Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing Group

Semester Thesis

PlanetLab Tests

Stefan Weber
weberste@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Stefan Schmid and Thomas Locher

Department of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2006

PlanetLab Tests

Stefan Weber

October 17, 2006

Abstract

In this thesis, we present an approach to perform measurements in the worldwide research network PlanetLab. We describe a framework for developing and deploying Java applications on PlanetLab and show how it can be used to develop data gathering applications. In particular, we implement a latency and a bandwidth test of which we also present some findings derived from its analyses.

Contents

1 Introduction

When developing distributed systems, it is often difficult to properly deploy them because access to possibly hundreds of distributed nodes all over the world is needed. Even though PlanetLab is providing such a network, it might still be helpful if a local simulation of the system could be made before the actual deployment. In order to make the simulation as realistic as possible, it can use network information, such as bandwidth and latency, that was measured beforehand in a real network. In addition, by analyzing the collected data, the network can be better understood and consequently, the simulation can be improved. It may even be possible to derive a realistic model for networks from that data.
In this thesis, we present a framework that simplifies the development and deployment of Java applications in general and network information gathering applications in particular. Furthermore, we also give an overview of the measurements we performed ourselves and provide an analysis of them.
The remainder of the thesis is organized as follows:
In Section 2 we give a short overview of PlanetLab itself. Then we present the architecture of the framework and its usage in Section 3. Section 4 contains an overview of the tests we did ourselves. The analyses of these tests are part of Section 5. The thesis concludes in Section 6.

2 PlanetLab

PlanetLab is an open platform for developing and deploying planetary-scale services and serves as a testbed for computer networking and distributed systems. It was established in 2002 and is today composed of over 700 nodes at 340 sites (October 2006). Access to the network is limited to persons affiliated with corporations and universities that host a node themselves.
A user is assigned to a *slice*. A slice is a set of allocated resources distributed across PlanetLab. To most users, this means shell access to a number of nodes, which can be chosen over a simple web interface by the user. As mentioned in the introduction, shell access is an important precondition in order to use a network for testing a distributed application. To simplify the handling of the nodes, for example to install software, PlanetLab users have contributed many helpful tools that are freely accessible.
Offering the above possibilities and the fact that PlanetLab itself is “part of the Internet” it is an ideal candidate for collecting network data that can be used in simulations and analyzed to get a better understanding of how global networks function.
A novice PlanetLab user can find further information in the *PlanetLab User's Guide* [8] and the *PlanetLab Quick Start Manual* [12].

2.1 Deploying the Application

When working with PlanetLab, one of the first barriers is the deployment of the application. Especially if, as in our testing application, several hundred nodes are involved, it is too cumbersome to do everything manually. But as mentioned above, there are quite a lot of tools which simplify these tasks. An overview of these tools can be found on the Contributed Software page [4] of the PlanetLab wiki [9].
For our needs we found *Codeploy* [2] helpful. It is an efficient, scalable deployment service for PlanetLab that is built on top of the *CoDeeN* [1] content distribution network. In most cases, `multiquery` and `multicopy`, two nice little tools that come with Codeploy were the tools of choice. While `multiquery` can be used to issue shell operations on a set of nodes, `multicopy` is useful to transfer files.

2.2 Installation of the JVM

We decided to implement the testing framework and consequently also the actual tests in Java. On the one hand, its dynamic class loading mechanism is very useful when deploying the application (more on that later) and, on the other hand, the network API is very simple but still powerful enough.
Hence, we first had to install the JVM on all the nodes. A nice tool to install the JVM and other applications is *Stork* [10], a software installation utility for PlanetLab. Unfortunately, there were some problems with it by the time we had to perform the installation. Therefore, we wrote a few shell scripts that, together with `multiquery` and `multicopy`, got the job done. The scripts can be found in the archive file that comes with this thesis. However, I would recommend to use Stork if possible.

3 Testing Framework

As mentioned in the introduction, our primary goal was to collect network information such as latency and bandwidth data. In order to make the data gathering as easy as possible and to minimize the time to distribute the test application, we decided that one of the nodes in PlanetLab has to act as a central authority that issues the test on all nodes involved in the test and collects the data from them once they have finished.
As these are actions that are common to probably any kind of testing one can imagine we built a testing framework that does this work. With the help of the framework, the developer of a test can focus on the actual test. It turned out that the testing framework can even be used for the development and/or deployment of any kind of Java application—not only in PlanetLab but basically in any kind of TCP/IP network.

3.1 Architecture

The testing framework has three major components. The *Master component* which is the main authority of the test and running on only one node, the *Slave component* which is the actual test and therefore is running on every node that participates in test and the *Launcher component* which is contacted by the Master component to deploy the Slave component (i.e. the actual test). Before these components are described in more detail, the workflow of a test is explained:

Workflow of a Test Similar as in JUnit [5], a test is made up of three phases:

1. *Setup Phase* This is the phase where the initialization is done. Usually, the main task of this phase is to load the classes that are needed for the test from the Master node and the test environment is set up automatically.
2. *Start Phase* This is the phase of the actual test. For example, in case of a ping test the nodes will mutually send ping packages and measure the time needed to receive an answer.
3. *Shutdown Phase* In this phase, usually some cleanup is done, for example terminating running servers and sending results back to the Master component.

