

Optimising the use of Storage Resource Broker to Host Manuscript Images and Virtual Exhibitions Online

Dr Michael Meredith & Professor Peter Ainsworth
Department of French
University of Sheffield
United Kingdom
E-mail: {M.Meredith, P.F.Ainsworth}@shef.ac.uk

Abstract

Through our research into hosting manuscript collections and virtual exhibitions using Storage Resource Broker as a file system, several potential pitfalls have arisen from its practical use, which are detailed in this paper. The main areas that are considered include uploading and retrieving data, optimal client-side read buffer sizes and user account permissions. While this paper focuses on accessing and manipulating many small files (typically less than 1MB in size) a review of performance metrics beyond this size is also provided for completeness.

1 Introduction

Storage Resource Broker (SRB) is a middleware application that facilitates a distributed file system (i.e. a Data Grid) which is presented to the user as a single hierarchical structure¹. Access to files stored on SRB can be controlled using user permissions. In addition to providing distributed file storage, the SRB system has the ability to store metadata about files, which can be searched and related back to the files. SRB has been developed and is distributed by the San Diego Supercomputing Centre.

Storage resource broker is currently used in our research and teaching environment to store images of digitised manuscripts and virtual exhibition content. These are accessible via a Java application that can be run on a desktop as a standard application, or from a website as an applet. To connect to the SRB system we use the Jargon interface provided by the San Diego Supercomputing Centre. The majority of the data held on our SRB system is write once, read many times. File permissions are however routinely adjusted, usually to add new users. A breakdown of the typical file distribution and usage is outlined below for our two major activities and section 2 outlines how the performance of SRB system is measured with respect to these types of files and their typical sizes.

1.1 Resources for manuscript image data

When hosting manuscript image data, each manuscript consists of multiple image files which are catalogued using a single XML file. Each page within a manuscript consists of a single JP2 (JPEG 2000) image file, which stores the high-resolution data, and a single JPEG image file, which is a low resolution thumbnail. Optionally, each page may contain a colour bar image in a JPEG format. The XML catalogue file is typically about 120.0KB in size, the JPEG thumbnails are approximately 1.5KB, the colour bar images are 11.5KB, and the JP2 files are around 7.9MB. A top-level XML collections file, which references the individual manuscript catalogue XML files, is also maintained and is approximately 1.2KB.

When browsing the manuscripts, first the top-level XML collections file is opened. With a selection of manuscripts available, the user can dig into a specific one. This requires the manuscript catalogue file to be loaded, which is followed by displaying a portion of thumbnails. The main JP2 files are only accessed by the user if they wish to view a manuscript page in full-detail.

In terms of browsing the manuscript collection it is therefore important to quickly access and read the data in from the smaller files, i.e. the XML catalogue files and JPEG thumbnails.

1.2 Resource for virtual exhibitions

¹ See http://www.sdsc.edu/srb/index.php/Main_Page for the SRB homepage

The data files used to host the virtual exhibitions can broadly be split into two different categories. The first type consists of XML files that script the look and feel of the exhibition, plus provide textual content to the user. These files are typically between 300 bytes and 32KB. The second type of data includes the non-textual media content such as images (possibly from the manuscript dataset outlined above), audio files (generally in MP3) and video files. These types of files can range from between a few kilobytes to many megabytes. Common actions, such as hiding or showing user controls, are scripted in a single XML file that is referenced multiple times by other XML script files to reduce redundancy. This means that to display a screen, many XML files may need to be loaded. For a given page within a virtual exhibition, there is typically an audio narration and one or more image files.

Therefore, in order to deliver a virtual exhibition to the end-user, quickly accessing many small files is a high priority, if only to make the user interface responsive. High-resolution images within a virtual exhibition require a similar level of attention as those described in section 1.1 (primarily because the underlying software technology is the same between the two instances). Audio and video files can be streamed from the SRB system so it is less time-critical to retrieve the whole of this data before a page is displayed to the user.

2 Measuring the Performance of SRB

The types of datasets and their uses underline the need to store and quickly access many small and medium-sized files. In our applications this is a critical component because they directly influence the responsiveness observed by the end-user. In section 3, the performance of uploading files of various sizes is analysed and a detailed review of the different upload mechanisms is provided. Section 4 looks at the performance of retrieving data from the SRB system, which includes file access times and optimal read buffer sizes for the client. Section 5 briefly examines the performance of setting user permissions and section 6 gives a review and concludes of the results obtained within this paper.

