Large-scale Deployment in P2P Experiments Using the JXTA Distributed Framework

Mathieu Jan, Gabriel Antoniu, Luc Bougé, Sébastien Monnet

Large-scale Deployment in P2P Experiments Using the JXTA Distributed Framework

Gabriel Antoniu[1], Luc Bougé[2], Mathieu Jan[1], and Sébastien Monnet[3]

[1] IRISA/INRIA, `Gabriel.Antoniu@irisa.fr`
[2] IRISA/ENS Cachan, Brittany Extension
[3] IRISA/University of Rennes I

Abstract. The interesting properties of P2P systems (high availability despite peer volatility, support for heterogeneous architectures, high scalability, etc.) make them attractive for distributed computing. However, conducting *large-scale experiments* with these systems arises as a major challenge. Simulation allows only to partially model the behavior of P2P prototypes. Experiments on real testbeds encounter serious difficulty with *large-scale deployment and control* of peers. This paper shows how an optimized version of the *JXTA Distributed Framework* (JDF) can help deploying, configuring and controlling P2P experiments. We report on our experience in the context of our JUXMEM JXTA-based grid data sharing service for various configurations.

1 How to test P2P systems at a large scale?

The scientifi c distributed systems community has recently shown a growing interest in the Peer-to-Peer (*aka* P2P) model [1]. This interest is motivated by properties exhibited by P2P systems such as high availability despite peer volatility, support of heterogeneous architectures and, most importantly, high scalability. For example, the *KaZaA* network has shown to scale up to 4,500,000 users, an unreachable scale for distributed systems based on the traditional client-server model.

However, the experimental validation phase remains a major challenge for designers and implementers of P2P systems. Validating such highly-scalable systems requires the use of large-scale experimentations, which is extremely diffi cult. Consider for instance popular P2P software, like *Gnutella* or *KaZaA*: workloads of these systems are not fully analyzed and modeled because the behavior of such systems cannot be precisely reproduced and tested [2]. Recently, P2P systems like *CFS* [3], *PAST* [4], *Ivy* [5] and *OceanStore* [6] based on smarter localization and routing schemes have been developed. However, most of the experiments published for these systems exhibit results obtained either by simulation, or by actual deployment on small testbeds, typically consisting of less than a few tens of *physical* nodes [7]. Even when larger scales are reached via emulation [8], no experimental methodology is discussed for automatic deployment and volatility control. For instance, failures are simulated by manually stopping the peers using the `kill` signal! There is thus a crucial need for infrastructures providing the ability to test P2P systems at a large scale. Several approaches have been considered so far.

Simulation. Simulation allows one to define a model for a P2P system, and then study its behavior through experiments with different parameters. Simulations are often executed on a single sequential machine. The main advantage of simulation is the reproducibility of the results. However, existing simulators, like *Network Simulator* [9], or *SimGrid* [10], need significant adaptations in order to meet the needs of a particular P2P system. This holds even for specific P2P simulators, like *ng-simulator* used by *NeuroGrid* [11]. Also, the technical design of the simulated prototype may be influenced by the functionalities provided by the simulator to be used, which may result in deviations from reality. Last but not least, simulators model simplified versions of real environments. Further validation by alternative techniques such as emulation or experiments in real environments is still necessary.

Emulation. Emulation allows one to configure a distributed system in order to reproduce the behavior of another distributed system. Tools like *dummynet* [12] or *NIST Net* [13] allow to configure various characteristics of a network, such as the latency, the loss rate, the number of hops between physical nodes, and sometimes the number of physical nodes (e.g., *ModelNet* [14] and *Emulab/Netbed* [15]). This way, networks with various sizes and topologies can be emulated. However, the heterogeneity of a real environment (in terms of physical architecture, but also of software resources) cannot be faithfully reproduced. More importantly, deployment of P2P prototypes is essentially left to the user: it is often overlooked, but it actually remains a major limiting factor.

Experiments on real testbeds. Real testbeds such as *GridLab* [16] or *PlanetLab* [17] are large-scale, heterogeneous distributed environments, usually called *grids*. They are made of several interconnected sites, with various resources ranging from sensors to supercomputers, including clusters of PC. Such environments have proven helpful for realistic testing of P2P systems. Even though experiments are not reproducible in general on such platforms, this is a mandatory step in order to validate a prototype. Here again, deployment and configuration control have in general to be managed by the user. This is even more difficult than in the case of emulation, because of the much larger *physical* scale.

