A DHT-based Infrastructure for Sharing Checkpoints in Desktop Grid Computing

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

Filipe Araujo, Luis Moura Silva
University of Coimbra, Portugal
{filipius, luis}@dei.uc.pt

Abstract

In this paper we present $Chkpt2Chkpt$, a desktop grid system that aims to reduce turnaround times of applications by replicating checkpoints. We target desktop computing projects with applications that are comprised of long-running independent tasks, executed in hundreds or thousands of computers spread over the Internet. While these applications typically do local checkpointing to deal with failures, we propose to replicate those checkpoints in remote places to make them available to other worker nodes.

The main idea is to organize the worker nodes of a desktop grid into a peer-to-peer Distributed Hash Table. Worker nodes can take advantage of this P2P network to keep track, share, manage and reclaim the space of the checkpoint files. We used simulation to validate our system and we show that remotely storing replicas of checkpoints can considerably reduce the turnaround times of the tasks, when compared to the traditional approaches where nodes manage their own checkpoints locally. These results make us conclude that the application of P2P techniques seems to be quite helpful in wide-scale desktop grid environments.

1. Introduction

In the last years, public-based computing projects such as SETI@home [21] and climateprediction.net [9] have emerged to capitalize the tremendous amount of idle computing power. Indeed, it is a well documented fact that CPU usage averages 5%, especially in desktop computers which are mainly used for running office-based applications [15]. The recent trend towards multicore CPU [13] seems to indicate that more processing power will remain unused in the near future. But resource idleness is not restricted to CPU: desktop grids are also interesting by the large amounts of available memory, disk storage and network bandwidth, as reported by Anderson and Fedak in [2]. Several middleware platforms like BOINC [1] and XtremWeb [17], have emerged and at least BOINC has been widely used in several projects of Internet-based grid computing [14].

However, the environment of desktop grids has some limitations, namely: a) the high volatility of computing nodes in the Internet; b) the need to deal with malicious attempts of sabotage of the application results; c) and the difficulty of managing the whole infrastructure. Internet-based desktop grids are also prone to network connectivity problems, when some of the nodes have to deal with network address translation (NAT) schemes and firewall systems. For this reason, desktop grids are mainly oriented to the master-worker paradigm or the so-called bag-of-task applications [8]. In this approach, a central supervisor orchestrates the whole computation and is responsible for the distribution of tasks amongst workers and for the final validation of results that have been computed by the nodes of the desktop grid.

Given the high volatility of resources, platforms like BOINC resort to application-level checkpointing. However, this approach presents a clear limitation, because all the checkpoint files of a node are stored locally. If this node fails, the local checkpoint files will not be available, and thereby they turn out to be useless. Martin et al. [18] reported that the climateprediction.net project would greatly improve efficiency with the existence of a mechanism to support the sharing of checkpoint files amongst worker nodes allowing the recovery of tasks in different machines. The alternative of storing checkpoints in the central supervisor is not feasible, since the central supervisor would become a very easy bottleneck.

In this context, we propose to make use of a P2P infrastructure for sharing checkpoint files that should be tightly integrated with the desktop grid environment. Under this approach, the worker nodes act as peers of a P2P Chord [24] Distributed Hash Table (DHT) that they use to track the checkpoint files. If the replicated checkpoint files are available in the P2P overlay, recovering from a failed task can be much more effective when compared with the private checkpoint model used in BOINC.

Although there are many recent examples of peer-to-peer file-sharing and backup applications, e.g. [12, 3, 23, 6], just to mention a few non-commercial systems, sharing checkpoints requires a different type of solution. To start, checkpoints become garbage in a deterministic (and often fast) way. For instance, as soon as some task is finished, all of its checkpoints should be discarded. Another difference concerns the usefulness of the checkpoints. While the utility of storing music files in one's disk is obvious, the same is not true with checkpoints. A system that replicates checkpoints needs to explicitly reward users that concede their space. On the other hand, in our solution, we can take advantage of the strict control that we have on the creation and placement of checkpoints. This sort of reasons make us think that creating a checkpoint replication system goes beyond a simple adaptation of existing file-sharing solutions.

