On Supporting Disconnected Operation in Grid Computing

Pavan Konanki and Ali R. Butt
Virginia Tech.
Blacksburg, VA 24061
{kpavan, butta}@cs.vt.edu

Abstract

In recent years, advancements in technology have made the use of mobile computers ubiquitous. The increasing capabilities of mobile devices forebode their playing a more involved role in computational grid environments. For example, they could assume the role of a temporary grid submission resource from which an application and data can be launched on the grid, and to which result-data is retrieved; or in the case where grid applications need remote data, they could act as a data source such as a sensor node. A major challenge in these emerging scenarios is that mobile devices are networked through intermittent wireless connections, and current grid environments cannot interact with resources that may disappear during job runs without costly application restarts/recovery, or worse may silently produce wrong results. In this paper, we address this issue and develop a transparent system that allows mobile devices to operate through network disconnections especially for remotely managing grid applications and data.

1 Introduction

Mobile computers and devices are becoming ubiquitous, and are used for tasks ranging from personal communication and entertainment to remotely managing business transactions. The increasing processing power of mobile computers makes them candidates for acting as computational Grid [7] resources. Instead of simply checking remote application status through web-based interfaces, the mobile computers of today can be used as active grid components: submitting and receiving application data, and providing critical data streams (e.g., sensor data, for consumption by other applications). A major hurdle in supporting such interactivity is the intermittent nature of wireless connections used by the mobile devices, as a disconnection may cause any application that depends on remote data to be suspended or to fail. Moreover, we cannot assume automatic continuation of work after connectivity is restored. Such automation requires special consideration and redesign of many grid components.

What is needed is an architecture that allows applications to continue or at least fail gracefully in the face of disconnections. Such a system should: (i) transparently handle connectivity failures; (ii) support continued operation by enhancing the availability of required data; (iii) provide a flexible interface that is applicable to a wide range of scenarios; (iv) utilize mobile resources judiciously; (v) support efficient synchronization of data upon reconnection; and (vi) keep manual intervention to a minimum. To this end, we propose a Dis-

connected Operation System (DCOS) that allows mobile resources to be integrated with the grid [1].

DCOS achieves disconnected operation through an innovative design that uses low overhead *client modules* running on the intermittently connected mobile devices, and *proxies* that are assumed to be always connected to the grid. The devices interact with the grid only through their associated proxies, which enables the devices to transparently operate over intermittent connections.

Disconnected operation has been supported by systems such as Coda [9] and Thor [6]. The novelty of our approach lies in its focus on grid applications that differ from traditional applications in that they are less interactive, long running, expensive to restart, and possibly data intensive. The existing systems are either impractical (simply fail) or extremely inefficient (use application restart) in such scenarios. To support disconnected operation for grid applications, we focus on: (i) removing centralized bottlenecks via the use of distributed proxies to enable the system to scale to a large number of nodes; (ii) minimizing overhead of bookkeeping operations, e.g., file-modification tracking; and (iii) keeping the framework independent of the file system and not necessitating kernel and middleware modifications.

2 System Design

We now present the design of DCOS. Each mobile device that wants to utilize DCOS services runs an instance of a DCOS client component. Each client component has an associated proxy component that runs on an always-connected, *wired*, grid node. The mobile device interacts with the grid only through the proxy. The client and proxy work together to make the network disconnections transparent to the applications on each end.

The DCOS operates in three phases. In the connected phase, the client prefetches data to the device in preparation for a possible disconnection. If the device looses connection, DCOS switches to the disconnected phase. In this phase, I/O's ("reads" and "writes") on the device are serviced from the cached data. Any data request from the grid for the data on the device is paused (delayed) at the proxy, which in essence prevents the grid request from failing. When the client regains connectivity, DCOS moves into the synchronization phase, where any modified data on the mobile device is synchronized with the grid, and any paused data requests from the grid are allowed to continue with the only noticeable effect of a delay.

Data prefetching and caching A key function of DCOS is to prefetch and cache data on the device so as to support continued operation in the face of a disconnection. While prefetching entire files simplifies the synchronization phase, the size of files vary from a few KBs to GBs and not all files are accessed in their entirety, making such an approach inefficient. Therefore, we chose to prefetch and cache files at the granularity of blocks.

The selection of an appropriate prefetching algorithm is crucial to the working of DCOS. We observed that the adaptive algorithm used by the Linux kernel works well for a variety of applications [4]. Therefore, we chose to use a similar algorithm in DCOS. However, if needed, the design of DCOS allows for easily plugging-in a different prefetching scheme.

Continued operation on cached data during disconnection entails synchronization upon reconnection. For this purpose, DCOS must track modifications to the cached data. Figure 1 shows an overview of the data structures used for tracking. The persistent *File Table* stores information about the files accessed on the client. It has an entry for each file modified during a disconnection, with tracking flags in-

[1] Although we target grid environments in this work, the flexible design of DCOS allows it use in other domains.

