Emulation platform for high accuracy failure injection in grids

Thomas HERAULT [a] Mathieu JAN [b] Thomas LARGILLIER [a,1]
Sylvain PEYRONNET [a] Benjamin QUETIER [a] Franck CAPPELLO [c]
[a] *Univ Paris Sud-XI FR-91405; LRI; INRIA Saclay; CNRS*
[b] *CEA, LIST, FR-91191*
[c] *INRIA Saclay FR-91893; Univ Paris Sud-XI; LRI; CNRS*

AbstractIn the process of developping grid applications, people need to often evaluate the robustness of their work. Two common approaches are, simulation where one can evaluates his software and predict behaviors under conditions usually unachievable in a laboratory experiment and experimentation where the actual application is launched on an actual grid. However simulation could ignore unpredictable behaviors due to the abstraction done and experimation does not guarantee a controlled and reproducible environment.

In this chapter, we propose an emulation platform for parallel and distributed systems including grids where both the machines and the network are virtualized at a low level. The use of virtual machines allows us to test highly accurate failure injection since we can "destroy" virtual machines and, network virtualization provides low-level network emulation. Failure accuracy is a criteria that notes how realistic a fault is. The accuracy of our framework is evaluated through a set of micro benchmarks and a very stable P2P system call Pastry since we are very interested in the publication system and resources finding of grid systems.

Introduction

One of the most important issue for the evaluation of a grid application is to monitor and control the experimental conditions under which this evaluation is done. This is particularly important when it comes to reproducibility and analysis of observed behavior. In grid software systems, the experimental conditions are diverse and numerous, and can have a significant impact on the performance and behavior of these systems. As a consequence, it is often very difficult to predict from theoretical models what performance will be observed for an application running on a large, heterogeneous and distributed system. It is thus often necessary and insightfull to complement the theoretical evaluation of parallel algorithms with simulations and experiments in the "real world".

However, even with detailed monitoring procedures, experiments in the real world are often subject to the influence of external events, which can prevent more detailed analysis. More importantly, the experimenters usually have access to only a small variety of distributed systems. In general, experimental conditions are not strictly reproducible in

[1]Corresponding author: Université Paris Sud XI, Bâtiment 490 91405 Orsay Cedex France; Email: thomas.largillier@lri.fr

the real world. The approach usually taken to broaden the scope of the evaluation consists in designing simulators, under which the experimental conditions can be as diverse as necessary. The "real world" experiments can then help to validate the results given by simulators under the reproduced similar conditions.

Still, simulators can only handle a model of the application, and it is hard to validate an implementation, or guarantee that the end user application will meet the predicted performance and behavior. Here, we study another tool for experimentations: emulators. Emulators are a special kind of simulator, which are able to run the final application, under emulated conditions. They do not make the same kind of abstraction as normal simulators, since they emulate the hardware parts of all the components of the real world infrastructure, and thus capture the complex interactions of software and hardware. Yet, since the hardware is emulated in software, the experimenter has some control on the characteristics of the hardware used to run the application.

Through this control, the experimenter can design an ad-hoc system, suitable for his experiments. Of course, the predicted performances must still be validated by comparison with experiments on real world systems, when such systems exist. But within an emulated environment, the experimenter can inject experimental conditions that are not accessible in a real environment, or not controlled. A typical example of such condition is the apparition of hardware failures during the experiment. With a real system, hardware failures are hard to inject, and hard to reproduce. In an emulated environment, hardware is software-controlled and the experimenter can design a reproducible scenario of fault injection to stress fault tolerant applications. This is crucial in fault-tolerant systems, since the impact of the timing and target of a failure can impact tremendously on the liveness and performance of the application.

Emulators can be designed at different levels of the software stack. A promising approach for emulators is the use of virtual machines (VM). A VM by itself fits partially the goals of parallel application emulators, since it emulates (potentially multiple) instances of a virtual hardware on a single machine. In addition to these virtual machines, we need to link them through a controllable network. In this chapter, we present V-DS, a platform for the emulation of parallel and distributed systems (V-DS stands for Virtual Distributed System) through virtualization of the machines and the network.