Given these ideas, in this paper, we propose and validate through simulation a desktop computing architecture called $Chkpt2Chkpt$, which couples the traditional central supervisor based approach with a Peer-to-Peer (P2P) Distributed Hash Table that enables checkpoint sharing. The purpose of $Chkpt2Chkpt$ is to reduce the turnaround time of bag-of-tasks applications. By reusing the checkpoints of interrupted tasks, nodes need not to recompute those tasks from the beginning. This considerably reduces the average turnaround time of tasks in any realistic scenario where nodes often leave with their tasks unfinished. Moreover, by reusing previous computations, our replication system also increases the throughput of the entire desktop grid system.

The rest of the paper is organized as follows: Section 2 presents an overview of the system. Section 3 presents the P2P infrastructure, and the main issues related to the use of the DHT. Section 4 describes our mechanism for garbage collection of useless files. Section 5 presents some preliminary results obtained by simulation, while section 6 outlines the related work. Finally, section 7 concludes the paper and presents venues for future work.

2. Overview

Our system is built upon the traditional model of public-computing projects, with a central supervisor coordinating the global execution of an application. Specifically, the central supervisor replies to a volunteer worker node (henceforth *worker*) request for work by assigning it a processing task. The worker then computes the assigned task, sending back to the central supervisor the results when it has completed the task. An application is comprised by a large number of independent tasks, and only terminates when all of its tasks are completed. Every task is uniquely identified by a number. Furthermore, we only consider sequential tasks, which can individually be 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 promote availability of checkpoints to increase the recoverability of the interrupted tasks, thereby improving the turnaround time of the applications.

Checkpoints are identified by a sequential number starting at 1. Workers self-organize to form a DHT[1], which they use to maintain the distributed checkpointing scheme and to keep track of the execution of tasks, in such a way that requires a minimal intervention of the central supervisor.

To describe the basic idea of the proposed system, we first expose the simple case of a single worker executing a task from the beginning to its end (see Figure 1). In this case, interaction occurs as follows: 1) The worker requests the central supervisor for a computing task, receiving a task and its respective input data. 2) The worker registers the task in the DHT, by selecting a regular peer-worker of the DHT to store a tuple called "worker-task info", which keeps information about the task. We deem this peer-worker as "guardian of i" ($guardian_i$). As we explain in Section 3, $guardian_i$ is a regular random node of the DHT. In general, different tasks have different guardians. 3) Each time the worker has to perform a checkpointing operation, it writes the checkpoint in its own disk and replicates it in some storage point which it selects accordingly to a given metric, like for instance, the network distance; 4) it uses the DHT to store a pointer to that storage point. This pointer is accessible by any other node of the DHT, using as key the pair formed by the task identifier and the checkpoint number. 5) Finally, when the worker node has completed the whole task it sends back the results to the central supervisor.

If a worker fails two things may happen: 1) if the worker recovers after a short period of time it resumes its previous task execution from the last checkpoint file maintained in its local disk. 2) If the worker fails for a period longer than a specified timeout then the central supervisor may redistribute that task to some other worker node. In this case, the new worker performs step 2 as before, but it may get in response from $guardian_i$, the number of the checkpoint where the task was left. In this case, the worker tries to fetch the checkpoint to resume the task. However, due to the departure of the previous $guardian_i$, the worker may not get any reply with the number of the checkpoint. In this scenario, the worker starts looking for the best checkpoint using a procedure that we describe in Section 3.5. After having found such checkpoint, the worker proceeds as explained before.

[1] It is not strictly necessary that all the nodes participate in the DHT, but only that they can access and write data on the DHT. To simplify description, we assume that all nodes belong to the DHT.

