Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-00-07

2000-01-01

Data Archiving with the SRB*

Jinghua Zhou

We use the SRB (Storage Request Broker) middleware to design and implement a storage archival system which will be used to archive Neuroscience data. As part of the design process, we developed and used an experimenter's workbench to measure SRB performance. These experiments improved our understanding of both the functionality and the performance of the SRB. This technical report describes the scripts in the experimenter's workbench, the archiving scripts, and performance measurements.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Zhou, Jinghua, "Data Archiving with the SRB*" Report Number: WUCS-00-07 (2000). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/284

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Data Archiving with the SRB*

Jinghua Zhou

WUCS-00-07

May 2000

Dept. of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130

Data Archiving with the SRB*

Jinghua Zhou
Dept. of Computer Science
Washington University
St. Louis, MO 63130

May 4, 2000

Abstract

We use the SRB (Storage Request Broker) middleware to design and implement a storage archival system which will be used to archive Neuroscience data. As part of the design process, we developed and used an experimenter's workbench to measure SRB performance. These experiments improved our understanding of both the functionality and the performance of the SRB. This technical report describes the scripts in the experimenter's workbench, the archiving scripts, and performance measurements.

1. Introduction

1.1 Motivation:

The Washington University NPACI (National Partnership for Advanced Computing Infrastructure) Program is focused on applying advanced technology towards some of the difficult tasks associated with brain mapping. In particular, it is cooperating with other NPACI partners in constructing a federated brain map database.

One challenging dimension deals with accommodating the tremendous rate of data acquisition from PET and MRI scanners: 20 GB per week. At this rate, the local cache will be exhausted in about 15 months. To reduce the impact of less active data sets on our data cache, a storage archival system using SRB (Storage Resource Broker) middleware was designed, implemented and evaluated. Routine procedures for archiving inactive data sets at our local data cache brainmap.arl.wustl.edu to the HPSS

* This research was sponsored by the National Science Foundation as part of the National Partnership for Advanced Computational Infrastructure (NPACI), contract ASI-9619020, and administered by the San Diego Supercomputer Center.

(High Performance Storage System) located at the SDSC (San Diego Supercomputer Center) were developed.

1.2 Technologies:

This project focuses on the design and implementation of a prototype storage archival system that involves four advanced component technologies.

- A terabyte RAID at Washington University as a local storage.
- A remote petabyte HPSS (High-Performance Storage System) at the San Diego Supercomputer Center (SDSC) as the archival storage system.
- A high-speed network -- vBNS (very high performance Backbone Network Service), a high-performance and high-bandwidth wide area network for advanced applications.
- The SRB (Storage Request Broker) middleware, which provides a uniform storage access interface to heterogeneous storage system.

1.3 Hurdles:

The archiving in this project is different from the traditional view of "archiving" in which objects are catalogued. Here there is no agreement among the neuroscientists about how the neuroscience data should be catalogued. Also, it is important to archive the data while keeping all data in their original file structure. Furthermore, we must accommodate the tremendous rate of data acquisition cache (about 20 GB per week during 80% of the weeks of last year). At this rate, the present capacity of local storage will be exhausted in about 15 months. An additional complication is that there are a large number (many thousands) of small files. "ftp" is too cumbersome and would require the user to remember the exact physical locations of files. Also, standard utilities do not help neuroscientists to easily exchange data among a diverse collection of tape archives, file systems, databases and digital libraries. However, the SRB middleware provides the facilities for data sharing and data distribution among diverse storage systems.

1.4 Accomplishments:

The major components of this project are shown below:

- Experiments: (more than 10 perl scripts)
  - Test S-command functionality
  - Test Container feature.
- Archiving: (more than 6 perl scripts)
  - Timing measurements
  - Neuroscience data archiving
  - Archival correctness verification
  - Data retrieval

1.5 Roadmap:

This report is organized as follows. Section 2 presents the technologies associated with the archival system in more details. Section 3 describes the archival system design, implementation, and evaluation. It also shows all experimental results and what work has been done with archiving. Section 4 gives an archiving guide and leads through running the archiving scripts. Section 5 leads a user through an experimenter's workbench guide.

2. Technologies

In the archival system, four advanced technologies are used:

- A terabyte local storage --RAID.
- A remote petabyte archival storage system -- HPSS.
- A high-speed network -- vBNS.
- A middleware – SRB.

More details about these four advanced technologies are described below.

2.1 Local Data Cache:

The Washington University neuroscience terabyte data cache is hosted on a 2-processor Sun Enterprise 450 with a DEC Storage Works RAID and 1GB of DRAM. This host (brainmap.arl.wustl.edu) is accessible through an OC-12 (622Mbps) ATM interface and 100Mbps Ethernet interface. The disk storage capacity is currently about 1.6 terabyte. This data cache holds the candidate data we are going to archive. Fig. 1 shows the network connectivity from the local cache to the archival storage system (HPSS), the local cache host (brainmap), and part of the WU network used by the neuroscientists. Basically, neuroscientists generate huge amounts of data everyday at the Washington University School of Medicine (WUSM). The largest data files are images, and the smallest are text files. As shown in the graph, there are several scanners affiliated with petsun-23. These scanners obtain images by scanning various brains. Also, the hosts stp.wustl.edu and v1.wustl.edu provide access to the SUMS (SUrface Management System) which provides Web access to brain surface data. Large amounts of data are generated, processed and pumped through the local ATM network to the local data cache brainmap.arl.wustl.edu. Because of the high rate of local data acquisition (around 20GB per week), the brainmap storage system will eventually not be large enough to hold all of the data. However, there is a petabyte high performance storage system (HPSS) at the San Diego Supercomputer Center (SDSC) which has a capacity of around 400TB. Also, the SRB middleware developed at SDSC provides a uniform access interface to heterogeneous storage system. Part of the SRB is the MCAT (Metadata CATalogue) server at ghidorah.sdsc.edu. So through the very fast Backbone Network Service (vBNS), the data at brainmap can be stored to the remote site HPSS using the SRB.

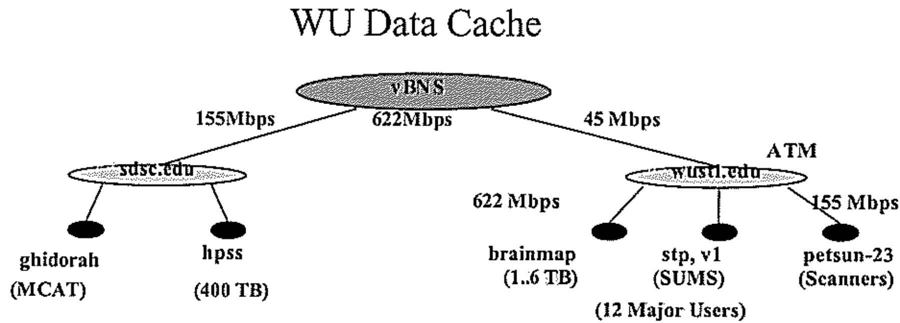

Fig. 1. Network connectivity from WU local cache to the archival storage system (HPSS)

2.2 Archival Storage (HPSS):

A remote petabyte HPSS (High-Performance Storage System) at the San Diego Supercomputer Center (SDSC) acts as the current archival storage system. The HPSS architecture, based on the IEEE Mass Storage Reference Model, is network-centered, including a high-speed network for data transfer and a separate network for control. The control network uses the DEC's Remote Procedure Call technology. In actual implementation, the control and data transfer networks may be physically separated or shared. An important feature of HPSS is its support for both *parallel* and *sequential* input/output (I/O) and standard interfaces for communication between processors (parallel or otherwise) and storage devices. In typical use, clients direct a request for data to an HPSS server. The HPSS server directs the network-attached storage devices to transfer data directly, sequentially, or in parallel to the client node(s) through the high speed data transfer network. The system can support any network environment, which provides either socket or IPI-3 interfaces. Fig. 2 shows the HPSS system architecture.

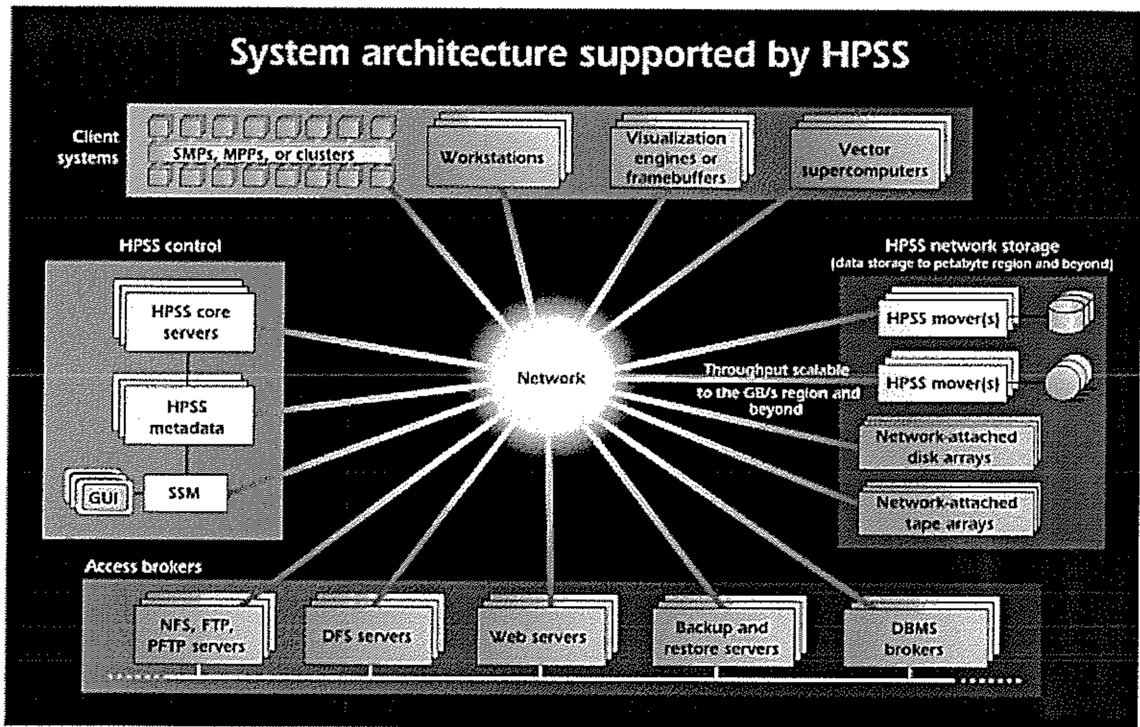

Fig. 2. The HPSS system architecture

2.3 High-Speed Network(vBNS):

vBNS (very high performance Backbone Network Service), is a high-performance and high-bandwidth wide area network for advanced applications. It operates at a speed of 622 Mbps (OC12) using MCI WorldCom's network of advanced switching and fiber optic transmission technologies. The vBNS relies on advanced switching and fiber optic transmission technologies, known as Asynchronous Transfer Mode (ATM) and Synchronous Optical Network (SONET). The combination of ATM and SONET enables very high speed, high capacity voice, data, and video signals to be combined and transmitted "on demand". The vBNS' speeds are achieved by connecting Internet Protocol (IP) through an ATM switching matrix and running this combination on the SONET network.

2.4 Middleware (SRB):

The SRB (Storage Request Broker) middleware provides a uniform storage access interface to heterogeneous storage system and glues together the data-handling environment. The design of the SRB server is based on the traditional network connected client/server model. The SRB server takes requests from applications through an application program interface, queries a meta data catalog for the physical location of the requested data, and accesses the data using the appropriate protocol via a resource-specific driver. The SRB hides the low-level details of accessing each store, and the SRB file

API provides a common Unix file-like interface to storage regardless of the underlying storage system, medium, or location.

SRB architecture

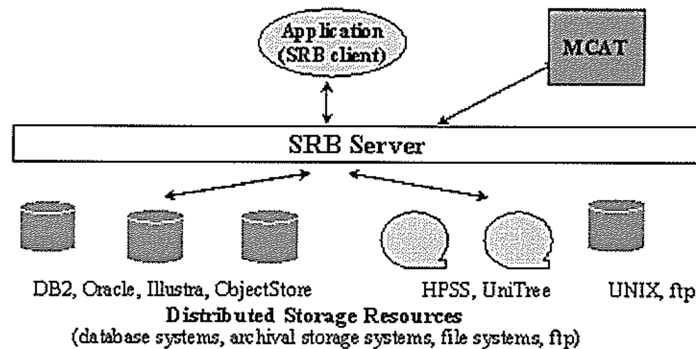

Fig. 3. A simplified view of the SRB middleware

Fig. 3 gives a simplified view of the SRB architecture. The model consists of four components: the distributed and diverse storage system, the meta data catalog (MCAT) service, SRB servers and the SRB clients.

The MCAT stores meta data associated with data sets, users and resources managed by the SRB. The MCAT server handles requests from the SRB servers. These requests include information queries as well as instructions for meta data creation and update. Client applications are provided with a set of API for sending requests and receiving response to or from the SRB servers. The SRB server is responsible for carrying out tasks to satisfy the client requests. These tasks include interaction with the MCAT service and performing I/O on behalf of the clients. A client uses the same common API to access every storage system managed by the SRB. The complex tasks of interacting with various types of storage system and OS/hardware architecture are handled by the SRB server.

This data archiving project used version 1.1.4 of the SRB. The MCAT server was running at ghidorah.sdsc.edu to handle requests from SRB severs, and the HPSS server was running at hpss.sdsc.edu. The current version of the SRB is 1.1.6, and version 1.1.7 is coming soon.