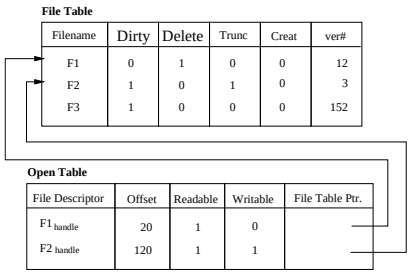

Figure 1. Data structures used in DCOS.

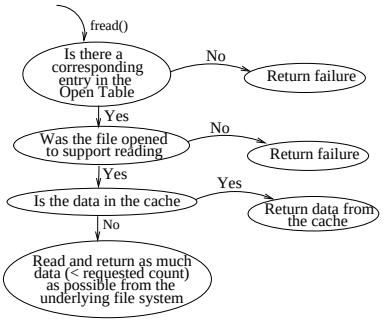

Figure 2. The sequence of events that follow an `fread` call.

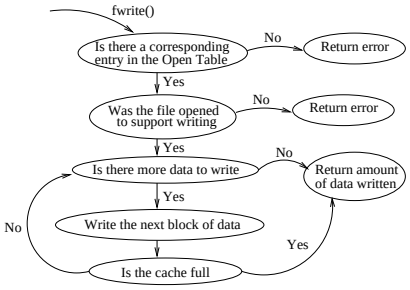

Figure 3. The sequence of events that follow an `fwrite` call.

dicating write, delete, truncate, and create operations. The *Open Table* stores information about the currently open files on the node. Figure 2 and 3 illustrate the uses of these data structures in handling two common operations, `fread` and `fwrite`. Note that DCOS does not require the exact time of a disconnection to be known for proper operation. This is an important design feature, especially in the presence of client-side caches that make instantaneous detection of disconnections (without kernel modifications) impossible.

Detecting conflicts It is possible that different clients end up writing to overlapping portions of a file while disconnected, leading to conflicts upon synchronization. DCOS maintains a per-file version number that indicates the current version of the file at the server. This number is incremented on each write at the server. The client fetches the version number along with the prefetched blocks, and stores this information in the *File Table*. In the synchronization process, the version number on the client is compared with that on the server to check if any other node has modified the file. Currently, all conflicts are flagged for manual inspection. However, we are working on applying automatic conflict resolution techniques [15, 1] to our system.

We adapt the `rsync` [14] algorithm for data synchronization in DCOS, which provides advantages such as optimal network bandwidth

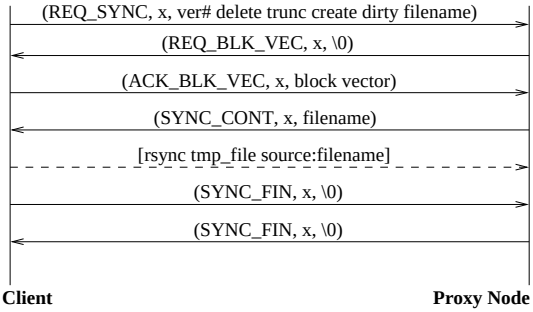

Figure 4. Messages exchanged during synchronization.

utilization when synchronizing files that differ only slightly. Figure 4 shows the key messages exchanged during the synchronization process.

3 Results

We have implemented DCOS using 7600 lines of C code that runs on Linux Kernel 2.6. The client component is implemented as an interposed library that intercepts the application I/O calls to the standard C Library [11], and does not require kernel modifications. Figure 5 shows the architecture of DCOS.

In order to detect conflicting updates to a file, the proxy registers a *listener* with the server. The *listener* signals the proxy if the file is changed. The proxy then uses this information to send a “cache-invalidate” message to its client if the client is connected, or buffers the message for delayed dispatch if the client is currently disconnected. Note that the *listener* is file system dependent, and our current implementation only supports the widely used

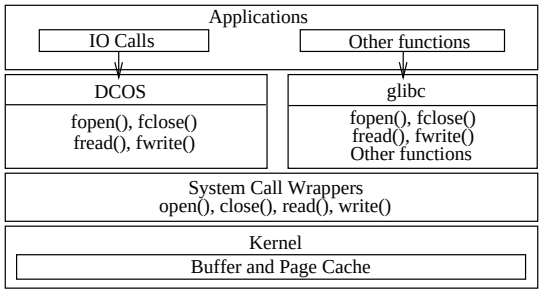

Figure 5. DCOS interactions with applications and the standard library.

Network File System [5][2].

Setup The experiments were conducted using a 32-node Condor [10] pool set up on departmental laboratory machines to emulate a grid environment. One machine served as the client that submits jobs to the pool, while another assumed the role of the client's proxy. The rest of the machines served as compute nodes in the pool. The focus of our experiments is to measure the amount of data transferred, which is directly proportional to the synchronization time.

Performance analysis To test the efficiency of the system for various scenarios, we prefetched and cached varying amounts of a data ranging from 512 KB to 2 GB. The cache was then subjected to a workload similar to the SPC-1 workload [8].

We calculated speedup as the total number of cached bytes of a file divided by the number of bytes transferred to synchronize the file. Figure 6 shows the speedup achieved for a representative sample of four different sizes of cached data as the amount of dirty data is varied. The results show that the data overhead is close to the lower bound for the studied cases.

