Towards Soft Real-Time Applications on Enterprise Desktop Grids

Derrick Kondo, Bruno Kindarji, Gilles Fedak, and Franck Cappello
Laboratoire de Recherche en Informatique/INRIA Futurs
Bat 490, Université Paris Sud, 91405 ORSAY Cedex, FRANCE
Corresponding author: dkondo@gmail.com

***Abstract*— Desktop grids use the idle cycles of desktop PC's to provide huge computational power at low cost. However, because the underlying desktop computing resources are volatile, achieving performance guarantees such as task completion rate is difficult. We investigate the use of buffering to ensure task completion rates, which is essential for soft real-time applications. In particular, we develop a model of task completion rate as a function of buffer size. We instantiate this model using parameters derived from two enterprise desktop grid data sets, evaluate the model via trace-driven simulation, and show how this model can be used to ensure application task completion rates on enterprise desktop grid systems.**

I. Introduction

For over a decade, the largest distributed computing platforms in the world have been desktop grids, which use the idle computing power and free storage of a large set of networked (and often shared) hosts to support large-scale applications. Desktop grids are an extremely attractive platform because there offer huge computational power at relatively low cost. Currently, many desktop grid projects, such as SETI@home [1], FOLDING@home [2], and EINSTEIN@home [3], use TeraFlops of computing power of hundreds of thousands of desktop PC's to execute large, high-throughput applications from a variety of scientific domains, including computational biology, astronomy, and physics.

Despite the huge return-on-investment that desktop grids offer, the platform's use has been limited to mainly task parallel, high-throughput applications. This is a consequence of the resources' volatility and heterogeneity. That is, the resources are volatile in the sense that CPU and host availability fluctuates tremendously over time. This is because the hosts are shared with the owner/user, whose activity is given priority over the desktop grid application; at any time, an executing desktop grid task may be preempted due to user activity (e.g. key/mouse activity, other user processes, and etc.). Moreover, the hosts are heterogeneous in terms of clock rates, memory sizes, and disk sizes, for example. As a result of such volatility and heterogeneity, broadening the range of desktop grid applications is a challenging endeavor.

In this paper, we focus on enabling soft real-time applications to execute on enterprise desktop grids; soft real-time applications often have a deadline associated with each task but can afford to miss some of these deadlines. While this problem entails a myriad of issues (such as timely data transfers), our goal is to achieve probabilistic guarantees on task completion rates via buffering. That is, we determine how large a buffer must be allocate to ensure that some fraction of tasks meet their corresponding deadlines. We concentrate particularly on achieving such guarantees on desktop grids in enterprise environments, for example a company's local area network. This is a challenging because task execution can be delayed or cancelled by users' preemptive activity or machine hardware failures.

A number of soft real-time applications ranging from information processing of sensor networks [4], real-time video encoding [5], to interactive scientific visualization [6], [7], [8] could potentially utilize desktop grids. An example of such an application that has soft real-time requirements is on-line parallel tomography [7]. Tomography is the construction of 3-D models from 2-D projections, and it is common in electron microscopy to use tomography to create 3-D images of biological specimens. On-line parallel tomography applications are embarrassingly parallel as each 2-D projection can be decomposed into independent slices that must be distributed to a set of resources for processing. Each slice is on the order of kilobytes or megabytes in size, and there are typically hundreds or thousands of slices per projection, depending on the size of each projection. Ideally, the processing time of a single projection can be done while the user is acquiring the next image from the microscope, which typically takes several minutes [9]. As such, on-line parallel tomography could potentially be executed on desktop grids if there were effective method for meeting the application's relatively stringent time demands.

To enable such soft real-time applications to utilize desktop grids, we investigate the use of buffering to ensure task completion rates. In particular, the contributions of this paper can be summarized as follows. First, we develop a model of the successful task completion rate as function of the server's buffer size. Second, in the process of verifying the model's assumptions, we show that the aggregate compute power of desktop grid systems can often be modelled using a normal distribution. Third, in the process of developing the model and running trace-driven simulations, we found several guidelines to be used when scheduling tasks with soft deadlines on volatile resources and describe them in detail.

The paper is structured as follows. In Section II, we formalize the specific problem that we address, describing in detail our application and platform model. Then, in Section III, we

characterize two desktop grid systems in an effort to instantiate parameters in our model using realistic values. In Section IV, we develop a model of task completion rate as function of buffer size. We describe in Section VI how our work in this paper relates to previous research. Finally, in Section VII, we summarize our conclusions and describe future research directions.

II. Problem Statement

In this section, we formalize the problem that we address in this paper. In particular, we detail the application and platform model shown in Figure 1. Table I summarizes the variable definitions given in this section.

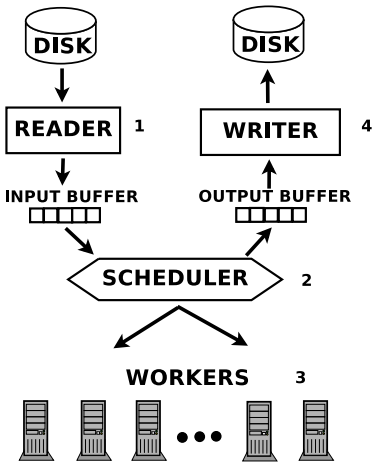


Fig. 1. Application and Platform Model

A reader periodically reads input data from disk into an input buffer (see Figure 1, step 1) of size b. We assume each input datum corresponds to a single task and use the term task synonymously with the term input datum. Specifically, the reader inserts a batch with H input tasks into the input buffer every C_{in} time units. Each task is assigned a deadline d, which denotes the maximum amount of time allowed to expire before the corresponding result becomes useless; d defines the soft real-time constraint of an application. The length of d is directly related to the length of the buffer and is given by

$$d = (b * C_{in})/H \tag{1}$$

We assume the application can tolerate a small percentage of tasks that fail to complete by the deadline.

The total number of tasks is T. We assume the execution of each tasks is independent of one another, and that all tasks are of equal size s, which denotes the amount of computational work (in floating point operations for example) required to complete a task. For tasks with large input data sizes, we assume that the data transfer time can be folded in with the task size s. A clear limitation of our platform model it that does not include a detailed and precise model of the network. Accurate network models is currently an open and difficult problem in parallel and distributed computing, and so, we postpone the inclusion of a more accurate network model for future work.

A scheduler is then responsible for scheduling a task from the input buffer to the set of N workers (see Figure 1, step 2). If a task's deadline passes, or if the buffer overflows with too many tasks, the scheduler will delete tasks from the buffer. However, once a task is scheduled, the scheduler does not have the ability to cancel an executing task. (Although implementable, most desktop systems such as BOINC [10], XtremWeb [11], and Entropia [12] do not provide preemption mechanisms.)

Once assigned a task, a worker downloads the task from the scheduler (see Figure 1, step 3) and computes a result during its idle time. If the task fails (for example, because of user keyboard/mouse activity, or hardware failure), we assume the scheduler will detect the failure immediately and can then reassign the task. (Fast failure detection can be done through the use of worker heartbeats or time out mechanisms). Or if the task completes, the worker uploads the result to the scheduler, which then removes the task from the input buffer and places the result in the output buffer . Upon completion, a writer then writes the results in the output buffer to disk (see Figure 1, step 4).

The workers are heterogeneous, unreserved and shared machines. As such, they can have different processor speeds, and at any given moment in a time, an executing task can be preempted because of the desktop user's activity. In particular, we denote the amount of computational power (in floating point operations, for example) usable *by the desktop grid application* between time t and $t + \delta$ on worker i as $p_i^{t,t+\delta}$ where δ is some time interval and $1 \leq i \leq N$. The value of $p_i^{t,t+\delta}$ ranges between 0 and $\delta *$ host's maximum number of operations per second.

Several factors can influence the value of $p_i^{t,t+\delta}$. For instance, if the desktop machine i is powered off or there is continuous user keyboard activity between time t and $t + \delta$, then $p_i^{t,t+\delta}$ would most likely be 0. If another process if running on the machine, then $p_i^{t,t+\delta}$ may be some fraction of the maximum possible value. If $p_i^{t,t+\delta}$ measures the compute power from an individual host, then the total amount of computational power $P^{t,t+\delta}$ available from the entire platform consisting of N workers is given by $\sum_{i=1}^{N} p_i^{t,t+\delta}$.

In our analysis based on previously described model, the performance metric we use is the percent of tasks that successfully complete before their corresponding deadlines, and we refer to this metric as the *task success rate* (or inversely, the *task failure rate*).

III. Characterization

Using the formulation described in the previous section, our goal is to construct a model to estimate the rate at which task deadlines are met as a function of the buffer size. To construct a such a model, one needs to first understand the characteristics of the platform's aggregate compute power during some interval of time. The aggregate compute power is directly related to the rate at which tasks that can be completed.

Because desktop grid hosts are volatile, clearly some fraction of the aggregate compute power cannot be used as tasks begin to execute but then fail to run to completion. Thus, one also needs to characterize the volatility of the platform and in particular, the rate at which tasks fail to meet their deadlines. The rate at which tasks fail to meet their corresponding deadlines is also directly related the task success rate.

In this section, we give a statistical characterization of these two variables $P^{t,t+\delta}$ (the aggregate compute rate) and f_{dl} (the rate at which a task does not complete by the deadline), and then use these variables in a model described later in Section IV.

A. Trace Data Sets

For both characterization and simulation purposes, we use the UC Berkeley (UCB) trace data set first described in [13]. The traces were collected using a daemon that logged CPU and keyboard/mouse activity every 2 seconds over a several weeks on about 85 hosts. The hosts were used by graduate students the EE/CS department at UC Berkeley. We use the largest continuously measured period between 2/28/94 and 3/13/94, and focus on the busiest periods of user activity between the hours of 10AM to 5PM. The traces were post-processed to reflect the availability of the hosts for a desktop grid application using the following desktop grid settings. A host was considered available for task execution if the CPU average over the past minute was less than 5%, and there had been no keyboard/mouse activity during that time. A recruitment period of 1 minute was used, i.e., a busy host was considered available 1 minute after the activity subsided. Task suspension was disabled; if a task had been running, it would immediately fail with the first indication of user activity.

The clock rates of hosts in the UCB platform were all identical, but of extremely slow speeds. In order to make the traces usable in our simulations experiments for modern-day applications, we transform clock rates of the hosts to a clock rate of 1.5GHz, which is a modest and reasonable value relative to the clock rates found in the other current desktop grid platforms. Doing so, we assume that the effects of user CPU and mouse/keyboard activity has remained constant over the years. Several works, such as [14] and [15], provide evidence that many desktop grids characteristics have remained similar over time.

In addition to the UCB data set, we use another trace data set collected from desktops at the San Diego Supercomputer Center (SDSC), first reported in [14]. This data set was collected by submitting compute-intensive tasks to about 200 hosts at SDSC through the Entropia desktop grid system [12], which ensured that a task would only run when the machine was free. When permitted to execute by the Entropia desktop grid system, each task would continuously compute floating-point operations and log the number of operations completed every 10 seconds. The tasks' output was then retrieved and used to create a continuous trace of CPU availability for each host. Because the tasks were executed through a real desktop grid system, the CPU availability perceived by the task is exactly the performance that would be obtained by a real compute-intensive desktop grid application. Using the above method, traces for over 200 hosts with host speeds ranging 179MHz to 3.0GHz over a 1 month period were obtained.

B. Normality of Aggregate Compute Power

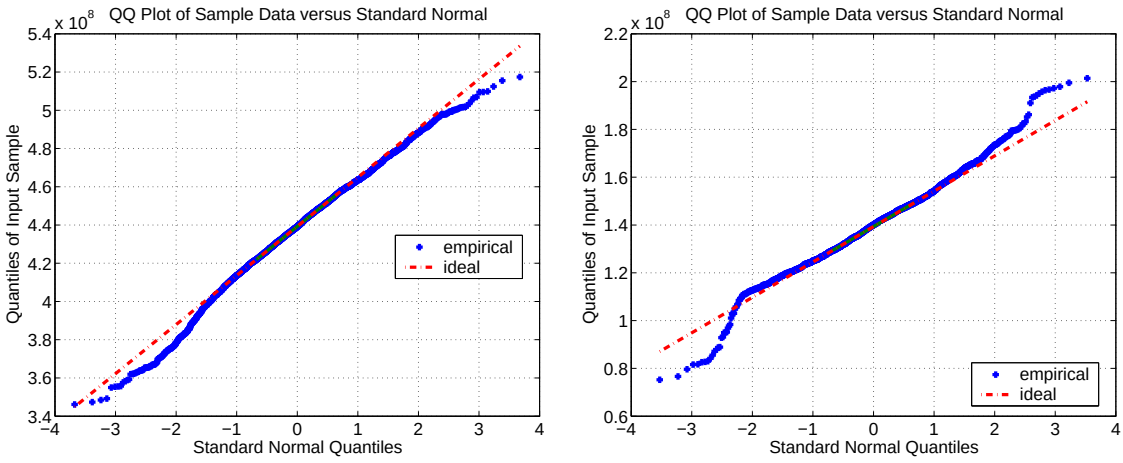
We hypothesize that $P^{t,t+\delta}$ can be modeled with a normal distribution. Intuitively, $P^{t,t+\delta}$ is the sum of several random variables $p_i^{t,t+\delta}$ for $1 \leq i \leq N$. The central limit theorem [16] states that the sum of a set of variates from any distribution with a finite mean and variance tends towards a normal distribution. By the central limit theorem, for large relatively large i, $P^{t,t+\delta}$ should be distributed normally.

To test this hypothesis, we first measured $P^{t,t+\delta}$ at thousands of different values of t and with δ equal to 60 second intervals. The rationale for using 60 second intervals is that task sizes would rarely be shorter than 60 seconds. We then conducted a parameter fit of this data set with a normal distribution using maximum likelihood estimation (MLE) [16].

MLE is a standard and the most popular technique for parameter estimation used for statistical modelling purposes. Intuitively, the MLE method finds parameter values for a particular distribution that maximize the probability that the data set came from that particular distribution. For the normal distribution, a MLE solver will attempt to maximize a likelihood function of μ and σ given the data set. After applying MLE to the data set, we found the estimate of the mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ to be $4.3829 * 10^8$ and $2.6136 * 10^7$ operations per second respectively for UCB trace data set. For the SDSC data set, the estimated mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ were $1.3998 * 10^8$ and $1.5630e + 07$ operations per second, respectively.

To measure the fit of the data set with respect to the normal probability distribution corresponding to the derived parameters, we conduct both a graphical test via a quantile-quantile (QQ) plot and a quantitative test via the Kolmogorov-Smirnov (KS) goodness-of-fit test. A QQ plot plots the quantiles of the empirical data versus the quantiles of the normal distribution. If the empirical data does in fact come from a normal distribution, the resulting plot will be linear (even if the empirical distribution is shifted or re-scaled relative to the normal distribution).

As depicted in Figures 2(a) and 2(b), we find that the QQ plots exhibits a strong linear relationship between the quantiles of the empirical and modelled distributions, and this in turn gives evidence that $P^{t,t+\delta}$ is distributed normally. The QQ plot for the SDSC data set is slightly skewed at the ends compared to the ideal. Nevertheless, the cumulative distribution function (CDF) of the empirical data versus the fitted is virtually indistinguishable. After careful visual inspection of the SDSC traces, we suspect that the skew (which occurs with relatively very few data points) is due to a few "abnormal" events, such as coordinated backups of all the hosts which would reduce the CPU availability of all hosts simultaneously resulting in relatively low values of $P^{t,t+\delta}$.



(a) UCB

(b) SDSC

Fig. 2. Quantile-quantile plot of empirical data for $P^{t,t+\delta}$

In addition, to comparing graphically the empirical data set with the hypothesized normal distribution, we use the Kolmogorov-Smirnov (KS) test. Intuitively, the KS test reflects the maximum difference between the observed CDF of the data set and the expected CDF of the test distribution.

For large data sets (such as our empirical data set for $P^{t,t+\delta}$), the KS test is "too sensitive" in the sense that it will almost always reject the null hypothesis that the data follow the normal distribution as it detects very minor differences between the empirical data and the data from the hypothesized distribution. For large data sets gathered from real environments, a certain amount of noise will occur in the data sets and one cannot expect these data sets to match the hypothesized distribution perfectly.

So instead of using the KS test on the entire data set as a whole we use the a similar method deployed in [17] to determine whether the sample data could have the hypothesized distribution. That is, we conduct many KS tests on subsamples randomly selected from the entire data set. In particular, we conducted KS tests on 1000 subsamples of the data, each with about 100 randomly chosen data points, and then determine the mean p-value to either reject or accept the null hypothesis.

We find that the resulting mean p-value of the KS tests with the UCB data to be 0.466, and the mean p-value of the KS test with the SDSC data to be .448, which are clearly above the traditional .05 threshold. While this should not be interpreted as absolute proof in support of the null hypothesis, it does provide additional quantitative evidence for the validity of our hypothesized distribution, which complements the results of our graphical tests.

The combination of our graphical test and KS test provide strong evidence that $P^{t,t+\delta}$ is distributed normally. So despite the volatility of desktop grid resources, we find that certain aggregate statistics of the computational power of these resources can be modelled accurately with probability distributions.

The fact that $P^{t,t+\delta}$ follows a normal distribution is useful for several reasons. First, one can accurately estimate the variance and determine confidence intervals for aggregate desktop grid performance. In past work [14], [13], [18], only the mean performance is presented and this point estimator does not reflect the volatility of the underlying system. This could have a number of important applications in multi-job scheduling (where the system is inundated with high-throughput and short-lived jobs, and the scheduler must give confidence intervals for throughput and response time), and also soft real-time applications that need guarantees on task completion rates. We give an example of the applying the normality result to the latter in the following section.

Second, when joining multiple desktop grids, it is possible to estimate the combined compute power easily. That is, given two desktop grids whose aggregate compute power are $P_1^{t,t+\delta}$ and $P_2^{t,t+\delta}$ with normal distributions $N(\mu,\sigma^2)$ and $N(\nu,\tau^2)$ respectively, then the combined desktop grid's aggregate compute power $P_{1,2}^{t,t+\delta}$ is distributed normally with expectation $\mu+\nu$ and variance $\sigma^2+\tau^2$ [16]. Obtaining such a statistic may be useful for estimating the cost and benefit (for example total compute power and reduction in variance) of joining two desktop grids infrastructures. The amalgamation of multiple desktop grids with different software infrastructures has been a topic of much past and present work [19], [20], [21] in the area of Grid computing.

C. Estimating deadline failure rates

Previously, we determined a method for modelling a desktop grid's aggregate compute power. However, an application, especially one with large task sizes, may not utilize all of the aggregate compute power of the system because of task failures. That is, during execution on a host, a task may in fact be preempted by user CPU or keyboard/mouse activity, for example; in this case a small fraction of the aggregate

compute power is not used and application completion does not progress.

Thus, to determine the *effective* compute power of a desktop grid system for an application with a particular task size, one must first determine the rate at which tasks fail to meet their deadlines. We determine the steady state failure rate as follows. Using trace driven simulation, we execute a job with an infinite number of tasks over the entire trace period and then determine the number of tasks that complete before their deadline. We do this for a range of buffer sizes to determine the failure rate as a function of buffer size. (Note that if a task failed to complete on the host chosen first, we would randomly pick another host on which the task would be restarted from scratch, and we would repeat this process until the deadline expired or the task was finished.)

Figures 3(a) and 3(b) show the task failure rates on the UCB and SDSC platforms respectively for deadlines in the range of 15 and 35 minutes for a 15-minute task (that is, a task that takes 15 minutes to execute on a 1.5GHz machine). Clearly, as d approaches infinity, the task failure rate will approach 0, and so $f_{dl}(d)$ will decrease sublinearly. While the function is slight curved, we believe that the function can be estimated sufficiently by a linear function for a relatively small range of deadlines; in particular, we believe that a reasonable range of task success rates (which we discuss later in Section IV) of interest to system developers (e.g. $\geq$80%) corresponds to a range of deadlines in which $f_{dl}(d)$ appears almost linear.

So, we determined the least squares fit among the data points (also shown in Figures 3(a) and 3(b)), and determined the following linear relationships between deadline and failure rate for the UCB platform: $f_{dl}(d) = -0.008 * d + 0.322$

Since $d = f(b) = (C_{in}/H) * b$,

$$f_{dl}(f(b)) = (-0.008 * C_{in} * b)/H + 0.322 \quad (2)$$

We found that for the UCB platform the mean and maximum error between the least-squares fit and simulation result to be 0.025 and 0.073 respectively.

For the SDSC platform, we found the following linear relationship between the failure rate and deadline: $f_{dl}(d) = -0.020 * d + 0.628$ with a mean and maximum error of 0.064 and 0.118 respectively. In a similar fashion, we determine the least-squares fit for applications with several other task sizes, and found fits with similarly small error values.