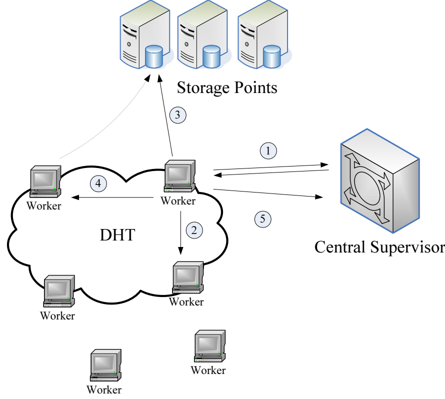

Figure 1. A summary of the existing interactions and of the components of the system

In our failure model, we assume that the central supervisor is protected by some replication mechanism and is always available despite the occurrence of some transient failures. On the contrary, we assume that worker nodes may fail frequently under a failure model which is either crash-stop or crash-recovery. Nodes that fail and then recover but loose their previous task can be seen as new nodes. Due to the high volatility of volunteer resources, nodes are very prone to fail, and thus can deprive, at least temporarily, the DHT from some state information and checkpoints. Thus, we consider as a typical case the possibility of a node not finding some information that it is looking for in the DHT. We have also considered byzantine failures of nodes in the design of our system. We inherit from existing solutions that use replication of computation to overcome some of these failures. However, managing the DHT under these conditions is an altogether different subject that we do not discuss here. See for instance [22].

An important issue that exists in $Chkpt2Chkpt$, as well as in many file sharing systems, is the garbage collection of shared files that are no longer useful. This is particularly relevant with checkpoint files, whose usefulness expires as soon as the task is completed. Thus, it is necessary to remove old checkpoints and the related management information for those checkpoints. We propose two approaches to erase the useless files: 1) pull and 2) push. Under the pull approach, nodes basically get information of task i from $guardian_i$ before they delete the stored information. Under the push approach, the node that finishes a task sends messages to remove all the administrative information related to the task. This is further explained in Section 4.

3. Description of $Chkpt2Chkpt$

3.1. Basic Components

We assume that task i is identified by i and that n sequential checkpoints are produced along the task execution. We identify the node that is working on task i as p_i, the j-th checkpoint of task i with the key $i : j$ and, the hash of this key as $hash(i : j)$, where $hash()$ is the hash function of the DHT. The DHT mechanism ensures that there is a single node responsible for holding key $i : j$, typically a node whose identifier is close to $hash(i : j)$ (to simplify we refer to this as the node $hash(i : j)$, although the real identifier of the node will usually be close but different). Any node in the DHT can reach the node $hash(i : j)$ in a deterministic way.

One of the principles that guided the design of this system was to reduce the burden of the central supervisor as much as possible and to keep all communications worker-initiated. Hence, the checkpointing service must be supported, as a best effort, by the nodes of the DHT. In particular, $Chkpt2Chkpt$ strongly relies on $guardian_i$, on the storage points, and on the indirection pointers to ensure proper operation of task i. The node $guardian_i$ serves to indicate that worker p_i is processing checkpoint j of task i. Nodes can determine the location of $guardian_i$ by computing the hash of key $i : 0$, $hash(i : 0)$. The guardian of i stores a tuple called "worker-task info of i" (WTI_i). The format of this tuple is (p_i, j, t_p), where p_i is the processor node of task i and j is the checkpoint being computed. Besides these elements, the WTI_i stores a timeout information, t_p, that serves to overcome workers that abandon a task. When the timeout t_p expires, $guardian_i$ sends a message to the central supervisor announcing a possible failure of p_i. This allows us to maintain timeouts with a much finer granularity than it is usually possible with minimal effort from the central supervisor. The advantage is that the system can recover from failures of worker nodes in a much faster way.

Ideally, we would like to maintain the following invariant in $Chkpt2Chkpt$, which we deem as INV: WTI_i exists in the node $guardian_i$ if and only if task i is being processed. From this, it follows that if there is no WTI_i in the node $guardian_i$, nodes can assume that the task is already finished (or yet to start). For performance reasons, we allow this invariant to be violated once in a while, as node $guardian_i$ can be down and task i can still be active.

At this point, we can use a concrete example to better illustrate the interaction with the DHT: consider that a node wants to fetch the last checkpoint available of task 43 (assume that it is in fact available). It issues a $get(43 : 0)$ operation. Assume that $43 : 0$ hashes to 578. The DHT will forward the request to the owner of key 578, which

may be node 581. Node 581 will reply with the number of the requested checkpoint, e.g., 3. Now, to get checkpoint 3, the requesting node issues a $get(43 : 3)$ operation. After another lookup operation in the DHT, this will give it a pointer to the storage point of the checkpoint.

