# Using Cliques of Nodes to Store Desktop Grid Checkpoints

Filipe Araujo, Patricio Domingues, Derrick Kondo, Luis Moura Silva

# Using Cliques of Nodes to Store Desktop Grid Checkpoints

Filipe Araujo
*CISUC, Department of Informatics Engineering, University of Coimbra, Portugal*
filipius@dei.uc.pt

Patricio Domingues
*School of Technology and Management, Polytechnic Institute of Leiria, Portugal*
patricio@estg.ipleiria.pt

Derrick Kondo
*Laboratoire d'Informatique de Grenoble INRIA Rhône-Alpes, France*
dkondo@imag.fr

Luis Moura Silva
*CISUC, Department of Informatics Engineering, University of Coimbra, Portugal*
luis@dei.uc.pt

**Abstract**

Checkpoints that store intermediate results of computation have a fundamental impact on the computing throughput of Desktop Grid systems, like BOINC. Currently, BOINC workers store their checkpoints locally. A major limitation of this approach is that whenever a worker leaves unfinished computation, no other worker can proceed from the last stable checkpoint. This forces tasks to be restarted from scratch when the original machine is no longer available.

To overcome this limitation, we propose to share checkpoints between nodes. To organize this mechanism, we arrange nodes to form complete graphs (cliques), where nodes share all the checkpoints they compute. Cliques function as survivable units, where checkpoints and tasks are not lost as long as one of the nodes of the clique remains alive. To simplify construction and maintenance of the cliques, we take advantage of the central supervisor of BOINC. To evaluate our solution, we combine simulation with some real data to answer the most fundamental question: what do we need to pay for increased throughput?

**Keywords:** Desktop grid, checkpointing, clique

## 1. Introduction

The enormous success of BOINC [1], fueled by projects like SETI@home or climateprediction.net, turned Desktop Grid (DG) communities into some of the most powerful computing platforms on Earth. This trend mounts on the motivation of volunteers eager to contribute with idle resources for causes of their interest. With an average utilization of CPU as low as 5% [6] and with new CPUs shipping with increasing number of cores, more and more resources should be available for grid computing in the near future.

Although bearing enormous potential, volatility of workers poses a great challenge to DG. To mitigate volatility, at certain points in the computation, the worker computes a checkpoint and stores it locally. This enables the *same* worker to resume computation from the last checkpoint, when it resumes computation. Unfortunately a worker can be interrupted by a single key stroke and can also depart from the project at any time. In this case computation of the worker in the task is simply lost.

One obvious limitation of storing checkpoints locally is that they become unavailable whenever the node leaves the project. Martin et al. [11] reported that climateprediction.net would greatly benefit form a mechanism to share checkpoint files among worker nodes, thus allowing the recovery of tasks in different machines. Storing the checkpoints in the central supervisor is clearly unfeasible, because this would considerably increase network traffic, storage space and, consequently, management costs of the central supervisor. Another option could be to use a peer-to-peer (P2P) distributed storage (refer to Section 4). We exclude this option for a number of reasons. First, P2P storage systems would typically require a global addressing scheme to locate checkpoints, thus imposing an unnecessarily high burden for storing replicas of a checkpoint. Unlike this, we can store all replicas of a checkpoint in nearby nodes, because we can afford to lose checkpoints and recompute respective tasks from scratch. This way, we trade computation time for simplicity. Moreover, in a P2P file system, replicas of a checkpoint are stored in arbitrary peers. We follow the approach of storing checkpoints in nodes that might use them. This is simpler, involves fewer exchanges of checkpoints and can allow nodes to use checkpoints they store to earn credits.