IV. Modelling Task Success Rate

Previously, we determined that the aggregate compute power can be modelled by a normal random variable, and we determined a function describing the linear relationship between failure rate and task deadline. In this section, we show the usefulness of these statistics by applying them to help enable a soft real-time application to run on an enterprise desktop grid. In particular, we construct a model of the task success rate as function of those statistics and other parameters, in particular the buffer size. For convenience, Table I summarizes the parameters that we first introduced in Section II.

Parameter	Definition
b	Buffer size
H	Tasks per batch
C_{in}	Period by which a batch of tasks is inserted into the input buffer
d	Task deadline
s	Task size
$p_i^{t,t+\delta}$	Compute power available between time t and $t+\delta$ for worker i
$P^{t,t+\delta}$	Aggregate compute power available between time t and $t+\delta$ for the entire desktop grid
T	Number of tasks in the application
N	Number of workers in the platform
f_{dl}	Rate at which tasks fail to meet their deadline
S	Fraction of tasks completed before their deadlines

TABLE I
PARAMETER DEFINITIONS.

The minimum buffer size is a function of the desktop grid's aggregate compute power $P^{t,t+\delta}$ and the rate at which tasks enter the buffer and is given by:

$$\begin{aligned} b &\geq ((H * s)/(C_{in} * P^{t,t+\delta})) * H \\ &\geq (H^2 * s)/(C_{in} * P^{t,t+\delta}) \end{aligned} \quad (3)$$

The reason for the second H in equation 3 is to account for the fact that H tasks are added to the input queue at each period C_{in}. So we need to scale the buffer size accordingly.

Intuitively, the task success rate should be equal to the frequency that $P^{t,t+\delta}$ is above some threshold minus the task failure rate given some deadline. To understand this, it is useful to consider each variable in isolation. Suppose that the resources are complete dedicated and so the deadline failure rate is 0. Then clearly the success rate is entirely dependent on $P^{t,t+\delta}$ being above some threshold. Now suppose that $P^{t,t+\delta}$ is very large relatively to the number and frequency of incoming tasks. Then clearly, the task success rate depends on the rate at which tasks can run to completion before the deadline.

More formally, the task success rate as a function of the buffer size can be given by:

$S(b) = Pr(P^{t,t+\delta} > \alpha) - f_dl(f(b))$

where α is some constant and $f_{dl}(f(b))$ can be estimated by the results of Section III-C.

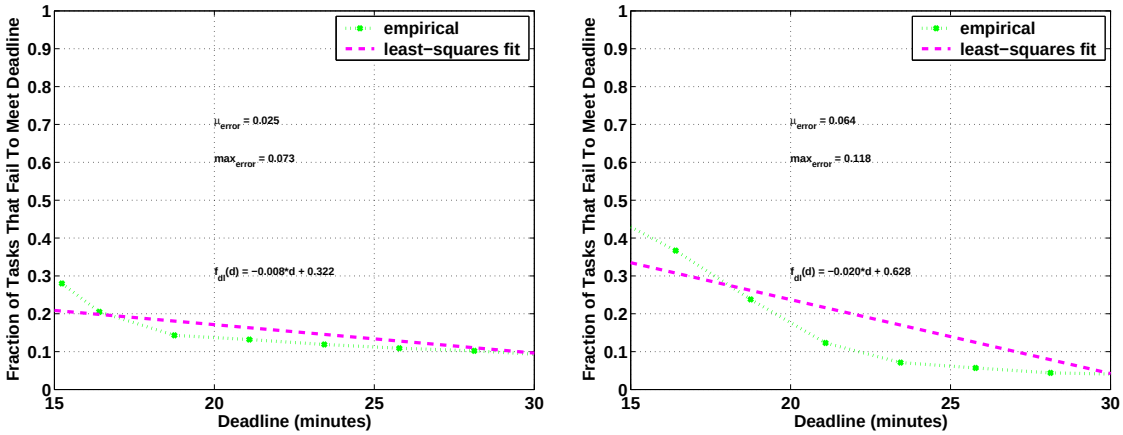
Moreover, using equation 3, we derive the following:

$Pr(P^{t,t+\delta} > \alpha) = Pr(P^{t,t+\delta} \geq (H^2 * s)/(C_{in} * b))$.

Then, for the UCB platform, the task success rate as a function of buffer size is given by:

$$\begin{aligned} S(b) = Pr(P^{t,t+\delta} > (H^2 * s)/(C_{in} * b)) - \\ ((-0.008 * C_{in} * b)/H + 0.322) \end{aligned} \quad (4)$$

The intuition behind the above equation is that tasks can fail to meet their deadlines for two reasons. Firstly, failure can occur if the aggregate compute power in the system dips below the incoming work rate. The probability that this occur is given by $Pr(P^{t,t+\delta} > (H^2 * s)/(C_{in} * b))$. Secondly, failure can occur if a task encounters (repeatedly) host failures. For example, even



(a) UCB

(b) SDSC

Fig. 3. Fraction of 15-minute tasks that fail to meet deadline.

if the aggregate compute power in the system is *always* greater than the incoming work rate, i.e., $Pr(P^{t,t+\delta} > (H^2 * s)/(C_{in} * b)) = 1$, host unavailability may still cause some tasks to fail in meeting their deadlines. This is particularly relevant to the UCB platform, where the intervals of availability tend to be quite small. (This is a consequence of the relatively strict criteria for which a host is considered available, i.e., the host cannot have any other user processes running, any keyboard/mouse activity is disallowed, and etc.)

We plot the success rate $S(b)$ as a function of the buffer size, represented by the solid blue line in Figures 4(a), 4(b), and 4(c) for $65 \leq b \leq 130$, which correspond to applications with 5-minute, 15-minute, and 25-minute tasks respectively. We set the number of tasks per batch H to be 64, which is large enough so that any host that was available always received a task so that the entire computing power of the platform would be utilized. We choose a minimum value of 65 for b since clearly b must be greater than or equal to H. Initially, for relatively small buffer sizes, $S(b)$ is relatively low, but as b increases $S(b)$ dramatically increases until a buffer size of 70, at which point the curve's "knee" occurs. Thereafter, $S(b)$ increases at a much lesser rate. For example, for an application with 15-minute tasks shown in Figure 4(b), a buffer size of 70 results in a .80 success rate, where as a buffer size of 120 only results in a 10% increase in success rate.

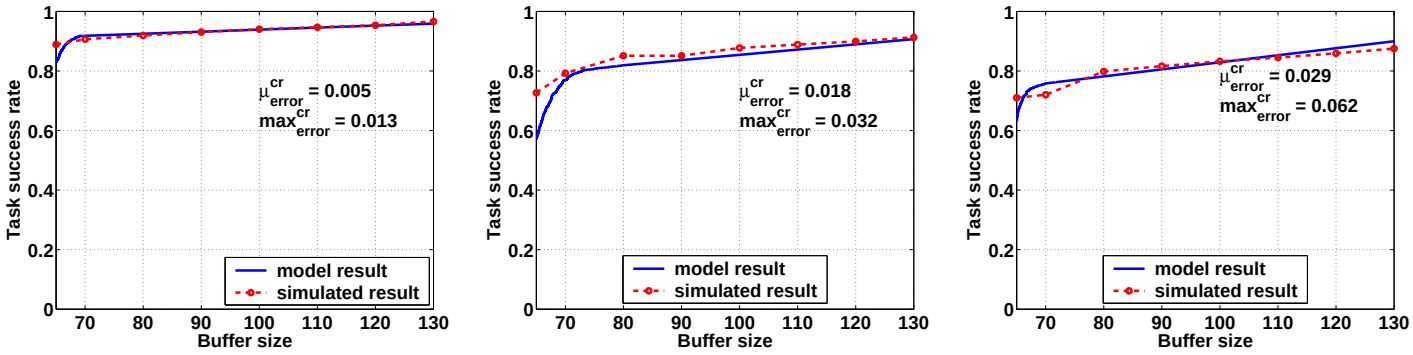
The shape of the curve can be explained as follows. For relatively small buffer size values, i.e., between 65-70, the success rate value is dominated by $Pr(P^{t,t+\delta} \geq (H^2 * s)/(C_{in} * b))$; $P^{t,t+\delta}$ is normally distributed with a relatively small variance, and so small changes in b dramatically affect $Pr(P^{t,t+\delta} \geq (H^2 * s)/(C_{in} * b))$. Moreover, because we set the batch size H so that the entire system is saturated with tasks, $Pr(P^{t,t+\delta} \geq (H^2 * s)/(C_{in} * b))$ dramatically changes for values of b close to H. For buffer sizes greater than 70, $Pr(P^{t,t+\delta} \geq (H^2 * s)/(C_{in} * b))$ becomes close to one, and so the success rate is dominated more by the deadline failure rate, which explains the more gradual increase in success rate with an increase with b. The above explanation applies similarly to the results for the SDSC platform shown in Figures 5(a), 5(b), and 5(c). We chose a slightly different range for the buffer size in these plots as the least-squares fit for f_{dl} was more accurate in this smaller range; specifically, we used the result in Figure 3(b) to determine a range of interest (as f_{dl} gives a lower bound on the success rate), and then "bootstrapped" by determining the f_{dl} in that new smaller range.

V. Model Applicability

One potential weakness in our model is that the steady-state failure rate f_{dl} was measured using a job with an infinite number of tasks. We were concerned that f_{dl} would in fact be too pessimistic because tasks submitted in the previous batches would interfere with the progress of tasks in the most recently submitted batch. That is, f_{dl} could be too pessimistic if tasks of previous batches that failed to complete immediately would remain in the queue (until their deadline passed) and thus delay the execution of tasks of the current batch such that these tasks would "starve" and miss their deadlines. In turn, this might affect the applicability of the model for jobs with a relatively small number of batches.

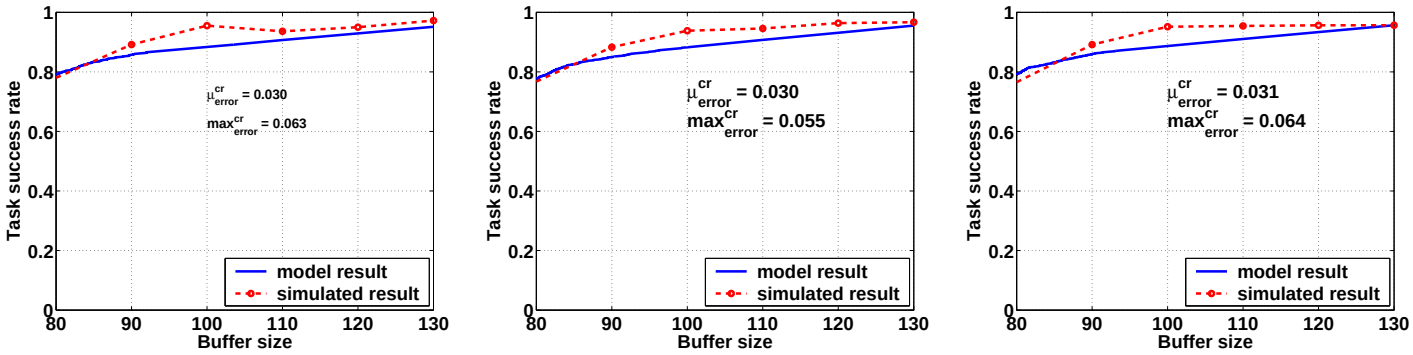
To determine the accuracy of f_{dl} and the result of including it in our model, we determined the mean success rate for an application with a relatively small number of batches (10). (This represents a worst-case scenario, i.e., increasing the number of batches would only improve our model's accuracy.) To do this, we ran trace-driven simulations of the execution of jobs with 5, 15 and 25-minute tasks and 10 batches.

We chose thousands of different starting points in the traces to begin a job and executed the job via trace-driven simulation. The red dotted line in Figures 4 and 5 shows the mean result of running these simulation experiments for various buffer sizes on the UCB and SDSC platforms, respectively. Also shown in those figures are the critical mean (μ_{err}^{cr}) and maximum errors



(a) 5 min. tasks (b) 15 min. tasks (c) 25 min. tasks

Fig. 4. Task success rate versus buffer size on UCB platform.



(a) 5 min. tasks (b) 15 min. tasks (c) 25 min. tasks

Fig. 5. Task success rate versus buffer size on SDSC platform.

($\max_{err}^{cr}$) between the model and simulation results; we define μ_{err}^{cr} and $\max_{err}^{cr}$ to be the mean and maximum errors for range in which $S(b)$ is greater than .80 because we believe that system developers will not be concerned as much with success rates below this threshold. We found that the maximum error between the model and simulation results to be .063 over both platforms' results, which indicates a precise fit between the model and simulated results.

The close fit between the model and simulation results can be explained by the way in which tasks are scheduled to hosts. Firstly, tasks are dropped from the buffer if there is no host on which the task can completed before its deadline. Virtually all desktop grid systems [22], [10], [11], [12] keep track host clock rate information, and so it is reasonably to assume that a scheduler can determine whether a host can complete a task by the deadline if it were completely dedicated to task execution. We found that this trivial policy resulted in significant (>10%) improvement in terms of task success rates. Secondly, tasks are dropped in First-In-First-Out (FIFO) order so that if the buffer ever overflows as incoming tasks are placed in the queue, then the oldest tasks are removed from the queue first. For these reasons, the negative affect of old tasks starving newly arrived ones is greatly reduced.

Ironically, what we believed was a weakness in the way we calculated f_{dl} was its strength. Because f_{dl} measures the rate at which tasks fail to meet their deadlines *when the system is in steady state*, f_{dl} takes into account the effect that older tasks have on the execution newly arrived ones. Alternatively, we had first erroneously measured f_{dl} by just using random incidence. In particular, we chose hundreds and thousands of starting points for a task on each host over the entire trace period, and determined whether the task could be completed by a particular deadline, *assuming the platform was entirely free*. This resulted in a failure rate that was inaccurate by 10% precisely because it did not consider the fact that other tasks already in the system could prevent newly arrived tasks from completing by their deadlines. While an error of 10% may seem negligible in absolute terms, if the curves in Figures 5 and 4 had been shifted down by 10%, the corresponding buffer sizes would have changed dramatically (and would therefore be not as accurate).

Nevertheless, in general, f_{dl} is still slightly pessimistic, and

as a result, $S(b)$ tends to be a lower bound on the true success rate. This explains why the $S(b)$ is slightly lower than the simulation results shown in Figures 4 and 5;

To apply the plots shown in Figures 4 and Figure 5, a desktop grid developer could use the graph to determine what buffer size is needed to achieve some desired success rate. For example, suppose an application consists of 15-minute tasks. Using Figure 4(b), if a 90% success rate for the application is acceptable, then a buffer size of about 120 would be needed, and the developer can then determine the amount of RAM necessary to make the success rate achievable. Conversely, if a desktop grid server already has some amount of memory, then a desktop grid developer could determine the corresponding buffer size and task success rate that will be result by such a configuration. Another way to use the result is determine the cost effectiveness of add more memory to a server. For example, if the server's current memory size can support a maximum buffer size of 65 tasks, it may be worthwhile to purchase slightly more memory to support a maximum buffer size of 70 tasks, thereby increasing the success rate by 20%. In contrast, if 120 tasks can already be stored in the server's memory, purchasing the same amount of memory to increase the number of buffered tasks to 125 will only improve the success rate by less than 3%.

VI. Related Work

There is related work in several different areas, namely desktop grid characterization, performance modelling, and scheduling for soft real-time applications. With respect to characterization, several papers [18], [14], [23] report the mean aggregate compute power of desktop grid systems, but do not detail the distribution. The work in [14] also reports failure rates, but the definition of failure rates is entirely different from that as defined in this paper. In [14], a failure is defined to be a task execution that fails run to completion. In this paper, a failure is a task that is not completed by a specific deadline; so although a task execution may fail to run to completion, the task may still successfully complete in time on a different host. As discussed in Section IV, using the latter definition of failure is essential for our model's accuracy. In [17], the authors use similar (and standard) statistical modelling techniques on desktop grid data sets. However, they model *availability intervals*, i.e., the time elapsed between two consecutive machine failures. In contrast, in Section III-B, we model the *aggregate compute power* of each desktop grid within some time period, which is entirely different from availability intervals, and thus arrive at different conclusions regarding its distribution.

In [24], [17], the authors give evidence that machine availability follows Weibull or hyperexponential distributions. While their models could potentially be incorporated into our model to determine task failure rates, we opted to take an empirical approach to model task failure rates as it was unclear whether the same probability distributions reported in [17] could model availability in our desktop grid traces. Moreover, the trace data sets used in that study were not available, so if we used the identical probabilistic model described in [17] to determine task failure rates, it would be difficult to verify our new model's accuracy.

Also, there has been much work in the area of performance modelling and scheduling on desktop grids [25], [26], [27], [28], [29], [30], [31]. To the best of our knowledge, none of these works use an application model where the goal is to determine the affect of buffer size on task completion rates with respect to a deadline. In addition, there has been some work on scheduling soft real-time applications on non-dedicated systems [6]. However, in these systems, the definition of CPU availability is quite different. That is, user activity (e.g. keyboard/mouse activity, or user processes) does not necessarily preempt a running desktop grid task; instead, the machines are shared between the desktop grid task and other users. As a result, the resources modelled in that study are much less volatile than those used here.

VII. Summary and Future Work

We described a closed-form model for task success rate as a function of buffer size. Our model assumed that the aggregate compute power of a desktop grid can be modelled as a normal probability distribution, and we showed the validity of our assumption via traces derived from two real desktop grid systems. Moreover, our model used the task deadline failure rate as a parameter, and we showed that such failure rates can be modelled adequately with a least-squares fit. Our model can be used by system developers who wish to determine an adequate buffer size for their (soft real-time) application to guarantee a certain task completion rate.

For future work, we plan on looking at replication techniques as an alternative means of improving the task success rate. Replication invariably "wastes" resources as they compute redundant tasks, and as such, it is only option when there are more hosts relative to the number of tasks. We hope to extend our current model to predict task success rate as a function of replication levels.

Acknowledgments

We gratefully acknowledge Kyung Ryu and Amin Vahdat for providing the UCB trace data set and documentation.

References

[1] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, G. Gedye, and D. Anderson, "A new major SETI project based on Project Serendip data and 100,000 personal computers," in *Proc. of the Fifth Intl. Conf. on Bioastronomy*, 1997.

[2] M. Shirts and V. Pande, "Screen Savers of the World, Unite!" *Science*, vol. 290, pp. 1903–1904, 2000.

[3] "EINSTEN@home," http://einstein.phys.uwm.edu.

[4] "Sensor Networks," http://www.sensornetworks.net.au/network.html.

[5] A. Rodriguez, A. Gonzalez, and M. P. Malumbres, "Performance evaluation of parallel mpeg-4 video coding algorithms on clusters of workstations," *International Conference on Parallel Computing in Electrical Engineering (PARELEC'04)*, pp. 354–357.

[6] J. Lopez, M. Aeschlimann, P. Dinda, L. Kallivokas, B. Lowekamp, and D. O'Hallaron, "Preliminary report on the design of a framework for distributed visualization," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, NV, June 1999, pp. 1833–1839.

[7] S. Smallen, H. Casanova, and F. Berman, "Tunable On-line Parallel Tomography," in *Proceedings of SuperComputing'01, Denver, Colorado*, Nov. 2001.

[8] I. Foster and C. Kesselman, Eds., *The Grid, Blueprint for a New Computing Infrastructure*, 2nd ed. Morgan Kaufmann, 2003, ch. Chapter 8: Medical Data Federation: The Biomedical Informatics Research Network.

[9] A. Hsu, Personal communication, March 2005.

[10] T. B. O. I. for Network Computing, http://boinc.berkeley.edu/.

[11] G. Fedak, C. Germain, V. N'eri, and F. Cappello, "XtremWeb: A Generic Global Computing System," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*, May 2001.

[12] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 597–610, 2003.

[13] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," in *Proceedings of SIGMETRICS'95*, May 1995, pp. 267–278.

[14] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien, "Characterizing and Evaluating Desktop Grids: An Empirical Study," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004.

[15] K. D. Ryu, "Exploiting idle cycles in networks of workstations," Ph.D. dissertation, 2001.

[16] R. Larsen and M. Marx, *An Introduction to Mathematical Statistics and its Applications*. Prentice Hall, 2000.

[17] D. Nurmi, J. Brevik, and R. Wolski, "Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments ," Dept. of Computer Science and Engineering, University of California at Santa Barbara, Tech. Rep. CS2003-28, 2003.

[18] A. Acharya, G. Edjlali, and J. Saltz, "The Utility of Exploiting Idle Workstations for Parallel Computation," in *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1997, pp. 225–234.