V-DS introduces virtualization of all the hardware of the parallel machine, and of the network conditions. It provides to the experimenter a tool to design a complex and realistic failure scenario, over arbitrary network topologies. To the best of our knowledge, this is one of the first systems to virtualize all the components of a parallel machine, and provide a network emulation that enables experimenters to study low-level network protocols and their interactions with failures.

This work is an extended version of the short paper presented at the ACM International Conference on Computing Frontiers 2009 in the paper entitled "High accuracy failure injection in parallel and distributed systems using virtualization"[14]. This chapter presents with more details the performance analysis that let us conclude to a better accuracy of failure injection using our tool, when compared with other tools, with new micro-benchmarks and experiments completing the previous results.

This chapter is organized as follows. Section 1 presents related work. Then, we give the design of our platform in section 2. Section 3 presents the experiments we made for testing the platform. Finally we conclude and introduce future works in the last section.

1. Related Work

Recently, the number of large-scale distributed infrastructures has grown. However, these infrastructures usually fall either into the category of production infrastructures, such as EGEE [2] or DEISA [17], or in the category of research infrastructures, such as PlanetLab [10]. To our knowledge, only one of the testbeds in the latter category, namely Grid'5000 [6], meets the mandatory requirement for performing reproducible and accurate experiments: full control of the environment.

For instance, PlanetLab [10] is a good example of testbed lacking the means to control experimental conditions. Nodes are connected over the Internet, and a low software reconfiguration is possible. Therefore, PlanetLab depends on a specific set of real-life conditions, and it is difficult to mimic different hardware infrastructure components and topologies. Consequently, it may be difficult to apply results obtained on PlanetLab to other environments, as pointed out by [13]. Grid'5000 [6] consists of 9 sites geographically distributed throught France. It is an example of a testbed which allows experiments in a controlled and precisely-set environment. It provides tools to reconfigure the full software stack between the hardware and the user on all processors, and reservation capabilities to ensure controllable network conditions during the experiments. However, much work remains to be carried out for injecting or saving, in an accurate and automatic manner, experimental conditions in order to reproduce experiments. Finally, Emulab [24] is an emulation platform that offers large-scale virtualization and low-level network emulation. It integrates simulated, emulated and live networks into a common framework, configured and controlled in a consistent manner for repeated research. However, this project focuses only on the full reconfiguration of the network stack. Moreover, Emulab uses extended FreeBSD jails as virtual machines. Inside jails, the operating system is shared between the real machine and the virtual machine, thus killing a virtual machine is the same as killing a process. It means that this framework may not simulate real (physical) machine crashes. To the contrary, our work uses Xen virtual machines, allowing to either shutdown the machine or crash it, which will leave the connections open. As will be demonstrated in the experiments section, this is a much more realistic crash simulation since a crashed machine never closes its connections before disappearing from the network.

Software environments for enabling large-scale evaluations most closely related to ours are [4] and [18]. [4] is an example of integrated environment for performing large-scale experiments, via emulation, of P2P protocols inside a cluster. The proposed environment allows the experimenter to deploy and run 1000000 peers, but at the price of changes in the source codes of tested prototypes and supporting only Java-based applications. Besides, this work concentrates on evaluating the overhead of the framework itself and not on demonstrating the strength of it by, for instance, evaluating P2P protocols at a large scales. In addition, the project provides a basic and specific API suited for P2P systems only. P2PLab [18] is another environment for performing P2P experiments at large-scale in a (network) controlled environment, through the use of Dummynet [20]. However, as for the previously mentioned project [4], it relies on the operating system scheduler to run several peers per physical node, leading to CPU time unfairness. Modelnet [23] is also based on Dummynet. It uses the same scheme except that the network

[2]EGEE Team. LCG. `http://lcg.web.cern.ch/`, 2004.

control nodes do not need to co-scheduled on the compute nodes. In Modelnet, network nodes are called *core nodes* and compute nodes *edge nodes*. But, as in P2PLab, multiple instances of applications are launched simultaneously inside *edge nodes*, consequently it relies again on the operating system to manage several peers.