In this context, we present *CliqueChkpt*, which follows from our previous approach in [5], where we used dedicated storage nodes to keep checkpoints of tasks. To make these checkpoints available to the community, workers self-organized into a DHT where they stored pointers to the checkpoints. In *CliqueChkpt*, we try to improve this system. First, we address the requirement of using dedicated storage nodes to hold the checkpoints. Second, we address the requirement for storing pointers to the checkpoints in the DHT, which raised the complexity of the system. To achieve this, we connect nodes in complete

graphs (i.e., graphs where all the vertices are connected to each other, also know as cliques). These cliques form small and independent unstructured P2P networks, where workers share all their checkpoints. This enables *CliqueChkpt* to easily achieve a replication factor that ensures checkpoint survivability despite frequent node departures. Unlike some P2P systems that are fully distributed and thus require considerable effort to find clique peers [10], we simply use the central supervisor for this purpose. We show that *CliqueChkpt* can achieve consistent throughput gains over the original BOINC scheme and we assess the bandwidth and disk costs that we need to pay for this gain.

The rest of the paper is organized as follows: Section 2 overviews *CliqueChkpt*. In Section 3 we do some preliminary evaluation of *CliqueChkpt*. Section 4 presents related work and Section 5 concludes the paper.

## 2. Overview of *CliqueChkpt*

In *CliqueChkpt*, upon request from a worker, the central supervisor assigns it one or more tasks. The worker then computes the tasks, sending back the results when it is done. We only consider sequential tasks, which can be individually broken into multiple temporal segments $(S_{t_1}, \ldots, S_{t_i}, \ldots, S_{t_n})$ and whose intermediate computational states can be saved in a checkpoint when a transition between temporal segments occurs. Whenever a task is interrupted, its execution can be resumed from the last stable checkpoint, either by the same node (if it recovers) or by some other worker. Our main goal is to increase throughput of the entire computation. In all that follows we consider single-replica computation. Extending this to scenarios where multiple replicas exist is straightforward.

### 2.1 *Chkpt2Chkpt*

*CliqueChkpt* follows from our previous work in a system called *Chkpt2Chkpt* [5]. In *Chkpt2Chkpt*, we had the following components: the central supervisor, the workers (including one particular type of worker, called the Guardian) and Storage Points. The workers self-organized into a DHT that served to store two kinds of information: indication of the current state of a task (owner and number of the checkpoint being computed) and pointers to previous checkpoints. This worked as follows (see Figure 1): $i$) worker requests task from server and gets its data; $ii$) worker registers task in the Guardian (which in fact is just a standard worker); $iii$) worker finishes a checkpoint and uploads it to a storage point; $iv$) client stores the pointer to the checkpoint, updates the Guardian (not shown) and $v$) sends results to the central supervisor.

The advantage of *Chkpt2Chkpt* over standard BOINC happens when there is some worker that departs leaving some computation unfinished. In this case, the cycle above changes slightly. Instead of restarting the task from

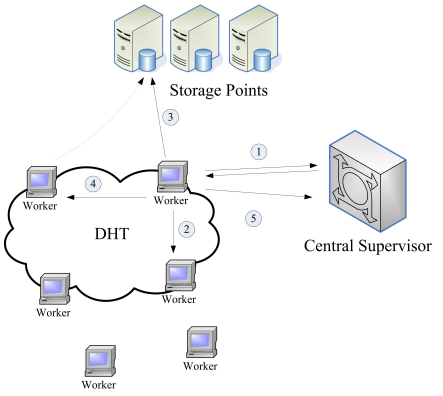


*Figure 1.* Overview of *Chkpt2Chkpt*

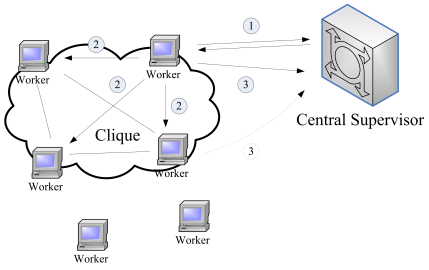


*Figure 2.* Overview of *CliqueChkpt*

the beginning, the worker can look for a recent checkpoint using the DHT. We showed in [5], that under certain scenarios our system could considerably decrease the turnaround time of tasks, for checkpoint availabilities starting at 40%.