Tracking overhead The overhead of I/O operations tracking is only 284 bytes per open file, as it is done through the File Table on the client and the version numbers on the data source. This approach is extremely lightweight and obviates the need to perform expensive log optimizations such as identifying and removing operations that cancel out each other.

Response to disconnections In this experiment, we set up a simulation environment of 1000 mobile clients, and distributed files from a file system trace (collected from our departmental machines) to the simulated clients. The sizes of the client-side caches were based on normal distribution with a mean of 512 MB. Upon disconnection, the clients dirtied data based on a Zipf [2] distribution.

Next, to simulate availability of various clients, we used the availability trace of 51663 machines in a large corporation over a consecutive 35-day (840-hour) period [3]. The trace contains the status of machines (up or failed) recorded hourly. We randomly selected 1000 nodes from the trace and used that information to simulate client disconnection and reconnection. We treated each hour as a simulation time unit, and for each time unit determined the number of clients that were reconnected and the amount of data that was transferred for synchronization. The results in Figure 7 show that the data transferred during synchronization is on average 4.85% of the total data stored on each client. This number implies that the cost of synchronization is also small.

In the next set of experiments, we evaluated the effectiveness of using an adaptive prefetching scheme for our DCOS cache. We used the same simulation setup as the one used before, but now loaded the clients, first, with a sequential application *cscope* [12] that access the files in looping references, and then with a random application *tpc-h* [13] that randomly accesses the files. For both cases, we recorded the number of misses for which our prefetcher failed to prefetch the data while connected, and hence resulted in the applications on the client to stop. Figure 8 shows the result. As expected, for *cscope* the prefetcher is able to prefetch the

[2]The file-system-dependent code is a small fraction of the overall code and support for other file systems can easily be added.

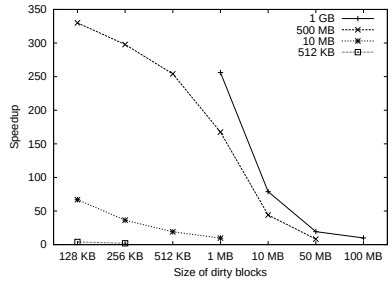

Figure 6. Speedups achieved during synchronization.

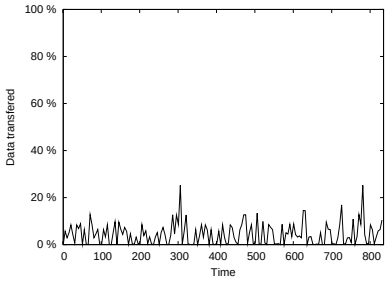

Figure 7. Amount of data transferred as clients failed and reconnected.

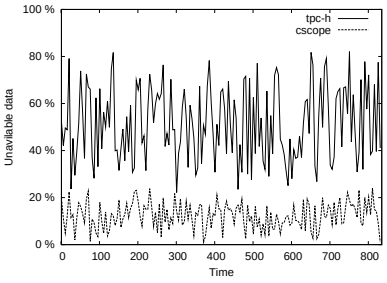

Figure 8. Amount of unavailable data under test workloads during disconnections.

data and provide better performance. However, for *tpc-h* the data accessed is fairly random and our adaptive prefetcher did not perform well resulting in a larger percentage of application data being unavailable. One possible solution is to design better prefetching schemes for random applications. Designing such schemes is orthogonal and complimentary to the proposed system, but is out of the scope of this paper.

4 Conclusion

We have presented the design and implementation of a system for supporting disconnected operation in grid environments, which is independent of the underlying file system to a large extent and obviates the need for kernel modifications. By using a proxy for each mobile device we are able to provide a transparent library that interposes itself between the applications and the kernel and manages disconnections through a combination of prefetching, caching, and synchronization.

References

[1] Concurrent Versions System. `http://www.nongnu.org/cvs/`

[2] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3(1):143–150, 2002.

[3] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS*, June 2000.

[4] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. SIGMETRICS*, June 2005.

[5] B. Callaghan. *NFS Illustrated*. Addison Wesley Longman, Inc., 2000.

[6] S. Chang and D. Curtis. An approach to disconnected operation in an object-oriented database. In *Proc. MDM*, 2002.

[7] I. Foster (Ed.) and C. Kesselman (Ed.). *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[8] B. S. Gill and D. S. Modha. Wow: Wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proc. USENIX FAST*, Dec. 2005.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[10] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. ICDCS*, June 1988.

[11] P. J. Plauger. *The standard C library*. Prentice-Hall, Inc., 1992.

[12] J. Steffen. Interactive examination of a c program with cscope. In *Proc. USENIX Winter Technical Conference*, Jan. 1985.

[13] Transaction Processing Council. `http://www.tpc.org`

[14] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, Canberra, Australia. Feb. 1999.

[15] A. Yip, B. Chen, and R. Morris. Pastwatch: a distributed version control system. In *Proc. 3rd USENIX NSDI*, May 2006.