Virtualization in Clouds and emulators serve different objectives: In Clouds, virtualization allows many users sharing the same hardware. In our emulation engines, virtualization is used to run, for a single user, many instances of virtual nodes on the same hardware. One of the main goal of the virtualization systems used in Clouds is to ensure that no data or program in one virtual machine can be accessed and corrupted from another virtual machine running on the same hardware. Thus virtual machine security (isolation) is a major concern. In contrary a virtualization environment for emulation should make easy the communication between virtual machines and since the same user is using all virtual machines, there is no need for security. In Emulation, one goal is to run the maximum number of virtual machines on the same hardware. This goal is not considered as essential for Clouds. These differences in goals are fundamental and as a consequence, the virtualization technologies developed for Clouds are not corresponding to the need of Emulators.

Finally, simulators, like Simgrid [7], GridSim [5], GangSim [11], OptorSim, [2], etc. are often used to study distributed systems. The main problem with simulation is that it successfully isolates protocols but does this at the expense of accuracy. Some problems that have been overlooked by the abstractions done in the simulation will not be exhibited by simulations but will be observed when the real application is launched, so conclusions from simulation may not be valid, like in [12].

2. V-DS Platform Description

The V-DS virtualization environment is composed of two distinct components: the virtualization environment for large-scale distributed systems and a BSD-module for the low-level network virtualization. Each component will be described in the following subsections.

2.1. Virtualization Environment for Large-scale Distributed Systems

V-DS virtualizes distributed systems entities, at both operating and network level. This is done by providing each virtual node its proper and confined operating system and execution environment.

V-DS virtualizes a full software environment for every distributed system node. It allows the accurate folding of a distributed system up to 100 times larger than the experimental infrastructure [19], typically a cluster.

V-DS supports three key requirements:

- Scalability: In order to provide insights on large-scale distributed systems, V-DS supports the folding of distributed systems. Indeed, thanks to Xen characteristics, it is possible to incorporate a large number of virtual machines on a single physical machine with a negligible overhead [1].

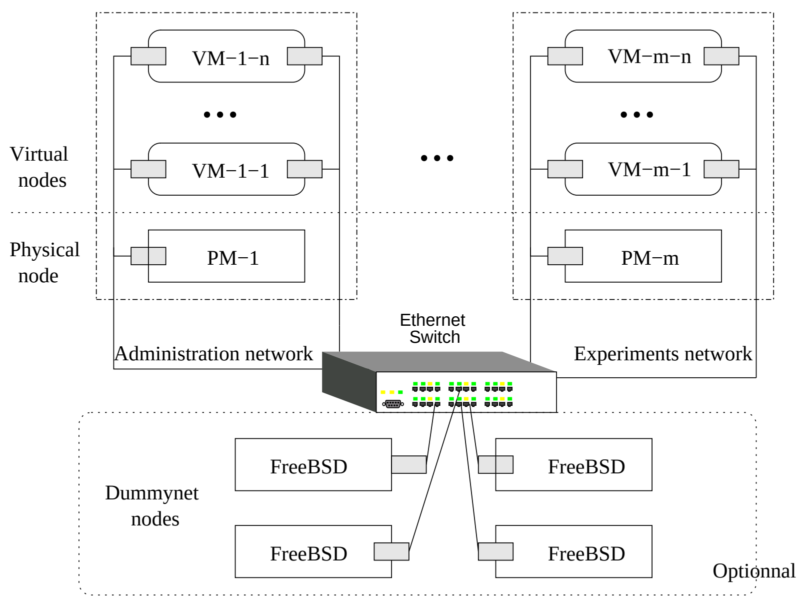

Figure 1. Overview of the architecture of V-DS.

- Accuracy: In order to obtain accurate behavior of a large-scale distributed system several constraints on the virtual machines (VMs) are needed. First the CPU must be fairly shared between VMs, then each VM must be isolated from the others, lastly the performance degradation of a VM must evolve linearly with the growth of the number of VMs. Using Xen allows V-DS to ensure all these requirements (see for example [19]).
- Adaptivity: the platform provides a custom and optimized view of the physical infrastructure used. For instance, it is possible to support different operating systems, and even different versions of the same operating system.

V-DS is based on the Xen [1] virtualization tool in version 3.2. Xen gets interesting configuration capabilities, particularly at the network level which is fundamental in the injection of network topologies. Compared to other virtualization technologies it has been demonstrated [19] that Xen offers better results.