3.2. Processing a Task

While processing a task i, a worker node p_i needs to perform two operations at $guardian_i$: increase the number of the checkpoint when it is done with the previous one and send some "heartbeat" message, to let $guardian_i$ know that it is alive, before t_p expires. As long as this timeout does not expire (or the timeout of the central supervisor, as we explain in Section 3.3), only processor p_i can change values in the WTI_i tuple (this is one of the small twists that we do to the normal operation of the DHT). However, it is possible that two workers try to process the same task i simultaneously, if $guardian_i$ or the central supervisor wrongly consider the first worker as failed and the central supervisor reassigns the task to the other. If this occurs, it is up to node $guardian_i$ to decide which node owns the task. It is either the first node that tries to take control of the task, if t_p is expired, or it is the initial worker, if t_p is not expired (we describe a third possibility in Section 3.3). One interesting way of providing the periodical heartbeat is to slightly modify the normal $put()$ operation of the DHT, such that rewriting the WTI_i tuple serves as a heartbeat. In this case, this operation will also serve without any modification as a watchdog for $guardian_i$, because it periodically rewrites the value j of the current checkpoint in the $guardian_i$. If $guardian_i$ fails, the DHT will automatically redirect messages to some other nearby node, thus providing for a transparent replacement of $guardian_i$.

3.3. Initiating and Resuming a Task

The central supervisor assigns three types of tasks: tasks that are being delivered for the first time and tasks whose previous execution attempts have exceeded the task timeout, either t_p (given by $guardian_i$) or t_t (given by the central supervisor). To improve the turnaround time, we need to set $t_p << t_t$. We aim to minimize the number of messages and the number of bytes exchanged between node p_i and the central supervisor. If the central supervisor is delivering task i for the first time, it can immediately send all the data to the worker and the worker can start to process the data as shown in Figure 1.

However, it may happen that a worker leaves without delivering its task. In this case, upon expiration of timeout t_p, $guardian_i$ sends a message informing the central supervisor of the likely interruption of task i, which puts task i in the redistribution list[2]. In the redistribution of a task, the central supervisor only sends the identifier of the task, say i, and flags the repetition to worker p_i, which must check at $guardian_i$ whether some other node is still processing this task, to avoid concurrent processing of the same data. If worker p_i finds that the previous owner of task i is still holding the task (because it came back to life, for instance), it gives up the task and tries to get a new one from the central supervisor.

Finally, we have the case where the central supervisor redistributes a task for which t_t has expired. Here, the new worker, say p_i will have the task regardless of the situation of the previous worker. To this end, p_i must instruct $guardian_i$ to check the new owner of the task in the central supervisor. In a redistribution of a task (regardless of the timer, t_p or t_t, which has expired), the new worker node always tries to fetch the last available checkpoint, as we describe in Section 3.5.

3.4. Separation of Processing and Storage

To ensure the availability of checkpoints, the worker node of a task should store replicas of its checkpoints in other nodes. In this way, we keep the sequential checkpoints of a running task available in case the worker that holds the task fails and its task is redistributed. Given this constraint, we consider two additional assumptions to build our replication system:

Assumption 1: nodes offering processing time may not have space available for storage of checkpoints. The system should explicitly manage a separation between processing and storage nodes. To discover storage points, worker nodes should use some out-of-band mechanism;

Assumption 2: it is not feasible to maintain a constant number of replicas of each checkpoint in the DHT. When nodes enter and leave the DHT, they cannot transfer big checkpoints from one node to another as the DHT changes, because nodes may be distant from each other and this would clog the network. We base this assumption on the work of Blake and Rodrigues [5].

To cope with Assumption 1, $Chkpt2Chkpt$ extends the contribution model of traditional public-computing projects. Besides donating CPU cycles, nodes can contribute to the P2P infrastructure with storage space and bandwidth. The system separates CPU donation from storage space and bandwidth volunteering. In fact, a node can provide CPU cycles (executing tasks), or volunteer storage and bandwidth (integrating the DHT or being a storage

[2] To avoid concurrent processing of the same task, $guardian_i$ sends another message to the central supervisor if a timed out worker ever recovers (due, for instance, to a machine with transient network access that gets reconnected to the network).

point) or, donate resources to both causes. To foster motivation for donors to volunteer space storage and bandwidth for the P2P infrastructure, a rewarding credit mechanism and associated ranking system, similar to the one employed to recompense CPU donation in public-computing projects, can be devised [1]. Under this scheme, a resource donor receives credits for the space storage effectively devoted to shared checkpoints. Furthermore, an added bonus can be provided whenever a locally-stored checkpoint is used to resume a task in another machine.