## 2.2 *CliqueChkpt*

In *CliqueChkpt*, we tackle the following shortcomings of our previous approach. *Chkpt2Chkpt* required the use of special dedicated nodes for storage and it depended on pointers stored in volatile nodes. To overcome the first problem, we store checkpoints in standard workers. Any node that stores a checkpoint can use it to get extra credits in the task, when the original worker fails. This gives nodes the motivation to increase storage space for checkpoins. To handle volatile nodes, we increase redundancy. Redundancy was harder to increase in *Chkpt2Chkpt*, because we used several pointers to handle tasks and checkpoint information. To replicate we would need to also replicate these pointers. This would raise complexity and, at the same time, it could fail to work, because pointers themselves are also highly volatile.

We now use a simple *ad hoc* peer-to-peer network, where nodes form independent and completely connected graphs — a clique (Figure 2). Each node of the clique requests tasks from the central supervisor (step 1). After reaching a checkpoint, the worker fully replicates the checkpoint inside its clique (step 2). Finally, the worker submits results to the central supervisor (step 3). If the worker fails any other worker from the clique can use the checkpoint to finish computation and submit the results (dashed step 3). We take advantage of the central supervisor role to handle the workers that belong to each clique. Since the central supervisor needs to know all the workers that exist in the system,

we use it to set clique membership. Management of the group itself, like node failure detection, reliable group communication, checkpoint download, upload and so on, is up to the workers in the clique and does not involve the central supervisor.

Workers have a system-wide variable, which tells the number of members in each clique. Whenever they boot or if they find at some moment that their clique is too small they request a new group from the central supervisor. To reduce the load in the central point, in some cases, specially when operational cliques still exist, workers should postpone this request until they really need to communicate with the central supervisor (for instance, for delivering results or requesting new tasks). The central supervisor keeps information of all the cliques that exist in the system.

Whenever a node requests a new clique, the central supervisor looks for an appropriate clique for the node and replies. This can involve three distinct situations: $i$) the client should join a new clique, $ii$) or the client already has a clique and this clique should be merged with some other, $iii$) or the clique should remain unchanged. Only the two first cases are of interest to us. A node can join a new clique or can request another clique to merge by sending a messages to some node of the clique (which can redistribute the message inside the clique afterward). One very important detail here is that some nodes can be inactive very often. For instance, in some configurations, if users interact with the system, BOINC middleware hands the CPU back to the user. We do a clear distinction between the communication group of the clique and the clique itself: the former is a subgroup of the latter (possibly the same). Workers should only consider their peers as missing from the clique (and thus form new cliques) after relatively large periods. For instance, simple heartbeats or disconnection if nodes keep TCP connections alive can serve to identify departing nodes.

At least one of the nodes in a clique should have a public accessible IP address[1]. This node must receive incoming connections from unreachable NAT peers to keep open ports at routers. This will make these nodes responsible for routing messages inside the clique. To pay for this effort, these nodes could be allowed to belong to several different cliques to raise their chance of resuming stalled computations and earn credits.

## 2.3 Definition of the Cliques

The central supervisor needs to have fast access to the size of the cliques. It can store this in a simple array ordered by clique size, starting from 1 and ending in the largest possible cliques (this array has necessarily a small number of entries, because cliques cannot be big). In a first approach the elements

[1]This is not strictly necessary as we could use a node outside the clique to serve as communication broker.