Figure 1 shows the general architecture of V-DS. Here, m physical machines called $PM{-}i$ running a Xen system are hosting n virtual machines named $VM{-}i{-}j$ with i the index of the physical machine hosting that virtual machine and j the index of the virtual machine. Thus, there are $n * m$ virtual machines (VM). All communications between these VM are routed to FreeBSD machines to 1) prevent them from communicating directly through the internal network if they are on the same physical machine, 2) add network topologies between VM.

2.2. Low-level Network Virtualization

One of the main advantages of the V-DS platform is that it also uses virtualization techniques for emulating the network. This allows the experimenter to emulate any kind of topology with various values for latency and bandwith on a cluster. For instance, we can run, in this framework, grid applications on clusters.

For the purpose of emulating the network, we use FreeBSD machines. Using BSD machines to virtualize the network is crucial to a reliably accurate network simulation, since BSD contains several very efficient tools to manipulate packages like ipfw[3] or Dummynet[20]. With these packages it is possible to insert failure (like dropping packets) in the network very easily. The platform is then capable of injecting realistic failures at the machine and network level.

There are three networks joining the virtual machines. The first is a classic ethernet network. Each virtual machine has its own ethernet card. The second one uses Myrinet cards and provides very fast links between the nodes. The last one offers layer 2 virtualization using the EtherIP protocol [16]. EtherIP bridges are set between any virtual machine and the corresponding BSD machine.

To set up the bridges, we use a topology file given by the user. The format we use for the topology is the dot format [4] which is easy to manipulate and to write. There are two different types of nodes for the topology. The Xen nodes representing the virtual machines and the BSD nodes representing routers. The dot language being very simple it is easy to generate well-known topologies such as rings, cliques, etc. As the language is well-spread, there also exist graphical tools to design specific topologies.

Using the topology file, we generate the routing table of each BSD machine. The routing is made through a kernel module. More precisely it is a netgraph[5] node called "ip_switch". This node works with ipfw, allowing the user to filter packets and to redirect them into a netgraph node. Here we filter all EtherIP packets.

These packets are examined by the module who stores its routing table in a kernel hash table. After modifying the IP header of the packet to correctly route it to the next hop, the packet is put back on the network. The packet may also enter a Dummynet rule before or after being rerouted. The module can also deal with ARP (Address Resolution Protocol) requests in which case it will forward the request to all his neighbors.

3. Experiments

In this section we present the experiments we perform in order to assess the performances and functionalities of our virtualization framework. All these experiments were done on Grid'5000 [6]. Grid'5000 is a computer science project dedicated to the study of grids, featuring 13 clusters, each with 58 to 342 PCs, connected by the Renater French Education and Research Network. For our experiments we used a 312-node homogeneous cluster composed of AMD Opteron 248 (2.2 GHz/1MB L2 cache) bi-processors running at 2GHz. Each node feature 20GB of swap and SATA hard drive. Nodes were interconnected by a Gigabit Ethernet switch. All our experiments were performed using a folding ratio of 10 (e.g. each physical node runs 10 virtual machines).

All the following experiments ran under Xen version 3-2, with Linux-2.6.18.8 for the Physical and Virtual Computing nodes, and BSD version 7-0PRERELASE for the network emulation. Since we were embedding a Java virtual machine on the Xen virtual

[3]`http://www.freebsd.org/doc/en/books/handbook/firewalls-ipfw.html`
[4]`http://www.graphviz.org/doc/info/lang.html`
[5]`http://people.freebsd.org/~julian/netgraph.html`

Requested value	Measured value - w/o NE	Measured value - with NE
250 Mbps	241.5 Mbps	235 Mbps
25 Mbps	24.54 Mbps	23.30 Mbps
2,5 Mbps	2.44 Mbps	2.34 Mbps
256 Kbps	235.5 Kbps	235.5 Kbps

(a) Bandwidth restraint

Requested value	Measured value - w/o NE	Measured value - with NE
10 ms	10.1 ms	10.2 ms
50 ms	52.2 ms	52.1 ms
100 ms	100.2 ms	100.4 ms
500 ms	500.4 ms	500.8 ms

(b) Latency restraint

Table 1. Respect of network restraint conforming measures

machines we needed it to be light. We chose the 1.5.0_10-eval version that fulfilled our needs and our space requirements.