[19] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing," in *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[20] O. Lodygensky, G. Fedak, V. Neri, F. Cappello, D. Thain, and M. Livny, "XtremWeb and Condor: Sharing Resources Between Internet Connected Condor Pool," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'03) Workshop on Global Computing on Personal Devices*, May 2003.

[21] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid," in *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000.

[22] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, 1988.

[23] W. Bolosky, J. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs," in *Proceedings of SIGMETRICS*, 2000.

[24] M. Mutka and M. Livny, "The available capacity of a privately owned workstation environment ," *Performance Evaluation*, vol. 4, no. 12, July 1991.

[25] L. Gong, X. Sun, and E. Waston, "Performance Modeling and Prediction of Non-Dedicated Network Computing, Tech. Rep. 9, 2002.

[26] X. Sun and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[27] M. Adler, Y. Gong, and A. Rosenberg, "Optimal sharing of bags of tasks in heterogeneous clusters," in *Proceedings of the Fifteenth Annual ACM symposium on Parallel Algorithms and Architectures, San Diego, California*, 2003.

[28] Y. Li and M. Mascagni, "Improving performance via computational replication on a large-scale computational grid," in *Proc. of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[29] S. Leutenegger and X. Sun, "Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System," in *Proc. of SC'93, Portland, Oregon*, 1993.

[30] G. Ghare and L. Leutenegger, "Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW," in *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

[31] S. Bhatt, C. F., F. Leighton, and A. Rosenberg, "An Optimal Strategies for Cycle-Stealing in Networks of Workstations," *IEEE Trans. Computers*, vol. 46, no. 5, pp. 545–557, 1997.

Characterizing Result Errors in Internet Desktop Grids

Derrick Kondo[1], Filipe Araujo[2], Paul Malecot[1], Patricio Domingues[3], Luis Moura Silva[2], Gilles Fedak[1], and Franck Cappello[1]

[1] INRIA Futurs, France
[2] University of Coimbra, Portugal
[3] Polytechnic Institute of Leiria, Portugal

Abstract. Desktop grids use the free resources in Intranet and Internet environments for large-scale computation and storage. While desktop grids offer a high return on investment, one critical issue is the validation of results returned by participating hosts. Several mechanisms for result validation have been previously proposed. However, the characterization of errors is poorly understood. To study error rates, we implemented and deployed a desktop grid application across several thousand hosts distributed over the Internet. We then analyzed the results to give quantitative and empirical characterization of errors stemming from input or output (I/O) failures. We find that in practice, error rates are widespread across hosts but occur relatively infrequently. Moreover, we find that error rates tend to not be stationary over time nor correlated between hosts. In light of these characterization results, we evaluated state-of-the-art error detection mechanisms and describe the trade-offs for using each mechanism.

1 Introduction

Desktop grids use the free resources in Intranet and Internet environments for large-scale computation and storage. For over 10 years, desktop grids have been one of the largest distributed systems in the world providing TeraFlops of computing power for applications from a wide range of scientific domains, including climate prediction, computational biology, and physics [1]. Despite the huge computational and storage power offered by desktop grids and their high return on investment, there are several challenges in using this volatile and shared platform effectively.

One critical issue is the validation of results computed by volatile and possibly malicious hosts. In large and complex distributed systems, errors in results are inevitable, and errors can stem from different sources. Some sources can be computational. For example, an error could result from a CPU miscalculation due to overclocking and overheating [2]. Other sources can be related to failures during application input or output (I/O). For example, if a machine crashes when the application is writing to an output file or checkpoint, only a partial number in-memory data blocks could have been flushed to disk (not necessarily

in order) [3], which would lead to an erroneous result[4]. Thus, effective error detection mechanisms are essential, and several methods have been proposed previously [7,8].

However, little is known about the nature of errors in real systems. Yet, the trade-offs and efficacy among different error detection mechanisms are dependent on how errors occur in real systems. Thus, we focus on characterizing errors, specifically I/O errors, in a real system by addressing the following critical questions. What is the frequency and distribution of host I/O error rates? How stationary are host I/O error rates? How correlated are I/O error rates between hosts? In light of the error characterization, what is the efficacy of state-of-the-art error detection mechanisms?

To help answer those questions, we deployed an Internet desktop grid application across several thousand desktop hosts. We then validated the results returned by hosts, and we analyzed the invalid results to characterize quantitatively the I/O error rates in a real desktop grid project.

2 Background

At a high-level, a typical desktop grid system consists of a server from which **workunits** of an **application** are distributed to a **worker** daemon running on each participating host. The workunits are then executed when the CPU is available, and upon completion, the **result** is returned back to the server. We define a result **error** to be any result returned by a worker that is not the correct value or within the correct range of values. We call any host that has or will commit at least one error an **erroneous host** (whether intentionally or unintentionally).

Workunits of an application are often organized in groups of workunits or **batches**. To achieve overall low rates for a batch of tasks, the individual error rate per host must be made small. Consider the following scenario described in [7] where a computation consists of 10 batches, each with 100 workunits. Assuming that any work unit error would cause the entire batch to fail, then to achieve an overall error rate of 0.01, the probability of a result being erroneous must be no greater than 1×10^{-5}. Many applications (for example, those from from computational biology [2] and physics [1]) require (low) bounds on error rates as the correctness of the computed results is essential for making accurate scientific conclusions.

3 Related Work

To the best of our knowledge, there has been no previous study that gives quantitative estimates of error rates from empirical data. Nevertheless, several mechanisms for reducing errors in desktop grids have been proposed. We discuss three

[4] While a number of mechanisms exist to ensure atomic writes or to detect file corruption, in practice, few if any are provided by desktop grid systems [4,5,6] or utilized by desktop grid applications themselves.

of the most common state-of-the-art methods [7,8,2] namely spot-checking, majority voting, and credibility-based techniques, and emphasize the issues related to each method.

The **majority voting** method detects erroneous results by sending identical workunits to multiple workers. After the results are retrieved, the result that appears most often is assumed to be correct. In [7], the author determines the amount of redundancy for majority voting needed to achieve a bound on the frequency of voting errors given the probability that a worker returns a erroneous result. Let the error rate φ be the probability that a worker is erroneous and returns an erroneous result unit, and let ε be the percentage of final results (after voting) that are incorrect.

Let m be the number of identical results out of $2m-1$ required before a vote is considered complete and a result is decided upon. Then the probability of an incorrect result being accepted after a majority vote is given by:

$$\varepsilon_{majv}(\varphi, m) = \sum_{j=m}^{2m-1} \binom{2m-1}{j} \varphi^j (1-\varphi)^{2m-1-j} \tag{1}$$

The redundancy of majority voting is $\frac{m}{1-\phi}$.

A more efficient method for error detection is **spot-checking**, whereby a workunit with a known correct result is distributed at random to workers. The workers' results are then compared to the previously computed and verified result. Any discrepancies cause the corresponding worker to be **blacklisted**, i.e., any past or future results returned from the erroneous host are discarded (perhaps unknowingly to the host).

Erroneous workunit computation was modelled as a Bernoulli process [7] to determine the error rate of spot-checking given the portion of work contributed by the host, and the rate at which incorrect results are returned. The model uses a work pool that is divided into equally sized batches.

Allowing the model to exclude coordinated attacks, let q be the frequency of spot-checking, let n be the amount of work contributed by the erroneous worker, let f be the fraction of hosts that commit at least 1 error, and let s be the error rate per erroneous host. $(1-qs)^n$ is the probability that an erroneous host is not discovered after processing n workunits. The rate which spot-checking with blacklisting will fail to catch bad results is given by:

$$\varepsilon_{scbl}(q, n, f, s) = \frac{sf(1-qs)^n}{(1-f) + f(1-qs)^n} \tag{2}$$

The amount of redundancy of spot-checking is given by $\frac{1}{1-q}$.

To address the potential weaknesses of majority voting and spot-checking, **credibility-based systems** were proposed [7], which use the conditional probabilities of errors given the history of host result correctness. The idea is based on the assumption that hosts that have computed many results with relatively few errors have a higher probability of errorless computation than hosts with a

history of returning erroneous results. Workunits are assigned to hosts such that more attention is given to the workunits distributed to higher risk hosts.

To determine the credibility of each host, any error detection method such as majority voting, spot-checking, or various combinations of the two can be used. The credibilities are then used to compute the conditional probability of a result's correctness.

Note that the methods described in this section were designed to detect errors from any source, including a computational source (for example, CPU miscalculations) or an I/O source (for example, a crash during a checkpoint). In the sections that follow, we determine the methods' efficacy in detecting errors caused by I/O failures.

4 Method

We studied the error rates of a real Internet desktop grid project called XtremLab [9]. XtremLab uses the BOINC infrastructure [4] to collect measurement data of desktop resources across the Internet. The XtremLab application currently gathers CPU availability information by continuously computing floating point and integer operations, and every 10 seconds, the application will write the number of operations completed to file. Every 10 minutes, the output file is uploaded to the XtremLab server.

In this study, we analyze the outputs of the XtremLab application to characterize the rate at which errors can occur in Internet-wide distributed computations. In particular, we collected traces between April 20, 2006 to July 20, 2006 from about 4400 hosts. From these hosts, we obtained over 1.3×10^8 measurements of CPU availability from 2.2×10^6 output files. We focused our analysis on about 600 hosts with more then 1 week worth of CPU time in order to ensure the statistic significance of our conclusions .

Errors in the application output are determined as follows. Output files uploaded by the workers are processed by a validator. The validator conducts both syntactical and semantics checks of the output files returned by each worker. The syntactical checks verify the format of the output file (for example, that the time stamps recorded were floating numbers, the correct number of measurements were made, and each line contains the correct number of data). The semantic checks verify the correctness of the data to ensure that the values reported fall in the range of feasible CPU availability values. Any output files that failed these checks were marked as erroneous. After careful inspection of output files that failed synatatic or semantic checks, we found that most of these output files were truncated prematurely or had scrambled output (perhaps due to an out-of-order flush of in-memory blocks [3], for example). As such, we assume that any output file that fails a syntactic or semantic check would have corresponded to an I/O error of an application (for example, an erroneous checkpoint) that would lead to an erroneous result [5]. To date, there has been little specific

[5] We do not believe the errors are due to network failures during transfers of output files. This is because BOINC has a protocol to recover from failures (of either the

data about error rates in Internet desktop environments, and we believe that the above detection method gives a first-order approximation of the I/O error rates for a real Internet desktop grid project.

5 Error Characterization

5.1 Frequency of Errors

The effectiveness of different methods by which errors are detected is heavily dependent on the frequency of errors among hosts. We measured the fraction of workunits with errors per host, and show the cumulative distribution function (CDF) of these fractions in Figure 1.

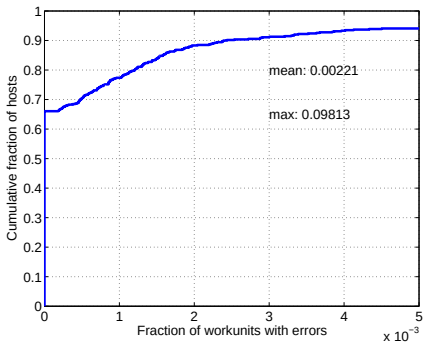


Fig. 1. Error Rates of Hosts in Entire Platform

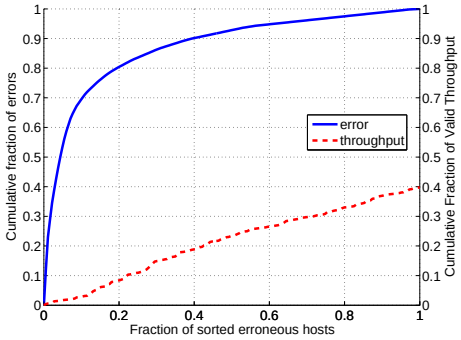


Fig. 2. Cumulative Error Rates and Effect on Throughput

We find that a remarkably high percentage of hosts (about 35%) returned at least one corrupt result in the 3 month time frame. Given that the overall error rate is low and a significant fraction of hosts result in at least one error, blacklisting all erroneous hosts may not be an efficient way of preventing errors.

An error rate of 0.002 may seem so low that error correction, detection and prevention are moot, but consider the scenario in Section 2 again where the desired overall error rate is 0.01. In that case, the probability of a result being erroneous must be no greater than 1×10^{-5}. If $\varphi = 0.002$ as shown in Figure 1, we can simply conduct a majority vote where $m = 2$ to achieve an error rate of about 1×10^{-5} and with a redundancy of about 2.0.

While spot-checking can achieve a similar error rate of about 1×10^{-5}, spot-checking requires a large number of workunits to be processed before achieving it. For example, to achieve a similar error rate of 1×10^{-5} via spot-checking where $q = 0.10$, $f = 0.35$ (from Figure 1), $s = 0.003$ (as shown in Table 1),

worker of server) during file transmission that ensures the integrity of files transfers [4].

Equation 2 requires that the number of workunits (n) processed by each worker be *greater* than 5300. While redundancy is lower at 1.11 compared to majority voting, if each workunit requires 1 day of CPU time (which is a conservative estimate as shown in [1]), it would require *at least* 14.5 years of CPU time *per worker* before the desired rate could be achieved. Even if we increase q to 0.25 (and redundancy is 1.33), spot-checking requires $n = 3500$ (or at least 9.5 years of CPU time per worker assuming a workunit is 1 day of CPU time in length).

Figure 2 shows the skew of the frequency of errors among those erroneous hosts. In particular, we sort the hosts by the total number of errors they committed, and the blue, solid plot in Figure 2, shows the cumulative fraction of errors. For example, the point (0.10, 0.70) shows that the top 0.10 of erroneous hosts commit 0.70 of the errors. Moreover, the remaining 0.90 of the hosts cause only 0.30 of the errors. We refer to the former and latter groups as **frequent** and **infrequent offenders**, respectively.

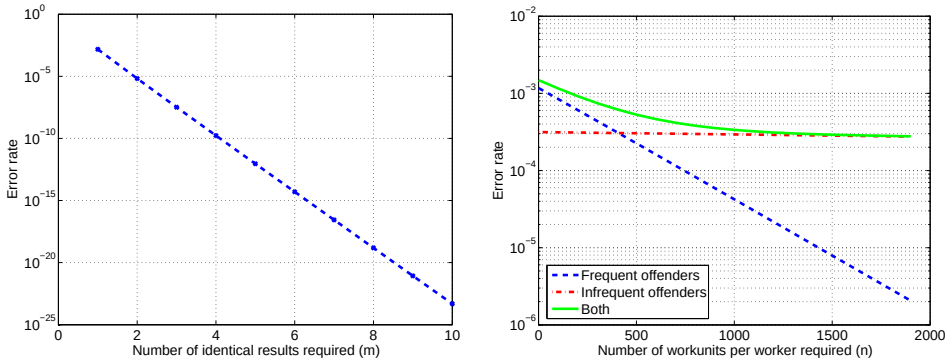
Figure 2 also shows the effect on throughput if the top fraction of hosts are blacklisted, assuming than an error is detected immediately and that after the error is detected, all workunits that had been completed previously by the host are discarded. If all hosts that commit errors are blacklisted, then clearly throughput is negatively affected and reduced by about 0.40. Nevertheless, blacklisting could be a useful technique if it is applied to the top offending hosts. In particular, if the top 0.10 of hosts are blacklisted, this would cause less than a 0.05 reduction on the valid throughput of the system while reducing errors by 0.70. One implication of these results is that an effective strategy to reduce errors could focus on eliminating the small fraction of frequent offenders in order to reduce the majority of errors without having a negative effect on overall throughput.

So we also evaluated majority voting and spot-checking in light of the previous result, by dividing the hosts into two groups, frequent and infrequent offenders based on the knee of the curve shown in Figure 2, but a similar problem described earlier occurs. The error rate for majority voting ε_{majv} is given by Equation 1, where $\varphi = f_{all} \times s_{frequent} \times f_{frequent} + f_{all} \times s_{infrequent} \times f_{infrequent}$. Note that f_{all} is simply the fraction of workers that could result in at least one error (0.35). $s_{frequent}$ (0.0335) and $s_{infrequent}$ (0.001) (see Table 1) are the error rates for frequent and infrequent offenders respectively. $f_{frequent}$ (0.10) and $f_{infrequent}$ (0.90) are the fraction of erroneous workers in the frequent and infrequent groups respectively.

We plot ε_{majv} as a function of m in Figure 3(a). We find that the error rate ε_{majv} decreases exponentially with m, beginning at about 1×10^{-5} for $m = 2$.

We also compute the error rate for spot-checking with blacklisting when dividing the hosts in terms of frequent and infrequent offenders. The error rate ε_{scbk} is given by the sum of the error rates for each grouping, $\varepsilon_{scbk,frequent}$ and $\varepsilon_{scbk,infrequent}$. Note that $\varepsilon_{scbk,infrequent}$ is given by substituting $f_{all} \times f_{frequent}$ for f and $s_{frequent}$ for s in Equation 2. $\varepsilon_{scbk,infrequent}$ can be calculated similarly.

Then we plot in Figure 3(b) $\varepsilon_{scbk,frequent}$, $\varepsilon_{scbk,infrequent}$, and ε_{scbk} as a function of n (the number of workunits that must be computed by each worker) where $q = 0.10$. The plot for the frequent offenders decreases exponentially; this



(a) With Majority Voting (b) With Spot-Checking and Blacklisting

Fig. 3. Error Rate Bounds

is because the error rate for the hosts is relatively high, and so after a series of workunit computations, the erroneous hosts are rapidly detected. The plot for the infrequent offenders decreases very little even as n increases significantly. This is because the error rates for the infrequent offenders are relatively low, and thus, increasing n does not improve detection nor reduce errors significantly. The effect of the net error rate ε is that it initially decreases rapidly for n in the range [0, 1000]. Thereafter, the error rate decreases little.

Thus, spot-checking acts as a low-pass filter in the sense that hosts with high error rates can be easily detected (and can then be blacklisted); however, hosts with low error rates remain in the system. If all frequent offenders are detected by spot-checking and blacklisted, then by Figure 2, this will reduce error rates by 0.70 (or equivalently, an error rate of 63×10^{-5}) and cause only a 0.05 reduction in throughput due to blacklisting. However, to reduce the error rate down to 1×10^{-5}, spot-checking must detect errors from both frequent *and* infrequent offenders. As shown by Figure 3(b), spot-checking will not efficiently detect errors from infrequent offenders because it requires a huge number workunits to be processed by each worker. From Figure 3(b), we conclude that spot-checking can reduce error rates down to about 2×10^{-4} quickly and efficiently. To achieve lower error rates, one should consider using majority voting. In the next section, we show that spot-checking may have other difficulties in real-world systems.

5.2 Stationarity of Error Rates

Intuitively, a process is stationary if its statistical properties do not change with time. In this section, we investigate how stationary the mean of the host error rate s is over time, and describe the implications for error detection mechanisms given our findings.

We measured the stationarity of error rates by determining the change in mean error rates over 96 hour periods for each host. That is, for every 96 hours

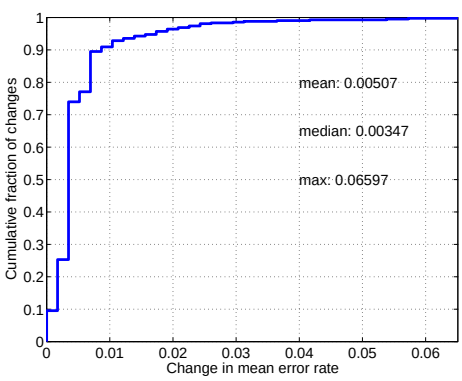


Fig. 4. Error Rate Stationarity

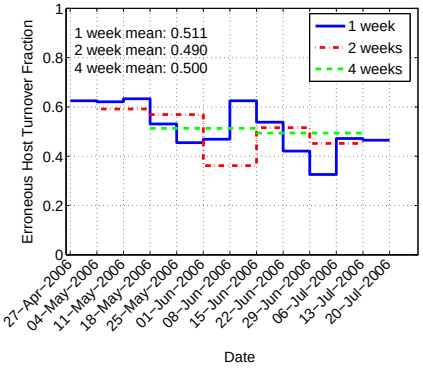


Fig. 5. Turnover Rate of Erroneous Hosts

of wall-clock time during which the worker had been active, we determined the mean error rate on each host, and measured the change in error rates from one period to the next . After close inspection of the results, we found that hosts often have long periods with no errors, and that when errors occurred, they occurred sporadically. Figure 4 shows the cumulative distribution function of error rate changes over all hosts. This results in Figure 4 show that workunit errors are not very stationary, and in fact, the error rate fluctuates significantly over time.

	Statistic		
Host Group	μ	σ	σ/μ
All erroneous	0.0034	0.018	3.48
Top 10% erroneous	0.0335	0.030	0.89
Bottom 90% erroneous	0.001	0.002	2.01

Table 1. Statistics for Host Error Rates over 96 hour Periods.

