PeerStripe: A P2P-Based Large-File Storage for Desktop Grids

Chreston Miller, Patrick Butler, Ankur Shah, and Ali R. Butt
Dept. of Computer Science, Virginia Tech.
Blacksburg, VA 24061
{chmille3, pabutler, ankur77, butta}@cs.vt.edu

ABSTRACT

In desktop grids the use of off-the-shelf shared components makes the use of dedicated resources economically nonviable and increases the complexity of design of efficient storage systems that are required to address the exponentially growing storage demands of modern applications that run on these platforms. To address this challenge, we present PeerStripe, a storage system that transparently distributes files to storage space contributed by participants that have joined a peer-to-peer (p2p) network. PeerStripe uses structured p2p routing to yield a scalable, robust, reliable, and self-organizing storage system. The novelty of PeerStripe lies in its ingenious use of striping and error coding techniques in a heterogeneous distributed environment to store very large data files. Our evaluation of PeerStripe shows that it can achieve acceptable performance for applications in desktop grids.

Categories and Subject Descriptors: D.4.7 [Organization and Design]: Distributed systems

General Terms: Design, Reliability, Experimentation.

Keywords: peer-to-peer, storage, striping, error-coding, desktop grids.

1. INTRODUCTION

In this paper, we propose PeerStripe, a p2p storage system that provides an economical and efficient storage solution for large data files. PeerStripe supports an elegant and simple design that allows for files to be stored on participating nodes that have joined a p2p overlay network. Our use of p2p networks ensures that PeerStripe has the features of scalability, self-organization, reliability, and composability for target environments of various sizes. Inspired by the local-area technique of RAID [3], a unique feature of PeerStripe is that instead of storing entire files on individual nodes, it splits the files into varying sized chunks and then stores these chunks separately on nodes distributed across a wide-area network. As a result, unlike previously proposed approaches such as PAST [6], the size of a file that can be stored in PeerStripe is not limited by the capacity of an individual node. Moreover, to provide fault tolerance against data loss due to losing a chunk of a distributed file, PeerStripe employs error coding at the granularity of chunks.

HPDC'07, June 25–29, 2007, Monterey, California, USA.
ACM 978-1-59593-673-8/07/0006.

2. DESIGN

First, PeerStripe uses the communication substrate provided by Pastry [5] to arrange the participating nodes in a p2p overlay network and enable storage sharing.

Second, to store a file PeerStripe splits it into chunks and stores the chunks in the storage pool. This enables PeerStripe to store very large data files as well as only retrieve needed portions of a file and not the entire file.

Third, PeerStripe employs erasure codes [2, 4] to provide fault tolerance against loss of chunks. Each chunk is named as *filename_ChunkNo*, e.g., *testImageFile_2* represents the second chunk of the file *testImageFile*. This naming convention is chosen as a means for easily determining the name of the file a chunk belongs to. Each chunk is then encoded using erasure codes. A chunk to be encoded is passed to an error coding algorithm that divides the chunk into n equal size blocks, calculates erasure codes across the blocks, and generates m encoded blocks. The encoded blocks for the chunk X are named *filename_X_ECB*, where ECB is the error coded block number and ranges from 1 to m. The error coded blocks are then stored in the p2p storage system using techniques similar to that used in PAST [6]. Due to the built-in redundancy of erasure codes, PeerStripe can retrieve the original chunk even if some of the m encoded blocks are lost due to failures of some kind.

Fourth, to determine the size of a chunk, PeerStripe uses our chunk naming convention and the information about the currently used erasure codes, and queries the nodes on which the encoded chunks will be stored using a `getCapacity` message. A response to the `getCapacity` message is the maximum amount of data a remote node can store. PeerStripe uses this information as the first chunk size. The process repeats until all the chunks of the file are stored.

In summary, to store a file, it is first split into chunks, and each chunk is then divided into n blocks and error coded to give m encoded blocks. The encoded blocks are then stored in the PeerStripe shared storage pool as shown in Figure 1. Similarly, to retrieve any portion of a file, PeerStripe first determines the number of the chunk to retrieve and the name of the required encoded blocks using our naming convention. Next, enough blocks per chunk are retrieved to allow decoding of the chunk. These chunks are then assembled together and returned to the user.

3. EVALUATION

We implemented PeerStripe with about 6000 lines of Java code using a freely available version of Pastry [5]. For these tests, we simulated a 500-node directly connected PeerStripe

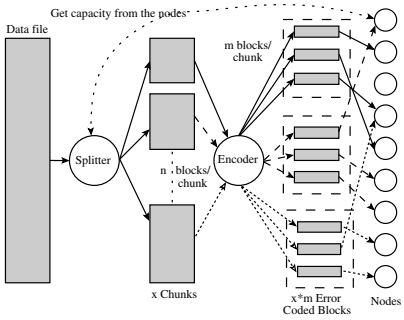

Figure 1: The various steps of storing a file in PeerStripe.