3.1. Impact of the Low-Level Network Emulation

We first measured the impact of the network emulation of V-DS on the network bandwidth and latency, using the netperf tool [22]. To do this, we used two version of V-DS: with network emulation at the high level only (when packets are slowed down by the router, but not encapsulated in an IP over ethernet frame), and with low-level network emulation, as described in section 2.

The experimental setup consisted in three physical machines: one running the BSD router, the other two running one virtual machine each. We configured the BSD router to introduce restraints on the network, either using low-level emulation with ethernet over IP, or without low-level emulation.

The results are summed up in table 1. The requested value represents the restraint imposed by the virtualization. In this table low-level network emulation is denoted as NE.

Regarding the bandwidth, the obtained values are very close to the requested ones, the difference being around 3%. This corresponds to the time spent in the traversing of the virtual layer. Netperf tests are realised at the TCP level, implying that part of the bandwidth is used for the TCP protocol.

When adding the low-level network emulation the bandwidth drops again for another 3%. This could be explained by the encapsulation needed by the etherip protocol for a full network emulation. There is no significant difference regarding the latency measures with the low-level emulation.

3.2. TCP Broken Connection Detection Mechanism

In this set of experiments, we stress the broken connection detection mechanism implemented in the TCP stack. Many applications rely on TCP detection mechanism to detect failures and implement their own fault-tolerance strategy, thus the efficiency of TCP failure detection has a significant impact on the efficiency of these applications. The failure detection mechanism of TCP relies on heartbeats, under the so-called pull model [3]: one peer sends a heartbeat to the other peer, and starts a timer; when a peer receives a heartbeat, it will send an acknowledgement back; if the acknowledge returns before the expiration of the timer, the sending peer assumes that the receiving peer is alive; if the timer expires before the reception of the acknowledge, the connection is broken.

This mechanism is controlled at the user level through BSD socket parameters: *SO_KEEPALIVE* enables the failure detection mechanism, which is tuned through *tcp_keepalive_time*, *tcp_keepalive_probes*, and *tcp_keepalive_intvl*. *tcp_keepalive_time* defines how long a socket can be without traffic before beginning the heartbeat protocol; *tcp_keepalive_probes* defines the number of heartbeats that can be lost on a socket before the connection is considered to be broken; *tcp_keepalive_intvl* defines the maximum time to wait before considering that a heartbeat has been lost.

To stress the failure detection mechanism of TCP, we designed three simple synthetic benchmarks. They all assume a single pair of client/server processes connected through TCP BSD sockets. In the first benchmark (*Send*), the client sends messages continuously to the server without waiting for any answer. In the second benchmark (*Recv*), the client waits for a message from the server. In the third (*Dialog*), the client and server are alternatively sending and receiving messages to/from each other.

In all those experiments the server is killed or destroyed right after the connection is established and we measure the elapsed time before the client realizes the connection has been broken. We set the *tcp_keepalive_time* to 30 minutes, the *tcp_keepalive_probes* to 9 and the *tcp_keepalive_intvl* to 75 seconds, which are the default on linux machines (except for the keepalive time, which was reduced to lower the duration of the experiments). As a consequence, when a machine crashes we expect the other side to notice the event within a period of approximately 41 minutes.

We then have two sets of experiments, one where the server process is killed and one where the machine hosting the process is destroyed. Each set of experiments includes the three benchmarks in both Java and ANSI C. Every benchmark is run twice, once where the SO_KEEPALIVE variable is on and once where it's off. The results are summed up in table 2.

Failure Injection Method	Language	Socket Option	Send	Receive	Dialog
Kill	C	-	0.2s	0.3s	0.2s
Kill	Java	-	N/A	N/A	N/A
Destroy	C	SO_KEEPALIVE	17min	41min	15min30s
Destroy	Java	SO_KEEPALIVE	∞	41min	15min30s
Destroy	C		17min	∞	15min30s
Destroy	Java		∞	∞	15min30s

Table 2. TCP failure detection times

In this table, the value ∞ means that after a long enough amount of time (several hours) exceeding significantly the expected time of the failure detection (41 minutes) the active computer has still not noticed that the connection has been broken. The N/A value means that the language or the system does not notify errors even if it detects a broken link. In the case of Failure Injection with the Kill method, the socket option SO_KEEPALIVE has no effect on the results.