3. Accomplishments

3.1 Overview

This data archiving project includes two parts: experiments and archiving. In the first part, experiments, we characterize the SRB's functionality, quantify the effect of key parameters on the performance of basic SRB operations to aid us in the design and evaluation of the archival system. The archiving part includes the design, implementation, and evaluation of our archival system.

In the experiments, we tested typical S-command usage, checked container size effects, verified SRB functionality and performance. In the design and implementation of the storage archival system, the Unix file system structure was reflected in the SRB collection structure. We can perform timing measurements for archiving each file, archive any directory which may have sub-directories, get archived data back from SRB storage system, test the archival correctness to make sure the right file with exact size is archived, and also get attributes of each archived file.

In a nutshell, then, the products of our effort include the following:

Experiments

- Perl scripts to test SRB functionality using SRB_1.1.4 version
- S-command usage and performance measurements
- Container performance measurements

Archiving

- Perl scripts for archiving, verification, and restoring
- Timing measurements of archiving performance using SRB_1.1.4 version
- Neuroscience data archival performance measurements

3.2 Experiments

3.2.1 Main SRB Concepts

The SRB middleware allows us to access a geographically distributed, heterogeneous storage system. Performance measurements of the two main SRB functions Sget and Sput were conducted to gain an understanding of their limitations in a WAN (Wide Area Network) setting. Before describing the experiments, we will introduce some important SRB concepts. Many of the concepts are similar to Unix file system concepts.

Collection: A collection is like a Unix directory. But unlike a directory, a "collection" is a logical name given to a set of data sets and is not limited to a single device or partition. The data sets grouped under a collection can be stored in heterogeneous storage devices.

Container: A container is like a set of contiguous disk blocks. All objects in a container are stored contiguously in physical storage. The archival storage system HPSS is a hierarchic storage system with two components: a disk cache and a magnetic tape which serves as the permanent storage. Data going to the HPSS is first stored in the disk cache for some number of days, and then eventually flushed to the magnetic tape (how often the disk cache is flushed is unknown). The HPSS allocates space in chunks of physical space that are atleast 8 KB. A high overhaead is associated with accessing the magnetic tape component (almost 100 seconds or more) after a disk copy has been flushed to tape storage. If there are lots of small files to be stored into the HPSS, the total tape access latency will be tremendous, and there will be large internal space fragmentation within each 8 KB storage chunk. Containers were designed to hide this high latency by allowing the user to pack small files into a single block (container) so that the container can be accessed as a single physical object.

There are two physical resources associated with a container: one plays the role of a cache, and the other acts as a permanent storage system. Usually the cache resource is a Unix file system, and the permanent storage system is the HPSS (with its own disk cache and backing tape store). After you add files to the container cache, you can flush the cached files in the container to the permanent storage system (e.g., the HPSS). Furthermore, if the container becomes completely filled, the SRB automatically renames the container by appending a time stamp and creates a new container with the original name. This automatic extension of containers is transparent to the user.

Object: An object is a file registered in the MCAT. It is the smallest storable unit in the SRB space.

S-command: An S-command is a Unix command which implements an SRB function. There is a corresponding function supplied as part of the C-language API. The primary S-commands used in our work include Sget/Sput for getting/putting data from/to the HPSS, Smkdir/Srmdir for making/removing a data collection, Smkcont/Srmcont for making/removing a container, Sinit/Sexit for beginning/ending an S-command session, and Ssyncont for flushing a container cache to permanent storage. These commands are described in more detail later.

The SRB system provides three user interfaces: a programmatic API, a command language interface (S-commands), and a web browser. With the programmatic API, users can write C or C++programs. But because this interface takes more programming time, we chose to use the S-command interface,

and wrote Perl scripts. Because these scripts were quite short and concise, less time was spent on programming.

3.2.2 Experiments and Results

In the SRB experiments, we tested the SRB's functionality, and measured its performance over a WAN. We were interested in answering several questions. First, what was the basic overhead of using a storage system middleware? Second, what was the effect on performance of moving data over a WAN that spanned approximately 2500 miles, the distance from St. Louis to San Diego via Chicago? Third, how well would the container abstraction hide the HPSS latency? We focused our performance measurements on the Sget and Sput S-commands since they were key operations in the archiving and restoring of data files. In the experiments and data archiving, we played the role of an SRB client that stored/retrieved data to/from the HPSS at hpss.sdsc.edu using the MCAT at ghidorah.sdsc.edu (i.e., the SRB host was ghidorah.sdsc.edu, and the SRB resource was *hpss*). Whencontainers were used, the container cache was at ghidorah.sdsc.edu (i.e., the SRB host was ghidorah.sdsc.edu and the SRB resource was *cont-sdsc*).

Several environment variables and parameter settings determine the overall performance of the SRB. The SRB client sends request to an SRB host which in turn goes to an MCAT server for metadata information (e.g., physical object location and access permissions). In our experiments, ghidorah.sdsc.edu was both the SRB server and MCAT server. In fact, ghidorah is the only MCAT server available. Ghidorah was chosen as the SRB host because performance is improved when the SRB server is colocated with the MCAT server. However, we did informally try other SRB servers to confirm this fact. The container resource was cont-sdsc which is a logical resource name for a Unix cach at ghidorah combined with the HPSS permanent storage at hpss.sdsc.edu. In most experiments, the performance results correspond to the case when the data was in the disk portion of the HPSS. The specific situation is described in the discussion of each experiment below. Finally, the TCP buffer size at brainmap.arl.wustl.edu was initially 32KB but was later increased to 256KB.

The Sput S-command puts local data to a remote site, and the S-command *Sget* gets an archived file back to local storage. These two commands are the main commands tested in our experiments. Our results show that *Sput* and *Sget* had similar performance when the TCP buffer size on brainmap.arl was 32KB. Recent preliminary experiments with a 256KB TCP buffer at brainmap.arl, the SRB client, showed that Sget performance was three to five times better than Sput peformance for large files. Our current thinking is that this is a result of assymetric routing. A measurement effort led by SDSC is just beginning to look at this issue and other related ones. This report shows only the *Sget* results since most of our measurements were done with a 32KB buffer size, and its performance was almost identical to that of the Sput measurements.

Fig. 4 shows the average *Sget* time and average bandwidth for retrieving a file from a collection stored in the HPSS disk cache (without a container). The average Sget time is shown in the left graph. Each point is the result of averaging the timing results from 10 executions of the same data retrieving operation. Data variation bars are not shown since the time spent for each retrieval showed very little variation. The right graph came from the computation: filesize/average Sget time.

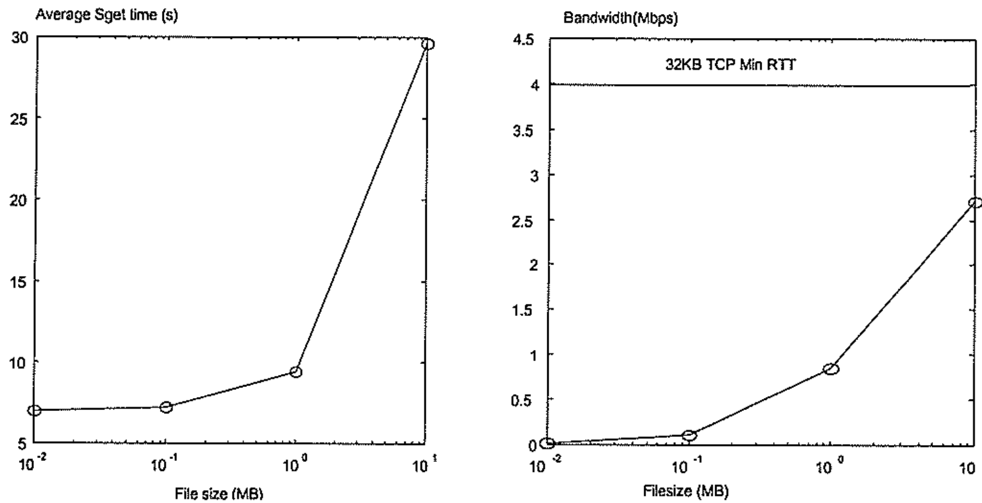

Fig. 4. Performance of SRB in retrieving a file from a collection.
Left: average *Sget* time versus file size. Right: average bandwidth versus file size

When the TCP buffer size was 32KB, we expected the maximum achievable bandwidth to be 4Mbps since the minimun RTT was 64ms (4 Mbps = 32KB/64ms). The figure illustrates that there is a large overhead (disk overhead from HPSS) of about 6 seconds for getting back each data set from a collection. Each data access requires accessing the MCAT for meta data information which takes time and results in the large overhead. The overhead typcially ranges from 5 to 7 seconds. However, this overhead is less significant for larger files when the transmission time dominates. In the right figure, one can see the bandwidth increases as file size grows. The experiment shows that it is more economical to archive large files.

Now let's compare the times used to retrieve a file from a collection and a container to see if the use of containers is worthwhile. In Fig. 5, there are two container curves in each graph because the SRB developers changed the container implementation in early April 2000, and we increased the TCP buffer size from 32KB to 256KB. The line marked "from container (initial)" shows the time (or bandwidth) to get data back from a container before the container implementation and TCP buffer size were changed. The line marked "from container" was collected after the change. The line marked

"from collection" shows the time (or bandwidth) to get data back from a collection; i.e., a container was not used. Because this time did not change much after the TCP buffer size was changed, only one line is shown in each graph.

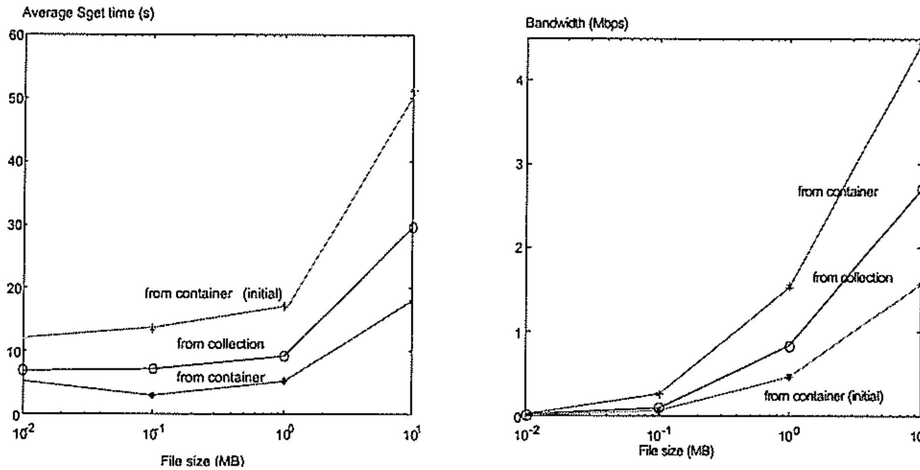

Fig. 5. Performance comparison of SRB in retrieving a file from a collection, from a container (initial, before the new implementation of container), from a container (after the new implementation). Left: average *Sget* time versus file size. Right: average bandwidth versus file size

Fig. 5 clearly shows that the current container implementation is superior to its earlier implementation and in fact, is faster than not using a container (the collection curve). For example, the minimum time to get an object from a collection is about 6 seconds (the "from collection" curve). But the minimum time to get an object from a container (that is stored in a collection), is about 3 seconds (the "from container" curve). Prior to the latest container implementation, the minimum Sget time was about 12 seconds (the "from container (initial)" curve).

Although not shown in these curves, the SRB developers claim that containers are also useful in hiding HPSS down time when writing to the HPSS since the container can be stored at the SRB host rather than the HPSS. Only when the container is flushed to the HPSS (using the Ssyncont S-command) does the HPSS need to be up. Furthermore, when reading from the HPSS, containers improve the overall access time if the access pattern follows the file clustering defined by the container loading. That is, files stored in the same container can be accessed as a group in only one HPSS tape access. This high penalty is amortized over every file access to the same container.

But how does container size affect the performance? Here, the container size is the upper bound on the container content size (i.e., the sum of the sizes of the files in the container). When a container is retrieved, the transmitted size of the container is the content size, not the container size. For example, a container can have a declared size of 100MB but only contain 30MB of files. In this case, only 30MB

of space is consumed, not 100MB. In Fig. 6, the left graph shows the average time to get a 1MB file back from a container when the data are stored in the HPSS disk (cache). The right graph shows the time after data has been pushed to the magnetic tape. You can see that the tape access overhead is about 100 seconds more for each file compared to the disk only access time. Here we have several 10MB and one 100MB containers fully filled with 1MB files. When the data is in the HPSS disk, it's better to use a larger container size (100MB) if you want to access more than 37 objects in a container; otherwise, you have to spend more time to access several small containers to get all files back. But when the data is in the HPSS tape system but not its disk cache, it is better to use a 100MB container if you are going to access more than 10 incontainer files (an *incontainer file* is a file stored in a container). So the effect of the container size depends on how many incontainer files you plan to access. You need prior-knowledge about the archival data access if you plan to take advantage of the container feature to reduce retrieval time.