We also computed statistics for *host* error rates over 96 hour periods. This characterizes s as defined in Section 3. Table 1 shows the mean, standard deviation, and coefficient of variation (which is the standard deviation divided by the mean) for all hosts , the top 10% of erroneous hosts, and the bottom 90% of erroneous hosts. We find that even for relatively long 96 hour periods, the host error rate is quite variable. In particular, the coefficients of variation for all hosts, the top 10%, and the bottom 90% are 3.48, 0.89, and 2.01 respectively.

To investigate the seasonality of errorless periods, we determined whether the set of hosts that err from time period to time period are usually the same hosts or different. In particular, we determined the erroneous host turnover rate as follows. For a specific time period, we determine which set of hosts erred, and then compared this set with the set of the hosts that erred in the following time period. The erroneous host turnover fraction is then the fraction of hosts in the first set that do not appear in the second set. We computed the erroneous host turnover fraction for time periods of 1 week, 2 weeks, and 4 weeks (see Figure 5). For example, the first segment at about 0.62 corresponding to the 1

week period between April 27 and May 4 means that only 0.62 of the hosts that erred between April 20 and April 27 also erred between April 27 and May 4 . On average, the turnover rate is about 0.50 for all periods, meaning that from time period to time period, 0.50 of the erred hosts will be newly erred hosts. That is, the 0.50 of erred hosts had *not* erred in the previous period.

One explanation for the lack of stationarity is that desktop grids exhibit much host churn, as users (and their hosts) often participate in a project for a while and then leave. In [10], the authors computed host lifetime by considering the time interval between entry and its last communication to the project. The host was considered "dead" if it had not communicated to the project for at least 1 month. They found that a host lifetime in Internet desktop grids was on average 91 days.

One implication is that mechanisms that depend on the consistency of error rates, such as spot-checking and credibility-based methods, may not be as effective as majority voting. Spot-checking depends partly on the consistency of error rates over time. Given the high variability in error rates and the intermittent periods without any errors, a host could pass a series of spot-checks, and thereafter or in between spot-checks, the host could produce a high rate of error. Conversely, an infrequent offender could have a burst of errors, be identified as an erroneous host via spot-checking, and then blacklisted. If this occurs with many infrequent offenders, this could potentially have a negative impact on throughput as shown in Figure 2. The same is true for credibility-based systems as a host with variable error rates could build a high credibility, and then suddenly, cause high error rates. Thus, the estimated bounds resulting from spot-checking or credibility-based methods may not be accurate in real-world systems. By contrast, majority voting is not as susceptible to fluctuations in error rates, as the error rate (and confidence bounds on the error rate) decrease exponentially with the number of votes Nevertheless, the effectiveness of majority voting could be hampered by correlated errors, which we investigate in the next section.

5.3 Correlation of Error Rates

Using the trace of valid and erroneous workunit completion times, we computed the empirical probability that any two hosts had an error at the same time. That is, for each 10 minute period between April 20 to July 20, 2006, and for each pair of hosts, we counted the number of periods in which both hosts computed an erroneous workunit, and the total number of periods in which both hosts computed a workunit (erroneous or correct). Using those counts, we then determined the empirical probability that any two hosts would give an error simultaneously, i.e., within the same 10 minute period. In this way, we determined the the empirical joint probability of two hosts having an error simultaneously.

We then determined the "theoretical" probability of two hosts having an error simultaneously by taking the product of their individual host error rates. The individual host error rates are given by dividing the number of erroneous workunits per host by the total number of workunits computed per host (as

described in Section 5.1). After determining the "theoretical" probabilities for each host pair, we then determined the difference between the theoretical and empirical probabilities for each host pair. If the error rates for each pair of hosts are not positively correlated, then the theoretical probability should be greater than or equal to the empirical, and the difference should be nonnegative.

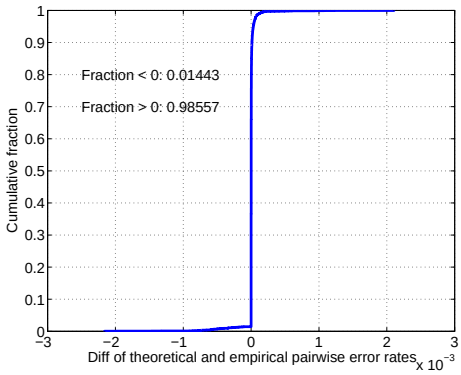


Fig. 6. Pairwise Host Error Rates

Figure 6 shows the cumulative distribution for the differences between theoretical and empirical pairwise error rates. We find most (0.986) of the theoretical pairwise error rates were greater than the empirical. This suggests that the error rates between hosts are not positively correlated. Moreover, only 0.01443 of the pairings had differences less than 0. After carefully inspecting the number of workunits computed by these host pairs, we believe these data points are in fact outliers due a few common errors made by both hosts over a relatively low number of workunits.

6 Summary

We characterized quantitatively the I/O error rates in a real Internet desktop grid system with respect to the distribution of errors among hosts, the stationarity of error rates over time, and correlation among hosts. In summary, the characterization findings were as follows. First, a significant fraction of hosts (about 35%) will commit at least a single error over time. Second, the mean error rate over all hosts (0.0022) is low. Third, a large fraction (e.g. about 70%) of errors result from a small fraction (e.g. 10%) of hosts. Fourth, error rates over time vary greatly and do not seem stationary. Error rates can vary as much as 3.48 over time. The turnover rate for erroneous hosts can be as high as 50%. Fifth, error rates between two hosts often seem uncorrelated. While correlated errors could occur during a coordinated attack or after worm propagation, we do not believe it is the most common source of errors in practice.

In light of these characterization findings, we showed the effectiveness of several error prevention and detection mechanisms namely blacklisting, majority voting, spot-checking, and credibility-based methods. We concluded the following. First, if one can afford redundancy or one needs an error rate to be less then 2×10^{-4}, then majority voting should be strongly considered. Second, if one can afford an error rate greater than 2×10^{-4} and can make batches relatively long (ideally with at least 1000 work units and at least 1 week of CPU time per worker), then spot-checking with blacklisting should be strongly con-

sidered. Third, fluctuations in error rates over time may limit the effectiveness of credibility-based systems.

For future work, we plan to address a limitation of this study. Our method measures the I/O error rates of only a single, compute-intensive application. While we believe this application is representative of most Internet desktop grid applications in terms of its high ratio of computation compared to communication, applications with different I/O or computation patterns could potentially differ in error rates. Thus, to study I/O and computational errors of another application, we will deploy a real scientific application that has easily verifiable results . Nonetheless, we believe this is the first study of a real project to give quantitative estimates of I/O error rates.

References

1. : Catalog of boinc projects. `http://boinc-wiki.ath.cx/index.php?title=Catalog_of_BOINC_Powered_Proje%cts`
2. Taufer, M., Anderson, D., Cicotti, P., III, C.L.B.: Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In: Proceedings of the International Heterogeneity in Computing Workshop. (2005)
3. Oltean, A.: How to do atomic writes in a file. `http://blogs.msdn.com/adioltean/archive/2005/12/28/507866.aspx` (December 2005)
4. Anderson, D.: Boinc: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA (2004)
5. Fedak, G., Germain, C., N'eri, V., Cappello, F.: XtremWeb: A Generic Global Computing System. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01). (May 2001)
6. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: Architecture and Performance of an Enterprise Desktop Grid System. Journal of Parallel and Distributed Computing **63** (2003) 597–610
7. Sarmenta, L.: Sabotage-tolerance mechanisms for volunteer computing systems. In: Proceedings of IEEE International Symposium on Cluster Computing and the Grid. (May. 2001)
8. Zhao, S., Lo, V.: Result Verification and Trust-based Scheduling in Open Peer-to-Peer Cycle Sharing Systems. In: Proceedings of IEEE Fifth International Conference on Peer-to-Peer Systems. (May 2001)
9. Malecot, P., Kondo, D., Fedak, G.: Xtremlab: A system for characterizing internet desktop grids (abstract). In: in Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing. (2006)
10. Anderson, D., Fedak, G.: The Computational and Storage Potential of Volunteer Computing. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06). (2006)

Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 23 (2007) 983–989
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Towards efficient data distribution on computational desktop grids with BitTorrent

Baohua Wei, Gilles Fedak*, Franck Cappello

Laboratoire de Recherche en Informatique/INRIA Futurs Bat 490, Université Paris Sud, 91405 Orsay Cedex, France

Received 27 November 2006; accepted 5 April 2007
Available online 18 April 2007

Abstract

Datacentric applications are still a challenging issue for large-scale distributed computing systems. The emergence of new protocols and software for collaborative content distribution over the Internet offers a new opportunity for efficient and fast delivery of a high volume of data. This paper presents an evaluation of the BitTorrent protocol for computational desktop grids. We first present a prototype of a generic subsystem dedicated to data management and designed to serve as a building block for any desktop grid system. Based on this prototype we conduct experiments to evaluate the potential of BitTorrent compared to a classical approach based on FTP data server. The preliminary results obtained with a 65-node cluster measure the basic characteristics of BitTorrent in terms of latency and bandwidth and evaluate the scalability of BitTorrent for the delivery of large input files. Moreover, we show that BitTorrent has a considerable latency overhead compared to FTP but clearly outperforms FTP when distributing large files or files to a high number of nodes. Tests on a synthetic application show that BitTorrent significantly increases the communication/computation ratio of the applications eligible to run on a desktop grid system.

Keywords: Desktop grid; Collaborative content distribution; BitTorrent

1. Introduction

For the last few years, the idea of high throughput computing over large sets of idle desktop computers has become more and more popular. This acceptance is partly due to the success of mainstream projects like SETI@Home [1] or Distributed.net [2] which have gathered a tremendous amount of computing power (for instance SETI@Home is claiming more than 100 TFlops as of January 2005), and the availability of generic software platforms, being open source like BOINC [3], XtremWeb [4], OurGrid [5], or commercial like Entropia [6] or Sun Grid Engine [7]. Nonetheless, desktop grids systems are still restricted to a few classes of applications: mainly the embarrassingly parallel applications (bag of tasks, master/slave) with a high computation/communication ratio. Therefore, a primary concern for a broader use of desktop grids is the ability to address a wider scope of applications. This paper focuses on multiparametric applications which feature a high volume of data inputs, highly shared among a large number of independent tasks. Such characteristics are frequent for trace-based simulations and bioinformatics applications which require access to large databases.

In this scenario of datacentric applications, the existing desktop grid systems face a scalability issue. One should expect that more computing resources also provides more network bandwidth and storage capacity. On the contrary, desktop grids systems like BOINC or XtremWeb rely on a centralized data service architecture. For instance, data distribution with BOINC relies on multiple http servers and tasks are described as a list of file locations, which could be a potential bottleneck when scheduling tasks sharing large input files.

Recent developments in content distribution such as collaborative file distribution (BitTorrent [8], Slurpie [9], Digital Fountain [10]) and P2P file sharing applications (EDonkey/Overnet [11], Kazaa [12]), have proven to be both effective, viable and *self-scalable*. Today, a significant part of the Internet bandwidth is used by P2P traffic [13]. The key idea, often featured as *parallel download* in the P2P applications, is to divide the file into chunks. Immediately after a peer

* Corresponding author.
E-mail address: fedak@lri.fr (G. Fedak).

0167-739X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.04.006

downloads a chunk from another peer, the peer serves the block for the other peers in the network, thus behaving itself as a server. Collaborative content distribution is a very active research topic and several promising strategies [14] such as the use of network coding [15], are proposed to improve the performance of the system.

However, the context of computational desktop grids shows specific characteristics when considering data involved in large multi-parametric applications. For instance, such applications are often composed of a large number of small files describing the parameters. Thus efficiency of the data diffusion mechanism should be evaluated according to its impact on the overall performance of a parallel application when scheduled on the Grid.

To evaluate the potential of the collaborative data diffusion approach, we focus on the BitTorrent protocol. Our methodology for evaluating BitTorrent is the following: we first design a data management prototype using the XtremWeb Desktop Grid as a reference architecture. We discuss the feasibility of a BitTorrent-based desktop grid architecture with two qualitative criteria: security and deployment. Using this prototype we conduct two sets of performance evaluation. The first one aims at measuring the basic performance in terms of latency and bandwidth and evaluating the scalability of BitTorrent for file transfers. The second one tests BitTorrent against a synthetic bag of tasks parallel application with shared input data. We conduct those experiments within a 65-nodes cluster and compare the results with a classical FTP-based solution used in centralized desktop grids. We demonstrate that even at that scale, BitTorrent improves the overall performance of the application by a factor up to 2.5.

The rest of the paper is organized as follows: Section 2 presents the challenge of data distribution for computational desktop grids. In Section 3 we present our prototype. Then, in Section 4, we conduct a performance evaluation of BitTorrent. Finally we conclude the paper in Section 5.

2. Challenge of efficient data distribution services for desktop grids

To achieve high scalability, an efficient data distribution faces two constraints. The first one is relative to the resources. It requires that the system can adapt: (1) to a very large number of resources, (2) to the high volatility of the resources, as computing nodes can join and leave the network at any time, which may preclude the computation of any fixed topology, (3) to the heterogeneity of network performances (nodes in a LAN, nodes at home, intermittent connections) and (4) to the change of dimension of the system (daytime vs nighttime, flash-crowd effect). The second issue is relative to the data sets of parameter-sweep data-intensive applications which are composed of: (1) large files highly shared (for instance, traces for a trace-driven simulator) and (2) parameter files which are small text files uniquely describing each task. Thus the system should provide both low latency for small messages and high bandwidth for bulk data transfers.

2.1. An overview of BitTorrent

BitTorrent is a popular file distribution system which aims at avoiding the bottleneck of FTP servers when delivering large and highly popular files. The key idea of BitTorrent is cooperation of the downloaders of the same file by uploading chunks of the file to each other.

A peer will first get all the information about the file to download in a special static file called a `.torrent`. A `.torrent` contains the SHA1 signature for all the chunks composing the file and the location of a central agent called the *tracker*, which helps the peers to connect to each other. In a BitTorrent network, trackers are responsible for keeping a list of information about the peers: IP address, port to contact, file downloads. When a peer requests a file, it first asks the tracker for a list of contact information about the peers who are downloading the same file. The tracker does not organize transfers of the chunks between the peers; all data movements are decided locally by a peer according to local information collected from its neighbours.

From this list of peers, the downloader asks its neighbors for the transfer of the file chunks. In the BitTorrent terminology, this operation of uploading is known as *unchoking*, the dual operation called *choking* is a temporary refusal of upload. The strategy for choking/unchoking relies on 3 rules: (1) no more than 4 file uploads are to run at a time, (2) selection of the upload is based on the best transfer rates averaged over the last 20 s time frame and (3) once every 30 s, an *optimistic unchoking* operation is performed, which randomly selects a fifth uploader. This operation helps to discover peers with a better bandwidth.

2.2. Related works involving BitTorrent

There have been several analyses of BitTorrent's performance through observation of large-scale utilization, through analytical modelling and through simulation of the protocol.

Measurement-based studies [8,16–18] of BitTorrent are conducted by analysing the logs of the tracker and instrumenting one client. Several trackers which correspond to different kind of files were investigated (the RedHat tracker for Linux distribution or SuprNova tracker for multimedia files). Thus data movements observed correspond to the interest that users have in these files and are characterized in terms of files availability and popularity, download volume and performance, geographical analysis, evolution of the seeds/downloaders ratio. From these observations, the authors conclude that BitTorrent is efficient for distributing large files when the number of nodes is high.

A model of BitTorrent based on fluid dynamics is proposed in [14], which quantifies the evolution of downloaders/seeders and the download time in function of nodes arrival/departure rates and network bandwidth. This study provides the evidence for the scalability of the BitTorrent and fairness of the built-in incentive mechanism. Performance evaluation of BitTorrent through simulations, in [19], shows similar results and they also show the efficiency under flash-crowd effects and fairness amongst peers in terms of volume served. In [15], authors

propose to use network coding to improve the performance of the BitTorrent protocol. With network coding, each node in the network is able to generate and transmit encoded pieces of data. Simulations shows an improvement of 2–3 times over unencoded versions of BitTorrent.

Overall, our work contrasts to others in that we: (i) measure experimentally an instrumented BitTorrent protocol, (ii) use a small to medium scale deployment where the tradeoff between FTP and BitTorrent is likely to occur, (iii) use real traces of communication as input for simulations.

3. Prototype for a data distribution subsystem

In this section we discuss the design of a prototype for a data distribution subsystem for computational desktop grids. By designing this prototype, we want to evaluate the impact of BitTorrent on the desktop grid architecture.

While this subsystem aims at being integrated in the XtremWeb desktop grid, we think that it is generic enough to serve as a building block for new systems or as replacement for existing centralized data repositories like in BOINC. In this section we briefly introduce XtremWeb, then we present our prototype data distribution subsystem and draw some guidelines concerning security, deployment and file management.

3.1. Overview of XtremWeb

XtremWeb [4] is an open source platform for desktop grids computing. XtremWeb offers a software infrastructure to gather the unused resources of PCs (CPU, network, storage) spread over LANs or the Internet to execute highly parallel applications. Its primary features permit multi-users, multi-applications and cross-domain deployments. XtremWeb follows the general vision of a large-scale distributed system turning a set of commodity resources (possibly volatile) into a runtime environment executing services (application modules, runtime modules or infrastructure modules) and providing volatility management. Typical parallel applications eligible to run on XtremWeb are embarrassingly parallel and an API is proposed to program them following the master/slave paradigm.

The architecture follows a three-tiers design (worker, coordinator, client), which is commonly found in desktop grids (see Condor [20] for a reference on this class of architecture). The design follows a set of three main principles: (1) a fault-tolerant architecture allowing the mobility of the clients, the volatility of the workers and intermittent crashes of the coordination service, (2) a set of security mechanisms based on autonomic decisions, (3) a connectionless and one-sided communication protocol.

XtremWeb allows a set of clients to submit task requests to the system which will execute them on workers. The three-tiers architecture adds a middle tier between client and worker nodes. Thus there is no direct peer-to-peer task submission and file transfer between clients and workers. The role of the third tier, called the coordinator, is (a) to decouple clients from workers and (b) to coordinate tasks execution on workers. The coordinator accepts task requests coming from several clients, distributes the tasks to the workers according to a scheduling policy, transfers application code and input file to workers if necessary, supervises task execution on workers, detects worker crash/disconnection, re-launches crashed tasks on any other available worker, collects and stores task results, delivers task results to client upon request. In the present implementation, the coordinator is implemented by a centralized node, eventually replicated. To ease the deployment phase regarding the connection issues raised by firewall and NAT configuration, all the communications are initiated by clients and workers toward the coordinator node.

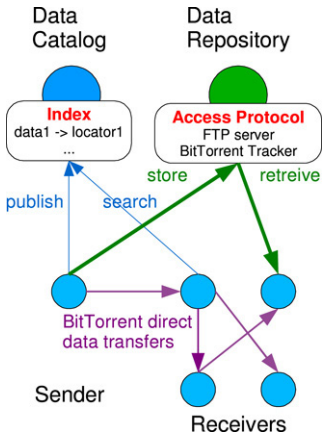


Fig. 1. Architecture of the prototype.

3.2. Architecture of the data distribution subsystem

Our architecture follows closely the three-tiers architecture described above. We enhance the middle tier with two entities dedicated to data management: the data catalogue and the data repository. Fig. 1 illustrates this design.

The data catalogue keeps track of the data and their location. Each datum is referred by a unique identifier (UUID) computed with random number, time-stamp and network address, and additional attributes for managing the integrity of the data (SHA1 signature, size of the file, type flags: binary, executable, text, compressed, architecture dependant). Among this information, a datum is associated with one or several locators which are addresses of data repositories and the required information to operate a file transfer (protocol, port, path, file name, login name and password).

The data repository stores the data and should be remotely accessed by the senders and receivers of files. The data repository can be managed by a client of the desktop grid system to avoid transfer to an intermediate or it can be used as a reliable storage for managing fault tolerance of the client (*launch and forget* mode). The data repository runs the necessary third-party software required to access the data, e.g. a FTP file server or a BitTorrent tracker.

