with the local statistics service, the network statistics service accepts one parameter to designate the buffering strategy to use.

5 Measurement modules

Measurement modules are meant to evaluate the variations of some performances metrics depending on the workload settings. But as emphasized by Raj Jain in [14], the definition of such performance metrics is a critical point in the modeling study itself. Thus, it would be pointless and probably impossible to try to provide measurement modules that exactly fit any of the *LoadBuilder* users' expectations.

Nevertheless, we managed to provide at least one measurement module for each of the system's activities we previously considered with the workload modules: CPU, memory, I/O, system and communications activities. By widely but imperfectly covering all the system's activities, these modules may happen to be useful in order to have an idea of the global performance variations and may detect unexpected phenomena.

Practically, these measurement modules are simply some instrumented and simplified versions of their corresponding synthetic workload services, except for the last one (communication). This instrumentation consists in collecting a few statistics about the module activity, mainly its throughput variations (the variations of the rate at which it performs the operations it is designed for, such as loop iterations per second for the CPU one).

Concerning the communications performances, we did not reuse the complex packet train scheme of the UDP/IP load module. The first obvious reason is that its complexity makes it very difficult to instrument. The second is that communications performances, that are currently our main topic of interest, are usually expressed using the two following metrics: latency and bandwidth.

Consequently, we designed two measurement procedures, one for the evaluation of the TCP/IP protocol performances and the other for the UDP/IP protocol. These two procedures use the same evaluation method based on a simple "ping-pong" scheme, in order to model the transmission time of the messages according to their length, and off course, the global workload of the system under study.

Thus, we implemented four services: an UDP/IP and a TCP/IP master services (the "ping" side), and an UDP/IP and a TCP/IP slave services for the ("pong") side.

The masters services (see algorithm figure 5) accept the following parameters :

- The name of peer workstation running the matching service;
- The port number (to possibly enable the simultaneous running of several experiments);
- The number of message lengths to evaluate and the two bounds of the message lengths interval: before it enters the main repetition loop (see the following algorithm description), the algorithm compute the set of message lengths it will use (uniform distribution of the lengths in the interval);
- The number of round-trips of the same message between the peers;
- The number of repetitions of each measurement;
- The results buffering strategy;
- A boolean to activate the random choice of the message lengths; more precisely, it does not tell to draw a random length in the interval, but only to randomly choose the next length in the set of the pre-computed lengths: this feature was added to avoid any possible correlation due to a monotonic variation of the message lengths;
- A delay fixing the sleeping time between the successive repetitions of the experiment;

The slaves services only accept two parameters: the port number and the name of its peer workstation.

6 Conclusion

By explaining the *LoadBuilder* environment design, implementations guidelines and functionalities, we attempted to show how this environment may help the experiment designer to put into practice experimental designs in a distributed workstation environment. But it

```
begin
    Compute the messages lengths;
    for r:=1 to number of repetition of the experiment
    do
        for l:=1 to number of message lengths
        do
            (randomly) choose the lth message length;
            transmit this length to the slave and wait for acknowledgment;
            start CLOCK;
            for i:=1 to number of round-trips
            do
                send a message of length l;
                receive a message of length l;
            done
            stop CLOCK;
            store the duration of these transmissions;
        done
        sleep(delay);
    done
end
```

Figure 5: Communication measurement: the slave service's algorithm

also exhibits the complexity of the experimental design itself, given the number of control parameters provided to control the workload, and the number of factors we have to study.

Nevertheless, it already provides enough functionalities to investigate new models of performances and behavior, including at least partially, the workload parameters. In particular, as a case study, we currently focus on a new model of the communication performances that include the local and network workload parameters, first in an homogeneous dedicated NOW interconnected via a 10 MBits Ethernet network, then on a different homogeneous network connected via high bandwidth and low latency networks (100 MBits Fast-Ethernet and 640 MBits Myrinet) and finally in an heterogeneous network.

From these modeling experiences, we expect to gain enough knowledge to exhibit automatic modeling procedures. In particular, we plan to define soon an interface between the *LoadBuilder* environment and the University of Massachusetts EKSL's CLIP tool[3], that

allows the user to define and run experiments designs and to collect the resulting data in a very simple manner.

It is also worth stressing that such behavior models are of great interest in the system administration area, since they allow a fast and accurate detection of unusual workload configurations.

Eventually, depending on the results of these modeling studies, we also plan to implement efficient multi-criteria Dynamic Load Balancing algorithms, firstly in order to validate the previous models and secondly to enhance the performances of the *C++//* object oriented, distributed environment for parallel applications, developped inside the SLOOP project[5].