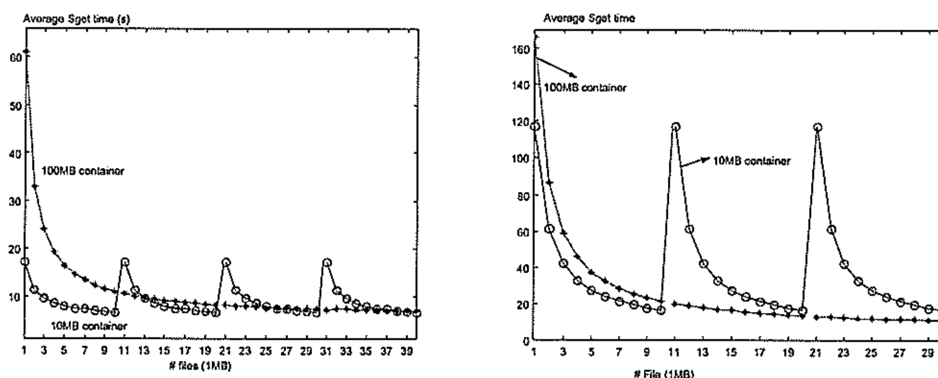

Fig. 6. Effects of container size on the SRB performance

Left: average *Sget* time versus file size. Right: average bandwidth versus file size

Now that we have an idea about the basic performance of the main S-commands, we will relate the performance results to the file size distribution of the neuroscience data. Table 1 gives a simple overview of the distribution of file sizes stored on brainmap.arl. In this table, a collection corresponds to a Unix directory. So, A through D in the table represents the Unix directories of four users. The distributions of the four directories are representatives of the local file system distribution. The surprising result is that there are a large number of small files! These small files are really important to us since it will affect our design and implementation of the data archival system. The next table will indicate the effect these small file sizes on performance.

	File numbers in Collection			
File Size Ranges	A	B	C	D
0 - 1KB	366	3,821	151	2,976
1KB – 10KB	794	7,346	3,362	969
10KB – 100KB	323	4,506	3,579	53,808
100KB -1MB	4,413	1,306	29,306	1,130
1MB – 10MB	0	780	1,846	1,718
10MB – 100MB	0	42	77	104
100MB – 1GB	0	7	0	5
>1GB	0	2	0	0

Table 1: File size distribution in our local storage system

Table 2 compares the SRB overhead time component with the transmission time component of the *Sget* time for the file sizes represented by collection D from Table 1. We show an overhead of 2 seconds which is the current minimum overhead under the best circumstances (i.e., uses containers). Although this is an optimistic overhead, continued improvements in the SRB implementation should make this a typical value soon. The transmission times are based on a single file of size equal to one-half of the upper end of the file size range (e.g., 5 KB for the range 1 KB to 10 KB) and using bandwidth values of 4 Mbps, 10 Mbps, and 32 Mbps. For example, if the file size range is 1-10KB and the overhead for each file is 2 seconds, the total overhead time in seconds is the number of files times 2 second, and the transmission time in seconds is the number of files times the transmission time of a 5 KB file: #files * (5 KB/Bandwidth). For the file size range 1-10KB, the overhead is 0.5 hours = 969*2 sec, and the transmission time is 9.7 seconds = 969*(5KB/4 Mbps). In this table, we listed three bandwidths: 4 Mbps was the maximum bandwidth we achieved when TCP buffer size was 32KB; 10 Mbps was what we had achieved in one way traffic from SDSC after the TCP buffer size was changed to 256KB; and 32 Mbps is the target bandwidth set by the SRB developers.

By looking at the table, we see that the overhead is quite large compared to the transmission time component at a bandwidth of 4 Mbps unless the file is at least 1 MB. As the effective bandwidth increases above 4 Mbps, a 2 second overhead becomes large relative to the transmission time even for a 1 MB file. But even if the effective bandwidth were 32 Mbps, the per file overhead still needs to be reduced substantially below 2 seconds since Table 2 shows that the overhead alone for collection D

amounts to over 33 hours! Even if the overhead is reduced to 1 second, the total overhead for this collection is over 16 hours.

If we place a deadline of 12 hours (one-half day) on any archiving run, we could accomplish this by archiving only large files (5 MB or larger at 4 Mbps; 500 KB or larger at 10 Mbps or 32 Mbps). But note that for collection D, we can not meet our deadline unless the overhead is reduced or aggregate small files locally (e.g., using the tar Unix utility). Although aggregation of small files before transmission is possible, we would lose the individuality of the small files unless we provided additional metadata (e.g., an index file).

File Size Range	File # in Collection D of Table 1	Overhead		Transmission Time		
		Each file	Total	4Mb/s	10Mb/s	32Mb/s
0-1KB	2,976	2 sec	1.7h	3sec	1.2sec	0.4sec
-10KB	969	2 sec	0.5h	9.7sec	3.9sec	1.2sec
-100KB	53,808	2 sec	29.9h	1.5h	0.6h	0.2h
-1MB	1,130	2 sec	0.6h	18.8min	7.5min	2.3min
-10MB	1,718	2 sec	1h	4.8h	1.9h	0.6h
-100MB	104	2 sec	3.5min	2.9h	1.2h	0.4h
-1GB	5	2 sec	10sec	1.4h	33.3min	10.4min
>1GB	0					

Table 2: Disk overhead for different file sizes

3.3 Archiving

From the above experiments, we know that the overhead time component is significant and can represent the majority of the total time for small files. Furthermore, some local Unix directories contain so many small files that they would take over a day to archive. Finally, the SRB implementation continues to evolve. Thus, the design of our archival scripts reflect this situation: command line arguments allow the user to flexibly select the minimum file size to be archived, an

archiving deadline (i.e., maximum allotted archiving time), whether to use the container abstraction, etc. A user (or script) can examine the file size statistics and select the appropriate set of command line options that would be most suitable for the situation. There are three categories of archiving scripts. These categories correspond to the three possible phases in an archival process: archiving, verification, and restoration (recovery). The following sections describe these scripts.

3.3.1 The Archiving Script

In our archiving system, the structure of our local Unix file structure is reflected in the SRB storage system. So there is a direct mapping from the logical location of a local file to the logical of an SRB object in the SRB storage space. A SRB collection name corresponds to a Unix directory name, and an SRB dataset corresponds to a regular Unix file. Thus, a Unix file located at /export3/jz5/xxx (for example) will appear as SRB collection PATH/export3/jz5 where PATH is an absolute pathname from the root collection. For example, if PATH=/neurodb/archiver/archive, the Unix file /export3/jz5/xxx would be located at /neurodb/archiver/archive/export3/jz5/xxx in the SRB space. Here neurodb is a collection; archiver, archive, export3, and jz5 are subcollections; and xxx is a dataset.

There are several versions of the archiving script which vary in small aspects. We describe the common features below and describe the details of the variations later. First, the brainmap *archiver* user account is the owner of the archiving scripts and is expected to be the account that executes archival scripts. Second, an archival script can read all files in the Unix file system even if a user has restricted access to a file my making it readable only by the owner. That is, the scripts give the *archiver* account root access to all files on the local host (e.g., brainmap.arl.wustl.edu). Third, the archiver user can select the minimum file size to be archived (i.e., he/she can define what it means to be a *large* file) to avoid the overhead of archiving small files. This feature allows the user to immediately accommodate the high rate of local data acquisition. Fourth, the archiver user can efficiently re-archive the same directory at a later time when small file archiving becomes practical (through a faster SRB implementation or different archiving algorithm). This efficient re-archiving is accomplished by checking a log file to skip files that have already been archived. Fifth, different versions of a Unix file appear as different SRB objects. Versions are maintained by appending a suffix that is the modification time of the Unix file. Sixth, the archiver user can set a deadline for an archival run. At the expiration of the deadline, the script is gracefully terminated. Finally, archiving scripts compute performance metrics which can be reviewed by the archiver user to identify performance problems.

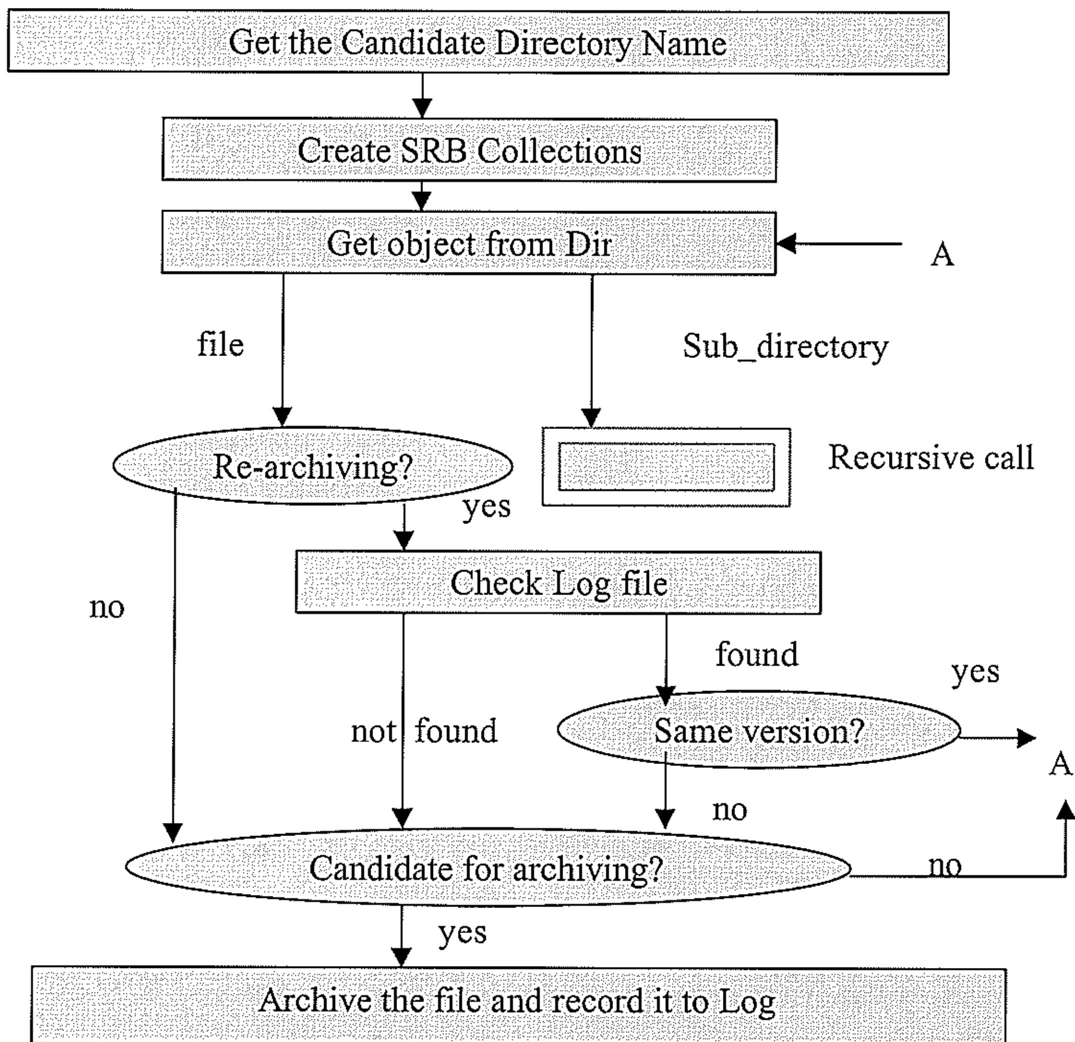

Fig. 7: The flow sheet of archival system

Fig. 7 shows a flowchart of an archiving script. The salient features include the following:

- Create an SRB collection for each Unix directory.
- Recursively call the archiving script to handle each Unix subdirectory.
- Optionally, check a log file before archiving if re-archiving.
- Log all files that are archived.

3.3.2 Verification Scripts

After archiving a directory or a file, the user may want to check whether a file has been archived with the right size or not, or whether the archiving process completed properly. Two verification scripts

allow you to do the following: 1) Get the archived objects' attributes (local path, file size and modification time); and 2) compare the attributes of objects in the SRB space with those in a Unix directory. This feature is very useful in providing feedback to the SRB developers. In the course of this project, some SRB bugs were discovered through this process. Running the verification scripts after each run will provide valuable information on the integrity of the archived data, the reliability of the SRB software, and increase user confidence in the archival process during periods when components are suspect (e.g., HPSS interruptions).

3.3.3 Recovery Script

When local data has been archived at a remote site, there must be a convenient way to get the data back. Since multiple versions of the local file may have been archived the user may want one, all, or some subset of these versions. The retrieval script allows the user to choose from several options: 1) Get the latest version; 2) get all versions; or 3) get a group of files which match a regular expression pattern. Furthermore, the selection criteria can be applied recursively to a whole directory.

3.3.4 Setuid Root

In order to give the *archiver* user access to all user files, the archiving Perl scripts are given root permission in a safe manner through a combination of *taintperl* usage, root C wrappers, and appropriate file permissions that limit how the scripts can be used and modified. Root permission can be easily given by marking a Perl script as owned by the root user and marking them as executable by all users. However, this approach opens up many security holes where it is quite easy for an ordinary user to gain root permissions. We follow the recommendations in Perl documents for writing setuid Perl scripts: write a C program wrapper that calls the Perl script, and use a special version of Perl (taintperl) that checks for potential security vulnerabilities. The C wrapper is compiled into a binary executable which is modifiable only by the root user making it difficult for an intruder to easily modify for their purpose. The Perl script is also modifiable only by the root user. In fact, many Unix OS kernels will disallow setuid scripts. Our approach allows us to safely run in root mode without depending on a Unix kernel setting.

The Perl Programming Manual describes what is required to write and run an untainted Perl script. In summary, the following rules are followed:

- The Perl script must begin with the line #/usr/bin/perl –T" so that taintperl will be called rather than the normal perl interpreter.
- All shell variables and environment variables (e.g., PATH) are initialized in the script.
- All variable values that have their values set from an external source (e.g., command line argument) are checked.
- Subshells are never created (e.g., $system ("... command... ")).

4. Archiving Guide

This section will give you a tour through the scripts that make up the Experimenter's Workbench and the archiving suite. This section is divided into subsections corresponding to the three major script categories for data archiving: archiving, verification, and restoring. Within each category, there are subsections which will discuss each script and guide you in using them.