To sum up, actually *deploying and controlling* a P2P system over large-scale platforms arises as a central challenge in conducting realistic P2P experiments. The contribution of this paper is to introduce an enhanced version of *JXTA Distributed Framework* (JDF) [18] and demonstrate its use in large-scale experiments for a P2P system in a grid computing context.

2 Deploying and controlling large-scale P2P experiments in JXTA

In this paper, we focus on deploying and controlling P2P experiments on grid architectures. Let us stress that we do not consider the *actual testing strategy*, i.e., which aspects of the behavior are monitored and how. We only address the technical infrastructure needed to support specific testing activities. As seen in Section 1, just *deploying* a P2P prototype on such a scale is a challenging problem.

2.1 The five commandments of large-scale P2P experiments

A tool aiming to facilitate large-scale P2P experiments should, at least, observe the following 5 commandments.

Commandment C1: You shall provide the ability to easily specify the requested virtual network of peers. Therefore, a specification language is required to define what kind of application-specific peers are needed, how they are interconnected, using which protocols, where they should be deployed, etc.

Commandment C2: You shall allow designers of P2P prototypes to trace the behavior of their system. Therefore, the tool should allow to retrieve the outputs of each peer, such as log files as well as result files, for off-line analysis.

Commandment C3: You shall provide the ability to efficiently deploy peers on a large number of physical nodes. For instance, a hierarchical deployment strategy may be useful when testing on a federation of distributed clusters.

Commandment C4: You shall provide the ability to synchronize peers between stages of a test. Indeed, a peer should have the possibility to wait for other peers to reach a specific state before going through the next step of a test.

Commandment C5: You shall provide the ability to control the simulation of peers' volatility. In typical P2P configurations, some peers may have a high probability of failure, while others may be almost stable. The tool should allow one to *artificially enforce* various volatile behaviors when testing on a (hopefully stable!) testbed.

2.2 The JXTA Project

The JXTA platform (for *juxtapose*) [19] is an open-source framework initiated by Sun Microsystems. JXTA specifies a set of language- and platform-independent, XML-based protocols. It provides a rich set of building blocks for P2P interaction, which facilitate the design and implementation of custom P2P prototypes. The basic entity is the regular peer (called *edge peer*). Before being able to communicate within the JXTA virtual network, a peer must discover other peers. In general, this is done through the use of specialized *rendezvous peers*. Communication between peers is direct, except when firewalls are in the way: in this case, *relay peers* must be used. Peers can be members of one or several *peer groups*. A peer group consists of peers that share a common set of interests, e.g., peers that share access to some resources. The reference implementation of JXTA is in Java. Its current version 2.2.1 includes around 50,000 lines of code. An implementation in C is under development.

2.3 The JXTA Distributed Framework Tool (JDF)

The purpose of JDF is to facilitate automated testing of JXTA-based systems. It provides a generic framework allowing to easily define custom tests, deploy all the required

```
<network analyze-class="prototype.test.Analyze">
  <profile name="Rendezvous">
        <!-- peer information -->
    <peer base-name="peerA" instances="1"/>
        <!-- rendezvous information -->
    <rdvs is-rdv="true"/>
        <!-- transport information -->
    <transports>
      <tcp enabled="true" base-port="13000"/>
    </transports>
        <!-- bootstrap information -->
    <bootstrap class=
              "prototype.test.Rendezvous"/>
  </profile>
  <profile name="Edge">
    <peer base-name="peerB" instances="1"/>
    <rdvs is-rdv="false">
      <rdv profile="Rendezvous"/>
    </rdvs>
    <transports>
      <tcp enabled="true" base-port="13000"/>
    </transports>
    <bootstrap class="prototype.test.Edge"/>
  </profile>
</network>
```

```
public class Peer extends JxtaBootStrapper {

    public static void main(String[] args) {
        Peer peer = new Peer();
        peer.start(args);
        peer.stop();
    }

    // Start the test on the local peer
    public void start(String[] args) {
        // Start JXTA locally
        super.startJxta();
        // Get a custom P2P service
        p2pService = group.lookupService(...);
        // Use the API of the service
        p2pService.doSomething();
    }

    // Stop the test on the local peer
    public void stop() throws Exception {
        // Stop the custom P2P service
        p2pService.stopApp();
        // Stop JXTA locally
        super.stopJxta();
    }

    // Store the local results
    protected void updateProperties() {
        super.updateProperties();
        // store own results ...
        setProperty(PROPERTY_TAG,
                       property_result);
    }
}
```

Fig. 1. An example of required input files: a network description file defining 2 profiles (left), and a basic Java test class inside the original JDF's framework from Sun Microsystems (right).

resources on a distributed testbed and run the tests with various configurations of the JXTA platform.