To cope with Assumption 2, we do as we explained before. The DHT does not hold the checkpoints, but only pointers to the checkpoints. The purpose of this indirection is to conserve network traffic, because checkpoints can be very large, like in the case of the climateprediction.net project where each checkpoint file has about 20 MB. To access checkpoints, nodes use the standard $get()$ functionality of the DHT. For instance, to access checkpoint j of task i, a node needs to issue a $get(i : j)$. Since there is yet another level of separation, this $get()$ operation returns an indirection pointer to the storage, instead of the storage itself.

3.5. Managing the Checkpoints

To retrieve checkpoint j of task i from a storage point, interested nodes get the value of the key $i : j$, which is a pointer holding all the needed information to reach the checkpoint. However, the checkpoint may be unreachable (for example, the machine holding it is down). In this case, the node starts a procedure with a logarithmic number of steps to find the last checkpoint available. The worker node will successively divide the space of keys $i : 1$, $i : 2$, $i : \ldots$, $i : n$ (assuming that n is the number of checkpoints) in approximately equal parts. First, it looks for checkpoint $\lceil n/2 \rceil$. If this checkpoint exists it will consider the interval $[\lceil n/2 \rceil, n]$, otherwise it will consider the interval $[1, \lceil n/2 \rceil)$. In either case, it will split this second interval in two and repeat the procedure until it finds the highest available checkpoint. For instance, if $n = 10$, the node will look for checkpoint 5. If checkpoint 5 exists, it will now look for checkpoint $\lceil (10 - 5)/2 + 5 \rceil = 8$ and so on, until it may find out that 7 is the highest available checkpoint. When setting limits for these intervals, the worker must also try some checkpoints beyond the limits it previously found to make sure that a negative answer is not due to a disappeared checkpoint. For instance, checkpoint 5 might have been missing, which would make the node restrict its search to the interval $[1, 5)$. However, it could be the case that checkpoints 6 and 7 were still there and the node would wrongly get checkpoint 4 as the last one.

An aspect that we evaluate experimentally and which is crucial to the performance of our scheme is the availability of the checkpoints. As referred before, we use indirection pointers to separate storage from the DHT. A consequence of this is that checkpoints may be lost due to the disappearance of the indirection pointers stored in the DHT. To overcome this problem, the processing node periodically refreshes the pointers to old checkpoints. It may also occur that indirection pointers are left hanging, either because the storage point has departed or because it deleted the checkpoint. In this case, the node looking for the checkpoint must try to fetch checkpoints with lower numbers.

4. Garbage Collection

To reclaim the space used by a task, the worker that finishes that task sends a message to every node that stored information of the task: WTI_i, pointers to checkpoints and to the storage points. In fact, as we show ahead, deletion of the WTI_i tuple requires more than a simple deletion message. This is the **push** approach. We also use a **pull** approach if for some reason a node is left with state of task i hanging. Nodes that store replicas of large checkpoints also use the pull approach if they need to recover space before the task ends. Although the storage point can immediately delete any replicas, it can also use a more graceful approach of fetching the WTI_i to know if the task is over or the checkpoint is old.

It may happen that when the worker node tries to delete the WTI_i, this tuple is temporarily unreachable, just to come back later and violate the invariant INV (Section 3.1). To avoid this inconsistency, when the worker node finishes task i, it stores n as the last checkpoint written, which means that the task ended. For garbage collecting purposes, reading n as the current checkpoint is the same as not finding the task — it just means that the task is not running. The $guardian_i$ must store this tuple for some time before deleting it, to ensure that a finished task cannot come back to life, due to some transient misbehavior of the DHT, capable of bringing an old WTI_i back. Finally, $guardian_i$ can only delete WTI_i with a checkpoint value lower than n after asking the central supervisor whether task i has already finished.