Since some scripts have a long parameter list, we present examples using a notation that will simplify the presentation. Example commands are shown in the form:

<command> <variable 1> <variable 2> . . .

where

<variable 1> = <value>

<variable 2> = <value>

. . .

For example,

archive *Dir1*

where

Dir1 = test1

is equivalent to entering the command "archive test1"; i.e., *Dir1* is equivalent to the shell variable $Dir1. By convention, we begin each variable with an uppercase letter (e.g., Dir1).

We summarize below the parameters used by the scripts we are going to talk about and give the meaning of each parameter:

Adir: Name of directory to be archived (recursively, if necessary)

Days: Archive files that have not been accessed in *Days* days (i.e., the inactivity criteria)

Cmndir: Absolute path of directory containing this command script (supports recursion)

Minsize: Archive files that are at least *Minsize* bytes in size (the *breakpoint*)

Coll: Name of collection where the archived *Dir* will be placed

First: Always set by the user to "yes" but is changed during a recursive call.

Chk: 1: Check log and archive only versions that have not been archived;

0: Do not check the log (i.e., force an archival of all files)

Dead: Execution deadline (allotted execution time)

Rdir: Recovery directory (where to place restored files) used by the*getback* script

Log: Log file (where to record archiving progress)

Fnm: Name of file to be retrieved from SRB space

Patn: A pattern (a regular expression)

The basic archiving cycle can involve the three steps of archiving, verification, and recovery. We give you a brief overview of the process below and more details in the sections which follow. Suppose you find a candidate directory 'Adir' to archive. You may call an archiving script to archive the directory by following the steps below. First, invoke one of the archiving scripts to archive the directory 'Adir' and record the archived file information (path, file size, archived date…) in the log file '*Log*'.

archall.hours Adir Days Cmndir Minsize Coll yes Chk Dead

Where the arguments Adir, Days, Cmndir, Minsize, Coll, First, Chk, and Dead are replaced by appropriate values. This command will archive all files starting at the directory Adir that have not been accessed in Days days and are at least Minsize bytes in size. The archall.hours script is located at Cmndir/archall.hours. The file will be placed in the collection Coll. The script will terminate after Dead time has expired. The value of Chk indicates whether the log file Log should be checked before archiving each file.

Second, call verification scripts to verify the archiving correctness. Verification requires two steps. First, recursively retrieve the file attributes from the SRB space and store them to the *objlog* file by calling the *archivedobj* script with the name of the collection (Coll) of interest:

archivedobj Coll –r

The flag –r indicates a recursive execution. If the –r is omitted, subcollections will not be examined. Note that the output of archivedobj is always to the file named *objlog*. Then, call the verification script *finalcompare* to verify the archiving correctness by comparing the contents of *objlog* with the archiving log file *Log*:

finalcompare Log objlog

Stdout messages indicate missing files and incorrect file attributes. Third, to recover all versions of the archived files from directory Adir, run the recovery script *getback*:

getback Coll Adir Rdir -a

The directory Adir can be replaced with a filename (Fname) or pattern (Patn). The modification date suffix of the SRB object name is removed from the recovered file if there is only one version of a file. The following sections will give more details about the archiving, verification and recovery process.

4.1 Archiving

There are four major archiving scripts:

archall.largefile: Archive only large files to collections (without using containers).

archallargefile.hours: Same as *archall.largefile* but with a deadline.

archall: Archive files so that small files go to containers and large files go to collections.

archall.hours: Same as *archall* but with a deadline.

Given the most recent performance tests, the script archall.hours is the most useful.

In the following descriptions, we first present the syntax of a script call and describe some of the command line arguments. Then we give one or more examples to illustrate different usage. The format of the examples follows the format "Command Argument ... where Argument=Value ... ."

The archiving scripts are written in Perl which recursively call themselves to archive subdirectories.

Syntax: archall.hours Adir Days Cmndir Minsize Coll First Chk Dead

Example 1: archall.hours Adir Days Cmndir Minsize Coll yes Chk Dead

where

Adir=jz5/srb
Days=5
Cmndir=/export1/jz5
Minsize=1024
Coll=archiver-wustl
Chk=0
Dead=3.3h

Description: Archive files from the directory 'jz5/ srb' which have not been accessed for at least 5 days. The archiving script (*archall.hours*) can be found by the path '/export1/jz5'. All files less than '1024' bytes should be placed in a container; all other files are stored in a collection without using a container. The archived files will be stored into sub-collections of the collection 'archiver-wustl' (which itself is a subcollection of the default subcollection /neurodb/archiver). Here setting *Chk* to 0 means that the system should assume that this is the first time the directory has been archived (even if it has been previously archived), and the script will not check the log file before archiving each file. If Chk were not 0, the log file will not be checked to avoid archiving a file version that has already been archived. The deadline is set to 3.3 hours; so the script will run for only 3.3 hours. The time is specified as a decimal number with a time unit indicator (e.g., 3.3h means 3 hours and 18 minutes). The time unit can be hours (h) , minutes (m) or seconds (s) (e.g., 3.3m). When the script exits, all

archiving statistics will be sent to a stdout file. The statistics include the elapsed time, the archival volume, and the effective bandwidth for each directory or subdirectory. The *Log* file for this archived directory "jz5/srb" will be named "jz5_srb.archivedfilelog" to record the archiving information (e.g., path, size…) of each file in this directory or its subdirectory.

The other syntax of the other archiving scripts is identical to the syntax of the *archall.hours* script except that *archall.largefile* and *archall* has no deadline argument. Other example scripts are shown below.

Example 2: archall.largefile.hours Adir Days Cmndir Minsize Coll yes Chk Dead
where the parameters are the same as in Example 1.

Only files that are at least Minsize bytes are archived. Containers are not used, and there is no execution deadline.

Example 3: archall Adir Days Cmndir Minsize Coll yes Chk
where the parameters are the same as in Example 1.

Archive all files in the directory Adir. Small files (those smaller than Minsize bytes) should be stored in containers, and large files should not.

Example 4: archall.hours Adir Days Cmndir Minsize Coll yes Chk Dead
where the parameters are the same as in Example 1.
This is the same as Example 3 except with an execution deadline.

4.1 Verification

There are three scripts that are used during verification:

listobj: Get the attributes of archived objects.
archivedobj: Call *listobj* to get the attributes of archived objects, and write relevant attributes (e.g., local file path, file size, archival date and modification time when archived) to the file '*objlog*'.
finalcompare: Get each record from the file *Log*, and search for a matching file name in the file '*objlog*'. Check whether the file has been archived or archived with the correct size.

The verification scripts are used to get archived objects' attributes back, compare against the " *objlog*" file, and then verify the archiving correctness. The following is the synopsis of these scripts.

Syntax: listobj Coll [–r] > Result

This command is used by the script *archivedobj* to retrieve attributes of previously archived files. *Coll* is the collection name you provided in the command line to the archiving script and is the collection or home collection where archived files are stored. The optional –r flag indicates that the collection should be traversed recursively; i.e., include sub-collections. Without the -r, only the attributes if objects stored in the collection *Coll* are retrieved, but not its sub-collections. The results are sent to the stdout file.

Syntax: archivedobj Coll [–r]

This script takes the output of *listobj* and filters out a subset of the file attributes for display to the *objlog* file.

Syntax: finalcompare Log objlog > Result

Log is the file where the archived file information (file path, size, and modification time when archived) is stored. *objlog* is the output file of *archivedobj* script. It stores the attributes of archived objects retrieved by *archivedobj* script. Error conditions (e.g., missing files and incorrect file sizes) are sent to the stdout file.

Example 5: listobj archiver-wustl -r

Display on stdout the attributes of all objects in the 'archiver-wustl' collection and all its sub-collections.

Example 6: archivedobj archiver-wustl -r

Send the interesting attributes (e.g., path, and size) of each object to the *objlog* file

Example 7: finalcompare Log objlog

Compare against the archived objects' log ' *objlog'* to see whether all files in the *Log* file are archived correctly.

4.2 File Recovery

The *getback* script allows you to retrieve one or more files from archival storage. You can retrieve the latest version or all versions of a file. When retrieving the latest version, the modification date suffix that was appended to the file name is stripped. When retrieving all versions, the suffix is not stripped. You can also retrieve a group of files whose name matches a pattern expressed as a Unix regular expression (e.g., *.gif). All recovered files are stored in a single user-specified file. Thus, if you retrieve all files from a collection that has sub-collections, the files that were original in subdirectories will appear in one directory but with a name prefix that indicates the original Unix directory structure. For example, a file xxx in subdirectory yyy of subdirectory zzz will be named zzz-yyy--xxx; i.e., one hyphen separates the directory components, and two hyphens separate the file name and the directory.

Syntax: getback Coll Patn Rdir [–a]

Coll is the name of the collection that will be searched for object names that match the pattern *Patn*. R*dir* is the pathname of the Unix directory where matching files will be placed. The optional -a flag indicates when you want all file versions retrieved; otherwise you will get the latest version. Note that any Unix metacharacters (e.g., *) must be quoted (e.g., * for *).

Example 8: getback Coll Patn Rdir –a

where

Coll= archiver-wustl

Patn=jz5/srb/*.c

Rdir=/export1/jz5/backup

This command retrieves the files matching the regular expression jz5/srb/*.c from the collection archiver-wustl into the directory /export1/jz5/backup. All file versions are retrieved.

Example 9: getback Coll jz5/srb Rdir –a

where the argument variables are the same as in Example 8.

It retrieves the directory jz5/srb which is stored in the collection archiver-wustl, from the SRB space with all versions of files and stores the retrieved files in the local directory /export1/jz5/backup.

Example 10: getback Coll jz5/srb Rdir

where the argument variables are the same as in Example 8.

This is the same as Example 9 except that it only retrieves the latest file versions.

5. Experimenter's Workbench Guide

This section describes the major scripts for running SRB performance measurements. Since the SRB implementation is still evolving, these scripts are very useful for periodic checks of the SRB's performance. Results of these measurement experiments provide useful feedback to the SRB developers, help to tune the SRB implementation and verify its functionality. The following are the experiment testbench scripts.

For simplicity, all command line arguments that appear in the syntax descriptions are listed below:

Dir: Directory name

Coll: Name of the collection where the files in *Dir* will be read from or written to

Dfile: Name of a file containing a list of data files that will be read from or written to the archival storage

Csfile: Name of a file containing a list of container sizes

Fnm: A file name

Cont: A container name

Reps: Number of repetitions (runs) of an operation

5.1 Simple S-command Test

Since the two S-commands Sput and Sget play a major role of in our archiving scripts, the measurement scripts test features of these commands such as whether to use containers, and the container size. The SRB host and SRB resource is set in the user's ~/.srb/.MdasEnv file. For example, typical settings are to use ghidorah.sdsc.edu as the SRB host, and cont-sdsc as the SRB resource. Cont-sdsc is a logical resource consisting of a container cache at ghidorah.sdsc.edu and permanent storage at the HPSS. All results are displayed on the stdout file.

The *simplesput* command times the writing of each file listed in the file *Dfile* to the archival storage *Reps* times and reports the maximum/minimum/average time and effective bandwidth. Containers are not used.

Syntax: simplesput Dfile Reps > Statistics

Example 11: simplesput datafile 5

The argument *datafile* here is the name of a regular Unix file containing a list of data files that will be copied to the SRB storage. Sput will be called five times to compute the performance statistics.

The *simplesget* command is the same as *simplesput* except that it uses the Sget command to get files from SRB storage.

Syntax: simplesget Dfile Reps > Statistics

The *dirsimpleput* command measures the time and effective bandwidth for writing the contents of an entire Unix directory to SRB storage.

Syntax: dirsimpleput Dir Coll > Statistics

In Example 12, the performance statistics for copying the contents of the Unix directory jz5/srb to the collection archiver-wustl will be reported.

Example 12: dirsimpleput jz5/srb archiver-wustl

The corresponding Sget measurement script is *dirsimpleget*:

Syntax: dirsimpleget Coll Dir > Statistics

5.2 Container Measurements

The SRB developers designed containers to pack small files into a physical block to hide the high latency of each small file access in the archival storage system. The key issue associated with containers is the effect of the container size. The *putcont* script is the equivalent of the *simplesput* script using containers. The only difference is that the container name must be specified and the container size is 100MB as default. The *putcont* script fills a container *Cont* with files listed in the file *Dfile*:

Syntax: putcont Dfile Cont > Statistics

Example 13 copies all files from the file *datafile* to the container *test-cont*:

Example 13: putcont datafile test-cont

The *getcont* script displays the statistics for reading the files from collection *Coll Reps* times.

Syntax: getcont Dfile Reps > Statistics

Example 14 displays the performance statistics for repeating the operation five times of copying all data from the container to the local space. The data has already been put to the SRB space by the *putcont* script.

Example 14: getcont datafile 5

The *getcont* and *putcont* scripts provide performance data for container usage so that we can decide whether it is useful to use containers in our archival system.

The *contfill* script is used to fill containers of different sizes to test the effect of different container sizes:

Syntax: contfill Fnm Csfile > Statistics

Example 15 measures the performance of filling containers whose sizes are listed in the file *contsizefile* with different instances of the Unix file t50k. For example t50k might be a 50 KB test file. The actual file names used are automatically generated. For example, if there are 100 files in each of two containers, their names will be t50k.00, t50k.01... t50k.099 and t50k.10, t50k.11, ... t50k.199.