When using the Kill failure injection method, one can see that the Linux operating system detects the failure at the other end almost instantaneously. This is due to the underlying TCP/IP protocol: the process is killed, but the operating system continues to work, so it can send the RST packet to the living peer, which will catch it and notify the process of a "failure". For the Java virtual machine, the socket is also notified as closed, but the language does not notify this closure as a failure: the code also has to check continuously for the status of the Input and Output streams, in order to detect that a stream was unexpectedly closed. In our JVM implementation, no exceptions were raised when sending on such a stream, and receptions gave null messages.

On the contrary, when using a more realistic destroy mechanism, the operating system of the "dead" peer is also destroyed. So, no mechanism sends a message to the living peer to notify of this crash. The living peer must rely on its own actions to detect failures, which is a more realistic behavior. We distinguish between the two cases when the SO_KEEPALIVE option is either on or not on the socket. In native Linux applications (ANSI C programs), the failure is always detected when the SO_KEEPALIVE option is on. TCP also uses the communications induced by normal traffic to detect a potential failure, that is why the detection time is lower for the Dialog and Send benchmarks.

In the case of the Recv benchmark, the living peer does not introduce communication in the network, so the system has to rely on the heartbeat procedure, which uses conservative values to detect the failure with a low chance of false positives, and a small perturbation of the network.

It is clear from these experiments that crash injection through complete destruction of the virtual machine, including the operating system, exhibit more accurate behavior than the simple destruction of a process, even using a forced kill method, because the underlying operating system will clean up the allocated resources, including the network resources.

3.3. Stress of Fault-Tolerant Applications

In order to evaluate the platform capabilities to inject failures, we stressed FreePastry which is an open-source implementation in Java of Pastry [21,9] intended for deployment on the Internet. Pastry is a fault-tolerant peer-to-peer protocol implementing distributed hash-tables. In Pastry every node has a unique identifier which is 128 bits long.This identifier is used to position the node on a 2^{128}-place oriented ring. A key is associated to any data, using a hash function, and each process of identifier $id < id'$ (where id' is the identifier of the next process on the ring) holds all data with key k such that $id \leq k < id'$. Then, by comparing the process identifiers and data keys, any process can route any message to a specific data. Shortcuts between nodes (called fingers in Pastry) are established to ensure logarithmic time to locate a node holding any data from any other node.

When a node is joining an existing ring, it gets a node id and initializes its leaf set and routing table by finding a "close" node according to a proximity metric. Then it asks this node to route a special message with its identifier as a key. The node at the end of the road is the one with the closest identifier and then the new node takes its leaf set and its routing table is updated with information gathered along the road. The new node will then send messages in the pastry network to update the routing table of all processes it should be connected to.

Pastry manages nodes failures as nodes departures without notification. In order to handle this kind of situation "neighbors" (nodes which are in each others leaf set) exchange keepalive messages. If a node is still not responding after a period T, it is declared failed and everyone in its leaf set is informed. The routing table of all processes that the departing process was connected to are then updated. This update procedure can take some time and is run during the whole life of the distributed hash table. At some point in time, the routes stop changing (they are stabilized), but the maintaining procedures for these routes continue to execute.

In order to validate the platform we looked at three things. First we evaluated the average time for the system to stabilize itself after all the peers had joined the network. Then we evaluated the average time needed for every node to know that a node was shut down or killed. In the first case we only kill a java process and in the second we "destroy" the virtual machine which is hosting the process.

The experiments go as follows. The first virtual machine (called the bootstrap node) creates a new ring and then every other virtual machine connects to it. We ask every node for its routing table every 200ms and log it whenever it changes together with a time stamp.

In order not to overwhelm the bootstrap node, we launch machines by groups of tens separated by a 1 second interval. The results for the first experiment are presented in figure 2 below.