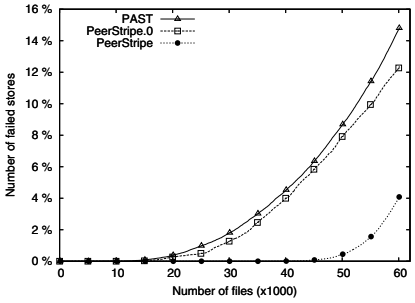

Figure 2: Total number of failed file stores as files are inserted.

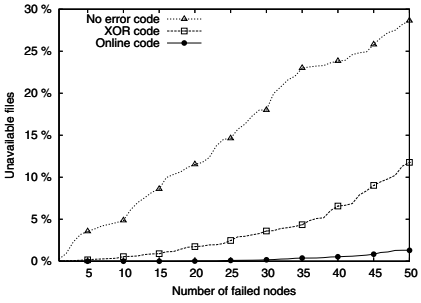

Figure 3: The number of unavailable files as nodes fail.

network using Pastry's built-in simulator. We also set up PAST [6] to run in our simulator environment. Moreover, we compared PeerStripe against a simpler scheme (PeerStripe.0) that used fixed size chunks of 4 MB.

Based on a recent study of available space on typical desktop machines [1], each simulated node was assigned a capacity using a normal distribution with mean and standard deviation of 8 GB and 2 GB, respectively, with a total capacity of 3915 GB. We drove our simulations using a file system trace collected from our departmental machines, which contains information for about 60k files with sizes ranging from 4 MB to 15.23 GB, with a mean and standard deviation of 39.4 MB and 247.4 MB, respectively. The total storage size required to store all the files in the trace is 2311 GB.

In the first set of experiments we measured the number of successful file stores as files from the trace were inserted into the system. Figure 2 shows the results for the three cases of PAST, PeerStripe.0, and PeerStripe. We observed that as the system utilization increases, the number of failed stores in PAST starts to increase, and it fails to store 14.8% of the total files. Similarly, PeerStripe.0 is able to perform slightly better, although it still fails to store 12.4% of the total files. Finally, PeerStripe is able to remedy the ill-effects of both PAST and PeerStripe.0, and results in only 4.1% failures; an improvement by a factor of 3.6 and 3.0 compared to PAST and PeerStripe.0, respectively. In terms of size, we observed that PAST and PeerStripe.0 are unable to store as much as 24.7% and 19.8% of the total size of data, respectively. In contrast PeerStripe failed to store 12.2% of the total data. This is an improvement by a factor of 2.0 and 1.6 compared to PAST and PeerStripe.0, respectively.

In the next experiment, we determined the number and size of chunks created under PeerStripe.0 and PeerStripe. On average, PeerStripe.0 created 2.6 times the number of chunks created under PeerStripe. The respective size of chunks was a factor of 5 larger in PeerStripe. The reduced number of chunks enables PeerStripe to avoid an unnecessarily large number of slow p2p look-ups, and shows the advantage of using varying size chunks over fixed size chunks.

Next, we evaluated the effectiveness of error coding in PeerStripe by distributing the files to the nodes and counting the total number of available files in the system as 50 randomly chosen nodes fail one-by-one. For this experiment we used a (2,3) XOR code as well as an online code that could tolerate two simultaneous failures per chunk. We counted a file as available only if all the chunks of the file could be retrieved. Figure 3 shows the percentage of total files that became unavailable as nodes failed under the three cases. The use of error coding resulted in 58.9% and 95.5% less failures for XOR code and online code, respectively, when 50 nodes failed. The overall number of failures for online code was negligible (1.3%), and almost zero for up to 25 failed nodes. Hence, error coding is an effective means for ensuring fault tolerance in PeerStripe.

In the next experiment, we determined the effect of participant churn on PeerStripe by studying the amount of data that is regenerated from other replicas/error-coded chunks as nodes leave the system due to failure. We failed up to 20% of the total participating nodes without any node recovery. We found that on average 3.43 GB of data was regenerated per failure after up to 20% of the nodes had failed, with a total of 343 GB being regenerated, and 9.17 MB of data lost under this extreme case. Finally, compared to the total data size of 2311 GB, the data recreated per failure is quite small, i.e., 0.15%. This shows that PeerStripe can handle participant churn well.

4. CONCLUSION

In this paper, we have presented the design and evaluation of PeerStripe. PeerStripe uses p2p overlay networks to establish a robust, scalable, and reliable distributed storage system. It employs the techniques of striping and error coding to support transparent storage of very large data files across distributed nodes, and exports a simple yet effective interface to users and applications. The evaluation of the system shows that it can store files that are larger than the capacity of individual participants, and gives acceptable performance in a dynamic setting. The efficient and simple design of our approach implies that it can be readily deployed and interfaced with different applications, and therefore can serve as a storage system for today's desktop grid environments.

5. REFERENCES

[1] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A Peer-to-Peer Enhancement for the Network File System. *Journal of Grid Computing*, 4(3):323–341, 2006.

[2] P. Maymounkov. Online Codes. Technical Report TR2003-883, New York University, Nov. 2002.

[3] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. SIGMOD*, 1988.

[4] J. S. Plank. Erasure codes for storage applications. Tutorial at *USENIX FAST-2005*, 2005.

[5] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.

[6] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, 2001.