The whole architecture is depicted in Figure 1 and is now described in more detail:

Master component The Master component is the central point of authority and is running on only one node, the *Master Node*. In order to tell the Launcher what to do, the Master sends command objects which, as the name implies, describes an action that has to take place.
In Figure 1, `Master` is the main logic of the component, `Commands` is the command repository, and `Nodes` contains a list of all the nodes that are part of the test. The `ResourceService` is used for transferring resources from the Master to the Launcher and vice versa and is described in Section 3.2.

Launcher component The Launcher component runs as a service on all the nodes that are part of the test. It listens on a predefined port with a `ServerSocket` until the Master component of a test establishes a connection. In Figure 1, `Launcher` is the main logic of the component. When a command is received from the `Master`, it is put into the `CommandQueue`. The `TestThread` serves as a sandbox where the commands can be executed without disturbing the `Launcher` (see Section 3.3). The `ResourceClient`

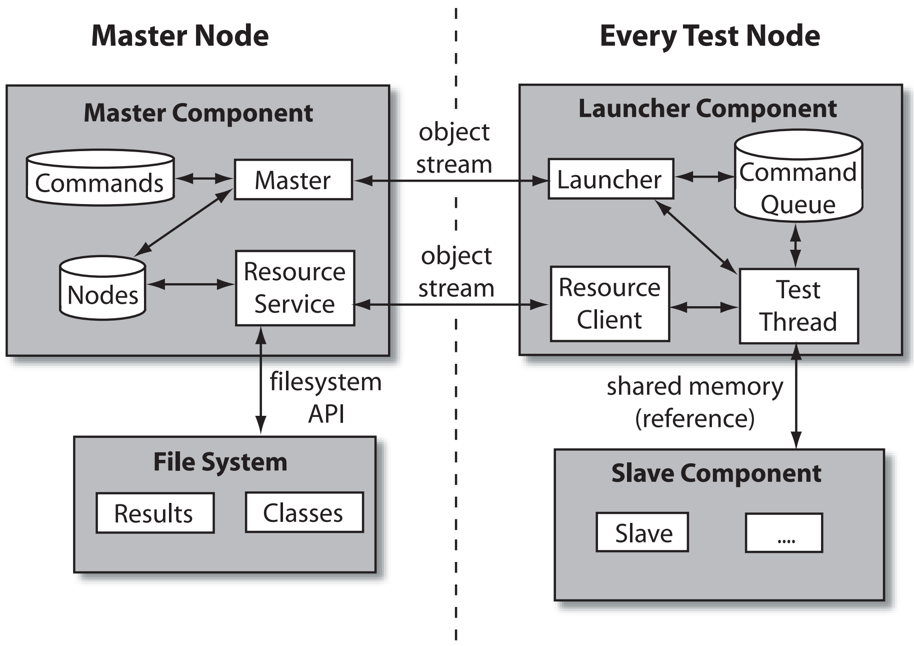


Figure 1: Architecture of the Test Framework

is a `ClassLoader` that is fetching resources from the Master's `Resource Service`.

Slave component The Slave component is the actual test implementation. Therefore, it has to be loaded from the Master before the test can be started. This is done with the `ResourceClient` of the Launcher component. In Figure 1, `Slave` is the main logic of the component. Depending on the actual test, there can be other entities in this component, which is indicated by the three dots.

Commands As already mentioned, commands are used by the `Master` to signal the `Launcher` what has to be done next. There are three main commands in the framework and each of them signals the Launcher to start one of the three phases of the workflow. These are the `SetupTestCommand`, the `StartTestCommand` and the `ShutdownTestCommand`.
If the user of the framework wants to introduce additional commands, this is also possible. The new command simply has to extend the abstract base class `Command`. The actual action is defined in the method `handleRequest` which is a template method that is called by the `TestThread` whenever the command is executed.

3.2 Loading Tests

As explained above, the actual code of the test is deployed during the setup phase. This is done with the help of the `ResourceService` that is listening on a predefined port on the Master node. It receives a `ResourceRequest` object that encodes the type of resource that is needed; when loading a component, this is most likely a Java class. Another possibility is that the *nodelist* to be used in the test can be loaded that way.
All the resources that can be loaded via the `ResourceService` have to be available on the Master's local filesystem. The filesystem is also used to store the tests that the Slave components are sending once the test has finished.

3.3 Running Tests

As the code of a test can be defined by the user of the framwork, it is most likely that from time to time there are errors in the code. If these errors let the `Launcher` quit or make it unusable, this would force restarting it, which is not very user-friendly. Therefore, the test runs in a test environment, the `TestThread`, which is running independently from the main thread and thus serves as a kind of sandbox. Every time the `Launcher` receives a `SetupTestCommand`, a previously running `TestThread` is interrupted and a new testing environment is set up for the new test. This means that if a previous test is blocked or still running and waiting for a chance to stop, the new test nevertheless is executed without any problems.

3.4 Implementing Tests

Each test needs a Master and a Slave component. To implement them, the abstract base class `Master` respectively `Slave` has to be extended.
The class `Master` has a few methods that do a lot of the work involved in every test, as for example sending the commands to the Launcher. Usually, the class that is extending `Master` uses several of the predefined methods and also contains a `main` method because the Master component is the one that has to be executed to start a test.
When implementing the Slave component there is a bit more to consider. The code that is part of the command objects is hard-coded. But of course, every test wants to do very specific things in each of the phases. Thus, the `Slave` class implements three methods `setup`, `start` and `shutdown` according to the phases of the workflow. Hence, code that should be executed when receiving the corresponding command has to be put in these methods by overriding them in the subclass. By default (i.e. if you don't override them), the methods do nothing.
Figure 2 shows how the `SetupTestCommand` of a test is executed. In this example, we assume that a ping test has to be done and thus the main logic of the Master component is called `PingMaster` and the main logic of