5. Results

In this section, we evaluate the advantages of replicating checkpoints to recover from failures. We compare by simulation the turnaround times of $Chkpt2Chkpt$, where each checkpoint is replicated exactly once, versus a typical private solution, where each worker stores its own checkpoints. We use the traditional definition of turnaround time, corresponding to the time that goes from the moment when the central supervisor distributes the task up to when it receives the results. We assume a homogeneous set of work-

$t_{checkpoint}$	n	t_{exec}	t_p	t_t
10	5	50	30	150

Table 1. Experiment settings

ers, with individual nodes prone to crash-stop and crash-recovery failures, both of them following a random geometric distribution. At discrete time intervals, we randomly decide whether the worker changes state with a probability that is fixed throughout the computation of the task. In the crash-stop model, a node can change from working to crashed and can never recover. The task only restarts when it is rescheduled to another worker. In the crash-recovery model, the worker node can change from working to crashed state and vice-versa with the same probability. In the private solution, a new worker must restart a reassigned task from the beginning, while in $Chkpt2Chkpt$ it can recover one of the previous checkpoints.

Under ideal execution conditions, that is, if run uninterrupted, a task requires t_{exec} time units to complete, with checkpointing occurring every $t_{checkpoint}$ time units, for a total of n checkpoints ($t_{exec} = n \cdot t_{checkpoint}$). Additionally, we consider that the execution pace of the tasks is dictated by two timeouts: the timeout of the entire task t_t and the timeout t_p (described in Section 3.1). We set $t_t = 3 \cdot t_{exec}$ (only for private checkpoints) and $t_p = 3 \cdot t_{checkpoint}$ (only for distributed checkpoints). If these timeouts expire, the entire task is *immediately* reassigned and restarted. In all cases, we set $t_{checkpoint}$ to be 10 time units and fixed the number of checkpoints per task (n) to 5.[3] Hence, we have $t_{exec} = 50$, $t_t = 150$ and $t_p = 30$ (see Table 1). When a task is reassigned to another worker, the private approach restarts the computation from scratch, that is, in checkpoint 1. In the distributed checkpoint solution, the new worker tries to fetch a previous checkpoint. Unless otherwise stated, the probability of recovering each of the previously saved checkpoints is set to 50% (we take the most recent one, that is, the one with the highest index).

The simulation results, corresponding to the average of at least 50 random trial points, are plotted in Figures 2 to 4. Figures 2 and 3 compare the execution times of private versus distributed solutions, when the failure rate increases, for the crash-recovery and crash-stop models, respectively. These execution times are relative to the minimum possible execution time, i.e., t_{exec}. The average count of failures that occur between consecutive checkpoint operations ($t_{checkpoint}$) is represented in the x-axis. The curves show that our scheme performs better for higher failure rates.

[3] As expected, when we keep the failure ratio constant and increase the number of checkpoints for the same task, the turnaround time clearly improves until some point and then becomes nearly constant. We do not show this graphic to conserve space.