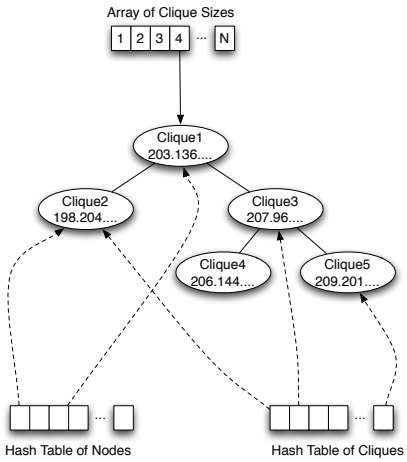


*Figure 3.* Internal structure of the central supervisor

of this array could point to a simple linked list with cliques of the same size. When needed, the central supervisor would just pick the first clique in the list and remove it from there (either to add an element or to merge with some other clique). However, to conserve bandwidth and reduce latency a simple approach would be to make cliques of nearby IP addresses, as these will often (but not always) reflect topological proximity. In this way, we can organize the cliques into ordered trees. To do this, we can compute the "average" IP address of a node in the clique or picking the address of one of the nodes. By keeping these trees balanced, searching a clique can have a time cost of $O(\log n)$, where $n$ is the number of nodes. Additionally, this requires, $O(n)$ space, which is reasonable if we consider that the server already uses this space to keep information of nodes. Correlated failures of nodes of the same clique pose little problem to *CliqueChkpt*, because tasks can be recomputed.

The former structure makes cliques accessible by their size and by their IP addresses. We also need some data structure that makes cliques accessible by their identification and also by their nodes. To access a particular clique in $O(1)$ time, both these structures could be hash tables. Deletion of old cliques occurs when new ones are formed from merging operations. To remove other cliques made of nodes that become inactive, the central supervisor must remove the workers first, one at a time. We show the entire structure in Figure 3.

Workers inside a clique control the activity of each other, such that they can tell whenever a clique peer is not computing its task anymore. In this case, active nodes can request the central supervisor to acquire the task. However, the central supervisor needs to have some mechanism to prevent faster nodes

from stealing checkpoints from slower ones. A related problem is to prevent nodes from getting undeserved credits that could be earned from checkpoints. To prevent these situations, we force nodes to periodically claim ownership of a task. If a worker fails to do so, it can be overrun by another worker. This can be combined with a Trickle mechanism [3] to give the appropriate credits to the right worker, when it reaches certain points in the computation. In this way, nodes can claim their credits as they go, they do not allow other nodes to steal their tasks, while at the same time, they can also claim ownership on a stalled task. Like in the standard BOINC case, the central supervisor must use replication to get some confidence that workers have done, in fact, real work.

## 3. Experimental Results

To evaluate the capabilities of our system, we built a simple event-based simulator. This simulator includes workers and a central supervisor. All workers are similar in CPU performance, all have public IP adresses and follow a fail-stop approach. In practical terms this means that all the nodes abandon the project at some random point in time. In Section 3.3, we evaluate such assumption against known figures for node attrition of climateprediction.net. Workers start in a sleeping state and wake up at random moments. Then, they request a task and compute it, repeating these two steps as many times as they can before thy leave. After another random period of time (with the same average length), new nodes enter the network in a fresh state (no task and no checkpoints). All the (pseudo) random times that govern the states of nodes follow an exponential distribution. We used $35$ nodes in the simulation, $200$ different tasks and $20$ checkpoints per task. Simulation time is set as the end of the $150^{th}$ task (whatever task it is) to exclude waiting times in the end. Currently in the simulator, we still do not take into consideration the time that it takes to create and transfer the checkpoints.

### 3.1 Evaluation of Clique Size

We first try to determine the ideal clique size. It turns out that this value depends heavily on nodes lifetimes. In Figures 4 and 5, we fixed node average lifetime and varied the size of the clique. The former figure shows throughput relative to the throughput achieved by (standard) nodes with private checkpoints. Figure 5 refers to the costs: we measure storage space as the average number of checkpoints that each active node stores on disk; and we measure bandwidth as the average number of times that each checkpoint is exchanged in the network[2].

[2]Computed as $\frac{E_{checkpoints}}{T_{checkpoints}}$, where $E_{checkpoints}$ is the total number of checkpoints exchanged in the entire simulation and $T_{checkpoints}$ is the total number of checkpoints that exist in the simulation.