2.1 Test environment

The empirical results have been obtained using SRB 3.3.1 installed on a Sun Fire V240 server with two 1.28GHz UltraSPARC IIIi CPUs and 2GB of RAM. The storage used for the SRB system on this server is held on a Sun storage array which uses RAID SCSI drives (striped and mirrored). The host computer is a Windows XP PC with an Intel Pentium D 2.8GHz processor and 3GB of RAM. There is a LAN network connection between the server and host where the minimum bandwidth connection between the two points is 100Mbps. Before the timings are started during the tests, an appropriate user has already been authenticated with the SRB server and a file system connection has been made with the Jargon interface.

The SRB server that these results are obtained from is not a clean installation and it contains live data which is typical of the types of data outlined in section 1. However, in order to maintain consistency between test sets, no new live data was uploaded to the server and the trash was routinely emptied for all users (Srmtrash -A) and the SRB server and underlying database was frequently vacuumed (install.pl vacuum) and re-indexed (install.pl index).

3 Uploading Data

SRB provides two different mechanisms for uploading data: single file upload and bulk file upload. The later approach allows multiple files to be uploaded as a single amalgamation which is decompiled on the server into its constituent files. Further details of this mechanism are given in section 3.2, which analyses its performance, after the individual approach is investigated in section 3.1. The final subsections of this section compare the two different approaches and draw comparative conclusions.

3.1 Individual file upload

To measure the performance of the SRB system at handling individual file uploads, test files of incrementally larger sizes were uploaded to the SRB system. The test file sizes ranged from 1KB to

64MB and doubled in size at each increment. For each of the file sizes, the upload process was repeated 1,000 times. The process of uploading a single file can be partitioned into the following steps:

- I.1) Open a new writable file on the SRB system
- I.2) Open the local file, Copy data from the local file to the SRB file, Close the local file
- I.3) Close the file on the SRB system
- I.4) Loop back to (1) until all files have been transferred

To eliminate the client-side local overhead incurred at step I.2 (opening, reading and closing a local file), the data to be transferred to the SRB system has been buffered in primary memory before being written to the SRB system. The following sections outline the performance results obtained from each of the key SRB system interaction processes.

3.1.1 Opening a new SRB file: step I.1

Figure 1 outlines the timings obtained for opening files for writing to where the files do not already exist within the SRB system before the open operation is issued. The performance values for opening a file across the different test sizes are consistent and therefore only one plot is given which is representative of the whole dataset.