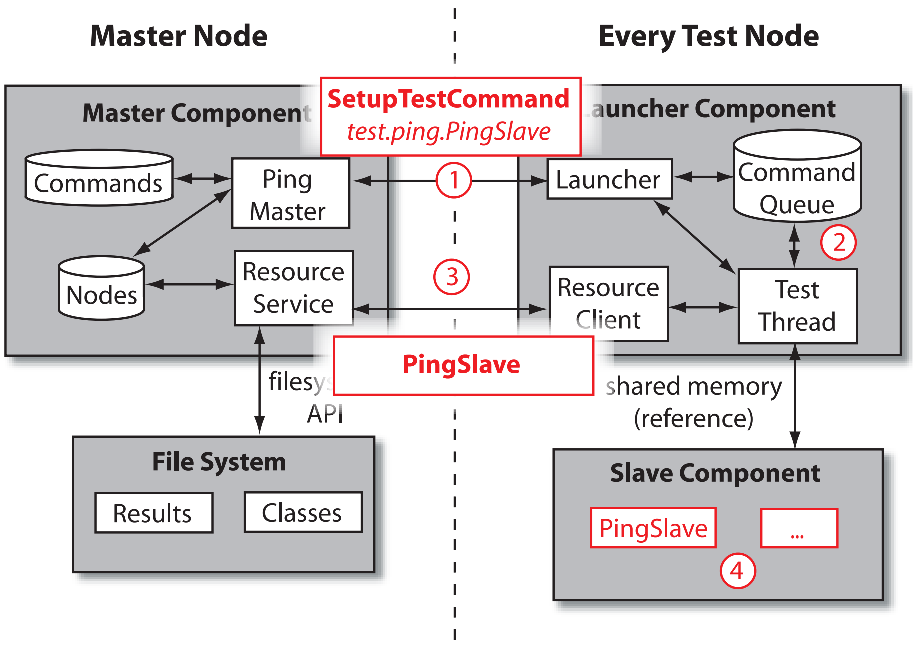


Figure 2: Issuing a SetupTestCommand

the Slave component is called `PingSlave`. At the beginning of the test, the Slave component is empty and we now explain how the whole component is loaded as part of the ping test's setup phase:

1. `PingMaster` gets a `SetupTestCommand` from the command repository and sends it via the object stream to the `Launcher`. The latter puts the command into the `CommandQueue`.

2. The `TestThread` fetches the command out of the queue and executes it.

3. As it is a `SetupTestCommand`, the Slave component has to be loaded. To do this, the `RemoteClient` can use the Master's `RemoteService`.

4. Finally, `PingSlave` and all the other resources part of the ping test are loaded.

We don't go further in explaining the usage here. The inclined reader is referred to the comments in the source code and the two example tests (latency and bandwidth) that we implemented. By the way, to distribute the Launcher component, the easiest way is to build a jar file containing the packages `launcher`, `slave`, and `shared` and using one of the methods mentioned in Section 2. For the Master node, a jar file containing the packages `shared`, `master`, and the package of the actual test can be uploaded and executed.

4 Tests

We implemented two tests using the testing framework. In this section, we give an overview of the results. An analysis is given in Section 5.
An important issue when implementing a test is the total runtime. When processing one node after the other, it usually takes too much time. To speed everything up, we always had 100 nodes that were executing the same command (i.e. the same test phase) in parallel. This reduced the total runtime significantly.
Another straightforward possibility to save time is to set a timeout when connecting to another node. In most cases, the connection is set up within a few seconds and if there is no connection after 10 seconds, it most likely doesn't work at all. However, the default timeout is set to over 3 minutes. As it happens quite regularly that two nodes cannot connect for whatever reason, a waiting time of 3 minutes significantly slows down the whole test. We therefore recommend to set the timeout to about 30 seconds.

You can find all the results as CSV files in the archive. The naming format is `result_{test}_{date}_at_{time}.csv` where `{test}` is substituted by the actual test (ping or bandwidth), `{date}` by the date in the format "yyyymmdd" and `{time}` by the European time when the test was running. In case of the bandwith tests, the time is omited as each test runs for about 24 hours anyway.

4.1 Latency

An interesting parameter in a network is the latency. As usual, we test it by sending a small packet to a node and measuring the time needed until an answer packet is received (ping).
It is easy to implement such a test in Java. First, however, we wanted to make sure that the JVM introduces no additional delay for whatever reason. Therefore, we compared the results of a ping test with the results achieved when using the `ping` command from the *iputils* package and didn't find significant differences.
Usually, the test was running on about 350 nodes—even though there is a total of over 700 nodes of which about 550 are production nodes [7]. To save time, it makes sense to only add the nodes to the test bench that are probably available. We propose to use *CoMon* [3], a monitoring infrastructure for PlanetLab, to get a list of currently alive nodes because by far not all production nodes are really available. For a list that can easily be copied

into a text file and later serve as nodelist for the test, the URL [6] can be used.
In every test, each node pinged every other node four or eight times in a row and the values were measured in milliseconds. A value of -1 indicates a problem. Most likely this is a connection problem because, when trying to ping these nodes manually with the `ping` command afterwards, it usually ended up with a 100% packet loss.
A value of 0 doesn't really mean that there is no latency, of course. The reason for these values is that Java gives access to the system time only in milliseconds. Hence, for two very fast nodes it is possible, that the time when the packet is sent is equal to when the packet is received. When analyzing or using the data, the 0 values can be a problem in certain calculations (e.g. division by zero), thus we incremented all the values by 1, which can be interpreted as kind of local overhead; of course, -1 values remain -1.