Example 15: contfill t50k contsizefile

The *contretrieve* script is used to measure the performance of reading all files back from a container to show the effect of different container sizes:

Syntax: contretrieve Fnm Csfile > Statistics

The *try2-2* script puts different size files into different sized containers, one file per container, to test the relationship between the container size and the content size of a container:

Syntax: try2-2 Dfile Csfile times > Statistics

There are other scripts which are formed by minor modifications of the above scripts. Since they have similar functionality, we will not list them here.

6. Conclusions

In this data archiving project, we installed the SRB in our local system. By doing experiments, we gained an understanding of SRB performance, verified SRB functionality, and quantified the effects of

key parameters on the performance of the basic SRB operations like Sput/Sget. We designed and implemented an archival system corresponding to the local file system distribution, reflected the local file structure in the SRB storage system (HPSS). We did perl programming to carry out the archiving, verification and recovery tasks. By running the experiment and verification scripts regularly, we provided valuable feed back (functionality and performance) to the SRB developers, cooperated with the developers to tune SRB performance and debugged SRB errors. With the SRB, we accessed distributed data. Right now, our data archiving scripts are running with root privilege to archive large files, to verify archiving success, to compute statistics on the archival process, to provide feedback to the SRB developers. We provided a facility to solve the local storage shortage problem, and provided valuable experience and performance data for constructing a future brain map database, and made it easier for sharing and publishing our local data in the future.

7. Acknowledgement

We would like to thank Dr. Ken Wong for advice during the conduct of this research project, Dr. Jerome R. Cox, Jr. for funding, and Sharon Stewart for providing useful scripts to show the local data characteristics. We would also like to thank Acrobat Raja, Mike Wan, and the other SRB developers for help in installing and understanding the SRB.