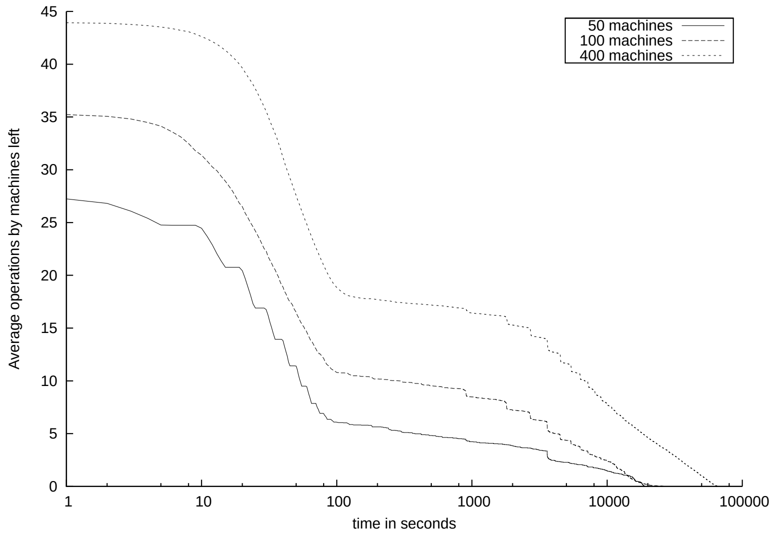

Figure 2. Average number of changes left by machines

It can be seen that for even small rings, composed of as few as 50 machines out of a possible 2^{128}, the time for the system to stabilize is huge (over 5 hours). This time increases with the numbers of machines and can still be over 18h for a ring as small as 400 machines.

To reduce the duration of the experiments, we made use of the fact that a majority of changes in the routing tables are made in the first few seconds of initialization. It appears that after only 100s more than 50% of the changes have been made. Thus we do not wait for the whole system to be stabilized before injecting the first failure, but we wait for the whole system to have made enough changes in the routing tables and for it to be in a relatively steady state. The first failure is injected 45min after the beginning of the experiment.

We call D-node the node we suppress from the ring, either by killing the process or destroying the machine. After suppressing the D-node we wait for 20 min for the nodes to update their routing tables. After this period we collect the routing tables and look for those which include the D-node. In those particular tables we search for the update that will make the D-node disappear from the routing table.

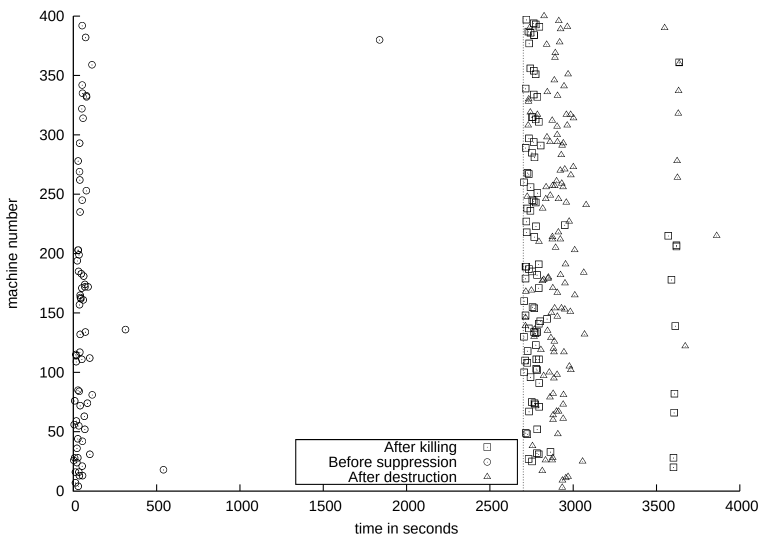

Figure 3. D-node deletion time

Figure 3 presents the cases when we "destroy" the virtual host of the process, and when we kill the process. Each dot in this figure represents the update of the routing table of process y, at a time x, concerning the D-node. The circles represent the modifications before the failure is injected, thus modifications due to the normal stabilization of FreePastry. The squares represent the modifications after the injection of the failure for the D-node in the case of process kill, and the triangles in the case of virtual host destruction. The vertical line represents the date of the failure injection at the D-node (45 minutes after the beginning).

The set of routing tables that include the D-node consists of 578 nodes over several experiments. In this set many nodes delete the D-node of their routing table before it is suppressed. As it can be seen on the figure, all these nodes do it very early in the stabilization and therefore we can consider that every node that deletes the D-node from its table after the suppression time does it thanks to the failure detection component of Pastry.

Since the routing table maintenance is done lazily in Pastry [8], it is natural that not every node updates its routing table, since in the experiments no messages are exchanged.