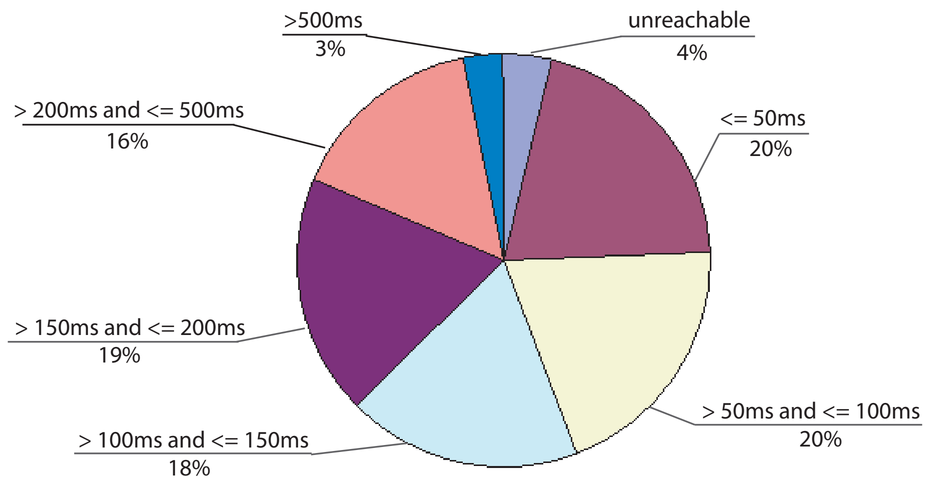


Figure 3: Overview of the Ping Results

Figure 3 provides an overview of the results. It shows the distribution of the ping values from our tests. There was a total of over 5 million pings and the distribution was more or less constant, irrelevant of time or date. So in PlanetLab, more than 90% of the nodes are connected with an acceptable latency (< 500ms) for most of the applications.
Unfortunately, sometimes there are extremly strange ping values ranging from 10s up to even several minutes. As we didn't find a reasonable explanation for this, we contacted the PlanetLab support mailing list. Vivek Pai assumes that "a number of nodes [...] have set their per-node bandwidth caps to such low levels that any nontrivial amount of traffic will find itself waiting on the network fair queueing" [13]. In addition, firewalls and loaded routers probably also lead to these problematic values. However, we don't

think that this is the only reason but did not have the time to get deeper into analyzing this problem.

4.2 Bandwidth

Another interesting network parameter is the bandwidth that we also wanted to test using the framework. While in the ping test several nodes were working in parallel, we had to make sure that in the bandwidth test no node is tested by two different nodes at the same time. This would falsify the test results as in such cases the bandwidth that is available for the slice is split between the two connections.
Also, to get reasonable results, it was not enough to send such small packets as with the ping test. But on the other hand, sending a big packet of about 1 MB size is not feasible as well because there are nodes that have a bandwidth of a few kByte/s only. Naturally, sending 1 MB from such nodes to every other node would take very long. Thus we decided to start with a packet of about 100 kByte and in case it is sent fast, we simply increase this size. The goal is that in the end, the transmission takes about 10s, which then easily allows to calculate an estimate of the bandwidth.

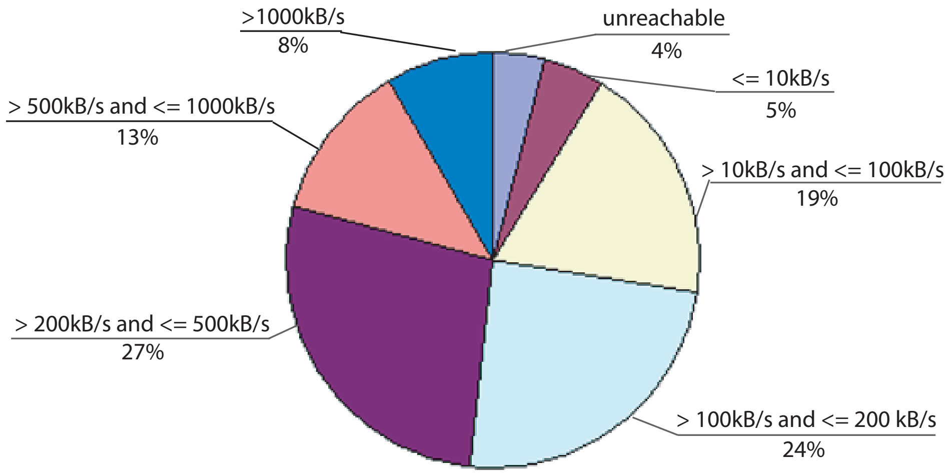


Figure 4: Overview of the Bandwidth results

Nodes with the value -1 indicate that something went wrong (as in ping). The value 0 is used whenever the remote node was the localhost. Figure 4 provides a rough overview of the results. All values are in kilobyte per second and the distribution shown in the figure is the result of almost 100'000 bandwidth tests. So over 90% of all the nodes were interconnected with a bandwidth higher than 10 kByte/s. By the way, many of the very high values occur between nodes that are located at the same site.

4.3 Working with the Results

The test writes the results in a simple CSV file that can be opened in *Excel*. Unfortunately, Excel doesn't support tables with more than 256 columns before version 2007 which makes it very cumbersome to work with the files. Therefore, we also developed a small framework that helps to perform operations on the result files. We will not go into detail here but the inclined reader can find the source code in the package `analysis`.
To employ the results in a simulation, the easiest way is to use the class `DataMatrix` from this package. It provides an internalization of the CSV file and therefore accessing the data is very easy and straightforward.