8. References

http://www.npaci.edu/DICE/SRB

http://www.vbns.net

http://www.sdsc.edu/hpss

Appendix (Source Code)

```
#! /pkg/gnu/bin/perl -T

# NAME :     archall
# PURPOSE:   archall some directory(may have subdir)
# SYNOPSIS:  archall Dir Days Path BrkPt Init First Chk
# EXAMPLE:
#            Dir=jz5/srb
#            Days=5
#            Path=/export1/jz5
#            BrkPt=1024
#            Init=archiver-wustl
#            First="yes"
#            Chk=0

# DESCRIPTION:
#            Dir: Directory to be archived
#            Days: Days, files in the Dir are not accessed
#            Path: Absolute path of a command script
#            BrkPt: Breakpoint
#            Init: Initial collection where the archived Dir will go
#            First: First time to create the collections. Always set to
#            "yes" to create collections once
#            Chk: Checking Log or not. "1" for checking log; "0" for not
#            checking log


$ENV{'PATH'}
='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustlsrb/bin:
/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/wustlsrb/SRB1_
1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

require "flush.pl";

if(($ARGV[0] =~ /^-h/) || ($#ARGV<4)){
   print "Usage: archall Dir Days Path BrkPt Init First Chk \n";
   print" Example: archall jz5/srb 5 /export1/jz5 1024 archiver-wustl
          yes 0 \n";
   print "number or argument now : $#ARGV \n";
   exit 0;
}

$totaltime=time;
system "Sinit";

if ($ARGV[2] =~ /^([-_\/\w.-_]+)$/) {
      $home = $1;                      # $home now untainted
    } else {
        die "Bad data in $ARGV[2]";
    }
if($ARGV[2] !~ /\/$/){ $home=$home . "/";}
$command=$home ."archall";             # archiving command for sub-routes


if ($ARGV[0] =~ /^([-_\/\w.-_]+)$/) {
      $directory = $1;                 # $directory now untainted
```

```
     } else {
        die "Bad data in $ARGV[0]";
     }

@totalsizes=split(" ",`du -s -k $directory`);

if (!chdir($directory)) {
      print "Can't cd to $directory\n";
      exit 0;
      }

$localdir=`pwd`;
$n=$ARGV[1];
open(LS, "ls -F -1 -a |") || die ("can't open $directory \n");

if ($directory=~/\/$/) {       # delete / at tail
      chop($directory );
}

if ($ARGV[4] =~ /^([-_\/\w.-_]+)$/) {
      $initcol = $1;           # $initcol now untainted
     } else {
        die "Bad data in $ARGV[4]";
     }

if ( $directory =~/^\//){
      $collpath=$initcol.$directory ;
   }else{
      $collpath=$initcol."\/".$directory ;
}

@dir=split(/\//,$collpath);
if ( $ARGV[4]=~/^\//){ shift(@dir);}

$archivedir="/".$dir[1];

for($i=0; $i<$#dir;$i++){
     $dir[0]= $dir[0]."_".$dir[$i+1];
                              # for container naming use
     if($i>=1){
      $archivedir=$archivedir. "/" .$dir[$i+1];
                              # record the archived directory
     }
}

$archivedirs=$localdir;

if ($archivedirs =~ /^([-_\/\w.-_]+)$/) {
      $archivedir = $1;        # $localdirs now untainted
     } else {
        die "Bad data in $archivedirs";
}

@forlog=split(/\//,$archivedir);

if ( $archivedir=~/^\//){ shift(@forlog);}
```

```
if($#forlog>=2){
   $logname=$forlog[0]."_".$forlog[1]."_".$forlog[2]."." .
   "archivedfilelog";
}
else{
     for($i=0; $i<$#dir;$i++){
         $forlog[0]= $forlog[0]."_".$forlog[$i+1];
     }
     $logname=$forlog[0]."." . "archivedfilelog";
}

$log=$home .$logname;

if(-e $log) {
      @loglines=split(" ", `wc -l $log`);
      $logline=$loglines[0];
}
else {$logline=0;}

open(LOG,">>$log");


if ($ARGV[5] =~ /^([-\/\w.]+)$/) {
     $firstcreating = $1;     # $firstcreating now untainted
    } else {
        die "Bad data in $ARGV[5]";
    }
if ($ARGV[6] =~ /^([-\/\w.]+)$/) {
     $archivedtimes = $1;     # $archivedtimes now untainted
    } else {
        die "Bad data in  $ARGV[6]";
}

$date=`date '+%m/%d/%y'`;
chop($date);

$sum=0; $size=100000000; $count=0; $dirsize=0;
$contname=$dir[0]."_". $count;
$contsource="cont-sdsc";
$firsttime="yes";
$found=0;
$incontfile=0; $outcontfile=0; $incontsize=0; $outcontsize=0;
$subtime=0; $logcheckingtime=0;

@col=split(/\//,$collpath);
if ( $collpath=~/^\//){ shift(@col);}
if($firstcreating eq "yes"){   # first time to create parent
collections.
    for($i=0; $i<=$#col;$i++){
 if($archivedtimes<1){          #first time to archive
              system "Smkdir", "$col[$i]";
       }
       system "Scd", "$col[$i]";
      }
        $firstcreating="no";
}
else{                              # create current collection
```

```
        for($i=0; $i<$#col;$i++){
              system "Scd", "$col[$i]";
          }
        if($archivedtimes<1){     #first time to archive
              system "Smkdir", "$col[$#col]";
        }
        system "Scd", "$col[$#col]";
}

while(<LS>){
        if((/^\.\/$/) || (/^\.\.\/$/)){next;}
        if((/^\./) && (/\/$/)) {print"can't archive .dir: $_\n"; next;}
        if (/\/$/) {                #--sub_dir
           chop; chop;

           $protime=time;
           system "$command", "$_", "$ARGV[1]", "$ARGV[2]", "$ARGV[3]",
           "$collpath", "$firstcreating", "$archivedtimes";
           $subtime +=time -$protime;
           next;
          }
                                    # not a sub_dir
        $line=$_;
        chop($line);

        if((/\@$/) ||(/\|$/)){
        print"don't archive symbolic link file or pipe file: $_"; next;
        }

        if($line =~/\*$/){          #executable file
                  chop($line);
        }


                                    #last modified time
        $mtime = (stat($line))[9];
        @gmtimes=gmtime($mtime);
        $day=$gmtimes[3];
        $month =1+ $gmtimes[4];
        $year =1900 +$gmtimes[5];

        if($day <= 9){
                $day="0".$day;
          }
        if($month <= 9){
              $month="0".$month;
          }

        $modifiedtime= $year.$month.$day;

        if ($line=~ /^([-_+\/\w.,-_+\~\^%$\#@\:]+)$/) {
             $file_name = $1;   # $file_name now untainted
        } else {
        print "Bad data in $line, at dir: $archivedir \n";
        next;
        }
```

```
   $filename=$archivedir. "/". $file_name;
   $filesize=-s $line;
   $dirsize=$dirsize +$filesize;

   $n=int(-A $line);                #how old is the file

   if($archivedtimes<1){            #first time to archive the file
         if($n <$ARGV[1]){next;}
   }
   else{ #check log to see whether the file was archived  before

         $logtime=time;
         @greps=`/pkg/gnu/bin/grep -F $filename  $log`;

         $logcheckingtime +=time -$logtime;

         if($#greps>=0){ #find it
               for($k=0; $k<=$#greps;$k++){
                     @times=split(" ",$greps[$k]);
                     if($times[1] eq $modifiedtime){$found=1; last;}
               }
               if($found==1){ $found=0; next;}
         }else{  # not been archived before
             if($n <$ARGV[1]){next;}
         }

   }

 $logline++;

 $srbobj= $line . "." . $modifiedtime;

 if($filesize>$ARGV[3]){   #large file goes to collection directly
     system "Sput", "-f", "$line", "$srbobj";
     $outcontfile++;
     $outcontsize=$outcontsize +$filesize;
 }
 else{                         # small files go to container now
     if($firsttime eq "yes"){ #first time to create container
         system "Smkcont", "-S", "$contsource", "-s", "$size",
         "$contname";
         $firsttime="done";
     }

     $incontfile++;
     $incontsize=$incontsize +$filesize;
     $sum=$sum + $filesize;

     if($sum > $size) {
         system "Ssyncont", "-d", "$contname";
         $count++; $sum=$filesize;
         $contname=$dir[0]."_". $count;

         system "Smkcont", "-S", "$contsource",  "-s",
         "$size", "$contname";
     }
     system "Sput", "-f", "-c", "$contname", "$line", "$srbobj";
```

```
        }
        printf LOG "$archivedir/$line $modifiedtime $filesize $date
        $mtime\n";
        &flush(LOG);
}

if($firsttime ne "yes"){
         system"Ssyncont", "-d", "$contname";
}


close(LS);
close(LOG);
system("Sexit");

$totaltime=time-$totaltime-$subtime;
$dirsize=$incontsize +$outcontsize;
if($totaltime<1 ){ $totaltime=1;}

##$throughputs=($totalsizes[0]*8)/(1024*$totaltime);
$throughputs=($dirsize*8)/(1024*1024*$totaltime);
$throughput=sprintf("%.4f\n",$throughputs);

if($firsttime ne "yes"){
      $numbcont=$count+1;
}
else{
      $numbcont=0;
}
print"\nStatistics of dir:$archivedir: \n";
print"The running_time for archiving $dirsize bytes in current dir is
(s): $totaltime\n";
print"Effective throughput(Mb/s): $throughput\n";
print"archiving_size(B): $dirsize\n";
print"Number of containers: $numbcont\n";
print"Container size(B) : $size\n";
print"Breakpoint(Byte): $ARGV[3]\n";
print"#files in containers: $incontfile\n";
print"#files out of containers: $outcontfile\n";
print"#bytes in containers: $incontsize\n";
print"#bytes out of containers: $outcontsize\n";
if($archivedtimes>0){
  print"checking_log_time for $logline lines log is (s):
$logcheckingtime\n";
}
```

```
#! /pkg/gnu/bin/perl -T

# NAME :     archall.hours
# PURPOSE:   archall.hours some directory(may have subdir)
# SYNOPSIS:  archall.hours Dir Days Path BrkPt Init First Chk T
# EXAMPLE:
#            Dir=jz5/srb
#            Days=5
#            Path=/export1/jz5
#            BrkPt=1024
#            Init=archiver-wustl
#            First="yes"
#            Chk=0
#            T=3.3h

# DESCRIPTION:
#            Dir: Directory to be archived
#            Days: Days, files in the Dir are not accessed
#            Path: Absolute path of a command script
#            BrkPt: Breakpoint
#            Init: Initial collection where the archived Dir will go
#            First: First time to create the collections. Always set to
#            "yes" to create collections once
#            Chk: Checking Log or not. "1" for checking log; "0" for not
#            checking log
#            T: Time value


$ENV{'PATH'}
='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustlsrb/bin:
/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/wustlsrb/
SRB1_1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

require "flush.pl";

if(($ARGV[0] =~ /^-h/) || ($#ARGV<4)){
   print "Usage: archall.hours Dir Days Path BrkPt Init First Chk T
   \n";
   print" Example: archall.hours jz5/srb 5 /export1/jz5 1024 archiver-
   wustl yes 0 3.3h \n";
   print "number or argument now : $#ARGV \n";
   exit 0;
}

$totaltime=time;
system "Sinit";

if ($ARGV[2] =~ /^([-_\/\w.-_]+)$/) {
        $home = $1;                          # $home now untainted
    } else {
        die "Bad data in $ARGV[2]";
    }
if($ARGV[2] !~ /\/$/){ $home=$home . "/";}

$command=$home ."archall.hours";
```

```
if ($ARGV[0] =~ /^([-_\/\w.-_]+)$/) {
        $directory = $1;                    # $directory now untainted
    } else {
        die "Bad data in $ARGV[0]";
}

@totalsizes=split(" ",`du -s -k $directory`);

if (!chdir($directory)) {
       print "Can't cd to $directory\n";
       exit 0;
      }

$localdir=`pwd`;
$n=$ARGV[1];
open(LS, "ls -F -1 -a |") || die ("can't open $directory \n");

if ($directory=~/\/$/) {                   # delete / at tail
      chop($directory );
}

if ($ARGV[4] =~ /^([-_\/\w.-_]+)$/) {
        $initcol = $1;                     # $initcol now untainted
    } else {
        die "Bad data in $ARGV[4]";
}

if ( $directory =~/^\//){
      $collpath=$initcol.$directory ;}
else{
      $collpath=$initcol."\/".$directory ;
}

@dir=split(/\//,$collpath);
if ( $ARGV[4]=~/^\//){ shift(@dir);}

$archivedir="/".$dir[1];

for($i=0; $i<$#dir;$i++){
         $dir[0]= $dir[0]."_".$dir[$i+1];
                                           # * for container naming use
     if($i>=1){
      $archivedir=$archivedir. "/" .$dir[$i+1];
                                           # record the archived dir
     }
}

$archivedirs=$localdir;

if ($archivedirs =~ /^([-_\/\w.-_]+)$/) {
        $archivedir = $1;                  # $localdirs now untainted
    } else {
        die "Bad data in $archivedirs";
}
@forlog=split(/\//,$archivedir);
```

```
if ( $archivedir=~/^\//){ shift(@forlog);}
if($#forlog>=2){
      $logname=$forlog[0]."_".$forlog[1]."_".$forlog[2]."." .
      "archivedfilelog";}
else{
    for($i=0; $i<$#dir;$i++){
         $forlog[0]= $forlog[0]."_".$forlog[$i+1];
    }
    $logname=$forlog[0]."." . "archivedfilelog";
}
$log=$home .$logname;

if(-e $log) {
      @loglines=split(" ", `wc -l $log`);
      $logline=$loglines[0];
}
else {$logline=0;}

open(LOG,">>$log");

if ($ARGV[5] =~ /^([-\/\w.]+)$/) {
        $firstcreating = $1;           # $firstcreating now untainted
    } else {
        die "Bad data in $ARGV[5]";
}
if ($ARGV[6] =~ /^([-\/\w.]+)$/) {
        $archivedtimes = $1;           # $archivedtimes now untainted
    } else {
        die "Bad data in  $ARGV[6]";
}
if ($ARGV[7] =~ /^([-\/\w.]+)$/) {
      $timeleft=$1;
      $unit=chop($timeleft);

      if($unit eq "h"){$timeinseconds=3600*$timeleft;}
      else{
        if($unit eq "m"){$timeinseconds=60*$timeleft;}
        else{$timeinseconds=$timeleft;}
      }
    } else {
      $timeleft="unknown";
    }


$date=`date '+%m/%d/%y'`;
chop($date);

$sum=0; $size=100000000; $count=0; $dirsize=0;
$contname=$dir[0]."_". $count;
$contsource="cont-sdsc";
$firsttime="yes";
$found=0;
$incontfile=0; $outcontfile=0; $incontsize=0; $outcontsize=0;
$subtime=0; $logcheckingtime=0;


@col=split(/\//,$collpath);
```

```
if ( $collpath=~/^\//){ shift(@col);}
if($firstcreating eq "yes"){    # first time to create parent
collections.
        for($i=0; $i<=$#col;$i++){
          if($archivedtimes<1){ #first time to archive
                 system "Smkdir", "$col[$i]";
          }
          system "Scd", "$col[$i]";
      }
        $firstcreating="no";
}
else{                             # create current collection
      for($i=0; $i<$#col;$i++){
            system "Scd", "$col[$i]";
        }
      if($archivedtimes<1){       #first time to archive
            system "Smkdir", "$col[$#col]";
      }
      system "Scd", "$col[$#col]";
}

while(<LS>){
      if((/^\.\/$/) || (/^\.\.\/$/)){next;}
      if((/^\./) && (/\/$/)) {print"can't archive .dir: $_\n"; next;}

      if (/\/$/) {                #--subdir
         chop; chop;
         if($timeleft ne "unknown"){
            if((time-$totaltime)>=$timeinseconds){
                  print"time_out for: $archivedir/$_/***\n";last;
            }
            else{$leftsecond=$timeinseconds-(time-$totaltime);
                 $timelefts=$leftsecond."s";
                 $protime=time;
                 system "$command", "$_", "$ARGV[1]", "$ARGV[2]",
                  "$ARGV[3]", "$collpath", "$firstcreating",
                  "$archivedtimes", "$timelefts";

                 $subtime +=time -$protime;
                 next;
            }
         }
         $protime=time;
         system "$command", "$_", "$ARGV[1]", "$ARGV[2]", "$ARGV[3]",
         "$collpath", "$firstcreating", "$archivedtimes";

         $subtime +=time -$protime;
         next;
      }
                              #-- not a sub_dir
      if($timeleft ne "unknown"){
           if((time-$totaltime)>=$timeinseconds){
                  print"time_out for: $archivedir/$_";last;
            }
      }
      $line=$_;
      chop($line);
```

```
if((/\@$/) ||(/\|$/)){
print"don't archive symbolic link file or pipe file: $_"; next;
}

if($line =~/\*$/){      #executable file
          chop($line);
}

#last modified time
$mtime = (stat($line))[9];
@gmtimes=gmtime($mtime);
$day=$gmtimes[3];
$month =1+ $gmtimes[4];
$year =1900 +$gmtimes[5];
if($day <= 9){
        $day="0".$day;
  }
if($month <= 9){
      $month="0".$month;
  }
  $modifiedtime= $year.$month.$day;

if ($line=~ /^([-_+\/\w.-_+\~\^%$\#@\:]+)$/) {
  $file_name = $1;      # $file_name now untainted
} else {
print "Bad data in $line, at dir: $archivedir \n"; next;
}

$filename=$archivedir. "/". $file_name;
$filesize=-s $line;
$dirsize=$dirsize +$filesize;