Annex

Keyword	Syntax	Description
QUIT	1	Exit from client
HELP	1,3	Online manual
RINFO	1,2,3,4,5,7	Query servers about services status
RUN	5,6	Spawn a new service
START	5,6	Spawn a new service
TERM	1,2,3,4,5,7	Terminate a service's process, preserving its state record
KILL	1,2,3,4,5,7	Terminate a service's process and remove its state record
GET	1,2,3,4,5,7	Collect standard output from a service's process
WAIT	1,2,3,4,5,7,8	Either wait for a given amount of time (8th syntax), or wait for a service's process to terminate
COLLECT	1,2,3,4,5,7	Transfer data saved by service's process (if any)
ECHO	1,8	Echo parameter (if any)

Table 1: The *LoadBuilder* client available keywords

Class	Name	Description
Synthetic Workload	LCPU	CPU bound workload
	LMEM	Memory bound workload
	LSYS	System bound workload
	LUSVR	UDP/IP network traffic producer
	LUCLNT	UDP/IP network traffic consumer
	LTPING	TCP/IP “ping-pong” traffic generators
	LTPONG	
Measurement procedures	MCPU	CPU throughput measurement
	MMEM	Memory throughput measurement
	MSYS	System throughput measurement
	UPING	UDP/IP network performance measurement (“ping-pong”)
	UPONG	
Statistics collectors	LSTAT	Workstation local statistics
	NSTAT	Network statistics

Table 2: The *LoadBuilder* available services

References

[1] A. GEIST & Al. *PVM - Parallel Virtual Machine : a user's guide and tutorial for networked parallel computing*. MIT press, 1994.

[2] LELAND & Al. On the self-similar nature of Ethernet traffic. *Computer Communication Review - ACM SIGCOMM'93 Conference Proceedings*, 23(4):183–193, Sept 1993.

[3] S. ANDERSON, A. CARLSON, D. WESTBROOK, D. HART, and P. COHEN. Tools for experiments in planning. In *Proc. of the 6th IEEE Int. Conf. on Tools with Artificial Intelligence*, pages 615–623. IEEE, 1994.

[4] R. ARPACI and Al. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. of the Joint Int. Conf. on Measurement & Modeling of Comp. Syst.*, Ottawa, 1995.

[5] F. BAUDE, F. BELLONCLE, JC. BERMOND, D. CAROMEL, O. DALLE, E. DARROT, O. DELMAS, N. FURMENTO, B. GAUJAL, P. MUSSI, S. PERENNES, Y. ROUDIER, G. SIEGEL, and M. SYSKA. The *SLOOP* project: Simulations, Parallel Object-Oriented Languages, Interconnection Networks. In *2nd European School of Computer Science, Parallel Programming Environments for High Performance Computing ESPPE'96*, pages 85–88, Alpe d'Huez, May 1996.

[6] Y. BERBERS, C. JACQMOT, W. JOOSEN, and E. MILGROM. UNIX and load balancing : a survey. In *EUUG'89*, Brussels, 1989.

[7] R. BOUTABA and B. FOLLIOT. Presentation of a multicriteria load balancing. In *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (ECOOP'92)*, Utrecht, Netherlands, 1992.

[8] L.-F. CABRERA. The influence of workload on load balancing strategies. In *USENIX Sum. Conf.*, June 1986.

[9] M. CALZAROSSA, G. HARING, G. KOTSIS, A. MERLO, and D. TESSERA. A hierarchical approach to workload characterization for parallel systems. In B. HertzBerger and G. Serazzi, editors, *LNCS(919)*. HPCN EUROPE, Springer Verlag, 1995.

[10] C. H. CAP and V. STRUMPEN. Efficient parallel computing in distributed workstation environnements. *Parallel Computing*, 19:1221–1234, 1993.

[11] Olivier DALLE. LoadBuilder: A tool for generating and modeling workloads in distributed workstations environments. In K. Yetongnon and S. Hariri, editors, *Proceedings of Parallel and Distributed Computing Systems (PDCS'96)*, volume 1, pages 248–253, Dijon (France), September 1996. International Society for Computers and their Applications (ISCA). ISBN: 1-880843-17-X.

[12] D. FERRARI and S. ZHOU. A load index for dynamic load balancing. In *Proc. of the Fall Joint Computer Conf.*, pages 684–690, Nov 1986.

[13] MPI Forum. 2nd European MPI workshop. In *MPI-2: Extensions to the message-passing interface*, Vienna, Jan 1996.

[14] Raj JAIN. *The Art of computer performance analysis : techniques for experimental design, measurement, simulation and modeling.* Wiley, 1991.

[15] Raj JAIN and Shawn A. ROUTHIER. Packet trains - measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, SAC-4(6):986–995, Sept 1986.

[16] W. LELAND and T. OTT. Load-balancing heuristics and process behavior. In *Performance'86 and ACM SIGMETRICS 1986, Vol 14*. ACM Performance Evaluation Review, May 1986.

[17] E. POZZETTI, V. VETLAND, J. ROLIA, and G. SERAZZI. Characterizing the resource demands of TCP/IP. In B. HertzBerger and G. Serazzi, editors, *LNCS(919)*. HPCN EUROPE, Springer Verlag, 1995.

[18] SUN Microsystems. *Network Programming Guide*, 1990.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399