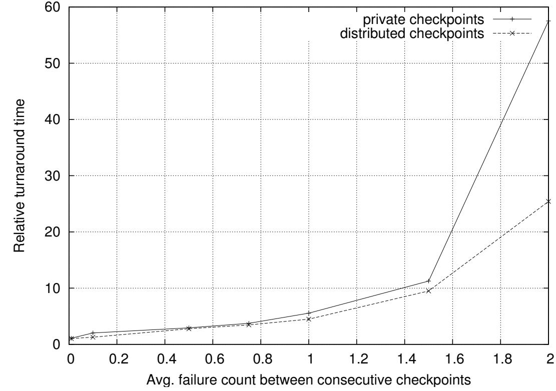

Figure 2. Turnaround times with crash-recovery failures

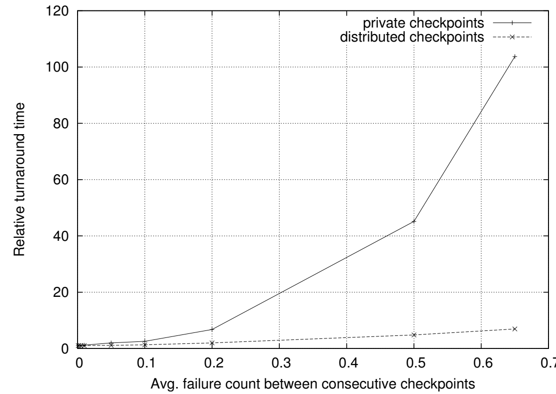

Figure 3. Turnaround times with crash-stop failures

This makes sense, because if failures are rare, i.e., if the environment is only lowly volatile or not volatile at all, there is no real need to share checkpoints. Under the crash-stop model, where a worker never returns to its task after failure, the distributed approach yields even better results when compared to the private approach. This is a consequence of the fact that the failure state lasts longer than the time needed to execute the task, a situation which degrades performance in the private approach. On the contrary, the shorter inter-checkpoint timeouts of the distributed approach, t_p, enable a faster reaction. This comparison is fair, because no dependency exists on the central supervisor to manage these per-checkpoint or per-process timeouts (except when they cause a redistribution). Finally, in Figure 4, we evaluate the impact of the probability of checkpoint availability (for a fixed crash-recovery probability). It is quite clear that the availability of checkpoints is crucial to

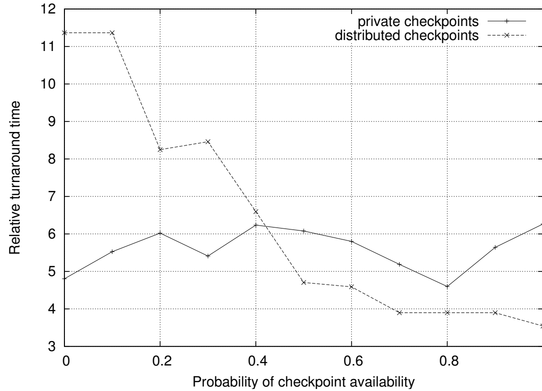

Figure 4. Turnaround time for varying checkpoint availability

the performance of our system: if availability is too small, like 40% or less, our system is of low utility (we may need to use more replicas of the checkpoints).

Hence, we believe that preliminary results show the validity of our approach and enable us to derive some conclusions about the advantages of using P2P techniques in desktop grids.

6. Related Work

In a previous work, we analyzed the effects of sharing checkpoints in local area environments, resorting to a centralized checkpoint server [16]. Likewise, Condor [25] relies on a central server for sharing checkpoint files and allows the migration of tasks for fault-tolerance and faster turnaround time. Both these approaches are limited to LAN environments. Condor-G [11] enables submission of jobs in remote clusters. However, unlike $Chkpt2Chkpt$, space of resources is not plane, because there is a clear separation between different clusters. For instance, this means that checkpointing cannot span across different clusters.

Antonelli et al. [4] propose a distributed approach for verification of results in volunteer computing. Although bearing some similarities to our work, this scheme is restricted to checkpoint verification and does not mention any use of P2P techniques.

Tritrakan and Muangsin [26] simulate the benefits of direct communication between a submitter machine that proposes a work and worker nodes in a desktop grid environment. The transfer of the needed files (input data and/or results) occurs directly between the submitter machine and the selected worker machine. When compared to our work, this approach uses a slightly different paradigm for grid computing that although interesting has not received so far much attention from the community of users.

Wei et al. [27] explore the use of BitTorrent [10] for circumventing the scalability issues that arise with large data files. They conclude that BitTorrent is effective for deploying large files required by a significant number of workers. Our approach is different, since we aim to promote file sharing directly between worker nodes.

Several works resort to structured DHT overlay networks, for achieving different purposes. For instance, WaveGrid is a peer-to-peer desktop grid system aimed to achieve fast turnaround execution times [28]. It resorts to the peer based model, where all peers can submit applications to be executed over the desktop grid system. To enable communication within peers, WaveGrid makes use of a CAN DHT overlay network [19]. The WaveGrid approach introduces serious problems related with security and accountability.

Another DHT-based infrastructure is proposed by Butt et al. [7]. They present a technique which uses a Pastry DHT [20] for resource discovery in distributed Condor pools [25] spread over several administrative domains, to overcome the restrictions of the statically defined flocking mechanism supported by Condor.