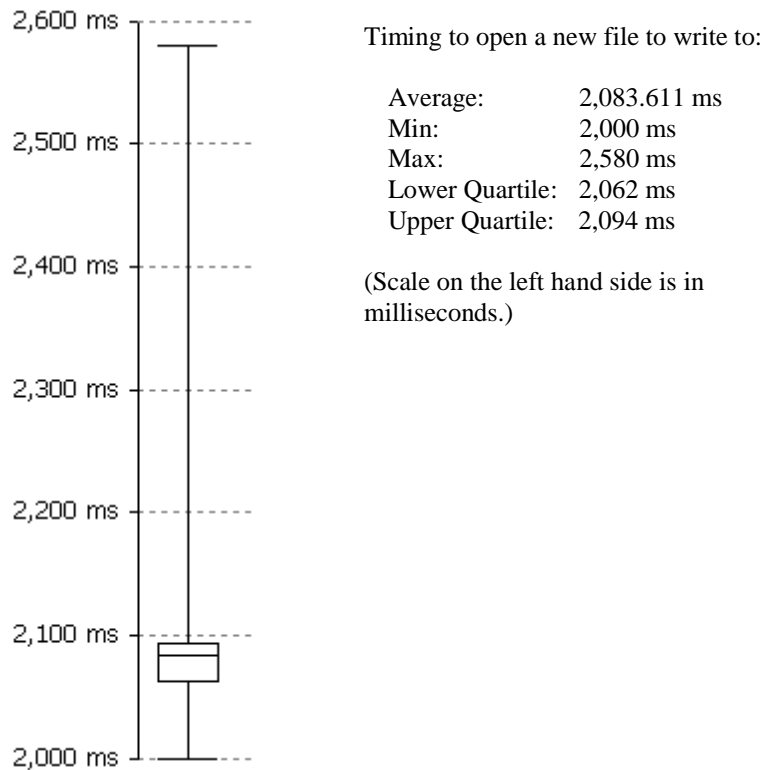


Figure 1: Timings obtained for opening a new file for writing to

Figure 1 illustrates a high level of consistency in the time it takes to open a new file ready to be written to. The main outcome to note from these results is the average time it takes to open a single file which is just under 2.1 seconds. Given that many of our files are quite small this overhead can be significant when compared to the time to transfer the actual data, as our empirical results demonstrate in the following section.

3.1.2 Transferring the data: step I.2

This section focuses on the transfer time incurred while uploading data to the SRB system (step I.2). It should be noted that data transfer time includes a flush to ensure all the data is written to the server before the file is closed. The results obtained are outlined in Figure 2 and tabularised in Appendix A.

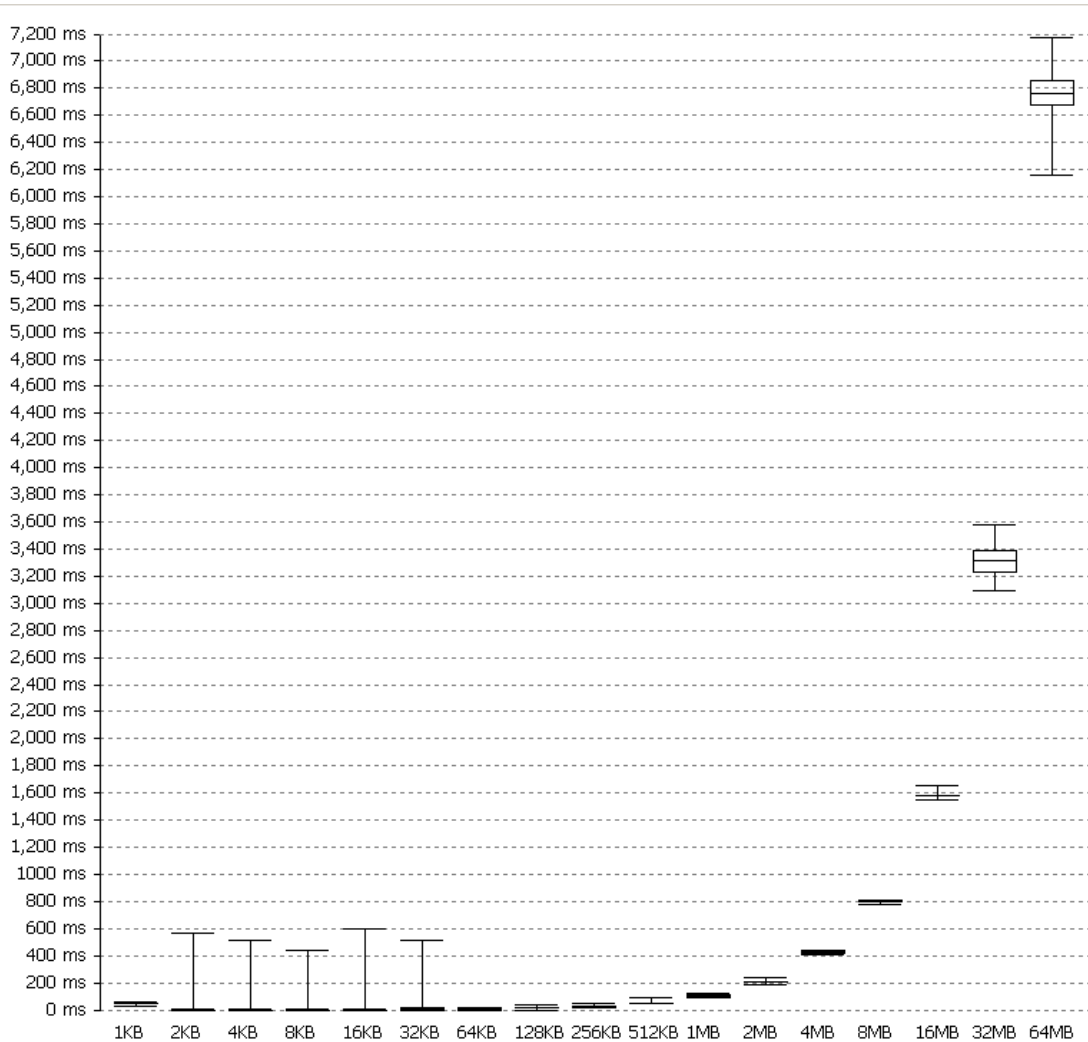


Figure 2: Transfer timings obtained for uploading data to the SRB server using a sample of 1,000 files for each size category