$n=int(-A $line);       #how old is the file

if($archivedtimes<1){   #first time to archive the file
      if($n <$ARGV[1]){next;}
}
else{ #check log to see whether the file was archived before
      $logtime=time;
      @greps=`/pkg/gnu/bin/grep -F $filename  $log`;

      $logcheckingtime +=time -$logtime;

      if($#greps>=0){ #find it
            for($k=0; $k<=$#greps;$k++){
                  @times=split(" ",$greps[$k]);
                  if($times[1] eq $modifiedtime){$found=1; last;}
            }
            if($found==1){ $found=0; next;}
      }else{              # not been archived before
          if($n <$ARGV[1]){next;}
      }

}

$logline++;
$srbobj= $line . "." . $modifiedtime;
```

```
        if($filesize>$ARGV[3]){ #large file goes to collection directly

          system "Sput", "-f", "$line", "$srbobj";
          $outcontfile++;
          $outcontsize=$outcontsize +$filesize;
        }
        else{                   # small files go to container now
           if($firsttime eq "yes"){ #first time to create container
              system "Smkcont", "-S", "$contsource", "-s", "$size",
              "$contname";
              $firsttime="done";
             }

          $incontfile++;
          $incontsize=$incontsize +$filesize;
          $sum=$sum + $filesize;
          if($sum > $size) {
              system "Ssyncont", "-d", "$contname";
              $count++; $sum=$filesize;
              $contname=$dir[0]."_". $count;
              system "Smkcont", "-S", "$contsource",  "-s", "$size",
              "$contname";
          }
          system "Sput", "-f", "-c", "$contname", "$line", "$srbobj";
        }
        printf LOG "$archivedir/$line $modifiedtime $filesize $date
        $mtime\n";
        &flush(LOG);
}
if($firsttime ne "yes"){
        system"Ssyncont", "-d", "$contname";
}


close(LS);
close(LOG);
system("Sexit");

$totaltimes=time-$totaltime;
$totaltime=$totaltimes-$subtime;
$dirsize=$incontsize +$outcontsize;

if($totaltime<1 ) { $totaltime=1;}

##$throughputs=($totalsizes[0]*8)/(1024*$totaltime);
$throughputs=($dirsize*8)/(1024*1024*$totaltime);
$throughput=sprintf("%.4f\n",$throughputs);

if($firsttime ne "yes"){
      $numbcont=$count+1;
}
else{
      $numbcont=0;
}
print"\nStatistics of dir:$archivedir: \n";
```

```
print"The running_time for archiving $dirsize bytes in current dir is
(s): $totaltime\n";
print"Effective throughput(Mb/s): $throughput\n";
print"archiving_size(B): $dirsize\n";
print"Number of containers: $numbcont\n";
print"Container size(B) : $size\n";
print"Breakpoint(Byte): $ARGV[3]\n";
print"#files in containers: $incontfile\n";
print"#files out of containers: $outcontfile\n";
print"#bytes in containers: $incontsize\n";
print"#bytes out of containers: $outcontsize\n";
if($archivedtimes>0){
  print "lines_of_log : $logline \n";
  print"checking_log_time for $logline lines log is (s):
$logcheckingtime\n";
}
```

```
#! /pkg/gnu/bin/perl -T

# NAME :     listobj
# PURPOSE:  list all archived files' attributes in some collections
# SYNOPSIS: listobj  (initcoll_name) -r
# EXAMPLE:  listobj (archiver-wustl) -r

# DESCRIPTION: recursivly list all objects in this initcoll_name
#collection if -r exits

$ENV{'PATH'}='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustl
srb/bin:
/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/wustlsrb/SRB1_
1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

if(($ARGV[0] =~ /^-h/) || ($#ARGV>1)){
   print "Usage: listobj (initcoll_name) -r\n";
   print" Example: listobj (archiver-wustl) -r \n";
   print "number or argument now : $#ARGV \n";
   exit 0;
}

system "Sinit";

if ($ARGV[0] =~ /^([-\/\w.]+)$/) {
        $directory = $1;                     # $directory now untainted
    } else {
        die "Bad data in $ARGV[0]";          # log this somewhere
    }

if ($ARGV[1] =~ /^-r$/) {
        $recursive = "yes";
} else{$recursive = "no";}

if($#ARGV>=0){
        @dir=split(/\//,$directory);
        if ( $ARGV[0]=~/^\//){ shift(@dir);}
        for($i=0; $i<$#dir+1; $i++){
            system "Scd", "$dir[$i]";
        }
}

if($recursive eq "yes"){
      system "Sls", "-r", "-l", "-L", "-1" ;
}
else {
      system "Sls", "-l", "-L", "-1" ;
}
system "Sexit";
```

```
#! /pkg/gnu/bin/perl -T

# NAME :    archivedobj
# PURPOSE:  list all archived files' attributes in some collections
# SYNOPSIS: archivedobj  initcollname -r
# EXAMPLE:  archivedobj archiver-wustl -r

# must use -r for recursivly getting objects

$ENV{'PATH'}='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustl
srb/bin:/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/
wustlsrb/SRB1_1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';


if($ARGV[0] =~ /^-h/){
   print "Usage: archivedobj initcollname -r\n";
   print" Example: archivedobj archiver-wustl -r\n";
   print "number or argument now : $#ARGV \n";
   exit 0;
}

$localdir=`pwd`;
 if ($localdir =~ /^([-_\/\w.-_]+)$/) {
         $home = $1;                      # $home now untainted
     } else {
         die "Bad data in $localdir";
     }
 if($localdir !~ /\/$/){ $home=$home . "/";}

 $command=$home."listobj";


if ($ARGV[1] =~ /^-r$/) {
        $recursive = "yes";
} else{$recursive = "no";}

if($#ARGV>=0){

      if ($ARGV[0] =~ /^([-\/\w.]+)$/) {
            $directory = $1;             # $directory now untainted
        }
      else {
                die "Bad data in $ARGV[0]";
      }
      @dir=split(/\//,$directory);
        if ( $ARGV[0]=~/^\//){ shift(@dir);}
            $extrasteps=$#dir+1;


      if($recursive eq "yes"){
            open(LS, "$command  $directory -r |") || die ("can't list
            obj\n");
      }
      else{ open(LS, "$command  $directory  |") || die ("can't list
      obj\n"); }
```

```
}
else{

      $extrasteps=0;
      open(LS, "$command |") || die ("can't list obj\n");

}
$log=$home ."objlog";
$constantstep=3;                              #/home/jz5.../
$finalstep=$constantstep + $extrasteps;   #/home/jz5.../test/.../
open(LOG,">$log");                    #attributes is writen to a file

while(<LS>){
      $line=$_;
        $line=~s/\s+//; #stripe the leading space( from 1 to unlimited)
      if ( $line=~/^C/){ next; }           #sub_collection name list

      if ($line =~/^\//) {                  #collection start:
            chop($line);
            @dir=split(/\//,$line);
            for($n=0; $n<$finalstep; $n++){
                  shift(@dir); #get of the hearder collection
            }

            for($i=0; $i<$#dir;$i++){
               $dir[0]= $dir[0]."/".$dir[$i+1];
            }
            if($#dir>=0){
                  $dir[0]= "/". $dir[0] ."/";   #file directory
            }

            else {$dir[0]="";}
            next;
      }
      else{                                       # objects

            chop($line);
            @file=split(" ",$line);

            # delete useless information: wustlsrb 0 cont-hpsss-sd

            shift(@file); shift(@file); shift(@file);

            if($file[0]=~/^IC\b/){
                   shift(@file); }                # IC--in container

            $size= $file[0];
            $times=$file[1];
            @dates=split("-",$times);
            $year=substr($dates[0], 2);
            $date=$dates[1]."/".$dates[2]."/".$year;
            $filename=$file[2];
            printf LOG "$dir[0]$filename $size $date\n";
            next;
      }

}
```

```
close(LS);
close(LOG);
```

```
#! /pkg/gnu/bin/perl -T

# name: finalcompare
# input two filenames
# check whether the files listed in the file log are archived correctly
# comparing time: n*n
# example: finalcompare log objlog

#log: stores the information of archived files
#objlog: stores the attributes of archived objects in SRB space

$ENV{'PATH'}='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustl
srb/bin:/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/wustls
rb/SRB1_1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

if ($ARGV[0] =~ /^([-\/\w.]+)$/) {
        $file0 = $1;                    # $file0 now untainted
    } else {
        die "Bad data in $ARGV[0]";
}

if ($ARGV[1] =~ /^([-\/\w.]+)$/) {
        $file1 = $1;                    # $file1 now untainted
    } else {
        die "Bad data in $ARGV[1]";
}


open(LOG, $file0 ) || die ("can't open $ARGV[0]) \n");

while(<LOG>){
      $line=$_;
      chop();

      if ($_ =~ /^([-_+\/\w.-_+\~\^%$\#@\:\s]+)$/) {
            $lines = $1;            # $_ now untainted, why????????
      } else {
            print "bad data in $_ \n"; next;
      }

      @datas=split(" ",$lines);
      $findingobj=$datas[0] . "." . $datas[1]; #filename.modifiedtime

      @greps=`/pkg/gnu/bin/grep -F $findingobj  $file1`;

      if($#greps>=0){               #find it
              @times=split(" ",$greps[0]);

              if($times[2] ne $datas[3]){
                 print"achiving date is different\n";
                 print "oringinal:$line archived:$greps[0]";
              }
              if($times[1] ne $datas[2]){
                      print"achiving file size is different\n";
                      print "oringinal:$line archived:$greps[0]";
```

```
                }

        }
        else{
                print "not be archived : $line";
        }
}
close(LOG);
```

```
#! /pkg/gnu/bin/perl -T

# NAME :    getback
# PURPOSE:  retrieve archived files from some collection
# SYNOPSIS: getback Init Dir/Fnm/Patn SDir -a
# EXAMPLE:
#           Init= archiver-wustl
#           Dir=jz5/srb
#           Fnm=xxx
#           Patn=jz5/srb/\*.c
#           SDir=/export1/jz5/backup

# DESCRIPTION:
#           Init: Initial collection where the archived Dir stored
#           Dir: Directory to be retrieved
#           Fnm: a file
#           Patn: a pattern
#           SDir: Storing directory, to store retrieved files


$ENV{'PATH'}='/pkg/gnu/bin:/bin:/usr/bin:/usr/ucb:/home/arl/staff/wustl
srb/bin:/home/arl/staff/wustlsrb/SRB1_1_4rel/bin:/home/arl/staff/wustls
rb/SRB1_1_4rel/utilities/bin';
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

if(($ARGV[0] =~ /^-h/) || ($#ARGV<0)){
   print "Usage: getback Init Dir/Fnm/Patn SDir -a \n";
   print "number or argument now : $#ARGV \n";
   exit 0;
}

$totaltime=time;
system "Sinit";

$localdir=`pwd`;

if ($localdir =~ /^([-_\/\w.-_]+)$/) {
         $home = $1;                     # $home now untainted
     } else {
         die "Bad data in $localdir";
}
if($localdir !~ /\/$/){ $home=$home . "/";}

$command=$home."archivedobj";
$log=$home. "objlog";


$isdir=0;
if ($ARGV[1] =~ /^([-_+\/\w.,-_+\*\?\~\^%$\#\:\>\<\s]+)$/){
        $directory = $1;                 # $directory now untainted
    } else {
        die "Bad data in $directory";
}

if($directory=~/\/$/) {                  # delete / at tail
        chop($directory );
```

```
      $isdir=1;
}

if ($ARGV[0] =~ /^([-\/\w.]+)$/) {
        $initcol = $1;                    # $initcol now untainted
    } else {
        die "Bad data in $initcol";
}

if ($ARGV[2] =~ /^([-\/\w.]+)$/) {
         $storedir= $1;                   # $storedir now untainted
    } else {
        die "Bad data in $storedir";
}

if($storedir !~ /\/$/){ $storedir=$storedir . "/";}
if(-e $ARGV[2]){ print " storing_dir exist\n";}
else{
        if(mkdir( $storedir, 0700)){ print "creating storing_dir\n";}
}

if(-d $directory){ $isdir=1;
       @dir=split(/\//,$directory);
}
else{
   if($isdir!=1){
      $isdir=0;
      $at=rindex($directory, "/");
      $puredir=substr($directory, 0, $at);
      $filenames=substr($directory, $at+1);

      if ($filenames =~ /^([-_+\/\w.,-_+\*\?\~\^%$\#\:\>\<\s]+)$/){
                $filename = $1;          # $directory now untainted
        } else {
                die "Bad data in $filenames";
      }


      @dir=split(/\//,$puredir);
   }
}
if ( $directory =~/^\//){
      shift(@dir);
     if($isdir==1){
        $collpath=$initcol.$directory ;
     }else{$collpath=$initcol.$puredir ;}
}
else{
     if($isdir==1){
        $collpath=$initcol."\/".$directory ;
     }else{$collpath=$initcol."\/".$puredir ;}
}
for($i=0; $i<$#dir;$i++){
         $dir[0]= $dir[0]."-".$dir[$i+1];
}
system "$command", "$collpath", "-r";
            #get all files' attributes(size...) from srb space
```

```
$version= "recent";
if ($ARGV[3] =~ /^-a$/) {
        $version = "all";
    }

open(LOG, $log ) || die ("can't open $ARGV[0]) \n");

if(($isdir==0) && ($filename =~ /[\*\?]/)){
                              #get multifiles matching the pattern
      $mypatern=paterns($filename);

      while(<LOG>){
            if ($_ =~ /^([-_+\/\w.-_+\~\^%$\#@\:\s]+)$/) {
               $lines = $1;
            }else{ print "bad data in $_ \n"; next; }

            @data=split(" ",$lines);

            if($data[0] =~ /$mypatern/){print"match** \n";
                  &getfile($lines);
            }
            else{ print"not match???? \n";
                  next;
            }
      }
      exit;
}

if($isdir==1){                              # get whole dir back
      while(<LOG>){
            if ($_ =~ /^([-_+\/\w.-_+\~\^%$\#@\:\s]+)$/) {
                  $lines = $1;}
            else{ print "bad data in $_ \n"; next; }
            &getfile($lines);
      }
}
else{                                       # get a file back
      if($version eq "all"){
            @greps=`/pkg/gnu/bin/grep -F $filename  $log`;
              $n=0; $find=0;
            if($#greps>=0){                 #find it
                   for($k=0; $k<=$#greps;$k++){
                            @times=split(" ",$greps[$k]);
                        $dot=rindex($times[0], ".");
                        if($dot<0){$filenm=$times[0];}
                        else{ $filenm=substr($times[0], 0, $dot);}

                        if($filename ne $filenm) { next;}
                        $find=1;
                        $localname=$storedir. $dir[0]. "--". $times[0];
                              $srbobj= $collpath . "/". $times[0];

                  print" srbobj: $srbobj; localname: $localname \n";
                         system "Sget", "-f", "$srbobj", "$localname";
                   }
```

```
                if($find==0){
                        print "this file is not archived <find==0>\n";
                        exit;
                }

            }
            else{  print "this file is not archived <grep notfind>\n";
                    exit;
            }
    }
    else{
             $tstamp=latestfile($filename);

                 if($tstamp eq "notfound"){
                         print "this file is not archived <notfound>\n";
                         exit;
                 }
                 else{
                         $localname=$storedir. $dir[0]. "-". $filename;

                          if($tstamp eq "notimestamp"){
                                $srbobj= $collpath . "/". $filename;
                          } else{
                                $srbobj= $collpath . "/". $filename
                               ."."."$tstamp;
                          }

                    print" srbobj: $srbobj; localname: $localname \n";
                    system "Sget", "-f", "$srbobj", "$localname";
                 }
      }
}
close(LOG);

sub latestfile{
      @greps=`/pkg/gnu/bin/grep -F $_[0]  $log`;
      $n=0; $find=0;
      if($#greps>=0){                                  #find it

          for($k=0; $k<=$#greps;$k++){
               @times=split(" ",$greps[$k]);
               $att=rindex($times[0], ".");
            if($att<0){$nm=$times[0];}
            else{ $nm=substr($times[0], 0, $att);}

            if($_[0] ne $nm) { next;}
                $timestamp=substr($times[0], $att+1);

                if(length($timestamp)==8){
                        $timepart[$n]=$timestamp;
                        $n++;
                        $find=1;
                }
                else{next;}

           }
           if($find==1){
```

```
                    $recentime=$timepart[0];