5 Analysis

After the overview of the tests and their results, we now want to have a closer look at the data. There are a lot of possibilities to analyze them. We do only a few simple analyses due to time schedule constraints here but these already help to understand the whole network a little bit better.
The problem with all the analyses was that, as already mentioned, there exist a few extreme values, for which a ping according to the test result takes several minutes. Of course, such values are not realistic and most probably do not come from a loaded network only. Even though there are very few, they have a strong influence when computing statistical values like mean, variance etc. So we often separate the results in categories and give the percentage of the results lying in these categories.

5.1 Latency

The mean of each of the ping tests is showed in the table below. Remark that the timezone is the Central European Summer Time (CEST).

date and time	**mean**
02.10.2006 at 11am	179ms
03.10.2006 at 10am	164ms
03.10.2006 at 07pm	191ms
04.10.2006 at 01am	177ms
04.10.2006 at 07am	204ms
04.10.2006 at 14pm	192ms
04.10.2006 at 18pm	255ms

We want to see if there is a variance of the global latency over the day. Our hypothesis is that the latency tends to be higher during american working hours. We assume that between six in the morning and noon (CEST) most people in the USA are not working. When computing the overall mean in this time frame, we get 183ms. For the mean of measurements outside of

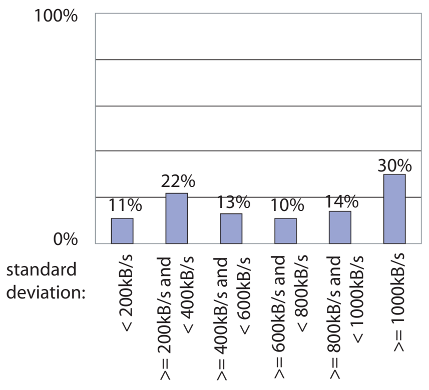
this frame, we get 202ms. So there seems to be a tendency for higher latency during the US working hours.
However, with respect to the fact that there are some extremly high ping values that possibly influence the mean very strong, we are not convinced with that. Thus, for every test, we build a mean for the pings which were less 5000ms and one for the pings that were higher. The results are listed in the following table:

date and time	**< 5000ms**	**> 5000ms**
02.10.2006 at 11am	157ms	17440ms
03.10.2006 at 10am	144ms	17962ms
03.10.2006 at 19pm	165ms	16126ms
04.10.2006 at 01am	163ms	13072ms
04.10.2006 at 07am	169ms	18692ms
04.10.2006 at 14pm	167ms	16255ms
04.10.2006 at 18pm	187ms	16036ms

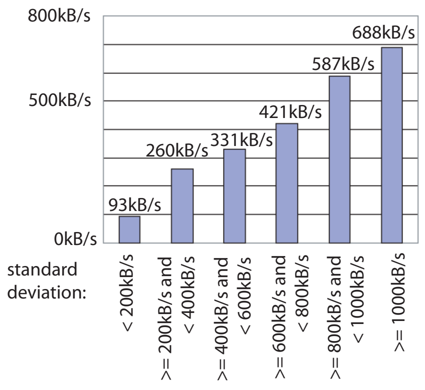
Of course, the first thing that comes up is that the unrealistic pings obviously have a very strong influence on the overall mean. So we build the mean for the two time frames as already done above, but this time, we only take into account the nodes from the group with pings less than 5000ms. For the time frame between six in the morning and noon (CEST) the overall mean now is 157ms and for the rest of the time it is 169ms. So the difference is not as big as before, but still, there seems to be a tendency to higher latency during US working hours. However, to ensure that this assumption is correct, a lot more data and more sophisticated analyzation techniques would be needed. Unfortunatley, we did not have any time left to go furhter into that.

5.2 Bandwidth

The global mean of all our bandwidth measurments is 442kB/s. Our main interest, however, is not the mean value. We want to know if the bandwidth is node- or link-dependent. We do this by calculating the standard deviation for every node. A low standard deviation indicates node-dependence and a high standard deviation indicates link-dependence.
Figure 5a shows the distribution of this calculation but obviously, there is no clear indication what the dependence is. So we additionally calculate the bandwidth mean for each of the groups, which is depicted in Figure 5b. This diagram shows a corellation of the variance groups and the corresponding mean bandwidth.
This is not not very surprising as this is the behavior one encounters all the time: High bandwidth nodes work very well with other high bandwidth nodes, but when they need to connect with a low bandwidth node, the latter is the lower bound for the resulting bandwidth. Low bandwidth nodes



(a) per node standard deviation



(b) mean bandwidths within the groups

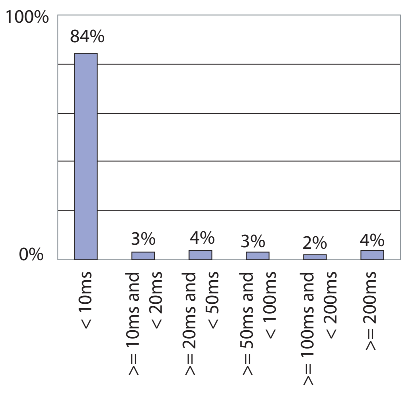
Figure 5: Dependency of the Bandwidth

however, have about the same bandwidth no matter what kind of node they are connected with.