Transfer times for the files only start to become significant above 512KB because the transfer of smaller files do not reach the bandwidth limit imposed by the physical network hardware. The 1KB transfers do however exhibit a curious artefact compared to other smaller file sizes. The transfer time for 1KB files consistently takes 47ms, however the larger file sizes up to 512KB all take less time to transfer the data; a 2KB file consistently transfers data in less than 3ms. The reason for this abnormality is currently unknown, although probably attributed to the way that the SRB client deals with data buffers and/or TCP/IP packet size and stuffing. A similar effect is also observed during the bulk upload section.

The results for smaller files also exhibit relative high outlier values, although as these values tend to be uncommon and typically less than 600ms they have been attributed to network and/or server congestion. Further analysis will therefore focus on data within the upper and lower quartile ranges as we are interested in typical performance results that are repeatable under general operating conditions.

Aside from these two artefacts in the results, the plot of Figure 2 demonstrates an expected exponential growth in transfer time that is consistent with the increase in file size. During the transfer test of the larger files, network activity was seen to be typically >80% on the 100Mbps network interface card of the client computer and thus it would be reasonable to conclude that the bottleneck in transfer for these tests is the network infrastructure and the SRB server was able to meet the demand. This is further supported by other SRB performance evaluation reports².

² Nadeem, S. A., Bayliss, C., Hewitt, M. P., "Storage Resource Broker Performance Evaluation Report", Technical Report for the UK Engineering Task Force, October 2007

3.1.3 Closing the SRB file – step 1.3

Figure 3 outlines the timings obtained for closing files once they have been written to. Before timings are taken to close a file, all the data has been flushed to the server during the transfer stage and therefore the close procedure simply permits the SRB system to tidy up any overhead it used to create new files and update the directory listings/MCAT database. The performance values for opening a file across the different test sizes are consistent and therefore only one plot is given which is representative of the whole dataset.

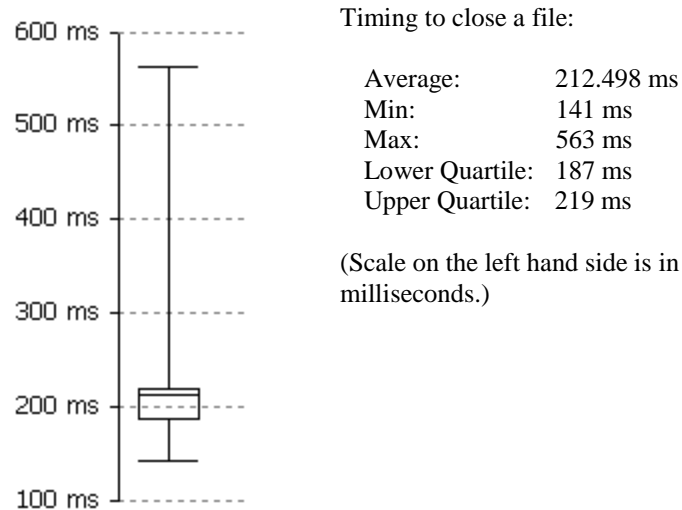


Figure 3: Timings obtained for closing a file

The results given in Figure 3 illustrate that a consistent amount of time is required to complete the close operation, which is much less than the time required to open a new file. What is worth noting is that the time to close a file is generally greater than the time to transfer files up to 2MB in size.

3.1.4 Individual Upload Conclusions

The process of copying data from one location to another, regardless of the destination, will always be subject to the overhead incurred when the new file is opened and closed. Therefore the fact that the SRB system also exhibits this delay is an expected outcome. What is of interest from these results is the significant amount of time it takes to perform these operations, which is considerably higher than via FTP running on the same server. The typical combined overhead is 2.3 seconds per file, which is approximately the same time it takes to perform the data transfer of 24MB. As file sizes increase this overhead becomes less significant, however for smaller files, the overhead is less trivial. The processing of smaller files is considered further after a review of uploading data in bulk is given in section 3.2.