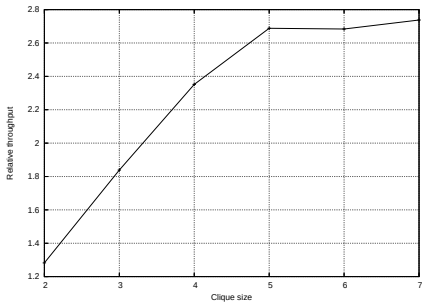


*Figure 4.* Relative throughput

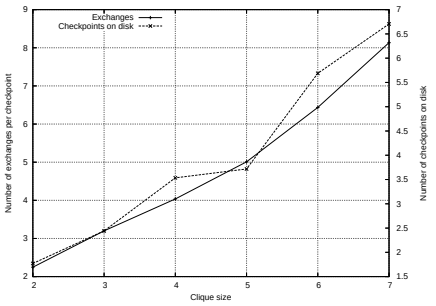


*Figure 5.* Chkpt. exchanges and space in disk

We can see that throughput goes up with clique sizes until it reaches a plateau. To conserve space we omit plots for different average lifetimes. If nodes live longer, we have little gain in increasing clique sizes and we get an approximately horizontal line in the throughput (with values below $1.5$). If nodes live shorter, there is a consistent gain in throughput with clique size. This suggests that for a given node lifetime there is an ideal clique size. We have rapid gains in throughput as we reach that size and no gains after that point.

Unlike throughput, bandwidth and disk space seem to grow linearly with clique size. This makes it difficult to pick the right size for the clique, as, in general, the designer may not know exactly the lifetime of nodes. Depending on the available resources, a conservative approach points to small cliques, as they promise improvements at controlled costs in bandwidth and disk space.

## 3.2 Comparison of Performance

In Figures 6, 7 and 8, we plot throughput, bandwidth consumption and occupied disk space, for varying worker average lifetimes. We consider cliques with 3, 5 and 7 nodes as well as a setting where nodes can upload and download all the checkpoints from storage points (as in *Chkpt2Chkpt*, except that in Figures 6, 7 and 8 nodes always succeed to download the checkpoints). This shows that an ideal storage point is unbeat by cliques, which, on the other hand, can achieve much higher throughputs than nodes with only private checkpoints. This gap widens as the failure rate of nodes goes up (this rate is computed as task duration divided by the average lifetime of a node).

## 3.3 A Simple Estimation of Trade-offs

In this paper, we basically propose to get more throughput in exchange of disk space and bandwidth. In this section we roughly quantify these trade-offs using figures from climateprediction [3]. Machine attrition should be around $2/3$, which means that around $2/3$ of the machines drop computation before

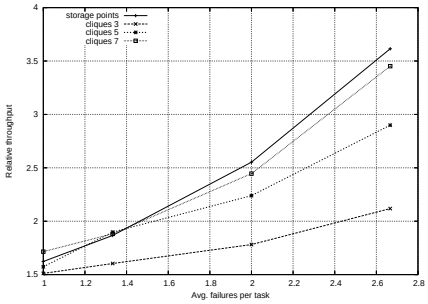


*Figure 6.* Rel. throughput for fail. rate

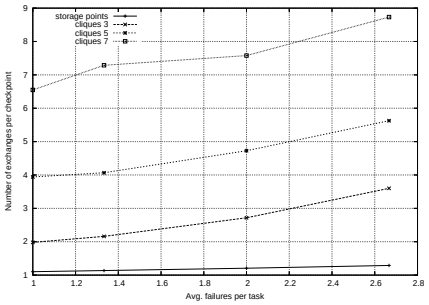


*Figure 7.* Avg. exchanges of each checkpoint

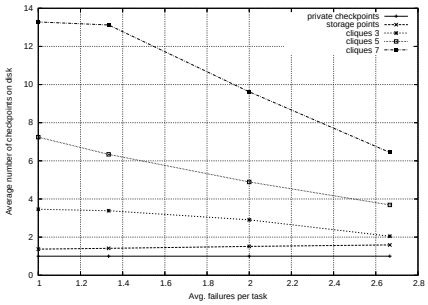


*Figure 8.* Average number of checkpoints on disk per active node

finishing an entire task. This means that our fail-stop model only applies to $2/3$ of the machines, while in $1/3$ of the machines we might not gain much from using cliques. Assume that due to their failures, these $2/3$ of the machines take $n$ times more to compute a task in average. Assume also that failure-free workers compute $T_{ff}$ tasks per time unit. This means that failure-prone machines will compute $2T_{ff}/n$ tasks per time unit, as they are twice as much, but $n$ times slower. We roughly evaluate throughput with all machines, $T_{all}$, and the gain in throughput, $G$, as:

$$T_{all} = T_{ff} + \frac{2T_{ff}}{n} = \frac{n+2}{n} T_{ff}$$

$$G = \frac{T_{all}}{T_{ff}} = \frac{\frac{n+2}{n} T_{ff}}{T_{ff}} = \frac{n+2}{n}$$

The gain $G$ is very sensitive to $n$ and tends to 1 (no gain) as $n$ grows. In Table 1, we show the gains in throughput and the costs of using cliques in the computation of a task. We assume the following data (taken from our previous experiments and from [3]): 20 checkpoints of 20MB for task; 40-day computation (so we have a checkpoint on every two days on average); and failure-prone

*Table 1.* Analysis of trade-offs

| Parameter | Storage Point | Clique 3 | Clique 5 | Clique 7 |
|---|---|---|---|---|
| Days saved | 18.6 | 13.2 | 17.7 | 17.8 |
| Avg. disk space per node (MB) | 31.8 | 41 | 73.8 | 128.8 |
| Avg. exchanges per node (MB) | 515.8 | 1439.8 | 2250.6 | 3492.6 |

nodes fail 2.66 times during those 40 days (this number matches Figure 4). It should be noticed that the frequency at which we publish checkpoints can be made totally independent from the actual frequency at which nodes produce local checkpoints. climateprediction.net creates local checkpoints every 15 minutes, but we can distribute a checkpoint only once in every two days. Table 1 shows that although the potential to save time in a task is considerable, bandwidth required might be beyond what is available for most computers today, if we think of a 20-30 day span to compute a task (after the improvements). On the contrary, disk space requirements are not too high.

## 4. Related Work

The idea of using cliques in distributed systems is not new. However, to our best knowledge, these have been used in different contexts and for different purposes. For example in LARK [10], nodes form ad hoc cliques with peers. The purpose of LARK is to do multicast at the application level. To multicast a message, nodes just send it to all the peers they know in different cliques. Most complexity of LARK comes from the need to create and maintain the cliques in a way that is, at the same time, efficient and tolerant to failures. Unlike LARK, we can greatly benefit from the central supervisor to discard all this complexity from the system. CliqueNet [9] also uses cliques, but for the sake of ensuring communication anonymity.

*CliqueChkpt* directly follows our previous work in *Chkpt2Chkpt*, where we used a DHT to store checkpoints and to manage some data related to the BOINC tasks [5]. Here, we try to remove some of the constraints existing in *Chkpt2Chkpt* and shift all the storage back to the volunteer nodes. Interestingly, there is one project called Clique [13], which targeted a lightweight distributed file system, where nodes self-organize into cliques. Clique also offered the possibility of disconnected operation because it includes a reconciliation protocol. Unlike this project, our cliques are logical and do not need to correspond to some topological division. Besides Clique there are many other systems that provide distributed storage. These often use DHTs to store either the files or pointer to files. Consider the case of PAST [8], Venti-DHash [14], Shark [2] or OceanStore [12], just to name a very small set. Most of these systems reach far

beyond what we really need for sharing checkpoints as they are fully fledged file systems, while all we need is to store some checkpoints in a purely best-effort basis.

With respect to storage requirements our system is closer to Condor [15] and Condor-G [4] (Condor for multi-institutional Grids), which have powerful mechanisms to generate checkpoints and migrate jobs. However, the specificities of these systems cannot be reproduced in DG systems. They push tasks to available computers (inside LAN or accessible to Globus in Condor and Condor-G, respectively) that share some sort of administrative ties. Unlike this, volunteers of a DG system are very loosely coupled and the only central entity is often over-utilized, under-financed and cannot be used to store checkpoints. In a previous work, we also analyzed the effects of sharing checkpoints in local area environments, resorting to a centralized checkpoint server [7].