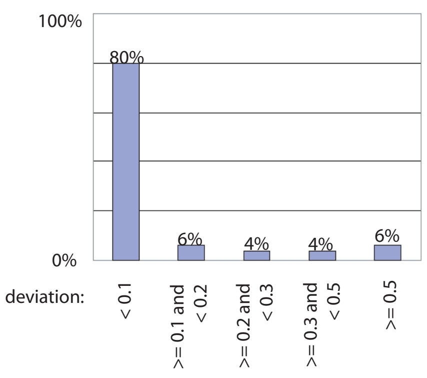
5.3 Symmetries

Another interesting analysis is to see if the measured values are symmetric. That is if for example a ping from node x to node y takes as long as one from node y to node x.

Latency A simple approach to analyze the symmetry of the latency is to calculate the mean of the pings from node x to node y, then calculate the mean of the pings from node y to node x and finally calculate the deviation of these two means.
Figure 6a depicts the distribution resulting from this calculation. Obviously, the links seem to be very symmetric. However, the problem when using absolut tolerances is, that for example a link with a ping value of 400ms in one direction and 405ms in the other is correctly considered to be very symmetric. But for a link with a ping value of 8ms, a deviation of 5ms is not good anymore. Therefore, in Figure 6b, we also show the distribution when using a relative tolerance with respect to the mean of the link's ping values. The symmetry is a little bit lower than with the absolute tolerance, but still, it is clear that most links are symmetric.



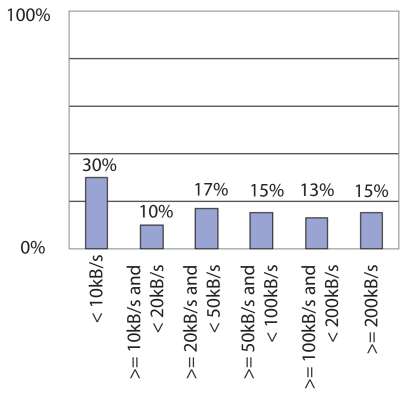
(a) absolute tolerance



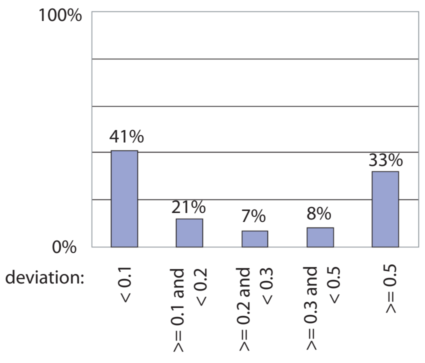
(b) relative tolerance

Figure 6: Symmetry of Latency

Bandwidth To analyze the symmetry of the bandwidth, we use the same approach again. That is, the deviation of the bandwith from node x to node y to the bandwith from node y to node x is calculated. Figure 7a shows the result of this calculation when using an absolute tolerance. This is obviously not very helpful, so we use a relative tolerance again. The distribution resulting from this correction is shown in Figure 7b, which is much more meaningful. There is a clear tendency that one group of links is symmetric while another group is extremly asymmetric.



(a) absolute tolerance



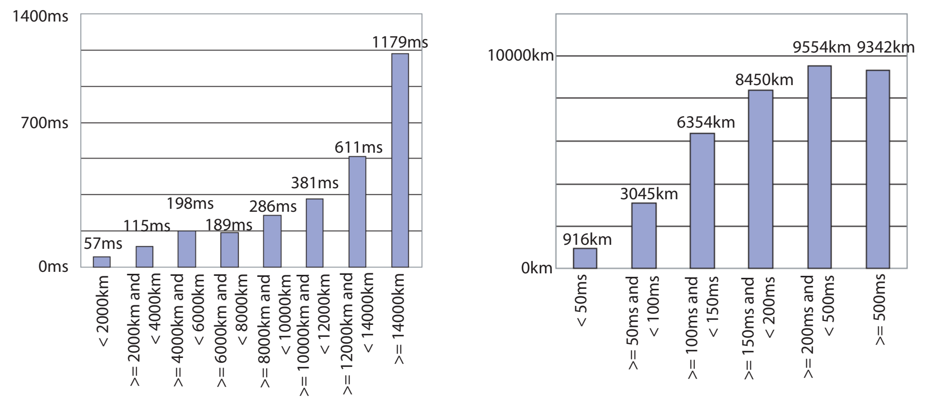
(b) relative tolerance

Figure 7: Symmetry of Bandwidth

A possible interpretation is that links between nodes with about the same bandwidth limitations are within the symmetric group, while on the other hand, links between nodes with very different bandwidths are within the asymmetric group because when transferring from a fast node to a slow one the bandwidth is higher than the other way round as the upload bandwidth is usually lower than the download bandwidth.

5.4 Correlation of Latency and Distance

An interesting analysis is to check if the latency is correlated in some way with the geographic distance between two nodes. Correlation is not used strictly mathematically here because the distance between two nodes is not a random variable. Anyway, we want to know if there is some kind of dependency between the distance and the latency.