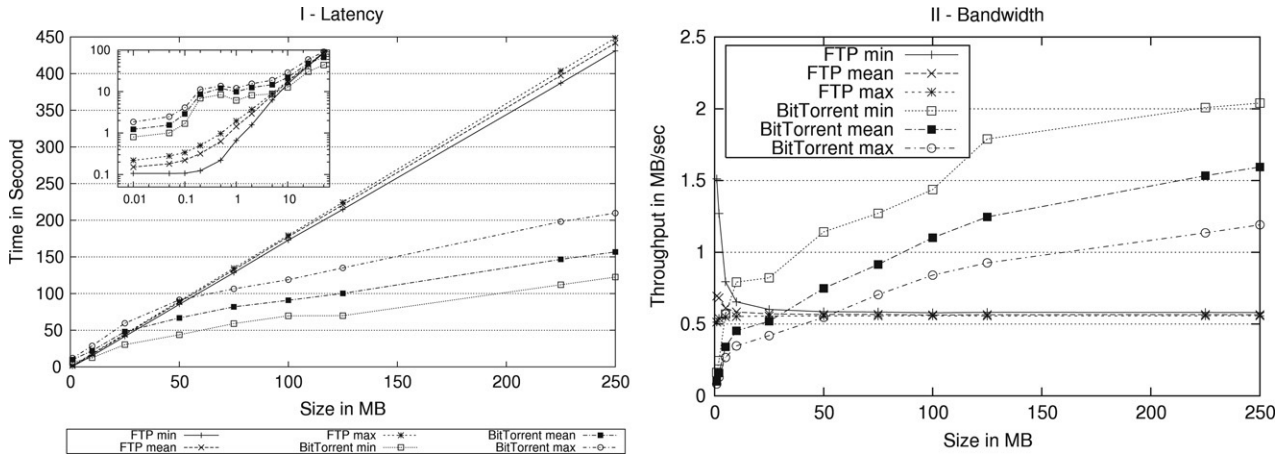


Fig. 2. The minimum, maximum and average completion time in seconds for the distribution of the file of a size varying from 1 to 250 MB to 20 nodes. The Fig. 2-I presents a close-up of the latency at logarithmic scale for the distribution of small files, with sizes between 10K and 50 MB. Fig. 2-II presents the bandwidth in MB/s.

3.3. Some remarks about security, deployment and transfers management

The design of this prototype raises some issues about security, deployment and transfers management.

A data management subsystem offers a remote storage facility, thus it has to prevent abuse of the service by malicious users who could store their own files on the server. When a task has completed, information to commit the results back to the data repository are available and could potentially be used to abuse the system. As a preventative action, the tasks submitter can retrieve the results as soon as a task is completed, therefore making the storage facility temporary and useless. Unfortunately, this solution might not be sufficient with FTP servers as the right management allows the creation and writing of new files in the server. In contrast, BitTorrent does not allow push operations; transfers are only initiated by download requests, which forbid unauthorized file creation.

Ease of deployment is a requirement for desktop grid systems. It makes components convenient to install and configure so that users can take advantage of existing infrastructure. Both BitTorrent and FTP implementations exist in multi-platform languages such as Java or Python which facilitates deployment on heterogeneous environments, and in native versions when performance is critical (servers side). However, the security issues raised below by FTP servers would suggests that the desktop grid system has control over the rights management of the server to create temporary user rights. This point could forbid location of the data repository on the client side, and therefore direct client-to-worker communications.

If transfers are trivial to manage with FTP, BitTorrent requires that nodes which own the data run BitTorrent to allow collaborative transfers. Thus, if a local mechanism exists to cache the data on the peer, the Data Repository should be able to trigger the BitTorrent client on the remote peer.

4. Experimental results

This section presents the experimental evaluation of BitTorrent for computational desktop grids. We begin with a description of the experimental conditions. Then we present the results obtained in our experiments.

4.1. Experiments setup

We have conducted our experiments in a predictable environment in order to evaluate the overhead and benefits of BitTorrent. The testbed is the *LRI Simulation Cluster* which is a 64-nodes cluster of heterogeneous ix86 machines. It is a set of single and dual CPU Athlon 32 1.5 GHz and Intel P-IV 2 GHz, each node with 885 MB RAM and interconnected with a 100 Mbps Ethernet switch. To stress the data server, it has been separated from the cluster. Due to the very dynamic nature of the BitTorrent protocol, every experiment was run 30 times, and the results reflect the averaged times.

The software configuration privileges Java implementation of the client part of the protocols in order to comply with realistic deployment on a heterogeneous platform and the native implementation for the server part of the protocol. The BitTorrent implementation evaluated is Azureus Version 2.2.0.2 (available on http://azureus.sourceforge.net), the BitTorrent tracker is the reference python implementation Version 3.9.0 (http://www.bittorrent.com), the FTP client is provided by the Apache Jakarta commons-net package Version 1.3.0 (http://jakarta.apache.org) and the FTP server is the Washington University FTP server Version 2.6.2 (http://www.wu-ftpd.org).

4.2. Basic performance of BitTorrent

This first experiment compares the performance of BitTorrent versus FTP when distributing a file to a set of 20 nodes. The file size varies from 1 to 250 MB. Fig. 2-I presents the minimum, maximum and average completion time in seconds for the file transfer. The time is measured on each receiver node and is averaged over 30 experiments. The close-up figure plots with a logarithmic scale, the latency in second for transferring small files (size between 10 KB to 50 MB).

This first result illustrates that for large files, the time to complete the file distribution for FTP grows linearly with the

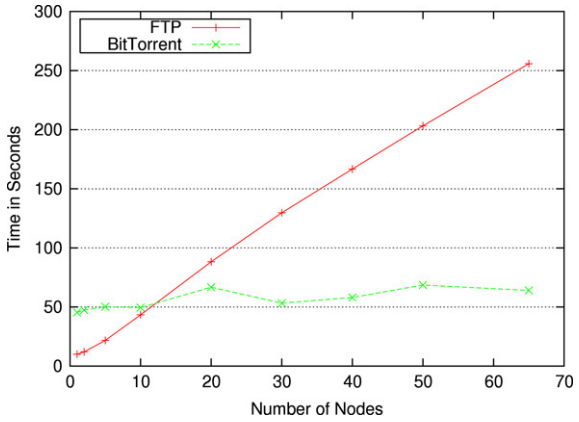


Fig. 3. Completion time for the distribution of a 50 MB file to a varying number of nodes. The vertical axis shows the time in seconds, the horizontal axis shows the number of nodes.

size of the file. In this experiment the bandwidth of the FTP server is shared by all downloaders. The access list of the FTP server is configured to allow more clients than the actual number of downloaders. One can observe that the maximum, the average and minimum curves for FTP are very similar, which shows that the server bandwidth is equally shared between the downloaders.

BitTorrent clearly outperforms FTP when the file size is greater than 20 MB. After this crossover point, the curve grows slightly with a slope approximately equals to 0.45. The cooperation between the nodes is effective to decrease the transfer time even if a modest number of hosts is involved (in this case 20 nodes). This shows a clear potential of using BitTorrent for large file transfer instead of FTP. The difference between the minimum, mean and maximum curves is discussed later in the paper.

If BitTorrent protocol seems appealing for large files, FTP is more efficient for small files transfer. Multi-parametric applications are often composed of a simulation code associated with one or several configuration text files, which describe the parameters of the execution. Thus, this kind of study requires the transfer of numerous small files.

When considering small file transfers, BitTorrent presents an overhead higher than FTP: about 0.8 and 0.1 s. The BitTorrent overhead is due to the various steps the protocol imposes before actually starting the file communication. First the downloader has to read the `.torrent` file to extract the information about the chunks and the tracker, next it has to contact the tracker to receive the list of peers. Then the downloader needs to wait for the seeder or another peer to upload the chunks of file, with the additional constraint that the upload queue is limited to four slots.

Finally Fig. 2-II presents the bandwidth as measured locally on the sender node. We observe that the bandwidth for the FTP protocol is kept constant due to the fair sharing of the server bandwidth among the downloaders. In contrast, BitTorrent shows a noticeable improvement of the throughput when the file size increases. When the number of nodes is low, BitTorrent is still effective for large file sizes.

With the following experiment, we evaluate the scalability of BitTorrent compared to FTP. The benchmark consists of a pool of nodes varying from 1 up to 64 which simultaneously download a 50 MB file. The file is located on a central FTP server or on the first BitTorrent client. Each node measures the time to complete the download and Fig. 3 presents the average time to transfer the file, compared to the number of workers.

As seen in the figure, the download time of BitTorrent remains stable as the number of resources increases while it linearly increases with FTP. This results shows that the scalability of BitTorrent is the one expected when the number of nodes is high. Another study [19] based on simulations implying more than 1000 nodes confirms this general trend. However, with a 50 MB file, there is a crossover point around 10 workers where FTP is more efficient than BitTorrent due to the overhead of BitTorrent.

4.3. Communication patterns of BitTorrent

The following experiment compares the profiles of the file distribution of the two protocols. We plot in Fig. 4 the download times when a file of 100 MB is delivered to 20 nodes. The experiment is repeated 30 times and each point on the plot represents the time for a node to complete the file transfer during one run. The horizontal axis represents the measures for every run. The upper three curves (maximum, mean and minimum) refer to the FTP measurements and the lower three ones to the BitTorrent measurements.

The figure shows that: (1) the download time is always lower for BitTorrent (2) while the distributions of the download time for FTP are quite homogeneous, BitTorrent suffers from less consistent performance. With BitTorrent, the download time can vary from a factor of three between the fastest and the slowest node.

In order to understand these variations we have instrumented the BitTorrent client. On each peer, we log every communication made to the other peers. The Fig. 5 presents the communication pattern of the BitTorrent protocol when diffusing a file

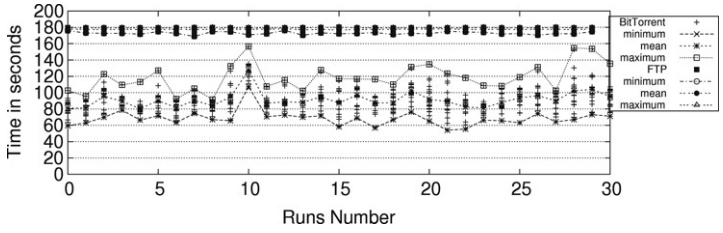


Fig. 4. Profiles of downloads for a 100 MB file by 20 machines: each point on the plot represents the time to completion for the file transfer for one node during one run. The three curves represent the maximum, mean and minimum for both FTP and BitTorrent.

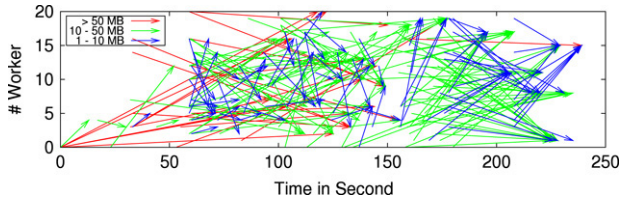


Fig. 5. Communication pattern of the BitTorrent protocol when diffusing a 250 MB file to 20 nodes. First point of a vector presents the beginning of a communication (start time, sender) and the second point presents the end of the communication. The colour of the vector presents the volume of data transmitted.

of size 250 MB to 20 nodes. A vector represents a communication from a sender to a receiver (vertical axis) during a period of time (horizontal axis) with no more than 10 s idle. We discarded vectors with a volume of data less than 1 MB; however, more than 99.45% of the total volume transmitted is presented.

The figure shows that: (1) nodes start to upload file chunks to other nodes before receiving the whole file, (2) largest communications are performed at the start of the diffusion and (3) the topology at the beginning of the diffusion represents a tree and a pipeline is organized to transmit the whole file to the last served nodes.

Future work should try to anticipate the construction of the topology in order to model the downloading time of individual nodes. This point can impact on the overall performance of application execution, as forecasting of a communication duration is required for appropriate scheduling decisions.

4.4. Evaluation on a synthetic multi-parametric application

The last set of experiments evaluate the potential of using BitTorrent in place of an FTP server for a multi-parametric synthetic application.

First, we consider an application with several computing/communication ratios. The application is composed of a set of n independent tasks, n being equal to the number of nodes involved. A task consists of two phases: a file transfer (20 MB) followed by an execution. Execution time of each task is $t_{\text{communication}} + t_{\text{computation}}$. The reference $t_{\text{communication}}$ is set to the time to transfer 20M between two nodes using FTP and the ratio $t_{\text{computation}}/t_{\text{communication}}$ varies from 0.1 to 10. We have used two metrics for computing the speed: *makespan* is the duration between the start of the first task and the completion of the last task. Nodes could be idle when the last nodes are still computing and thus be attributed to another application. We have used a second metric *average* which represents the average execution time on each node.

The result shown in Fig. 6 presents the speed of BitTorrent compared to FTP for a varying number of nodes. Speed of BitTorrent increases with the number of nodes and the communication ratio to reach a factor 2.5 when r is 10 and n is 70 with *average* metric and 2.3 when considering *makespan*. We note that *average* shows better results than *makespan* which is due to the differences in downloading time between the nodes (*makespan* considers the maximum downloading time). On the contrary, when the number of nodes is small and when the communication ratio is high, FTP outperforms BitTorrent owing to the large overhead when transmitting small files.

The last experiment confronts BitTorrent and FTP with node volatility. This experiment evaluates the transfer failure rate, that is the probability that a host will become unavailable before a transfer is complete. We base this model on host availability characterization proposed by Kondo and All. [21], which gives the probability that a host will become unavailable according to the duration of a task. We have combined this failure rate with a linear model of the results presented in Fig. 7. The higher efficiency of BitTorrent permits decrease of the transfer failure rate by a factor of 2.8.

5. Conclusion

Collaborative data distribution has become a key technology on the Internet. In this paper, we investigated BitTorrent as a protocol for data diffusion in the context of computational desktop grids. We designed a prototype and found that even if desktop grid architectures often rely on centralized coordination components, they can easily integrate this P2P technology without fundamental changes on their model of security or deployment.

We conducted an experimental performance evaluation of the protocol on a LAN cluster and showed that BitTorrent is efficient for large file transfers, scalable when the number of nodes increases, but suffers from a high overhead when transmitting small files. Compared with an FTP-based centralized desktop grid, the execution of a multi-parametric

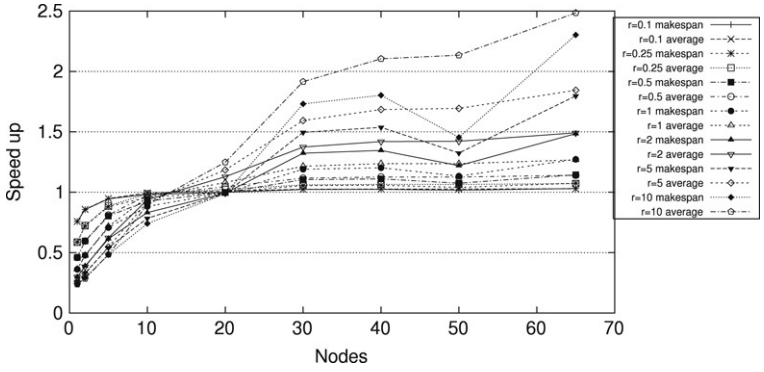


Fig. 6. Speed up of BitTorrent versus FTP on a synthetic application. We plot curves for ratios communication/computation r equal to 0.1, 0.25, 0.5, 1, 2, 5, 10 and two metrics for the execution time *makespan* and *average*.

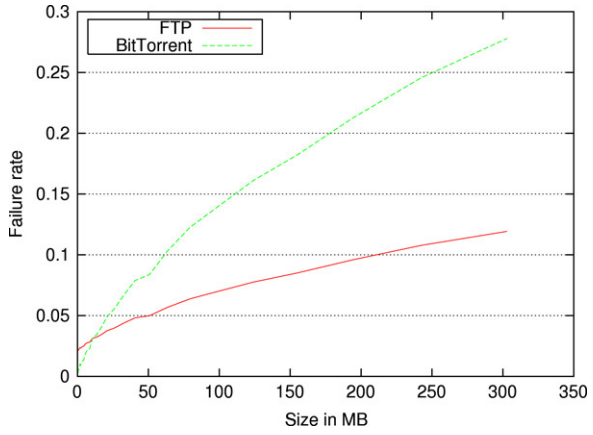


Fig. 7. Transfer failure rate versus transfer size (in MB).

application demonstrates that BitTorrent is able to execute applications with a higher communication/computation ratio and to reduce the fault probability, which are two requirements for a broader use of desktop grids. However, due to its high overhead, a misuse of BitTorrent, e.g. for small files or files that are not shared enough, results in a performance penalty.

Therefore we think that future work around the integration of desktop grid and collaborative data distribution should focus on: (1) improving the latency of BitTorrent, (2) experimenting with other P2P protocols and evaluating the cost and benefit of indexing/publishing/searching data, (3) designing multi-protocols data desktop grid systems, (4) experimenting with other deployments (ADSL/Multi-LAN/WAN) and investigating how BitTorrent performs with various physical topologies and (5) designing BitTorrent-aware scheduling strategies for datacentric applications.

References

[1] D. Anderson, S. Bowyer, J. Cobb, D. Gedye, W.T. Sullivan, D. Werthimer, Astronomical and Biochemical Origins and the Search for Life in the Universe (Proc. of the Fifth Intl. Conf. on Bioastronomy), 1997.
[2] Distributed.net, RSA Labs' 64bit RC5 Encryption Challenge.
[3] D. Anderson, BOINC: A system for public-resource computing and storage, in: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA, 2004.
[4] F. Cappello, S. Djilali, G. Fedak, F.M. Thomas Hérault, V. Néri, O. Lodygensky, Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid, Future Generation Comput. Sci.
[5] N. Andrade, W. Cirne, F. Brasileiro, P. Roisenberg, OurGrid: An approach to easily assemble grids with equitable resource sharing, in: Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, 2003.
[6] A. Chien, B. Calder, S. Elbert, K. Bhatia, Entropia: Architecture and performance of an enterprise Desktop Grid system, J. Parallel Distrib. Comput. 63 (5) (2003) 597–610.
[7] W. Gentzsch, Sun Grid Engine (SGE): A Cluster Resource Manager, 2002.
[8] B. Cohen, Incentives build robustness in BitTorrent, in: Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, 2003.
[9] R. Sherwood, R. Braud, B. Bhattacharjee, Slurpie: A cooperative bulk data transfer protocol, in: Proceedings of IEEE INFOCOM, 2004.
[10] J.W. Byers, M. Luby, M. Mitzenmacher, A. Rege, A digital-fountain approach to reliable distribution of bulk data, in: Proc. of the ACL SIGCOMM, 1998.
[11] EDonkey, Edonkey, overnet homepage. http://www.edonkey200.com/, January 2002.
[12] FastTrack, P2P Technology. KaZaA Media Desktop. http://www.fasttrack.nu/, January 2002.
[13] CAIDA, CAIDA, the Cooperative Association for Internet Data Analysis: Top applications (bytes) for subinterface 0[0]: SD-NAP traffic, 2002.
[14] D. Qiu, R. Srikant, Modeling and performance analysis of BitTorrent-like peer-to-peer networks, SIGCOMM Comput. Commun. Rev. 34 (4) (2004) 367–378.
[15] C. Gkantsidis, P. Rodriguez, Network coding for large scale content distribution, in: Proceedings of IEEE/INFOCOM 2005, Miami, USA, 2005.
[16] M. Izal, G. Urvoy-Keller, E.W. Biersack, P.A. Felber, A.A. Hamra, L. Garces-Erice, Dissecting BitTorrent: Five months in a Torrent's lifetime, in: Proceedings of Passive and Active Measurements, PAM, 2004.
[17] J.A. Pouwelse, P. Garbacki, D. Epema, H.J. Sips, A measurement study of the BitTorrent peer-to-peer file sharing system, Tech. Rep. PDS-2004-007, Delft University of Technology, 2004.
[18] B.N.L. Anthony Bellissimo, P. Shenoy, Exploring the use of BitTorrent as the basis for a large trace repository, Tech. Rep., University of Massachusetts, 2004.
[19] A.R. Bharambe, H. Cormac, V.N. Padmanabhan, Understanding and deconstructing BitTorrent performance, Tech. Rep., Microsoft Research, 2005.
[20] M.J. Litzkow, M. Livny, M.W. Mutka, Condor — A hunter of idle workstations, in: Proceedings of the 8th International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, Washington, DC, 1988, pp. 104–111.
[21] D. Kondo, M. Taufer, C.L. Brooks, H. Casanova, A. Chien, Characterizing and evaluating Desktop Grids: An empirical study, in: IPDPS 2004, IEEE/ACM International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, 2004.

Baohua Wei is a Ph.D. candidate in computer science from Paris South University (France). He is a member of the INRIA Grand Large project. His work focus is on data management for desktop grid.

Gilles Fedak is a junior researcher in the INRIA Grand Large team. His research interests include desktop grids and global computing systems. He is involved in several projects targeting middleware, application scheduling, resources characterization, data management for desktop grids: XtremWeb, BitDew, DLSLab and XtremLab. He organizes the '06 and '07 editions of the Global and P2P Computing Workshops. He received his Ph.D. in computer science from Paris South University, France, in 2003.