The exact cause for the open and close overhead is unknown but the majority can probably be attributed to the SRB system connecting and performing MCAT operations such as lookup and updates. Evidence to support this was seen during these tests where the overhead times were seen to grow over time if a re-indexing of SRB system was not performed routinely between tests. It should however be noted that within our test set we were uploading many thousands of files which were subsequently deleted. This will have cluttered the MCAT database and thus the catalogue would not normally need re-indexing as frequently unless it was under a heavy load.

3.2 Bulk file upload

The bulk upload operation allows multiple files to be uploaded to the SRB system as a single concatenated file which the server decompiles and registers. The major benefit of this is that only one SRB file open and close operation is needed on the client-side for many files, a process that section 3.1 demonstrated to take a non-trivial amount of time. The MCAT operations also appear to be carried out

more quickly, which is probably due to there being functions that handle such operations more efficiently. The bulk upload operation does however suffer from the need to copy the data twice: the first is to the concatenated file and second is from the concatenated file to the individual files. The double copy is however performed server-side so the second copy is subject only to the data rate of the hard drives and not the network.

Procedurally, the stages required to achieve a bulk upload of files to the SRB system can be described in the following steps:

- B.1) Open a new “temporary” file on the SRB system which the data will be written to
- B.2) Open the local file, Copy data from the local file to the SRB file, Close the local file
- B.3) Create a new metadata record locally (client-side) that contains information which the SRB server will use to decompile the merged file into its constituent parts
- B.4) Loop back to (2) until all local files have been transferred
- B.5) Close the SRB file
- B.6) Register the bulk upload with the SRB server using the metadata records. This will allow the server to decompile the bulk file

Similar to the individual file upload performance measures, the client-side overhead incurred at step B.2 has been eliminated by buffering the data in primary memory before being written to the SRB system. The performance measures obtained also eliminate step B.3 because this happens on the client-side only and requires negligible time. However, in order to meet these conditions we are not using the Jargon helper methods in the *SRBFile* class to perform the bulk upload operations. This is because the methods do not immediately facilitate the transfer of data held and buffered in primary memory but require an actual file. Furthermore, the helper bulk upload methods within Jargon selectively choose to upload the file individually or within the bulk collection depending on its file size; files less than a given threshold are uploaded using the bulk mechanism while those above the threshold are done individually. The Jargon constant that controls this within the *SRBFile* class is called *MAX_BULK_FILE_SIZE* and is set at 2MB. Unfortunately, the method to register the metadata records with the SRB server (step B.6) is not publically exposed within the Jargon API. To make this method available we have changed the *void SRBFileSystem.srbBulkLoad(int catType, String bulkLoadFilePath, SRBMetaDataRecordList[] rl)* method from private to public accessibility.

To measure the performance of the bulk upload mechanism, 200 files of identical size were uploaded at a time. The test file sizes were increased from 1KB to 16MB, doubling in size at each increment. Each bulk upload operation for a specific file size was repeated 100 times. The performance results obtained for transferring the data (step B.2) are given in Figure 4 and Appendix B.1. Figure 5 and Appendix B.2 provide the timings measured to perform the bulk upload registration process (step B.6). Timings for opening and closing the SRB file have not been provided in this section as they are in agreement with those recorded for individual file uploads; Figure 1 and Figure 3 respectively.

Figure 4 demonstrates an expected exponential growth in transfer times, similar to that described in section 3.1. Like the transfer times for individual files, Figure 4 exhibits a much higher transfer time for 1KB files than its neighbours. The consistency with which this occurs indicates that there is something fundamental happening with these transfers that needs further exploration.

The exponential growth in registration times is also expected given that more data needs to be copied by the SRB server. The range of results obtained for any given file size is however larger than other performance timings have revealed. This can probably be attributed to network congestion, particularly for the larger upload sizes, and drive fragmentation because the server load by other users was low during these tests (one flaw in these experimental timings is that a drive defragmentation was not performed). However, despite the larger range, the typical results are consistent enough to perform further analysis on them.