                    for($i=0; $i<=$#timepart;$i++){
                           if($recentime<$timepart[$i]){
                                 $recentime=$timepart[$i];
                           }
                    } return  $recentime;
              }
              else {return "notimestamp";}
          }
          else{ return "notfound";}
}


sub paterns{
      $patern=$_[0];
      if($_[0]=~/\./){ $patern=~s/\./\\\./g;}

      if($_[0]=~/\?/){$patern=~tr/\?/\./;}
      if($_[0]=~/\*/){$patern=~s/\*/\.+/;}
      $patern="\(". $patern ."\)";

      if(($_[0]=~ /^(\w)/) ||($_[0]=~ /(\w)$/)){
            $patern ="\^".$patern."\$";}
      return $patern;
}


sub getfile{
             @datas=split(" ",$_[0]);

             if($version eq "all"){
                 $localname=$storedir. $dir[0]. "--". $datas[0];
                 $srbobj= $collpath . "/". $datas[0];
                 print" srbobj: $srbobj; localname: $localname \n";
                 system "Sget", "-f", "$srbobj", "$localname";
             }
             else{              # get files with latest version back
                 $dot=rindex($datas[0], ".");
                 if($dot<0){
                     $filenm=$datas[0];
                     $localname=$storedir. $dir[0]. "--". $datas[0];
                     $srbobj= $collpath . "/". $datas[0];
                  print" srbobj: $srbobj; localname: $localname \n";
                     system "Sget", "-f", "$srbobj", "$localname";
                  }
                  else{
                     $filenm=substr($datas[0], 0, $dot);
                     $tstamp=latestfile($filenm);
                     $localname=$storedir. $dir[0]. "--". $filenm;
                     $srbobj= $collpath . "/". $filenm ."."$. $tstamp;
                  print" srbobj: $srbobj; localname: $localname \n";
                     system "Sget", "-f", "$srbobj", "$localname";
                  }
           }
}
```

```
! /pkg/gnu/bin/perl

# NAME :     simpleput
# PURPOSE:   test average sput time to a collection
# SYNOPSIS   simplesput Dfile Times
# EXAMPLE:   simplesput datafile 5
# DESCRIPTION:
#            Dfile: a file contains  all  files for the testing purpose.
#            Times: times of repeated operations


if(($ARGV[0] =~ /^-h/) || ($#ARGV<1)){
   print "Usage: simplesput Dfile Times \n";
   exit 0;
}

open(TESTDATA, "$ARGV[0]") || die ("can't open $ARGV[0]");

$j=0;
while(<TESTDATA>){
                chop;
                $testfile[$j]=$_;
                $j++;
}
$numb=$#testfile;
system("Sinit");


 for($j=0; $j<=$numb; $j++){

      $filesize=-s $testfile[$j];
      $iterator=$ARGV[1];
      $sum=0; $max=0; $min=0; $first=1;
      while($iterator){
            $times=time;
            `Sput -f  $testfile[$j] $testfile[$j]`;
            $times=time-$times;

            $sum +=$times;

            if($first){
               $max=$min=$times;
               $first=0;
            }else {
               if($max<$times){
               $max=$times;}
               if($min>$times){
                 $min=$times;}
               $first=0;
            }
      $iterator--;
      }
      $avgs=$sum/$ARGV[1];
        $avg=sprintf("%.5f\n",$avgs);
      $throughputs=($filesize*8)/(1024*1024*$avg);
      $throughput=sprintf("%.4f\n",$throughputs);
```

```
      write;
 }

close(TESTDATA);
system("Sexit");

format STDOUT=
@<<<<<<<<< @<<<<<<<<<<<< @<<<<<<<<<<<<<<< @<<<<<<<<<<< @<<<<<<<<<<<
@<<<<<<<<<<<<<<<<<<
$max, $min, $avg, $testfile[$j], $throughput , $filesize
.


format STDOUT_TOP=
Page @<<<<<
$%

just sput time
<---------------------------------->
Max(s)      Min(s)      Average(s)    InContFile  throughput filesize
--------   ----------   -----------   ----------  ---------- ----------
-----
--------   ----------   -----------   ----------  ---------- ----------
-----
.
```

```
#! /pkg/gnu/bin/perl

# NAME :     simpleget
# PURPOSE:   test average sget time to a collection
# SYNOPSIS   simplesget Dfile Times
# EXAMPLE:   simplesget datafile 5

# DESCRIPTION:
#            Dfile: a file contains  all  files for the testing purpose.
#            Times: times of repeated operations

if(($ARGV[0] =~ /^-h/) || ($#ARGV<1)){
   print "Usage: simplesget Dfile Times \n";
   exit 0;
}

open(TESTDATA, "$ARGV[0]") || die ("can't open $ARGV[0]");
$j=0;
while(<TESTDATA>){
                chop;
                $testfile[$j]=$_;
                $j++;
}
$numb=$#testfile;
system("Sinit");


 for($j=0; $j<=$numb; $j++){

      $filesize=-s $testfile[$j];
      $iterator=$ARGV[1];
      $sum=0; $max=0; $min=0; $first=1;
      while($iterator){
            $times=time;
            `Sget -f  $testfile[$j] $testfile[$j]`;
            $times=time-$times;

            $sum +=$times;

            if($first){
               $max=$min=$times;
               $first=0;

            }else {
               if($max<$times){
               $max=$times;}
               if($min>$times){
                 $min=$times;}
               $first=0;
            }
      $iterator--;
      }
      $avgs=$sum/$ARGV[1];
        $avg=sprintf("%.5f\n",$avgs);
      $throughputs=($filesize*8)/(1024*1024*$avg);
      $throughput=sprintf("%.4f\n",$throughputs);
```

```
      write;

 }

close(TESTDATA);
system("Sexit");
print:" repeating times: $ARGV[1]\n";

format STDOUT=
@<<<<<<<<< @<<<<<<<<<<<< @<<<<<<<<<<<<<<< @<<<<<<<<<<< @<<<<<<<<<<<
@<<<<<<<<<<<<<<<<<
$max, $min, $avg, $testfile[$j], $throughput , $filesize
.

format STDOUT_TOP=
Page @<<<<<
$%

just sget time (thoughput unit: Mb/s)
<---------------------------------->
Max(s)      Min(s)      Average(s)    InContFile  throughput filesize
--------   ----------   -----------   ----------  ---------- ----------
-----
--------   ----------   -----------   ----------  ---------- ----------
-----
.
```

```
#! /pkg/gnu/bin/perl

# NAME :     dirsimpleput
# PURPOSE:   test average sput time to a collection
# SYNOPSIS   dirsimpleput Dir Init
# EXAMPLE:   dirsimpleput jz5/srb archiver-wustl
# DESCRIPTION:
#            Dir:  a directory
#            Init: initial collection in which the archived directory
#            will be stored.


if(($ARGV[0] =~ /^-h/) || ($#ARGV<0)){
   print "Usage: dirsimpleput Dir Init \n";
   exit 0;
}
$totaltime=time;
system("Sinit");
        @totalsizes=split(" ",`du -s -k $ARGV[0]`);
            $times=time;
            `Sput -r -f $ARGV[0] .`;
            $times=time-$times;

      $throughputs=($totalsizes[0]*8)/(1024*1024*$times);
      $throughput=sprintf("%.4f\n",$throughputs);

print " dirname for sput:$ARGV[0] \n";
print" dir size: $totalsizes[0]\n";
print " throughput(Mb/s):   $throughput \n";
`date`;
$totaltime=time-$totaltime;
print"script run time(s): $totaltime\n";
```

```
#! /pkg/gnu/bin/perl

# NAME :    dirsimpleget
# PURPOSE:  test average sget time to a collection
# SYNOPSIS  dirsimpleget Init Dir
# EXAMPLE:  dirsimpleget archiver-wustl jz5/srb

# DESCRIPTION:
#           Dir: a directory
#           Init: initial collection in which the archived directory
#           will be stored.


if(($ARGV[0] =~ /^-h/) || ($#ARGV<0)){
   print "Usage: dirsimpleget Init Dir\n";
   exit 0;
}
$totaltime=time;
system("Sinit");
        @totalsizes=split(" ",`du -s -k $ARGV[0]`);
            $times=time;
            `Sget -r -f $ARGV[0] $ARGV[1]`;
            $times=time-$times;

      $throughputs=($totalsizes[0]*8)/(1024*1024*$times);
      $throughput=sprintf("%.4f\n",$throughputs);

print " dirname for sput:$ARGV[0] \n";
print" dir size: $totalsizes[0]\n";
print " throughput(Mb/s):   $throughput \n";

$totaltime=time-$totaltime;
print"script run time(s): $totaltime\n";
```

```
#! /pkg/gnu/bin/perl

# NAME :     getcont
# PURPOSE:   test average sget time from a container
# SYNOPSIS   getcont Dfile Times
# EXAMPLE:   getcont datafile 5

# DESCRIPTION:
#            Dfile: a file contains all files for the testing purpose.
#            Times: times of repeated operations


if(($ARGV[0] =~ /^-h/) || ($#ARGV<1)){
   print "Usage: getcont Dfile Times \n";
   exit 0;
}

open(TESTDATA, "$ARGV[0]") || die ("can't open $ARGV[0]");

$j=0;
while(<TESTDATA>){
                chop;
                $testfile[$j]=$_;
                $j++;
}
$numb=$#testfile;

system("Sinit");

for($j=0; $j<=$numb; $j++){

      $filesize=-s $testfile[$j];
      $iterator=$ARGV[1];
      $sum=0; $max=0; $min=0; $first=1;
      while($iterator){
            $times=time;
            `Sget -f  $testfile[$j] $testfile[$j]`;
            $times=time-$times;

            $sum +=$times;

            if($first){
               $max=$min=$times;
               $first=0;
            }else {
               if($max<$times){
               $max=$times;}
               if($min>$times){
                 $min=$times;}
               $first=0;
            }
      $iterator--;
      }
      $avgs=$sum/$ARGV[1];
      $avg=sprintf("%.5f\n",$avgs);
      $throughputs=($filesize*8)/(1024*1024*$avg);
      $throughput=sprintf("%.4f\n",$throughputs);
```

```
      write;

 }

$timec=time;
system("Ssyncont -d $contname");
$timec=time-$timec; print "syn-time: $timec\n";

close(TESTDATA);
system("Sexit");
print:" repeating times: $ARGV[1]\n";

format STDOUT=
@<<<<<<<<< @<<<<<<<<<<<< @<<<<<<<<<<<<<<< @<<<<<<<<<<< @<<<<<<<<<<<
@<<<<<<<<<<<<<<<<<<
$max, $min, $avg, $testfile[$j], $throughput , $filesize
.


format STDOUT_TOP=
Page @<<<<<
$%

just sget time (thoughput unit: Mb/s)
<---------------------------------->
Max(s)       Min(s)       Average(s)    InContFile  throughput filesize
--------   ----------   -----------   ----------  ---------- ----------
-----
--------   ----------   -----------   ----------  ---------- ----------
-----
.
```

```
#! /pkg/gnu/bin/perl

# NAME :     putcont
# PURPOSE:   test average sput time to a container
# SYNOPSIS   putcont Dfile Cont
# EXAMPLE:   putcont datafile test_cont

# DESCRIPTION:
#            Dfile: a file contains all files for the testing purpose.
#            Cont: a container name to contain all files listed in the
#            file Dfile


if(($ARGV[0] =~ /^-h/) || ($#ARGV<1)){
   print "Usage: putcont Dfile Cont \n";
   exit 0;
}

open(TESTDATA, "$ARGV[0]") || die ("can't open $ARGV[0]");
$j=0;
while(<TESTDATA>){
                chop;
                $testfile[$j]=$_;
                $j++;
}
$numb=$#testfile;

system("Sinit");
$contname=$ARGV[1];

system("Smkcont -S cont-sdsc $contname");

for($j=0; $j<=$numb; $j++){
       $filesize= -s $testfile[$j] ;
       $times=time;
       `Sput -f -c $contname $testfile[$j] $testfile[$j]`;
       $times=time-$times;

       $throughputs=($filesize*8)/(1024*1024*$times);
       $throughput=sprintf("%.4f\n",$throughputs);


       write;

 }
$timec=time;
system("Ssyncont -d $contname");
$timec=time-$timec; print "syn-time: $timec\n";

close(TESTDATA);
system("Sexit");

format STDOUT=
 @<<<<<<<<<< @<<<<<<<<<< @<<<<<<<<<< @<<<<<<<<<<<<<<<<<
$times $testfile[$j], $throughput , $filesize
.
```

```
format STDOUT_TOP=
Page @<<<<<
$%

just sput time
<------------------------------->
  put(s)        InContFile  throughput filesize
 -----------    ----------  ---------- ---------------
 -----------    ----------  ---------- ---------------
.
```

```
#! /pkg/gnu/bin/perl

# name : contfill

# SYNOPSIS  contfill testdatafile containersizefile

# testdatafile: input file to a container
# containersizefile: a file contains different container sizes


if(($ARGV[0] =~ /^-h/) || ($#ARGV<0)){
 print "Usage: contfill testdatafile containersizefile\n";
   exit 0;
}
$totaltime=time;

open(CONTSIZE, "$ARGV[1]") || die ("can't open $ARGV[1]");
$j=0;

while(<CONTSIZE>){
        chop;
      $size[$j]=$_;
      $j++;
}
$sizes=$#size;

system("Sinit");

$filesize=-s $ARGV[0];
$contsource="cont-sdsc";
$contsource1="brainmap-wustl-container";

for($i=0; $i<=$sizes; $i++){

     $contnames=$size[$i]."."."$ARGV[0];
     $sum=0;
     system("Smkcont -S $contsource -s $size[$i] $contnames");
     $sumsize=0;
     $n=0;
     while(($sumsize<$size[$i]) && (($sumsize+ $filesize)<=$size[$i])){

            $localfile=$ARGV[0]. "." . $i. $n;
            $putime[$n]=time;
            `Sput -f -c $contnames $ARGV[0] $localfile`;
            $putime[$n]=time-$putime[$n];
            $sum=$sum +$putime[$n];
            if($putime[$n]){
              $bws=($filesize*8)/(1024*1024*$putime[$n]);
            }
            else{$bws=0;}
            $bw=sprintf("%.5f\n",$bws);
            $sumsize=$sumsize + $filesize;
            $n++;
            write;
    }
      $timesyn=time;
      system("Ssyncont -d $contnames");
```

```
        $timesyn=time-$timesyn;
        $sum =$sum +$timesyn;
        $avgs=$sum/$n;
        $BWss=($filesize*8)/(1024*1024* $avgs);
        $BWITH=sprintf("%.5f\n",$BWss);
        print" avgput time:  $avgs \n";
        print " BW: $BWITH \n";
        print "total files in container: $n \n";


}

close(CONTSIZE);
system("Sexit");
$totaltime=time-$totaltime;

print"\n total running time of this script is: $totaltime\n";

format STDOUT=
@<<<<<<< @<<<< @<<<<<<<<<<< @<<<<<<< @<<<<<<<<<<
$ARGV[0], $n, $size[$i] , $putime[$n-1],$bw
.


format STDOUT_TOP=
Page @<<<<<
$%

Sputime to container:   (# files per container)   (bw=Mbit/second)
<-------------------------------->
InContFile NO.Copy ContSize Sputime (Sput)BWith
---------  -----  ---------- ------- -------
---------  -----  ---------- ------- -------
.
```

```
#! /pkg/gnu/bin/perl

# name : contretrieve

# SYNOPSIS  contretrieve testdatafile containersizefile

# testdatafile: input file to a container
# containersizefile: a file contains different container sizes

if(($ARGV[0] =~ /^-h/) || ($#ARGV<0)){
 print "Usage: contretrieve testdatafile containersizefile\n";
   exit 0;
}
$totaltime=time;

open(CONTSIZE, "$ARGV[1]") || die ("can't open $ARGV[1]");
$j=0;

while(<CONTSIZE>){
        chop;
      $size[$j]=$_;
      $j++;
}
$sizes=$#size;

#print "$numb $sizes\n";

system("Sinit");
$filesize=-s $ARGV[0];

for($i=0; $i<=$sizes; $i++){

     $contnames=$size[$i]."."$ARGV[0];
     $sum=0;
     $sumsize=0;
     $n=0;
     $numb=int($size[$i]/$filesize); $a=1;

    while(($sumsize<$size[$i]) &&(($sumsize+ $filesize)<=$size[$i])){

            $localfile=$ARGV[0]. "." . $i. $n;
            $putime[$n]=time;
            $obj=$ARGV[0]."."."copy";
            `Sget -f $localfile $obj`;
            $putime[$n]=time-$putime[$n];
            $sum=$sum +$putime[$n];
            if($putime[$n]){
             $bws=($filesize*8)/(1024*1024*$putime[$n]);
            }
            else{$bws=0;}
            $bw=sprintf("%.5f\n",$bws);

            if($n==int(($numb*$a)/16)){
            ## list time for getting some part of total files

            print" time for getting $a/16 of total files: $sum\n";
                  if($a<2){$a+=1;} else {$a+=2;}
```

```
            }
            $sumsize=$sumsize + $filesize;
            $bwav=($sumsize*8)/(1024*1024*$sum);
            $bwavs=sprintf("%.5f\n",$bwav);
            $n++;
            write;
        }
          $timesyn=time;
          system("Ssyncont -d $contnames");
          $timesyn=time-$timesyn;
          $sum =$sum +$timesyn;
          $avgs=$sum/$n;
          $BWss=($filesize*8)/(1024*1024* $avgs);
          $BWITH=sprintf("%.5f\n",$BWss);
          print" avgput time:  $avgs \n";
          print " BW: $BWITH \n";
          print "total files in container: $n \n";


}

close(CONTSIZE);
system("Sexit");
$totaltime=time-$totaltime;

print"\n total running time of this script is: $totaltime\n";

format STDOUT=
@<<<<<<< @<<<< @<<<<<<<<<<< @<<<<<<< @<<<<<<<<<< @<<<<<<<<<<<
$ARGV[0], $n, $size[$i] , $putime[$n-1],$bw, $bwavs
.


format STDOUT_TOP=
Page @<<<<<
$%

Sgetime from container:   (# files per container)   (bw=Mbit/second)
<---------------------------------->
InContFile NO.Copy ContSize Sputime (Sput)BWith
---------  -----  ---------- ------- -------
---------  -----  ---------- ------- -------
.
```