Franck Cappello holds a senior researcher position at INRIA. He leads the Grand-Large project at INRIA, focusing on large scale distributed systems issues. He has initiated the XtremWeb (desktop grid) and MPICH-V (fault-tolerant MPI) projects. He is currently the director of the Grid5000 project, a nation wide computer science platform for research in grid and P2P. He has authored more than 60 papers in the domains of high performance programming, desktop grids, grids and fault-tolerant MPI. He has contributed to more than 30 programme committees. He is editorial board member of the "International Journal on GRID Computing" and steering committee member of IEEE HPDC and IEEE/ACM CCGRID. He was the general chair of IEEE HPDC'2006.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BitDew: A Programmable Environment for Large-Scale Data Management and Distribution

Gilles Fedak — Haiwu He — Franck Cappello

N° 6427

January 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6427--FR+ENG

DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

BitDew: A Programmable Environment for Large-Scale Data Management and Distribution

Gilles Fedak , Haiwu He , Franck Cappello

Thème SYM — Systèmes symboliques
Équipes-Projets Grand Large

Rapport de recherche n° 6427 — January 2008 — 24 pages

Abstract:

Desktop Grids use the computing, network and storage resources from idle desktop PC's distributed over multiple-LAN's or the Internet to compute a large variety of resource-demanding distributed applications. While these applications need to access, compute, store and circulate large volumes of data, little attention has been paid to data management in such large-scale, dynamic, heterogeneous, volatile and highly distributed Grids. In most cases, data management relies on ad-hoc solutions, and providing general approach is still a challenging issue.

To address this problem, we propose the BitDew framework, a programmable environment for automatic and transparent data management on computational Desktop Grids. This paper describes the BitDew programming interface, its architecture, and the performance evaluation of its runtime components. BitDew relies on a specific set of meta-data to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction. The Bitdew runtime environment is a flexible distributed service architecture that integrates modular P2P components such as DHT's for a distributed data catalog and collaborative transport protocols for data distribution. Through several examples, we describe how application programmers and Bitdew users can exploit Bitdew's features. The performance evaluation demonstrates that the high level of abstraction and transparency is obtained with a reasonable overhead, while offering the benefit of scalability, performance and fault tolerance with little programming cost.

Key-words: Data Management, Desktop Grids

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

BitDew: un environnement programmable pour la gestion et la diffusion des données à large échelle

Résumé : Les Grilles de PCs utilisent les capacités de calcul, de communication et de stockage d'ordinateurs personnels distribués sur plusieurs LAN ou sur l'Internet. La gestion des données dans ces grilles de grande échelle, dynamiques, hétérogènes, volatiles et hautement distribuées est un défi qui doit être relevé pour étendre l'usage des Grilles de PCs.

Nous proposons le logiciel BitDew, un environnement programmable pour la gestion et la distribution des données sur les Grilles de PCs. Ce rapport de recherche présente l'interface de programmation de BitDew, son architecture ainsi que les évaluations de performance des composants de l'environnement d'exécution. Nous décrivons l'API qui, avec un haut niveau d'abstraction et de transparence, contrôle les opérations de gestion des données : cycle de vie, distribution, placement, réplication et la tolérance aux pannes. Notre environnement d'exécution repose sur une architecture distribuée flexible, qui intègre de façon modulaire des composants P2P tels qu'une DHT pour implémenter un catalogue distribué de données et BitTorrent pour la distribution des données. Dans ce rapport de recherche nous effectuons une évaluation de performance de ces composants, et nous évaluons la scalabilité et l'efficacité de l'environnement lors de l'exécution de l'application BLAST.

Mots-clés : Gestion des données, Grille de PC

1 Introduction

Enabling Data Grids is one of the fundamental efforts of the computational science community as emphasized by projects such as EGEE [14] and PPDG [32]. This effort is pushed by the new requirements of e-Science. That is, large communities of researchers collaborate to extract knowledge and information from huge amounts of scientific data. This has lead to the emergence of a new class of applications, called *data-intensive* applications which require secure and coordinated access to large datasets, wide-area transfers and broad distribution of TeraBytes of data while keeping track of multiple data replicas. The Data Grid aims at providing such an infrastructure and services to enable data-intensive applications.

Our project, BitDew[1], targets a specific class of Grids called Desktop Grids. Desktop Grids use computing, network and storage resources of idle desktop PCs distributed over multiple LANs or the Internet. Today, this type of computing platform forms one of the the largest distributed computing systems, and currently provides scientists with tens of TeraFLOPS from hundreds of thousands of hosts. Despite the attractiveness of this platform, little work has been done to support data-intensive applications in this context of massively distributed, volatile, heterogeneous, and network-limited resources. Most Desktop Grid systems, like BOINC [4], XtremWeb [15], Condor [28] and OurGrid [6] rely on a centralized architecture for indexing and distributing the data, and thus potentially face issues with scalability and fault tolerance.

However, we believe that the basic blocks for building BitDew can be found in P2P systems. Researchers of DHT's (Distributed Hash Tables) [38, 30, 34] and collaborative data distribution [12, 20, 16], storage over volatile resources [1, 11, 40] and wide-area network storage [9, 27] offer various tools that could be of interest for Data Grids. To build Data Grids from and to utilize them effectively, one needs to bring together these components into a comprehensive framework. BitDew suits this purpose by providing an environment for data management and distribution in Desktop Grids.

BitDew is a subsystem which could be easily integrated into other Desktop Grid systems. It offers programmers (or an automated agent that works on behalf of the user) a simple API for creating, accessing, storing and moving data with ease, even on highly dynamic and volatile environments.

BitDew leverages the use of metadata, a technics widely used in Data Grid [23], but in more directive style. We define 5 different types of metadata : *i*) *replication* indicates how many occurrences of data should be available at the same time in the system, *ii*) *fault tolerance* controls the resilience of data in presence of machine crash, *iii*) *lifetime* is a duration, absolute or relative to the existence of other data, which indicates when a datum is obsolete, *iv*) *affinity* drives movement of data according to dependency rules, *v*) *transfer protocol* gives the runtime environment hints about the file transfer protocol appropriate to distribute the data. Programmers tag each data with these simple attributes, and simply let the BitDew runtime environment manage operations of data creation, deletion, movement, replication, as well as fault tolerance.

The BitDew runtime environment is a flexible environment implementing the APIs. It relies either on centralized or and distributed protocols for indexing, storage and transfers providing reliability, scalability and high performance. In this paper, we present the architecture of the prototype, and we describe in depth the various mechanisms used. We also provide detailed quantitative evaluation of the runtime environment on two environments : the GRID5000 experimental Grid platform, and DSL-Lab, an experimental platform over broadband ADSL.

Through a set of micro-benchmarks, we measure the costs and benefits, components by components, of the underlying infrastructures. We run communication benchmark in order to evaluate the overhead of the BitDew protocol when transferring files and we assess fault-tolerant

[1] BitDew can be found at `http://www.bitdew.net` under GPL license

capabilities. And finally we show how to program a master/worker application with BitDew and we evaluate its performance in a real world Grid deployment.

The rest of the paper is organized as follows. Section 2 presents the background of our researches. In Section 3, we present the API and the runtime environment of BitDew. Then in Section 4, we conduct performance evaluation of our prototype, and Section 5 presents a master/worker application. Finally we present related work in Section 6 and we conclude the paper in Section 7.

2 Background

In this section we overview Desktop Grids characteristics and data-intensive application requirements. Following this analysis, we give the required features of BitDew.

2.1 Desktop Grids Characteristics

Desktop Grids are composed of a large set of personal computers that belong both to institutions, for instance an enterprise or a university, and to individuals. In the former case, these home PCs are volunteered by participants who donate a part of their computing capacities to some public projects. However several key characteristics differentiate DG resources from traditional Grid resources : *i*) performance; mainstream PCs have no reliable storage and potentially poor communication links, *ii*) volatility; PCs can join and leave the network at any time and appear with several identities, *iii*) shared between their users and the desktop grid applications, *iv*) scattering across administrative domains with a wide variety of security mechanisms ranging from personal routers/firewalls to large-scale PKI infrastructures.

Because of these constraints, even the simplest data administration tasks, are difficult to achieve on a Desktop Grid. For instance, to deploy a new application on a cluster, it is sufficient to copy the binary file on a network file server shared by the cluster nodes. After a computation, cluster users usually clean the storage space on the cluster nodes simply by logging remotely to each of the compute nodes and by deleting recursively the temporary files or directories created by the application. By contrast, none of the existing Desktop Grids systems allows such tasks to be performed because : *i*) a shared file system would be troublesome to setup because of hosts connectivity and volatility and volunteers churn, and *ii*) remote access to participant's local file system is forbidden in order to protect volunteer's security and privacy.

2.2 Requirements to Enable Data-Intensive Application on Desktop Grids

Currently, Desktop Grids are mostly limited to embarrassingly parallel applications with few data dependencies. In order to broaden the use of Desktop Grids we examine several challenging applications and outline their needs in terms of data management. From this survey, we will deduce the features expected from BitDew.

Parameter-sweep applications composed of a large set of independent tasks sharing large data are the first class of applications which can benefit from BitDew. Large data movement across wide-area networks can be costly in terms of performance because bandwidth across the Internet is often limited, variable and unpredictable. Caching data on local workstation storage [21, 31, 40] with adequate scheduling strategies [35, 41] to minimize data transfers can improve overall application execution performance.

Moreover, the work in [22] showed that data-intensive applications in high energy physics tend to access data in groups of files called "filecules". For these types of applications, replication of

groups of files over a large set of resources is essential to achieve good performance. If data are replicated and cached on local storage of computing resources, one should provide transparent fault tolerance operation on data.

In a previous work [41], we have shown that using a collaborative data distribution protocol BitTorrent over FTP can improve execution time of parameter sweep applications. In contrast, we have also observed that the BitTorrent protocol suffers a higher overhead compared to FTP when transferring small files. Thus, one must be allowed to select the correct distribution protocol according to the size of the file and level of "sharability" of data among the task inputs.

The high-level of collaboration in e-Science communities induces the emergence of complex workflow applications [7]. For instance in the case of multi-stage simulations, data can be both results of computation and input parameters for other simulations. To build execution environment for applications with task and data dependencies, it requires a system that can move data from one node to another node according to dependency rules. A key requirement of the system is to efficiently publish, search and localize data. Distributed data structures such as the DHT proposed by the P2P community might fulfill this role by providing distributed index and query mechanisms.

Long-running applications are challenging due to the volatility of executing nodes. To achieve application execution, it requires local or remote checkpoints to avoid losing the intermediate computational state when a failure occurs. In the context of Desktop Grid, these application have to cope with replication and sabotage. An idea proposed in [25] is to compute a signature of checkpoint images and use signature comparison to eliminate diverging execution. Thus, indexing data with their checksum as is commonly done by DHT and P2P software permits basic sabotage tolerance even without retrieving the data.

2.3 BitDew Features

Previously, we profiled several classes of "data-bound" applications and we now give the expected features to efficiently manage data on Desktop Grids.

- Fault tolerance: the architecture should handle frequent faults of volatile nodes which can leave and join the network at any time.
- Scalability: the architecture should provide a decentralized approach when a centralized approach might induce a performance bottleneck.
- Replication: to achieve application performance, the system should allow data replication and distributed data cache.
- Efficiency: the architecture should provide adequate and user optional protocols for both high throughput data distribution and low latency data access.
- Reliability: interrupted transfers should be automatically resumed or canceled according to the programmer's preference.
- Simplicity: the programming model should offer a simple view of the system, unifying the data space as a whole.
- Transparency: faults and data location should be kept hidden from the programmer.

We have designed our system to address each of those design goals in mind. In this paper, we give evidence for the system's manageability, scalability, efficiency, and simplicity by performing a set of micro-benchmarks and by deploying a real scientific application.

Security issues are not specifically addressed in this paper, because existing solutions in literature could be applied to our prototype. In fact, a relevant analysis of security for Data Desktop Grid has been done by [29], where is also proposed a protocol to maintain data confidentiality given that data will be stored on untrusted nodes. Authors use methods known as Information Dispersal Algorithms (ISA) which allows one to split a file into pieces so that by carefully dispersing the pieces, there is no method for a single node to reconstruct the data. Another well known issue is protection against data tampering, which has been addressed in the literature under the generic name of "results certification" [36]. It is a set of methods (spot-checking, voting, credibility) to verify that results computed by volunteers are not erroneous (for example, because of intentional modifications by malicious participants). Result certification is also mandatory to protect the DG storage space. In public DG system, the information to upload task result could be exploited by malicious participants. We assume that a result checker such as the *assimilator* of BOINC exists to sort between correct and incorrect results. Furthermore, in a Volunteer Computing setup, BitDew should be run in a confined environment, such as a sandbox [24] to protect volunteer's privacy and security. For future work, we will show the system's ability to deal with security issues.

Also, in this paper we consider data as immutable. However one could leverage the built-in distributed data catalog to provide data consistency. For example, authors in [8] have proposed an *entry consistency* protocol for a P2P system with mutable data.

3 BitDew Architecture

In this section we detail the BitDew architecture: programing interface, runtime environment and implementation.

3.1 Overview

Figure 1 illustrates the layered BitDew software architecture upon which distributed application can be developed. The architecture follows strict design rules : each layer is composed of independent components; components of the same layer do not interact directly and a component of an upper layer only interacts with components of the immediate lower layer.

The uppermost level, the APIs level, offers the programmer a simplified view of the system, allowing him to create data and manage their repartition and distribution over the network of nodes. The programming model is similar to the Tuple Space model pioneered by Gelernter [17] in the Linda programming system; it aggregates the storage resources and virtualizes it as a unique space where data are stored. The BitDew APIs provide functions to create a slot in this space and to put and get files between the local storage and the data space. Additional metadata, called data attributes, are managed by the ActiveData API and help to control the behavior of the data in the system, named *replication*, *fault tolerance*, *placement*, *lifetime* and *distribution*. It also provides programmers event-driven programming facilities to react to the main data life-cycle events: creation, copy and deletion. Finally the TransferManager API offers a non-blocking interface to concurrent file transfers, allowing users to probe for transfer, to wait for transfer completion, to create barriers and to tune the level of transfers concurrency.

The intermediate level is the service layer which implements the API : data storage and transfers, replicas and volatility management. The architecture follows a classical approach commonly found in Desktop Grids: it divides the world in two sets of nodes : stable nodes and volatile nodes. Stable nodes run various independent services which compose the runtime environment: Data Repository (DR), Data Catalog (DC), Data Transfer (DT) and Data Scheduler (DS). We call these nodes the *service hosts*. The fault model we consider for service node is the transient

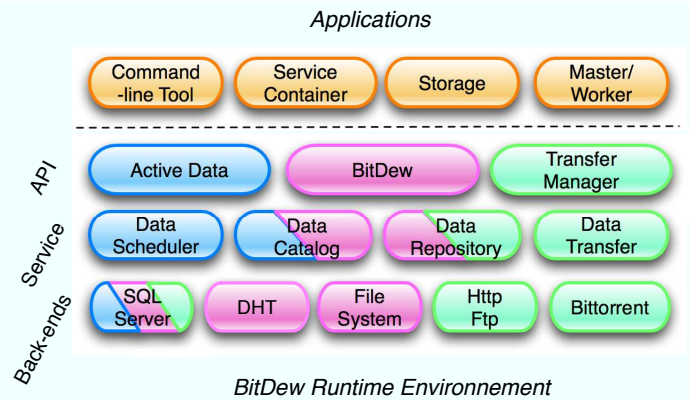


Figure 1: The BitDew software architecture. The upper part of the figure shows distributed applications designed using BitDew. Lower parts are the three layers composing the BitDew run-time environment : the API layer, the service layer and the back-ends layer. Colors illustrate how various components of different layers combine together. For instance, the TransferManager API uses two services : Data Repository and Data Transfer, which in turn use three back-ends : SQL Server, Http/FTP protocol and BitTorrent protocol.

fault where a host is assumed to be restarted by administrators after a failure. Volatile nodes can either ask for storage resources (we call them *client hosts*) or offer their local storage (they are called *reservoir hosts*). Classically in DG, we use a *pull model*, where volatile nodes periodically contact service nodes to obtain data and synchronize their local data cache. Failures of volatile nodes is detected by the mean of timeout on periodical heartbeats. Usually, programmers will not use directly the various D* services; instead they will use the API which in turn hides the complexity of internal protocols.

The lowermost level is composed of a suite of back ends. The Bitdew runtime environment delegates a large number of operations to third party components : 1) Meta-data information are serialized using a traditional SQL database, 2) data transfers are realized out-of-band by specialized file transfer protocols and 3) publish and look-up of data replica are enabled by the means of DHT protocols. One feature of the system is that all of these components can be replaced and plugged-in by the users, allowing them to select the most suitable subsystem according to their own criteria like performance, reliability and scalability.

3.2 Data Attributes

The key feature of BitDew is to leverage on metadata, called here Data Attributes. Though, metadata are not only used to index, categorize, and search data, as in other Data Grids System, but also to control dynamically repartition and distribution of data onto the storage nodes. Thus, complexity of Desktop Grids systems is hidden to the programmers who is freed from managing data location, host failure and explicit host to host data movement.

Instead, the runtime environment interprets data attributes and schedule data to host in order to satisfy the constraints expressed by the attributes. The following is the list of attributes a user can set :

`replica`: gives the number of instances of a datum that should exist at the same time in the system. The runtime environment will schedule new data transfers to hosts if the number of owners is less than the number of replica. As nodes are volatile there might be more replicas

in the system than what is specified by this attribute because the runtime environment will not issue orders for data deletion.

`fault tolerance`: indicates what the runtime environment should do if a reservoir host holding a data replica fails. If the data is resilient to host crash (the *fault tolerance* attribute is set), the data will be scheduled to another node so that the number of available replicas is kept at least equal to the value of the *replica* attribute over time. If the data are not marked as fault tolerance, the replica will be unavailable as long as the host is down.

`lifetime`: defines data lifetime, that is precise time after which a datum can be safely deleted by the storage host. The lifetime can be either absolute or relative to the existence of the other data. In the latter case, a datum is obsolete when the reference data disappear.

`affinity`: defines the placement dependency between data. It indicates that data should be scheduled on a node where other data have been previously sent. The *affinity* attribute is stronger than *replica*. That is, if data A is r_a *replica* and is distributed to r_n nodes, then if a datum B has a placement dependency over A, it will be replicated over r_n nodes whatever the value of r_b or r_a is.

`transfer protocol`: specifies to the runtime environment the preferred transfer protocol to distribute the data. Users are more knowledgeable to select the most appropriate protocol according to their own criteria like the size of data and the number of nodes to distribute these data. For example a large file distributed to a large number of nodes would be preferably distributed using collaborative distribution protocol such as BitTorrent or Avalanche [41].

3.3 Application Programing Interfaces

We will now give a brief overview of the three main programming interfaces which allows the manipulation of the data in the storage space (BitDew), the scheduling and programming (ActiveData) and the control of file transfer (TransferManager).

To illustrate how APIs are put into action, we'll walk through a toy program which realizes a network file update and works as follows : one master node, the Updater, copies a file to each node in the network, the Updatee, and maintains the list of nodes which have received the file updated. The Listing 1 presents the code of the Updater[2], implemented using Java.

```
1  public class Updater {
2    //list of hosts updated
3    Vector updatees = new Vector();
4
5    public Updater(String host, int port, boolean master) {
6
7      //initialize  communications and APIs
8      Vector comms=ComWorld.getMultipleComms(host, "RMI", port, "dc","dr","dt","ds");
9      BitDew bitdew = new BitDew(comms);
10     ActiveData activeData = new ActiveData(comms);
11     TransferManager tranferManager = new TransferManager(comms);
12
13     if (master) {
14       //this part of the code will only run on the master
15       File fic = new File("/path/to/big_data_to_update");
16       Data data = bitdew.createData(fic);
17       bitdew.put(data, fic); //copy file to the data space
18       //attribute specifies that the data should be send to every node using the
             BitTorrent protocol, and has a lifetime of 30 days
19       Attribute attr = bitdew.createAttribute("attr update = { replicat=-1, oob=
             bittorrent, abstime=43200}");
20       activeData.schedule(data, attr); //schedule data
21       activeData.addCallback(new UpdaterHandler()); //install data life-cycle event
             handler
22     } else {
```

[2]For sake of clarity we have simplified some elements of syntax, however the full source code is available in the BitDew source package.

```
23        //this part of the code will be executed by the other nodes,
24        activeData.addCallback(new UpdateeHandler()); //install data life-cycle event
              handler
25      }
26    }
27  }
```

Listing 1: The Updater example.

```
1   public class UpdaterHandler extends ActiveDataEventHandler {
2     public void onDataCopyEvent(Data data, Attribute attr) {
3       if (attr.getname().equals("host")) {
4         updatees.add(data.getname());
5       }
6     }
7   }
8   public class UpdateeHandler extends ActiveDataEventHandler {
9     public void onDataCopyEvent(Data data, Attribute attr) {
10      if (attr.getname().equals("update")) {
11        //copy file from the data space
12        bitdew.get(data, new File("/path_to_data/to/update/"));
13        transferManager.waitFor(data); //block until the download is complete
14        Data collectorData = bitdew.searchData("collector");
15        //sends back to the updater the name of the host
16        activeData.schedule( bitdew.createData(getHostByName()),
17                             activeData.createAttribute("attr host = { affinity = " +
                                 collector.getuid() + "}");
18
19      }
20    }
21    public void onDataDeleteEvent(Data data, Attribute attr) {
22      //delete the corresponding file
23      if (attr.getname().equals("update"))
24        (new File("/path_to_data/to/update/")).delete();
25    }
26  }
```

Listing 2: Data life-cycle events handlers installed in the Updater example.