JDF is based on a regular Java Virtual Machine (JVM), a Bourne shell and `ssh` or `rsh`. File transfers and remote control are handled using either `ssh`/`scp` or `rsh`/`rcp`. JDF assumes that all the physical nodes are visible from the control node. JDF is run through a regular shell script which launches a distributed test. This script executes a series of elementary steps: install all the needed files; initialize the JXTA network; run the specified test; collect the generated log and result files; analyze the overall results; and remove the intermediate files. Additional actions are also available, such as killing all the remaining JXTA processes. This can be very useful if the test badly failed for some reason. Finally, JDF allows one to run a sequence of such distributed tests.

A distributed test is specified by the following elements. 1) A *network description file* defining the requested JXTA-based configuration, in terms of edge peers, rendezvous peers, relay peers and their associated Java classes, and how they are linked together. This description is done through the use of profiles, as shown on the left side of Figure 1. Two profiles are described. Profile `Rendezvous` specifies a rendezvous peer configured to use TCP. Its associated Java class is `protoype.test.Rendezvous`. Profile `Edge` specifies an edge peer. It is configured to use a peer with Profile `Rendezvous` for its rendezvous peer through TCP a connection. The `instance` attribute of the `peer` tag specifies how many peers of the profile will be launched on a given physical node. 2) The set of Java classes describing the behavior of each

```
<profile name="RendezVous1" replicas="1"/>      <profile name="RendezVous1" replicas="1"/>
<profile name="RendezVous2" replicas="1"/>      <profile name="RendezVous2" replicas="1"/>
<profile name="Edge1" replicas="9"/>            <profile name="Edge1" replicas="99"/>
<profile name="Edge2" replicas="9"/>            <profile name="Edge2" replicas="99"/>
```

Fig. 2. A small JXTA network (left), and a large one (right).

peer. These Java classes must *extend* the framework provided by JDF, in order to easily start JXTA, stop JXTA and save the results into files, as shown on the right side of Figure 1. These result files are collected by JDF on each physical node and sent back to the control node to be analyzed by an additional Java class specified by the user (`prototype.test.Analyze` in the example). 3) A *node file* containing the list of physical nodes where to deploy and run the previously described JXTA network, as well as the path of the JVM used on each physical node. 4) An optional file containing the list of librairies to deploy on each physical node (a default one is provided if omitted).

3 Improving and extending the JXTA Distributed Framework

We are currently building a prototype for a data-sharing service for the grid, based on the JXTA platform. Therefore, we need a tool to easily deploy and control our prototype for large-scale experiments. We describe how we improved the JDF tool, with the goal of making it better observe the 5 commandments stated above.

3.1 Improving JDF functionalities

Commandment C1 requires JDF to provide the ability to easily specify the requested virtual network of peers. Developers can easily modify the number of peers hosted on each physical node, but JDF requires that a *specific* profile must be explicitly defined for each physical node, which is obviously not scalable. To facilitate testing on various large scale configurations, we introduced the ability to specify that a single profile can be shared by multiple physical nodes. This is specified by the `replicas` attribute. Let us assume we deploy JXTA on 2 clusters of 10 nodes. Each cluster runs one JXTA rendezvous peer, to which 9 edge peers are connected. Peers with profile `Edge1` are connected to the peer with profile `Rendezvous1` and similarly for peers with profiles `Edge2` and `Rendezvous2`. In the original version of JDF, this would require 2 rendezvous profiles plus 2×9 edge profiles. In contrast, our enhanced version of JDF requires only 2 rendezvous profiles plus 2 edge profiles. An excerpt of the corresponding new JDF network file is shown on the left side of Figure 2.

Now, imagine we want to run the same experiment on a larger JXTA configuration, in which 99 edge peers are connected to a rendezvous peer in each of two 100-nodes clusters. Thanks to the `replicas` attribute, it suffices to substitute 99 for 9 in the network description file. In the original JDF, this would require to define 2×99 edge profiles plus 2 rendezvous profiles!

Other enhancements mainly include improved performance for various phases of JDF execution. For instance, only the modified files are transmitted when updating the deployed environment. The ability to use NFS instead of `scp`/`rcp` has also been added when deploying a distributed test on a NFS-enabled cluster.