Similarly to the work we present here, there are many other systems that use DHTs to manage data, from file systems to replicas of entire systems. For instance, Venti-DHash is a cooperative backup system, which couples the Venti backup system with an Internet peer infrastructure for archiving snapshots of file systems [23]. Venti-DHash uses DHash, which is a Chord-based distributed hash table (DHT). Pastiche [12] is a peer-to-peer backup system that resorts to a Pastry [20] DHT for the identification and organization of redundant data for saving space storage. Unlike our application-level checkpointing and unlike Venti-DHash, which acts at the block level, Pastiche makes replicas at the machine level.

Other interesting approaches to create file systems are Shark [3] and Kosha [6]. The main asset of Shark lies in its cooperative-caching mechanism, in which mutually-distrustful clients use a DHT to exploit each others' file caches to reduce load on an origin file server. Kosha aims to harvest unused storage of desktop machines within a LAN environment. It uses a structured overlay network to provide location and mobility transparency, load balancing, and file replication.

7. Conclusion and Future Work

In this paper, we used a DHT to extended the traditional desktop grid architecture with decentralized replicas of checkpoint files. With this technique any node in the grid can resume a failed task provided that the checkpoint file is available in the P2P infrastructure. Almost all interactions

needed to replicate checkpoints are decentralized among the DHT, thus containing the load on the central supervisor. To maintain the basic assumptions of existing architectures, we keep all the interactions involving the central supervisor strictly worker-initiated. Our preliminary simulations show that our proposed scheme can considerably reduce the turnaround time of tasks when there is a possibility of node failures. These results allow us to conclude that the use of P2P techniques in desktop grids seems to be a promising approach, which should be further researched.

We are currently implementing a PlanetSim simulation of our system, using the Chord [24] DHT. Additionally, one of the points that we intend to explore with greater detail in the future, is the use of techniques to increase the robustness of $Chkpt2Chkpt$ to failures caused by malicious users, either isolated or colluded.

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] D. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *6th International Symposium on Cluster Computing and the Grid CCGRID06*, Singapore, 2006.

[3] 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.

[4] D. Antonelli, A. Cordero, and A. Mettler. Securing distributed computation with untrusted participants. Technical report, University of California at Berkeley, 2004.

[5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 2003.

[6] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 51, Washington, DC, USA, 2004. IEEE Computer Society.

[7] R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of condors. In *Supercomputing 2003*, Phoenix, Arizona, 2003.

[8] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The MyGrid approach. In *2003 International Conference on Parallel Processing (ICPP'03)*, pages 407–416, October 2003.

[9] climateprediction.net (http://climateprediction.net), 2006.

[10] B. Cohen. Incentives build robustness in BitTorrent. In *First Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

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

[12] L. Cox, C. Murray, and B. Noble. Pastiche: making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, 36:285–298, 2002.

[13] M. Creeger. Multicore CPUs for the masses. *ACM Queue*, 3(7), September 2005.

[14] Distributed info (http://distributedcomputing.info/).

[15] 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.

[16] P. Domingues, J. G. Silva, and L. Silva. Sharing checkpoints to improve turnaround time in desktop grid. *AINA*, 1:301–306, 2006.

[17] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing system. In *1st Int'l Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 582–587, Brisbane, 2001.

[18] 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.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[20] 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)*, Nov. 2001.

[21] SETI@home (http://setiathome.berkeley.edu/), 2006.

[22] A. Singh, M. Castro, A. Rowstron, and P. Druschel. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[23] E. Sit, J. Cates, and R. Cox. A DHT-based Backup System. *Proceedings of the 1st IRIS Student Workshop*, 2003.

[24] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, San Diego, August 2001.

[25] 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.

[26] K. Tritrakan and V. Muangsin. Using peer-to-peer communication to improve the performance of distributed computing on the internet. In *19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, volume 2, pages 295–298, Mar. 2005.

[27] B. Wei, G. Fedak, and F. Cappello. Collaborative data distribution with BitTorrent for computational desktop grids. In *4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*, 2005.

[28] D. Zhou and V. Lo. WaveGrid: A scalable fast-turnaround heterogeneous peer-based desktop grid system. In *20th International Parallel & Distributed Processing Symposium (IPDPS)*, April 2006.