Before a user can start data manipulation, he firstly has to attach the host to the rest of the distributed system. For this purpose, a special class called CommWorld will set up the communication to the remote hosts executing the different runtime services (DC, DR, DT, DS). The result of this operation is a set of communication interfaces to services passed as parameters to the APIs constructor. After this step, the user does never have to explicitly communicate with service hosts. Instead the complexity of the protocol is kept hidden unless programmer wishes to perform specialized operations.

In the Updater example we have assumed that all D* services are executed on a single centralized node. However, in real world, it might be necessary to run several service nodes in order to enhance reliability and scalability or to adjust with an existing infrastructure where data are spread over multiple data servers. The BitDew approach to cope with distributed setup is to instantiate several APIs, each one configured with its own vector of interfaces to the D* pool.

Data creation consists of the creation of a slot in the storage space. This slot will be used to put and get content, usually a file, to and from that slot. A data object contains data meta-information: `name` is the character string label, `checksum` is an MD5 signature of the file, `size` is the file length, `flags` is a OR-combination of flags indicating whether the file is compressed, executable, architecture dependent, etc... The BitDew API provides methods which compute these meta-information when creating a datum from a file. Data objects are both locally stored within a database and remotely on the Data Catalog service. Consequently data deletion implies both local and remote deletion.

Once slots are created in the storage space, users can copy files to and from the slots using dedicated functions. However users have several ways of triggering data movement either explic-

itly or implicitly. Explicit transfers are performed via `put` and `get` methods, which copy data to the storage space slots. Implicit transfers occur as a result of affinity placement, fault tolerance or replication and are resolved dynamically by the Data Scheduling service.

This is precisely the role of the ActiveData API to manage data attributes and interface with the DS, which is achieved by the following methods: *i*) `schedule` associates a datum to an attribute and order the DS to schedule this data according to the scheduling heuristic presented in paragraph 3.4.3; *ii*) `pin` which, in addition, indicates the DS that a datum is owned by a specific node. Besides, ActiveData allows programmer to install *handlers*, those are codes executed when some events occur during data life cycle : creation, copy and deletion.

The API provides functions to publish and search data over the entire network. Data are automatically published by hosts by means of a DHT. Moreover, the API also gives the programmer the possibility to publish any key/value pairs so that the DHT can be used for other generic purpose.

3.4 Runtime Environment

We review now the various capabilities provided by the BitDew service layer of the runtime environment.

3.4.1 Indexing and locating data

The data's meta-information are stored both locally on the client/reservoir node and persistently on the Data Catalog (DC) service node.

For each data published in the DC, one or several Locators are defined. A Locator object is similar to URL, it gives the correct information to remotely access the data: file identification on the remote file system (this could be a path, file name, or hash key) and information to set up the file transfer service (for instance protocol, login and password).

However, information concerning data replica, that is data owned by volatile reservoir nodes, are not centrally managed by DC but instead by a Distributed Data Catalog (DDC) implemented on top of a DHT. For each data creation or data transfer to a volatile node, a new pair data identifier/host identifier is inserted in the DHT.

The rationale behind this design is the following : as the system grows, information in the DDC will grow larger than information in the DC. Thus, by centralizing information in the DC, we shorten the critical path to access to a permanent copy of data. On the other hand, distributing data replica information ensures scalability of the system for two reasons : *i*) DHTs are inherently fault-tolerant; thus it frees the DC to implement fault detection mechanisms and *ii*) the search requests are distributed evenly among the hosts, ensuring effective load-balancing.

3.4.2 Storing and Transferring data

The strength of the framework depends on its ability to adapt to various environments in term of protocols (client/server vs. P2P), of storage (local vs. wide area), of security level (Internet vs. Grid). To provide more flexibility, we have separated data access in two different services : Data Repository (DR) is an interface to data storage with remote access and Data Transfer service (DT) is responsible for reliable out-of-band file transfer.

The Data Repository service has two responsibilities, namely to interface with persistent storage and to provide remote access to data. DR acts as a wrapper around legacy file server or file system, such as Grid Ftp server or local file system. In a Grid context, DR is the solution to map BitDew to an existing infrastructure.

BitDew does not propose new protocol to transfer data from node to node, instead, data are moved by out-of-band transfer.

The role of Data Transfer (DT) is to launch out-of-band transfers and ensure their reliability. If several transfers of the same data occur in parallel (for a broadcast, for example), it is the responsibility of the file transfer protocol to leverage this concurrency. This is finally what happens when collaborative file transfer are being used, but this is transparent to the system.

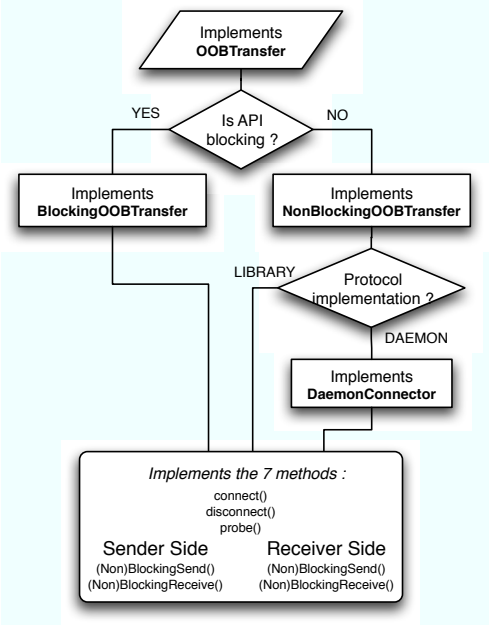


Figure 2: Flowchart to implement out-of-band transfer. To plug-in a new file transfer protocol, a programmer has to implement the OOBTransfer interface. Programmer chooses the blocking (resp. non blocking) interface if the method protocol are blocking (resp. non blocking). DaemonConnector is a helper interface for protocol provided as daemon instead of library. Finally, it is sufficient to write 7 methods : to open and close connection, to probe the end of transfer and to send and to receive file from the sender and the receiver sides.

Transfers are always initiated by a reservoir or client host to DT, which manages transfer reliability, resumes faulty transfers, reports on bandwidth utilization and ensures data integrity. Transfer management relies on a principle called *receiver driven transfer*. The sender of a datum will periodically pool the receiver to check the state of the transfer, because receiver can verify the size and the integrity, using the MD5 signature, of the received data. This mechanism, while simple, ensures support for a broad range of different data transfer protocols.

Figure 2 presents the framework to integrate existing or new file transfer protocols, client/server or P2P, with blocking or non-blocking communication, whose implementations are provided as libraries or as daemons. Note, that the former is very popular for P2P protocol where a daemon runs forever in the background and a GUI issues search and download order. So far, we support HTTP, FTP and BitTorrent, both as a library with Azureus[3] and as a daemon with BTPD and we tested the framework with SMTP, POP and edonkey.

[3]The Azureus BitTorrent Client: `http://azureus.sourceforge.net`

3.4.3 Scheduling data

Implicit data movement on the grid is determined by the Data Scheduling service (DS). The role of the DS service is to generate transfer orders according to the hosts' activity and data attributes.

Algorithm 1 presents the pseudo-code of the scheduling algorithm. Periodically, reservoir hosts contact the data scheduler with a description of the set of data hold in their local cache Δ_k. The data scheduler scans the list of data to schedule Θ, and according to data attributes, makes a scheduling decision which consists of a new set of data Ψ_k returned to the reservoir host. Reservoir host can safely delete obsolete data ($\Delta_k \setminus \Psi_k$), keep the cached data validated by the DS ($\Delta_k \cap \Psi_k$) and download newly assigned data ($\Psi_k \setminus \Delta_k$).

First step of the scheduling algorithm determines which data should be kept in reservoir cache. It is defined as the set of data both present in the reservoir cache Δ_k and in the DS data set Θ and whose lifetime, either absolute or relative, has not expired. In the second step, new data are scheduled to the reservoir host by filling Ψ_k. Two conditions trigger attribution of data to reservoir host. The first one is the dependency relations: if the reservoir cache Δ_k contains data which has a dependency relation with a datum missing from Δ_k, then this datum is added to Ψ_k. The second one is the replica attribute: if the number of active owners $\Omega(D_i^k)$ is less than the the value of *replica* attribute then this data is added to Ψ_k. The scheduling algorithm stops when the set of new data to download ($\Psi_k \setminus \Delta_k$) has reached a threshold.

Finally the Data Scheduler implements support for fault tolerance. For each data is maintained a list of *active* owners updated at each synchronization of reservoir hosts. Faults of owners are detected through timeout on the last synchronization. If a datum has the *fault tolerance* attribute, the faulty owner is removed from the list of active owners, otherwise the list is kept unchanged. As a consequence, the data will be scheduled again to a new host.

For now the scheduling has been designed to fulfill metadata specification without focus on performance. In future, we will investigate specific data repartition policies, cache management strategies and coordinated tasks scheduling heuristics.

3.5 Implementation

We have used the Java programming environment to prototype BitDew with Java RMI for the communication and Apache Jakarta Java JDO (`http://jakarta.apache.org`) with JPOX (`http://jpox.org`) which permits transparent objects persistence in a relational database. Each object is referenced with a unique identifier AUID, a variant of the DCE UID.

We have used two different database back-ends. MySQL (`http://mysql.com`), is a well-known open-source database, and HsqlDB (`http://hsqldb.org`) is an embedded SQL database engine written entirely in Java. Jakarta Commons-DBCP provides database connection pooling services, which avoids opening new connection for every database transaction. We have implemented data transfer with the client/server FTP protocol, respectively the client provided by the apache common-net package and the ProFTPD FTP server (`http://www.proftpd.org/`) and with the BTPD BitTorrent client (`http://www.murmeldjur.se/btpd/`). The distributed data catalog uses the DKS DHT [2].

Overall our first version of the software, while implementing most of the features described in the paper, includes less than 17000 lines of code. Initial release is available at `http://bitdew.net` under GNU GPL.

Algorithm 1 SCHEDULING ALGORITHM

Require: $\Theta = \{D_1, \cdots, D_m\}$ the set of data managed by the scheduler
Require: $\Delta_k = \{D_1^k, \cdots, D_n^k\}$ the data cache managed by the reservoir host k
Require: $\Omega(D_i^k) = \{k, \cdots\}$ the set of reservoir host owning data D_i
Ensure: $\Psi_k = \{D_1^k, \cdots, D_o^k\}$ the new dataset managed by the reservoir host k

$\Psi_k \leftarrow \emptyset$
{Step 1 : Remove obsolete data from cache}
for all $D_i^k \in \Delta_k$ **do**
  if $((D_i^k \in \Theta) \wedge (D_i^k.lifetime.absolute > now()) \wedge (D_i^k.lifetime.relative \in \Theta))$ **then**
    $\Psi_k \leftarrow \Psi_k \cup \{D_i^k\}$
    if $((D_i^k.faultTolerant == true)$ **then**
      update $\Omega(D_i^k)$
    end if
  end if
end for
{Step 2 : Add new data to the cache}
for all $D_j \in (\Theta \setminus \Delta_k)$ **do**
  {Resolve affinity dependence}
  for all $D_i^k \in \Delta_k$ **do**
    if $((D_j.affinity == D_i^k) \wedge (D_j \notin \Delta_k))$ **then**
      $\Psi_k \leftarrow \Psi_k \cup \{D_j\}$
      $\Omega(D_j) \leftarrow \Omega(D_j) \cup \{k\}$
    end if
  end for
  {Schedule replica}
  if $((D_j.replica == -1) \vee (D_j.replica < |\Omega(D_j)|)$ **then**
    $\Psi_k \leftarrow \Psi_k \cup \{D_j\}$
    $\Omega(D_j) \leftarrow \Omega(D_j) \cup \{k\}$
  end if
  if $(|\Psi_k \setminus \Delta_k| \geq MaxDataSchedule)$ **then**
    break
  end if
end for

return Ψ_k

4 Performance Evaluation

In this section, we present performance evaluation of the BitDew runtime environment. The experiments evaluate the efficiency and scalability the core data operation, the data transfer service, and the data distributed catalog. We also report on a Master/Worker bioinformatics application executed over 400 nodes in a Grid setup.

4.1 Experiments Setup

Experiments were conducted in 3 different testbeds. To measure precisely performances of basic data operations within an environment where experimental conditions are reproducible, we run

Cluster	Cluster Type	Location	#CPUs	CPU Type	Frequency	Memory
gdx	IBM eS-erver 326m	Orsay	312	AMD Opteron 246/250	2.0G/2.4G	2G
grelon	HP Pro-Liant DL140G3	Nancy	120	Intel Xeon 5110	1.6G	2G
grillon	HP Pro-Liant DL145G2	Nancy	47	AMD Opteron 246	2.0G	2G
sagittaire	Sun Fire V20z	Lyon	65	AMD Opteron 250	2.4G	2G

Table 1: Hardware configuration of the Grid testbed which consists in 4 Grid5000 clusters.

micro-benchmarks on the *Grid Explorer (GdX)* cluster which is part of the Grid5000 infrastructure [10].

To analyze BitDew behavior on a platform close to Internet Desktop Grids, we conducted experiments with the DSL-Lab platform [4]. DSL-Lab is an experimental platform, consisting of a set of PCs connected to broadband Internet. DSL-lab nodes are hosted by regular Internet users, most of the time protected behind firewall and sharing the Internet bandwidth with users' applications. DSL-lab offers extremely realistic networking experimental conditions, since it runs experiments on the exact same platform than the one used by most of the desktop Grids applications. Technically, it's a set of 12 Mini-ITX nodes, Pentium-M 1Ghz, 512MB SDRam, with 2GB Flash storage.

The third testbed, used for scalability tests, is a part of Grid5000: 4 clusters (including GdX) of 3 different sites in France. Note that, due to the long running time of our experiments, we were not able to reserve the 2000 nodes of Grid5K. The hardware configuration is shown in table 1. All of our computing nodes are installed Debian GNU/Linux 4.0 as their operating systems.

As for software, we used the latest version of the software package available at the time of the experiment (July to December 2007). Java SDK 1.5 for 64 bits was the Java version we used for the experiments. BLAST used is NCBI BLAST 2.2.14 for Linux 64 bits.

4.2 Core data operation

We first report on the performance of basic data operations according to the database and communication components.

The benchmark consists of a client running a loop which continuously creates data slot in the storage space, and a server running the Data Catalog service. This basic operation implies an object creation on the client, a network communication from the client to server (payload is only few kilobytes) and an a write access to the database to serialize the object. Every ten seconds the average number of data creations per second (dc/sec) is reported. Table 2 shows the peak performance in thousands of dc/sec. This benchmark is representative of most of the data operations executed by the different D* services when the runtime environment manages data transfers, fault tolerance and replication.

The experimental configuration is as follows: with the *local* experiment, a simple function call replaces the client/server communication, with *RMI local* the client and server are hosted on the same machine and a RMI communication takes place between them; with *RMI remote* client and server are located in two different machines. We have used two different database engines

[4]DSL-lab: `http://dsllab.org`. Note that DSL-lab is currently an early prototype. At the time of the experiments, only 12 nodes were available.

	without DBCP		with DBCP	
	MySQL	HsqlDB	MySQL	HsqlDB
local	0.25	3.2	1.9	4.3
RMI local	0.21	2.0	1.5	2.8
RMI remote	0.22	1.7	1.3	2.1

Table 2: Performance evaluation of data slot creation (dc/sec) : the number is expressed as thousands of data creation per second.

	Min	Max	Sd	Mean
publish/DDC	100.71	121.56	3.18	108.75
publish/DC	2.20	22.9	5.05	7.02

Table 3: Performance evaluation of data publishing in the centralized and distributed data catalog : the number are expressed as pairs (dataID,hostID) create per second.

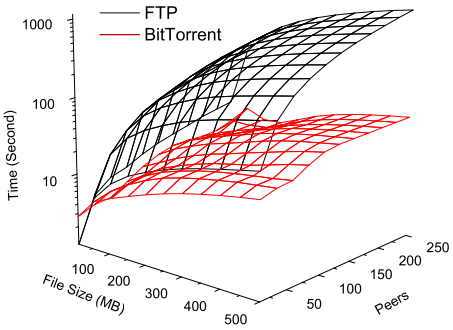
MySQL and HsqlDB; each database can be accessed either directly (*without DBCP*) or through (*with DBCP*) the use of the connection pooling service DBCP.

Preliminary results show that the latency for a remote data creation is about $500\mu sec$ when considering the HsqlDB database with DBCP. Using an embedded database provides a performance improvement of 61% over a more traditional MySQL approach, but comes at a price of manageability. Indeed, there exist numerous language bindings and third-party tools for MySQL, which can make the system more manageable. This lack of performance is certainly due to the networked client server protocol imposed by the MySQL Java JDBC connector. Moreover we can see that MySQL without the use of a connection pool is clearly a bottleneck when compared to the network overhead. However, a single service is able to handle more than 2 thousand data operations per second and we think that there is room for further performance improvements by using multi-threaded remote service invocations and by enclosing burst of data operations in a single remote invocations.

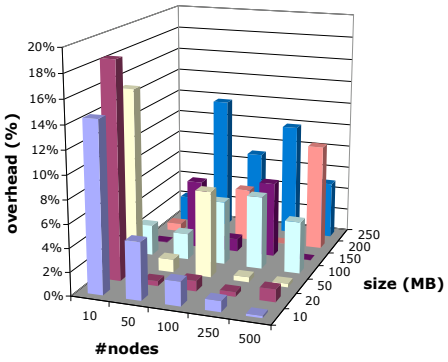
The next experiment evaluates the effectiveness of the Distributed Data Catalog (DDC) through the measurement of the DHT publish and search mechanism. The benchmark consists of an SPMD program running on 50 nodes. After a synchronization, each node will publish 500 pairs of dataID, hostID values, an operation which is executed every time a host completes a data transfer. We measure the time elapsed between the first data published to the last publication in the DHT, and we report in Table 3 the total time to create the data. One can observe that indexing 25000 data in the DDC takes about 108 sec. We conducted a similar experience with the DC and we found out that DDC is 15 time slower than DC. However, it is not fair to compare a DHT with a centralized approach as DC service do not implement fault-tolerance. Nevertheless, this result validates the design decision presented in paragraph 3.4.1 to rely both on a centralized Data Catalog service to provide fast access to data and a DHT for data replica hold by volatile nodes.

4.3 Data transfer

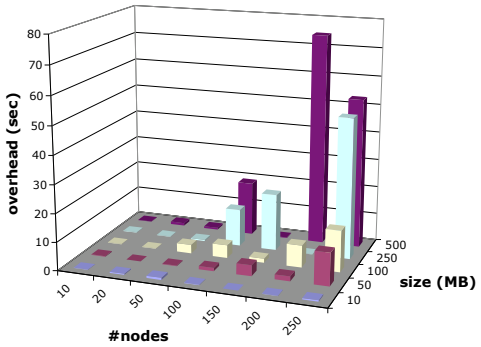
The following experiment evaluates the overhead of BitDew when transferring data, and Figure 3 presents the results. In a previous study [41], we have compared the BitTorrent and FTP protocols for computational Desktop Grids. Here BitDew issues and drives the data transfer, providing file transfer reliability. As a consequence, both the BitDew protocol and the file transfer



(a) BitDew performance with two file transfer protocols : BitTorrent and FTP.



(b) Overhead of BitDew with FTP against FTP alone in percentage of the communication time.



(c) Overhead of BitDew with FTP against FTP alone in seconds.

Figure 3: Evaluation of the BitDew overhead when transferring files.

protocol run together at the same time. As clusters have limited number of nodes and to provoke a greater impact of the BitDew protocol, we setup the experiment so that the D* services, the FTP server and the BitTorrent seeder run on the same node. To generate a maximum of BitDew traffic, we have configured the DT heartbeat to monitor transfer every 500 *ms* and the DS service to synchronize with the scheduler every second.

The benchmark works as follows: BitDew replicates data, whose size varies from 10 to 500MB, to a set of nodes whose size ranges from 10 to 250. Two file transfer protocols are measured: FTP and BitTorrent.