3.2 Enhancing JDF to interact with local batch systems

An important issue when running tests on grid infrastructures regards the necessary interaction between the deployment tool and the resource allocator available on the testbed. Many testbeds are managed by batch systems (like PBS [20], etc.), which dynamically allocate physical nodes to jobs scheduled on a given cluster. To observe commandment C3 regarding efficient deployment, we have enhanced JDF to make it interact with local batch systems via Globus, or directly with PBS.

JDF takes a static node file as input, which explicitly lists the physical nodes involved. On testbeds using dynamic resource allocators, such a list cannot be provided at the time of the job submission. Our enhanced version of JDF allows the node file to be dynamically created once the job is actually scheduled, using the actual list of physical nodes provided by the batch system.

This solves the problem on a single cluster. However, conducting P2P experiments on a federation of clusters requires to co-schedule jobs across multiple batch systems. This problem is beyond the scope of this paper, but we are confident that JDF will easily take advantage of any forthcoming progress in this domain.

3.3 Controlling the simulation of volatility

One of the most significant features of P2P systems is to support a high volatility. We think that a JDF-like tool should provide the designer with the possibility of running his system under various volatility conditions (Commandment C5). Using a stable testbed, the tool should provide a means to enforce node failures in a controlled way. The original version of JDF did not consider this aspect. We have therefore developed an additional facility to generate a *failure control file* from various volatility-related parameters. This file is given as an input to JDF, in order to control peer uptime. The file is statically generated before the test is launched, based on (empirical) statistical laws and on relevant parameters (e.g., the MTBF of the network peers). At runtime, the uptime of each peer is controlled by an additional service thread, which is started by JDF. This thread is used to kill the peer at the specified time. This way, the volatility conditions for a given test can be changed by simply varying the MTBF parameter. Note that using a pre-computed failure control file also enhances the reproducibility of the test.

As of now, the failure control file is generated according to a simple statistical law, where peer failures are independent. It would be interesting to consider dependencies between peers: the failure of one specific peer may induce the failure of some others. These dependencies could be associated with probabilities, for instance, the failure of peer A induces the failure of peers B and C with a probability of 50%. This should make it possible to simulate complex correlated failures, for instance, network partitions. Another direction would be to let the failure injection mechanism take into account the peer status. The JDF failure control threads could be used to monitor the state of their respective peers and regularly synchronize to generate correlated failures.

4 Case study: towards large-scale experiments in JUXMEM

We are using this enhanced version of JDF to deploy and control our JXTA-based service called JUXMEM (for Juxtaposed Memory) [21]. We believe that these results can be applied to other JXTA-based services, such as the Meteor project [22]. Actually, we were able to deploy and run Meteor's peers using JDF without any modification in the prototype.

4.1 The JUXMEM Project: a JXTA-based grid data-sharing service

JUXMEM is designed as a compromise between DSM systems and P2P systems: it provides *location transparency* as well as *data persistence* in a *dynamic environment*. The software architecture of the data-sharing service mirrors a hardware architecture consisting of a federation of distributed clusters. The architecture is therefore *hierarchical*. Its ultimate goal is to provide a data sharing service for grid computing environments, like DIET [23].

JUXMEM consists of a network of peer groups, called *cluster* groups, each of which generally corresponds to a physical cluster. All the *cluster* groups are enclosed by the *juxmem* group, which includes all the peers who are members of the service. Each *cluster* group consists of a set of peers which provide memory for data storage (*providers*). A *manager* monitors the providers in a given *cluster* group. Any peer can use the service to allocate, read or write to data as a *client*. All providers which host copies of the same data block make up a *data* group, uniquely identified by an ID. Clients only need to specify this ID to read/write a data block: the platform transparently locates it. JUXMEM can cope with peer volatility: each data block is replicated across a certain number of providers, according to a redundancy degree specified at allocation time. The number of replicas is dynamically monitored and maintained through dynamic replication when necessary.

4.2 Experimenting with various configurations

For our experiments, we used a cluster of the Distributed ASCI Supercomputer 2 (DAS-2) located in The Netherlands. This cluster is made of 72 nodes, managed by a PBS scheduler. Once it is scheduled, the JDF job deploys JXTA peers over the available cluster nodes using `ssh`/`scp` commands. Figure 3 reports the time needed to deploy, configure and update a JUXMEM service on a variable number of physical nodes using our optimized version of JDF.