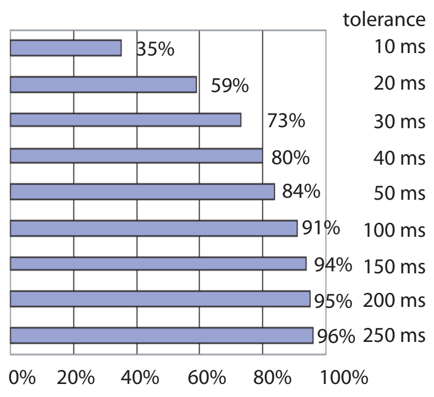
(a) grouped by distance (b) grouped by latency

Figure 8: Correlation of Latency and Distance

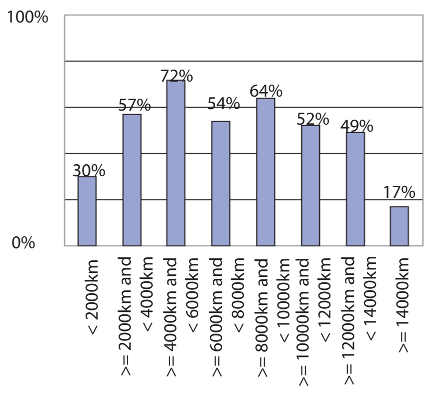
The first step we do is to group the node pairs by their distance and then calculating the mean of all ping values within each of the groups. Similarly, we do it the other way round by grouping by ping value and calculating the mean of the distances. The result, as shown in the Figures 8a and 8b, indicates a strong correlation between distance and latency. The small discrepancies between 4000km and 8000km and with ping values higher than 500ms are due to the already mentioned very high ping values. There are very few of them but they have a strong influence on the mean value.

It would be nice to be able to estimate the latency when the distance is known. To check if this works, we calculate the mean of the quotient $\frac{distance}{ping}$, which is 41. Thus, the estimation of the ping value is $\frac{distance}{41}$. Figure 9a shows the quality of this estimation where the tolerance is the maximum error the estimate is allowed to have from the value actually measured. This shows that the estimate starts to make sense with a tolerance of about 100ms. This is good for pings that are in the region of 1000ms, but very bad for pings in the region of 50ms.

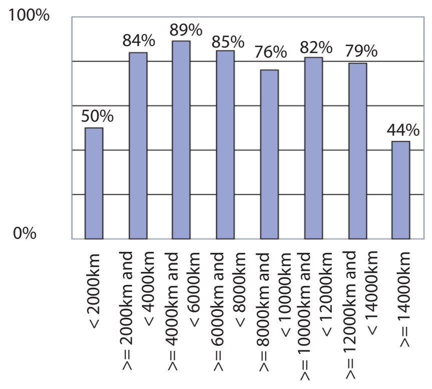
To avoid this problem, we use a tolerance relative to the mean of all nodes forming a group of specific distances. For example, with a relative tolerance of 10%, a node whose ping target is 500km away lies in the group with a mean of 57ms as Figure 8a shows. Thus, the estimate is good enough if the error is not bigger than 5.7ms. Figures 9b, 9c and 9d show the results when using relative tolerances of 10%, 20% and 30% respectively. The diagrams show that with a relative tolerance of 30%, most of the estimates are within the allowed clearance.



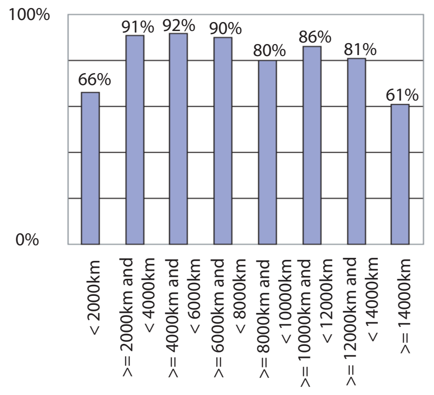
(a) absolute tolerance



(b) 10% relative tolerance



(c) 20% relative tolerance

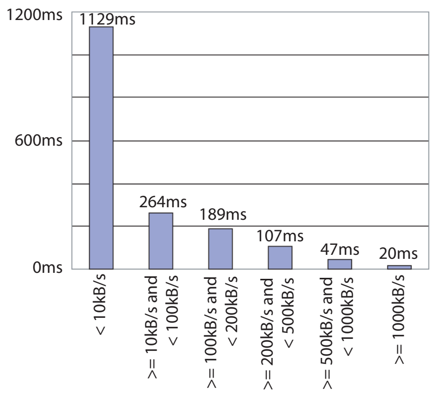


(d) 30% relative tolerance

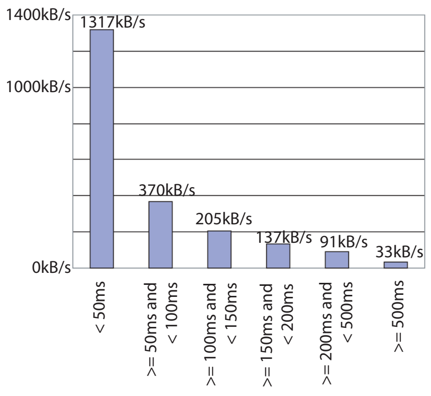
Figure 9: Matching of Estimations

5.5 Correlation of Latency and Bandwidth

We also want to know if latency and bandwidth are correlated. To do so, we use the same approach as in the previous section. At first, we group the node pairs by their bandwidth and then calculate the mean of all ping values within each of the groups. Again, we also do it the other way round by grouping by ping value and calculating the mean of the bandwidths. The result is shown in Figures 10a and 10b and indicates a correlation between bandwidth and latency.