Figure 3a presents the completion time in seconds for the file distribution, that is the time between the beginning of the file replication and the last node ending the file transfer, averaged over 30 experiments. The first result shows that BitTorrent clearly outperforms FTP when the file size is greater than 20MB and when the number of nodes is greater than 10, providing more scalability when the number of nodes is large.

As expected measurement of BitDew running BitTorrent and FTP are similar to our previous study where FTP and BitTorrent alone were compared. To investigate precisely the impact of the

out-of-band transfer management of BitDew on file transfer performances, we choose to compare the performances of file transfer performed by the FTP protocol alone against BitDew and FTP.

We choose not to evaluate the BitDew overhead over BitTorrent for the following reasons : *i*) we have shown in [41] that BitTorrent exhibits varying and unpredictable performances, which would have affected the quality of the measures, *ii*) we want to generate high load on the server and it is well known that BitTorrent can adapt to low bandwidth servers. Also, one can argue that there exists efficient and scalable FTP server such as GridFTP server. However, for this experiment, the main bottleneck is the server bandwidth, so the efficiency of the FTP server does not affect the measure of the BitDew overhead.

Figure 3b shows the overhead of BitDew with FTP against FTP alone in percentage of the file transfer time. One can observe that the impact is stronger on small files distributed to a small number of nodes. Before, launching a data transfer, BitDew has to communicate with the DC service to obtain a location of the data, to the DR service to obtain a description of the protocol, and finally to the DR service to register the data transfer. Obviously these steps add extra latency. Figure 3c shows the overhead of BitDew with FTP against FTP alone in seconds. BitDew overhead increases with the size of the file and with the number of nodes downloading the file, which shows that the overhead is mainly due to the bandwidth consumed by the Bitdew protocol. For instance, distribution of a 500 MB file to 250 nodes, in approximately 1000 sec, generates at least 500000 requests to the DT service. Still, our experimental settings are stressful compared to real world settings. For instance the BOINC client contacts the server only if any of the following occurs: when the user's specified period is reached, whose default is 8.5 hours or when a work unit deadline is approaching and the working unit is finished. By analyzing the logs of the BOINC based XtremLab[5] project, we have found that, after filtering out clients which contact the server less than two times in a 24 hours period, the mean time between two requests is 2.4 hours. Thus, even a very responsive periodical heartbeat of 1 minute generate an equivalent workload on the DT service if the number of clients exceeds 30000, implying a reasonable degradation on file transfer performance less than 10%.

4.4 Fault tolerance data operation

The next experiment aims at illustrating a fault tolerance scenario. We run the experiment in the DSL-Lab environment. The scenario consists of the following : we create a datum with the following attribute : replica = 5, fault tolerance = true and protocol = "ftp", which means that the runtime will constantly tries to maintain the number of data replica, even in the presence of host failures. At the beginning of the experiment, the data are owned by 5 nodes. Every 20 seconds, we simulate a machine crash by killing the BitDew process on one machine owning the data, and we simultaneously simulate a new host arrival by starting BitDew on an other node.

We measure the elapsed time between the arrival of the node and the schedule of data to this node, as well as the time to download the data. Figure 4 shows the Gantt chart of the experiment and the bandwidth obtained during the download. One can observe a waiting time of 3 seconds before the download starts, which is due to the failure detector. BitDew maintains a timeout to detect host failure, which is set to 3 times of the heartbeat period (here 1 second). We can also observe a great variation in the communication performance between the hosts. This can be explained by the difference of service quality between the various Internet Service Providers and by the fact that bandwidth consuming applications might be running at the same time.

[5] `http://xtremlab.lri.fr`

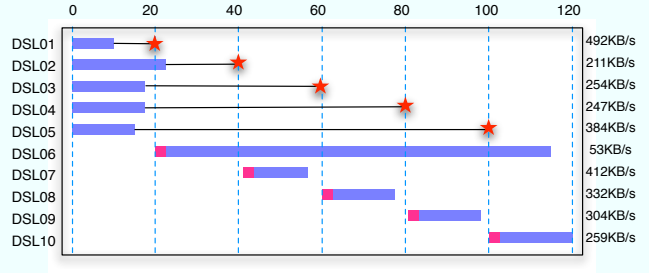


Figure 4: Evaluation of Bitdew in a faulty scenario. The Gantt chart presents the main events: red box the waiting time, blue box the downloading time and red star indicates a node crash. The rightmost part of the graph presents the bandwidth obtained during the file transfer.

5 Programing a Master/worker Application

In this section, we present an example of a Master/worker application developped with BitDew. The application is based on NCBI BLAST (Basic Local Alignment Search Tool), BLAST compares a query sequence with a database of sequences, and identifies library sequences that resemble the query sequence above a certain threshold. In our experiments, we used the *blastn* program that compares an amino acid query sequence against a protein sequence database. The input DNA sequences used were taken from the GeneBank database and the DNA databases were taken from the National Center for Biotechnology Information.

```
attribute Application = { replication = -1, protocol = "BitTorrent"
}
attribute Genebase = { protocol = "BitTorrent", lifetime = Collector, affinity =
    Sequence
}
attribute Sequence = { fault tolerance = true, protocol = "http", lifetime =
    Collector, replication = x
}
attribute Result = { protocol = "http", affinity = Collector, lifetime = Collector
}
Collector attribute {
}
```

Listing 3: Attributes Definition

In a classical MW application, tasks are created by the master and scheduled to the workers. Once a task is scheduled, the worker has to download the data needed before the task is executed. In contrast, the data-driven approach followed by BitDew implies that data are first scheduled to hosts. The programmer do not have to code explicitly the data movement from host to host, neither to manage fault tolerance. Programming the master or the worker consists in operating on data and attributes and reacting on data copy.

With this application, there exists three sets of data : the *Application* file, the *Genebase* file, and the *Sequence* files. The *Application* file is a binary executable files which has to be deployed on each node of the network. The replication attribute is set to -1, which is a special value which indicates that the data will be transferred to every node in the network. Although the size is small (4.45 MB), the file is highly shared, so it is worth setting the protocol to BitTorrent.

Each task depends on two data: the *Genebase* data is a compressed large archive (2.68 GB), and the *Sequence* which is the parameter of the task. The previous experience has shown that FTP is an appropriate protocol to distributes sequence which are small text files, unique

to each tasks, and BitTorrent is efficient to distribute *Genebase* shared by all the computing nodes. We define an affinity between a *Sequence* and a *Genebase*, which means that BitDew will automatically schedule transfer of *Genebase* data wherever a *Sequence* is present. This ensures that only nodes actually participating in the computation will download and store the *Genebase* files. Once the *Genebase*, the *Application* and at least one *Sequence* files are present in the worker's local cache, the worker can launch BLAST computation.

At the end of the computation, the tasks will produce a *Result* file which has to be retrieved by the master node. The master creates an empty *Collector* and `pin` this data. Each worker set an affinity attribute from *Result* data to the *Collector* data. By this way, results will get automatically transferred to the master node. At the end of the experiment, it is wise to delete data and to purge the workers' local cache. However, some files are large and should be kept persistent on workers' cache for the next execution. An elegant way is to set for every data a relative lifetime to the *Collector*. Once the user decides that he has finished his work, he can safely delete the *Collector*, which will obsolete remaining data.

Setting the *fault tolerance* attribute for the sequence data ensures that the tasks will be rescheduled if the host failed. The replication attribute of the sequence also affects the scheduling of data and tasks on the hosts. For instance, to implement the scheduling strategy presented in [26], one would simply keep a replication to 1 when the number of tasks is less than the number of hosts and dynamically increase the value of *replication* attribute when there are more hosts available than remaining tasks.

In Figure 5, we use the protocols FTP and BitTorrent respectively as transfer protocol. The x axis represents the number of workers used in our experiment, the y axis represents the total execution time: the time to broadcast the Genebase and Sequence to query, more the execution time of BLAST application for searching gene sequence in Genebase. When the number of workers is relatively small (10 and 20), the performance of BitTorrent is worse then FTP. But when the number of workers still increases from 50 to 250, the total time of FTP increases considerably, in contrast the line for BitTorrent is nearly flat.

For further experiments, we run our M/W application with BitDew on a part of Grid5000: 400 nodes of 4 clusters in 3 different sites (see 1). Breakdown of total execution time, in transfer time, unzip time, execution time is shown in Figure 6. The last 2 columns show the mean time for 4 clusters. Obviously, the transfer protocols used by BitDew play an important role over application performance because most of the time is spent for transferring data in network. In this case, using BitTorrent protocol to transfer data can gain almost a factor 10 of time for delivering computing data.

6 Related Work

The main efforts to enable data-intensive application on the Grid were initiated by projects that address the issues of data movement, data access and metadata management on the Grid. Representative example includes GridFTP [3] and GFarm [39]. GridFTP is a protocol to support secure, reliable and high performance wide-area data transfers on the Grid, by implementing striped transfers from multiple sources, parallel transfers, tuned TCP usage, failure detection and GSS authentication. The GFarm file system enable parallel processing of data intensive application. OGSA-DAI [7] is an effort to provides middleware for data access and data integration in the Grid. Metadata management is one of the key technique in Data Grids [37]. Metadata Catalog Service provides mechanism for storing and accessing descriptive metadata and allows users to query for data items based on desired attributes [13]. To provide high availability and scalability, metadata replica can be organized in a highly connected graph [23] or distributed in

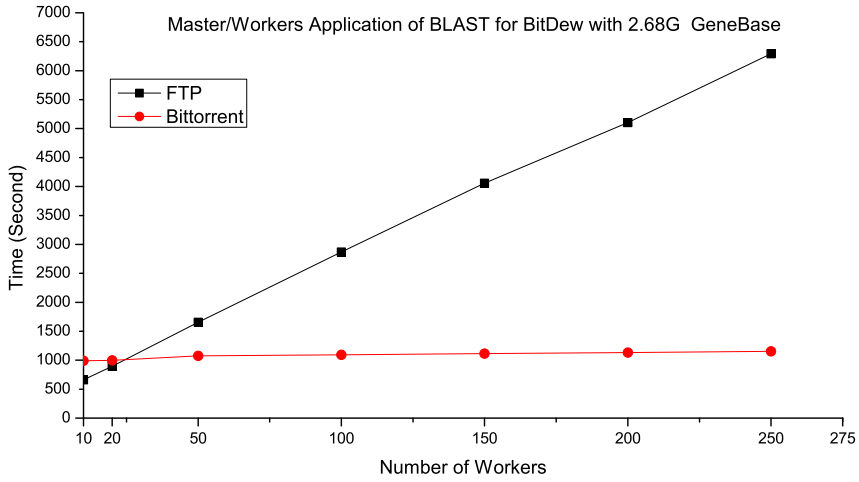


Figure 5: BitDew performances on a Master/Worker application. The two lines present the average total execution time in seconds for the BLAST application with a large Genebase of 2.68GB, executed on 10 to 250 nodes and with file transfer performed by FTP and BitTorrent

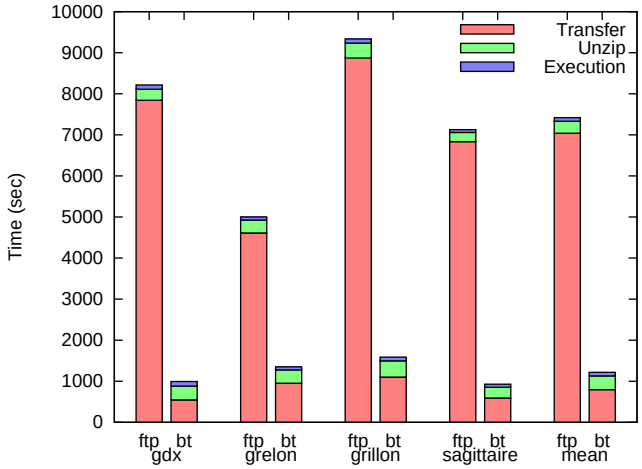


Figure 6: Breakdown of total execution time in time to transfer data, time to unzip data and Blast execution time by cluster. The rightmost values is the time average on the whole platform.

P2P network [33]. Beside descriptive information, metadata in BitDew also expresses *directive* information to drive the runtime environment. However Desktop Grid differs significantly from traditional Grid in terms of security, volatility and system size. Therefore specific mechanisms must be used to efficiently distribute large files and exploit local resource storage.

Several systems have been proposed to aggregate unused desktop storage of workstation within a LAN. Farsite [1] builds a virtual centralized file system over a set of untrusted desktop computers. It provides file reliability and availability through cryptography, replication and file

caching. Freeloader [40] fulfills similar goals but unifies data storage as a unique scratch/cache space for hosting immutable datasets and exploiting data locality. Nevertheless these projects offer a file system semantic for accessing data that is not precise enough to give users (or an agent that work on behalf of the user) control over data placement, replication and fault tolerance. We emphasize that BitDew provides abstractions and mechanisms for file distribution that work on a layer above these systems, but can nevertheless work in cooperation with them.

Oceanstore [27], IBP [9], and Eternity [5] aggregate a network of untrusted servers to provide global and persistent storage. IBP's strength relies on a comprehensive API to remotely store and move data from a set of well-defined servers. Using an IBP "storage in the network" service helps to build more efficient and reliable distributed application. Eternity and Oceanstore uses cryptographic and redundancy technologies to provide deep storage for archival purpose from unreliable and untrusted servers. In contrast with these projects, BitDew specifically adds wide-area resource storage scavenging, using P2P technics to face scalability and reliability issue.

JuxMem [8] is a large scale data sharing service for the Grid which relies on a P2P infrastructure to provide a DSM-like view of the data space. It is built over the JXTA middleware and features data replication, localization, fault tolerance, and a specialized consistency model. Bitdew differs in its approach to build a flexible and modular runtime environment which allows one to integrate a new protocol for data transfer, for DHT's, or to access remote storage. We believe that a key requirement for a Grid data sharing service is its ability to integrate with other Grid standards and utilities.

As BitDew's data distribution mechanism is built by using BitTorrent protocol, one could argue that performance over a wide-area could be severely limited by the transmission of redundant data blocks over bottleneck links. However, recent techniques involving network coding and file swarming have alleviated these concerns. The Avalanche protocol [19, 18], which effectively ensures that only unique and required datum is transmitted through these links, could be easily integrated within BitDew's framework.

7 Conclusion

We have presented BitDew, a programmable environment for large-scale data management and distribution that bridges the gap between Desktop Grid and P2P data sharing systems. We have detailed a programming model that provides developers with an abstraction for complex tasks associated with large scale data management, such as life cycle, transfer, placement, replication and fault tolerance. While maintaining a high level transparency, users still have the possibility to enhance and fine tune this platform by interacting directly with the runtime environment.

BitDew's runtime environment is an evolution of traditional Desktop Grid architectures which takes advantage of local storage of the resources. We have proposed a flexible and modular environment which relies on a core set of independent services to catalog, store, transfer and schedule data. This runtime environment has been designed to cope with a large number of volatile resources, and it has the following high-level features: reliable data transfers, automatic replication and transparent data placement. To achieve scalability, the BitDew architecture can apply P2P protocols when a centralized approach might induce a performance bottleneck. We have conducted a performance evaluation of a distributed data catalog implemented with the DKS DHT and evaluate BitDew's protocol overhead over the FTP file data transfer. We have presented a master/worker application with a performance evaluation which show the potential by relying on an efficient data distribution subsystem.

Desktop grids can integrate BitDew in three complementary ways. First, BitDew can serve as a multi-protocol file transfer library, featuring concurrent, reliable and P2P transfers. BitDew

would be a means of leveraging future enhancements of P2P protocols without modifying the Desktop Grid system. Second, a Desktop Grid could be enhanced with a distributed storage service based on BitDew, which would allow data management tasks (for example, lifetime and replication management) that are currently impossible to perform on existing DG systems. Finally, BitDew could facilitate the execution of data-intensive applications. This is the subject of our future works, which will aim at building a Data Desktop Grids system, providing the following features: sliced data, collective communication such as gather/ scatter, and other programming abstractions, such as support for distributed MapReduce operations.

Acknowledgment

Authors would to thank Derrick Kondo for his insightful comments and correction throughout our research and writing of this report.

Experiments presented in this paper 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 and RENATER and other contributing partners (see `https://www.grid5000.fr`).

Experiments presented in this paper were carried out using the DSL-Lab experimental testbed, an initiative supported by the French ANR JCJC program (see `https://www.dsllab.org`) under grant JC05_55975.

References

[1] A. Adya and all. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, 2002.

[2] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications. In *The 3rd International CGP2P Workshop*, Tokyo, 2003.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of Super Computing (SC05)*, 2005.

[4] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *proceedings of the 5th IEEE/ACM International GRID Workshop*, Pittsburgh, USA, 2004.

[5] R. Anderson. The Eternity Service. In *Proceedings of Pragocrypt '96*, 1996.

[6] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[7] M. Antonioletti and all. The Design and Implementation of Grid Database Services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17:357–376, February 2005.

[8] G. Antoniu, L. Bougé, and M. Jan. JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[9] A. Bassi, M. Beck, G. Fagg, T. Moore, J. S. Plank, M. Swany, and R. Wolski. The Internet BackPlane Protocol: A Study in Resource Sharing. In *Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002.

[10] R. Bolze and all. Grid5000: A Large Scale Highly Reconfigurable Experimental Grid Testbed. *International Journal on High Peerformance Computing and Applications*, 2006.

[11] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A Peer-to-Peer Enhancement for the Network File System. In *Proceeding of International Symposium on SuperComputing SC'04*, 2004.

[12] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, 2003.

[13] E. Deelman, G. Singh, M. P. Atkinson, A. Chervenak, N. P. C. Hong, C. Kesselman, S. Patil, L. Pearlman, and M.-H. Su. Grid-Based Metadata Services. In *SSDBM04*, Santorini, Greece, June 2004.

[14] Enabling Grids for E-Science in Europe.

[15] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing Platform. In *CCGRID'2001 Special Session Global Computing on Personal Devices*, 2001.

[16] Y. Fernandess and D. Malkhi. On Collaborative Content Distribution using Multi-Message Gossip. In *Proceeding of IEEE IPDPS*, Rhodes Island, 2006.

[17] D. Gelernter. Generative Communications in Linda. *ACM Transactions on Programming Languages and Systems*, 1985.

[18] C. Gkantsidis, J. Miller, and P. Rodriguez. Anatomy of a P2P Content Distribution System with Network Coding. In *IPTPS'06*, California, U.S.A., 2006.

[19] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive View of a Live Network Coding P2P System. In *ACM SIGCOMM/USENIX IMC'06*, Brazil, 2006.

[20] C. Gkantsidis and P. Rodriguez. Network Coding for Large Scale Content Distribution. In *Proceedings of IEEE/INFOCOM 2005*, Miami, USA, March 2005.

[21] A. Iamnitchi, S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *proceeding of 15th IEEE International Symposium on High Performance Distributed Computing HPDC 15*, Paris, 2006.

[22] A. Iamnitchi, S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *HPDC 2006*, Paris, 2006.

[23] H. Jin, M. Xiong, S. Wu, and D. Zou. Replica Based Distributed Metadata Management in Grid Environment. *Computational Science - Lecture Notes in Computer Science, Springer-Verlag*, 3994:1055–1062, 2006.

[24] K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *5th International Workshop in Grid Computing (Grid 2004)*, Pittsburgh, 2004.

[25] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello. Characterizing Result Errors in Internet Desktop Grids. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2007.

[26] D. Kondo, A. Chien, and H. Casanova. Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. In *ACM Conference on High Performance Computing and Networking (SC'04)*, Pittsburgh, 2004.

[27] J. Kubiatowicz and all. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[28] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 104–111, Washington, DC, 1988. IEEE Computer Society.

[29] J. Luna, M. Flouris, M. Marazakis, and A. Bilas. Providing security to the Desktop Data Grid. In *2nd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid'08)*, 2008.

[30] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*. MIT, 2002.

[31] E. Otoo, D. Rotem, and A. Romosan. Optimal File-Bundle Caching Algorithms for Data-Grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2004. IEEE Computer Society.

[32] PPDG. From Fabric to Physics. Technical report, The Particle Physics Data Grid, 2006.

[33] A. Reinefeld, F. Schintke, and T. Schatt. Scalable and Self-Optimizing Data Grids. *Annual Review of Scalable Computing, Singapore University Press*, 6:30–60, 2004.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 2001.

[35] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[36] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[37] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of SuperComputing'03*, Phoenix, Arizona, USA, November 2003.

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[39] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proc. of the 2nd IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid'02)*, 2002.

[40] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. L. Scott. FreeLoader: Scavenging Desktop Storage Resources for Scientific Data. In *Proceedings of Supercomputing 2005 (SC'05)*, Seattle, 2005.

[41] B. Wei, G. Fedak, and F. Cappello. Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005*, Seatle, 2005.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399