The control node executes a sequential loop of `scp`/`ssh` commands in order to deploy JUXMEM and its whole environment on each physical node. The total size of all libraries for JXTA, JDF and JUXMEM is about 4 MB. As expected, the *deployment* time is thus linear with the number of physical nodes, e.g., it takes 20 s for 16 nodes, 39 s for 32 nodes.

Of course, this initial deployment step is required only once per physical node, so its cost can be "shared" by a sequence of experiments. Usually, these experiments consist of incremental modifications to some specific libraries. In this case, only the modified files need to be transmitted and updated on each physical node. For instance, the size of JUXMEM's `jar` file is around 100 kB. Modifying JUXMEM only requires the transmission of this file. This is much faster, as reported by the *update* curve.

As seen in Section 2.3, the JDF network description file defines what types of JXTA peers are requested, how they are interconnected, etc. The configuration step on a given

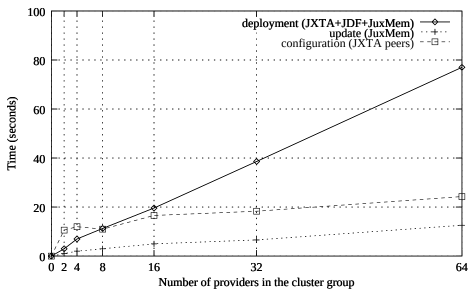

Fig. 3. Time needed to deploy, update and configure the JuxMem service on various network sizes

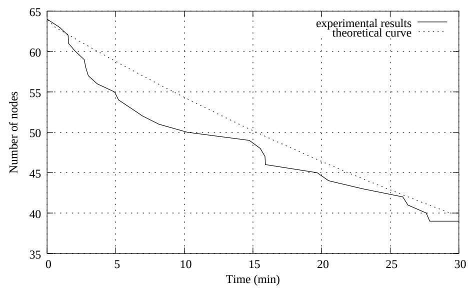

Fig. 4. Number of remaining peers in an experiment with a MTBF set to 1 minute

physical node consists in 2 phases: 1) based on this description, generate the specific JXTA configuration file of each peer; 2) for each peer, update this file with its specific list of rendezvous peers. As expected, the *configuration* curve shows that this takes a time which increases slowly and linearly with the number of physical nodes.

Note that, in order to reach larger scales, each physical node can host several peers. This is easily specified using the `instances` attribute of the `peer` tag in the JDF network description file. For instance, we have been able to deploy 10 peers on each node without any significant overhead, amounting to a total of 640 peers without any additional effort.

Finally, once the configuration is complete, JuxMem is started by invoking a JVM for each peer. This initialization time is dependent on the number of peers launched on each physical node. According to commandment C4, JDF should provide a way to synchronize the initialization of JXTA peers. This could be handled by a JDF control thread, through a mechanism similar to the failure control. This feature is still being implemented at this time.

4.3 Experimenting with various volatility conditions

The simplest way to model architecture failures is to provide a global MTBF. Our enhanced version of JDF allows the programmer to generate a failure schedule according to a given MTBF. We carried out a sample experiment on 64 physical nodes with a failure control file generated to provide a MTBF of 1 minute: the uptime in minutes for each physical node follows an exponential distribution with a rate parameter of $\frac{1}{64}$. The files automatically collected by JDF allow to check which peer went down, and when. Figure 4 reports the results of a specific single run of 30 minutes: 25 peers were killed. Observe that these results are somewhat biased as peers do not start exactly at the same time, and the clocks are not perfectly well-synchronized.

5 Conclusion

Validating P2P systems at a large scale is currently a major challenge. Simulation is nowadays most-widely used, since it leads to reproducible results. However, the significance of these results is limited, because simulators rely on a simplified model of reality. More complex validation approaches based on emulation and execution on "real" large-scale testbeds (e.g., grids) do not have this drawback. However, they leave the deployment at the user's charge, which is a major obstacle in practice.

In this paper, we state ***five commandments*** which should be observed by a deployment and control tool to successfully support large-scale P2P experiments. Our contribution consists in enhancing the *JXTA Distributed Framework* (JDF) to fulfill some of these requirements. This enhancement mainly includes a more precise and concise specification language describing the virtual network of JXTA peers, the ability to use various batch systems, and also to control the volatility conditions during large-scale tests. Some preliminary performance measurements for the basic operations are reported.