## 5. Conclusions and Future Work

In this paper we proposed a system called *CliqueChkpt*, where we organize workers of BOINC into complete graphs (cliques) to share checkpoints of tasks. This involves the central supervisor to manage the groups, but does not use neither this central component, nor any other dedicated machines to store checkpoints. Additionally, nodes storing checkpoints can potentially use them to earn CPU credits, which serves as motivation for volunteers to donate resources. Our simulations suggest that *CliqueChkpt* can bring considerable advantage over private checkpoints when tasks are very long, as in projects like climateprediction.net. To demonstrate the feasibility of our scheme, we used some figures from climateprediction.net to produce a rough estimate of advantages, as well as some costs involved. This analysis showed that while there is a huge potential for these schemes, bandwidth can be a major hurdle.

As we referred before, our work has some limitations that we intend to tackle in the future, namely in the simulator, as we are not considering the times needed to produce and exchange a checkpoint. Concerning the use of cliques, we believe that there is considerable room for reducing the costs involved in exchanging the checkpoints. In fact, we use a very straightforward scheme that always downloads all the checkpoints missing from a node when there is a change in the clique of that node. Just to name one possibility, we could reduce the number of checkpoint replicas in each clique and use short timeouts to detect worker failures.

## Acknowledgments

This work was supported by the CoreGRID Network of Excellence, funded by the European Commission under the Sixth F.P. Project no. FP6-004265.

# References


[1] D. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[2] S. Annapureddy, M. Freedman, and D. Mazieres. Shark: Scaling File Servers via Cooperative Caching. *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), Boston, USA, May*, 2005.

[3] C. Christensen, T. Aina, and D. Stainforth. The challenge of volunteer computing with lengthy climate model simulations. In *1st IEEE International Conference on e-Science and Grid Computing*, pages 8–15, Melbourne, Australia, 2005. IEEE Computer Society.

[4] Condor-g. http://www.cs.wisc.edu/condor/condorg/.

[5] P. Domingues, F. Araujo, and L. M. Silva. A DHT-based infrastructure for sharing checkpoints in desktop grid computing. In *2nd IEEE International Conference on e-Science and Grid Computing (eScience '06)*, Amsterdam, The Netherlands, December 2006.

[6] P. Domingues, P. Marques, and L. Silva. Resource usage of windows computer laboratories. In *International Conference Parallel Processing (ICPP 2005)/Workshop PEN-PCGCS*, pages 469–476, Oslo, Norway, 2005.

[7] P. Domingues, J. G. Silva, and L. Silva. Sharing checkpoints to improve turnaround time in desktop grid. In *20th IEEE International Conference on Advanced Information Networking and Applications (AINA 2006), 18-20 April 2006, Vienna, Austria*, pages 301–306. IEEE Computer Society, April 2006.

[8] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, Schoss Elmau, Germany, May 2001.

[9] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report TR2003-1890, Cornell University Computing and Information Science Technical, February 2003.

[10] S. Kandula, J. K. Lee, and J. C. Hou. LARK: a light-weight, resilient application-level multicast protocol. In *IEEE* $18^{th}$ *Annual Workshop on computer Communications (CCW 2003)*. IEEE, October 2003.

[11] A. Martin, T. Aina, C. Christensen, J. Kettleborough, and D. Stainforth. On two kinds of public-resource distributed computing. In *Fourth UK e-Science All Hands Meeting*, Nottingham, UK, 2005.

[12] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[13] B. Richard, D. Nioclais Mac, and D. Chalon. Clique: A transparent, peer-to-peer collaborative file sharing system. Technical Report HPL-2002-307, HP Laboratories Grenoble, 2002.

[14] E. Sit, J. Cates, and R. Cox. A DHT-based backup system, 2003.

[15] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation Practice and Experience*, 17(2-4):323–356, 2005.