(a) grouped by bandwidth

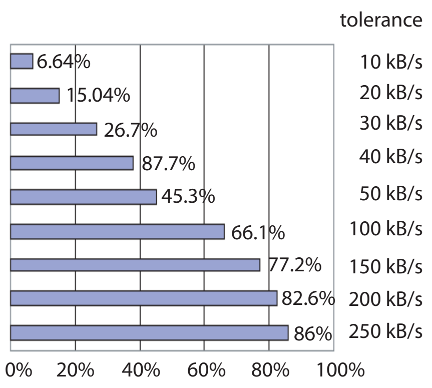


(b) grouped by latency

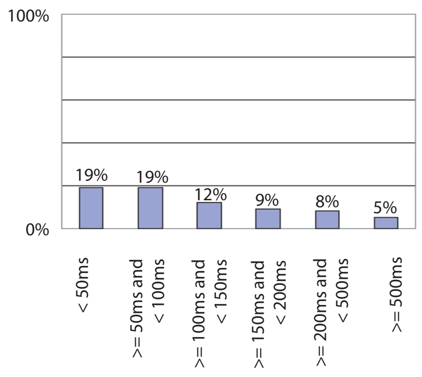
Figure 10: Correlation of Latency and Bandwidth

It would also be nice to be able to estimate the bandwidth when knowing the ping value. To check if this works, we calculate the mean of the product $bandwidth * ping$, which is 25418. Thus, the estimation of the bandwidth value is $\frac{25418}{ping}$. Figure 11a shows the accuracy of the estimation. The tolerance is the maximum error that the estimate is allowed to have. The estimates start to make sense with a tolerance of about 200kB/s. Of course, an estimation error that high is acceptable for bandwidths over 1000kB/s but not at all for lower bandwidths.

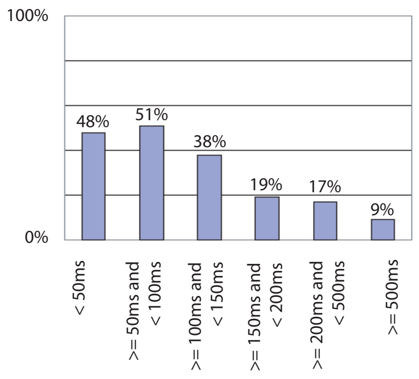
To better understand the quality of the estimation, we use a tolerance relative to the mean bandwidth of all nodes forming a group of specific latencies. For example with a relative tolerance of 10%, a link with a ping value of 70ms allows an estimation error of 37kB/s. Figures 11b, 11c and 11d show the results when using relative tolerances of 10%, 20% and 30% respectively. Not surprisingly, the estimation of the bandwidth is harder than estimating the bandwidth as presented in Section 5.4.



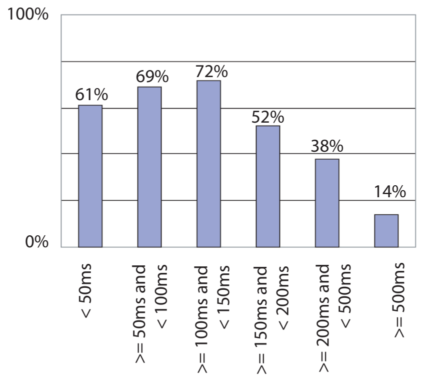
(a) absolute tolerance



(b) 10% relative tolerance



(c) 20% relative tolerance



(d) 30% relative tolerance

Figure 11: Matching of Estimations

6 Conclusion

We present a framework that can be used to develop and deploy network tests in the PlanetLab environment. The framework is very flexible and can even be used for other kinds of applications unlike network tests. With the help of this framework it is very easy to collect large amounts of network information such as latency and bandwith. Analyzing this data helps to understand the characteristics of the underlying network. Moreover, the data can be used to run realistic simulations. It has for example been used to study the DHT eQuus [11] for stretch and load balancing.

There is also a lot of open work that can be done in this direction: The testing framework can be enhanced and made more flexible. For example the Launcher should be configurable and updateable without the need of redeploying and restarting, a caching mechanism for Slave components could

be added or the resource service and its usage could be generalized. Additionally, the problems with the tests (i.e. the extremly high values for pings) should be taken care of.
Much more time has also to be invested in collecting data and analyzing it in more detail and with more sophisticated techniques. For example, the ping values and the bandwidths can be analyzed node-specific and link-specific or the analysis about the variance of the ping over the time, as started in Section 5.1, can be deepened. Furthermore, it would be nice to have a visualization tool that allows to load the result CSV files and display the statistics or analyses of choice.

References

[1] CoDeen. `http://codeen.cs.princeton.edu`.

[2] Codeploy. `http://codeen.cs.princeton.edu/codeploy`.

[3] CoMon. `http://comon.cs.princeton.edu`.

[4] Contributed Sofware. `https://wiki.planet-lab.org/twiki/bin/view/Planetlab/ContributedSoftware`.

[5] JUnit. `http://www.junit.org`.

[6] List of currently alive PlanetLab nodes. `http://summer.cs.princeton.edu/status/tabulator.cgi?format=nameonly&table=table_nodeviewshort&select='resptime%20%3E%200'`.

[7] PlanetLab Node Lists. `https://www.planet-lab.org/db/nodes/nodelists.php`.

[8] PlanetLab User's Guide. `http://www.planet-lab.org/doc/UsersGuide.php`.

[9] PlanetLab Wiki. `https://wiki.planet-lab.org/twiki/bin/view/Planetlab/WebHome`.

[10] Stork. `http://www.cs.arizona.edu/stork`.

[11] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System. In *6th IEEE International Conference on Peer-to-Peer Computing (P2P), Cambridge, United Kingdom*, September 2006.

[12] Luzius Meisser. PlanetLab Quick Start Manual.

[13] Vivek Pai. Private Conversation.