Further enhancements are needed for JDF to fully observe the five commandments. A hierarchical, tree-like scheme for the `ssh`/`rsh` commands could be used to balance the load of copying files from the control node to other physical nodes, along the lines of [24]. We plan to integrate a synchronization mechanism for peers to support more complex distributed tests. The final goal is to have a rich generic tool allowing to deploy, configure, control and analyze large-scale distributed experiments on a *federation of clusters* from a single control node. We intend to further develop this tool, in order to use it for the validation of JUXMEM, our JXTA-based grid data sharing service. JDF could also be very helpful for other JXTA-based services, and the approach can be easily generalized to other P2P environments.

Acknowledgments The authors thank Thilo Kielmann's group from the Vrije Universiteit for making the DAS-2 testbed available to us. We also thank Bernard Traversat, Senior Architect of Project JXTA at Sun Microsystems, for his useful comments, as well as the reviewers, for their suggestions.

References

1. Milojicic, D.S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z.: Peer-to-peer computing. Technical Report HPL-2002-57, HP Labs (2002) available at `http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf`.
2. Gummadi, K.P., Dunn, R.J., Saroiu, S., Gribble, S.D., Levy, H.M., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: 19th ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing, NY, ACM Press (2003) 314–329
3. Dabek, F., Kaashoek, F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise, Banff, Alberta, Canada (2001) 202–215
4. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise, Banff, Alberta, Canada (2001) 188–201
5. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy a read/write peer-to-peer file system. In: 5th Symposium on Operating Systems Design and Implementation (OSDI '02), Boston, MA (2002) 31–44

6. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: OceanStore: An architecture for global-scale persistent storage. In: 9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000). Number 2218 in Lecture Notes in Computer Science, Cambridge, MA, Springer-Verlag (2000) 190–201
7. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the OceanStore prototype. In: 2nd USENIX Conference on File and Storage Technologies (FAST '03), San Francisco, CA (2003) 14
8. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. Technical Report CSD-03-1299, UC Berkeley (2003) Available at `http://oceanstore.cs.berkeley.edu/publications/papers/`.
9. The ns manual (formerly ns notes and documentation). `http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf`
10. Casanova, H.: SimGrid: A toolkit for the simulation of application scheduling. In: First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia (2001) 430–441
11. Joseph, S.: P2P metadata search layers. In: Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003). Number 2872 in Lecture Notes in Computer Science, Bologna, Italy, Springer-Verlag (2003)
12. Rizzo, L.: Dummynet and forward error correction. In: 1998 USENIX Annual Technical Conference, New Orleans, LA (1998) 129–137
13. Carson, M., Santay, D.: NIST Net - a Linux-based network emulation tool. (2004) To appear in special issue of Computer Communication Review.
14. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J., Becker, D.: Scalability and accuracy in a large-scale network emulator. In: 5th Symposium on Operating Systems Design and Implementation (OSDI '02), Boston, MA (2002) 271–284
15. White, B., Lepreau, J., Stoller, L., Ricci, R., Newbold, S.G.M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: 5th Symposium on Operating Systems Design and Implementation (OSDI '02), Boston, MA (2002) 255–270
16. Allen, G., Davis, K., Dolkas, K.N., Doulamis, N.D., Goodale, T., Kielmann, T., Merzky, A., Nabrzyski, J., Pukacki, J., Radke, T., Russell, M., Seidel, E., Shalf, J., Taylor, I.: Enabling applications on the grid: A GridLab overview. International Journal of High Performance Computing Applications **17** (2003) 449–466
17. PlanetLab: an open community testbed for planetary-scale services. `http://www.planet-lab.org/pubs/2003-04-24-IntelITPlanetLab.pdf`
18. JXTA Distributed Framework. `http://jdf.jxta.org/`
19. The JXTA project. `http://www.jxta.org/`
20. The Portable Batch System. `http://www.openpbs.org/`
21. Antoniu, G., Bougé, L., Jan, M.: JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. Kluwer Journal of Supercomputing (2004) To appear. Preliminary electronic version available at URL `http://www.inria.fr/rrrt/rr-5082.html`.
22. Project Meteor. `http://meteor.jxta.org/`
23. The DIET project: Distributed interactive engineering toolbox. `http://graal.ens-lyon.fr/~diet/`
24. Martin, C., Richard, O.: Parallel launcher for clusters of PC. In Imperial College Press, L., ed.: Parallel Computing (ParCo 2001), Naples, Italy, World Scientific (2001) 473–480