When we only kill the pastry process to suppress the D-node after 45 minutes we can see on figure 3 that a lot of nodes react in a very short period of time to the suppression of the D-node. Comparing the points distributions for Kill and Destruction, we can see that nodes detect the failure in a shorter period of time in the case of kill than in the case of destruction. Since behaviors in the two cases is different we can consider that "destroying" a machine is more accurate since the stressed application must rely on its own failure detection mechanism, and the behavior of this application may be influenced by the asynchronism and the timings of the failure detection mechanism used. The figure also demonstrates that the active failure detection mechanism of FreePastry is effective and the distributed hash table is able to stabilize even with accurate failure injection.

Conclusion and Future Work

In this chapter, we presented an emulation platform for grids where both the machines and the network are virtualized at a low level. This allows an experimenter to test realistic failure injection into applications running on distributed architectures, such as grids. We evaluated the interest of our approach by running a classical fault-tolerant distributed application: Pastry.

We are in the process of developping a fault injection tool to work with the platform. it will be an extension of the work started in the tool Fail [15]. The interest of this work is that using Xen virtual machines will allow to model strong adversaries since it is possible to have virtual machines with shared memory. These adversaries will be stronger since they will be able to use global fault injection strategies.

Part of this work is already available on the web[6] and a tutorial is also available online[7]. This version does not include the layer 2 network virtualization because it is not packaged yet and will be available as soon as possible.

Acknowledgements. Experiments presented in this chapter were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners.

[6]`http://www.lri.fr/~quetier/v-ds/v-ds-1.4.tgz`
[7]`http://www.lri.fr/~quetier/vgrid/tutorial`

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.

[2] William H. Bell, David G. Camerona, Luigi Capozza, A. Paul Millar, Kur t Stockinger, and Floriano Zini. Optorsim - a grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications,*, 17(4), 2003.

[3] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Erik Buchmann and Klemens Böhm. How to run experiments with large peer-to-peer data structures. *Parallel and Distributed Processing Symposium, International*, 1:27b, 2004.

[5] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource manageme nt and scheduling for grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 14(13-15), 2002.

[6] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages 99–106, Seattle, USA, November 13-14 2005. IEEE/ACM.

[7] H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings of the IEEE International Symposium on Cluster Computing and the G rid (CCGrid'01),Brisbane, Australia*, pages 430–437, may 2001.

[8] M. Castro, P. Druschel, Y.C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. 2003.

[9] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, December 2002.

[10] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[11] C. Dumitrescu and I. Foster. Gangsim: A simulator for grid scheduling studies. In *Proceedings of the IEEE International Symposium on Cluster Computing and the G rid (CCGrid'05), Cardiff, UK*, may 2005.

[12] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Trans. Netw.*, 9(4):392–403, 2001.

[13] Andreas Haeberlen, Alan Mislove, Ansley Post, and Peter Druschel. Fallacies in evaluating decentralized systems. In *In Proceedings of IPTPS*, 2006.

[14] Thomas Hérault, Thomas Largillier, Sylvain Peyronnet, Benjamin Quétier, Franck Cappello, and Mathieu Jan. High accuracy failure injection in parallel and distributed systems using virtualization. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 193–196, New York, NY, USA, 2009. ACM.

[15] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. Technical Report 1420, Laboratoire de Recherche en Informatique, Université Paris Sud, October 2005.

[16] R. Housley and S. Hollenbeck. EtherIP: Tunneling Ethernet Frames in IP Datagrams. RFC 3378 (Informational), September 2002.

[17] Ralph Niederberger. DEISA: Motivations, strategies, technologies. In *Proc. of the Int. Supercomputer Conference (ISC'04)*, 2004.

[18] Lucas Nussbaum and Olivier Richard. Lightweight emulation to study peer-to-peer systems. *Concurr. Comput. : Pract. Exper.*, 20(6):735–749, 2008.

[19] Benjamin Quétier, Vincent Neri, and Franck Cappello. Scalability Comparison of Four Host Virtualization Tools. *Journal of Grid Computing*, 5:83–98, 2006.

[20] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[22] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. Netpipe: A network protocol independent performace evaluator. In *In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[23] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kosti 'c, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and i mplementation*, pages 271–284, New York, NY, USA, 2002. ACM Press.

[24] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, 2002.