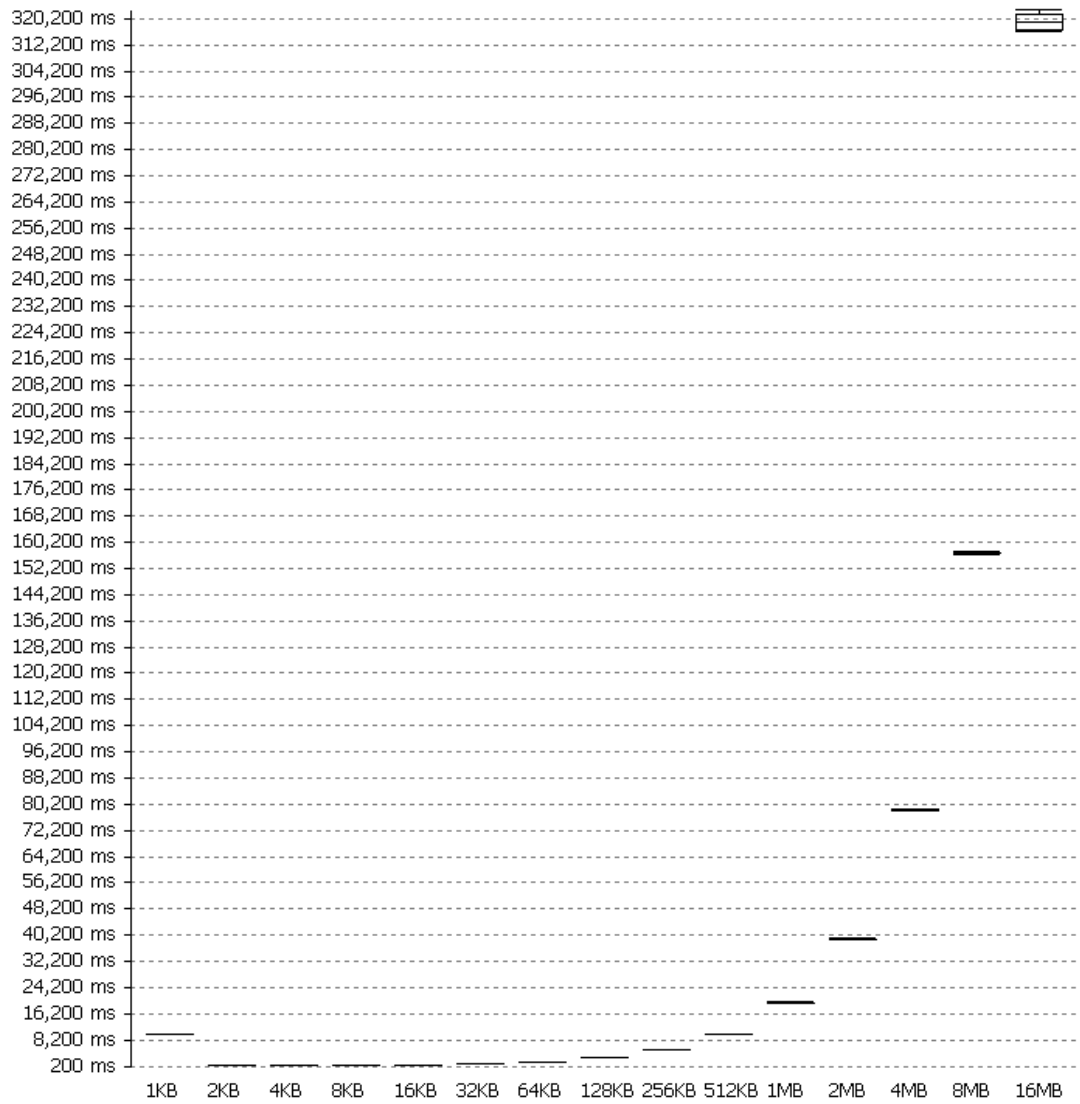


Figure 4: Data transfer measures obtained for uploading data to the SRB server using the bulk upload mechanism. 200 files of each size were uploaded at the same time. Note that 32MB file transfers have not been included on this plot to increase resolution of other results due to exponential growth although values are presented in Appendix B.2.

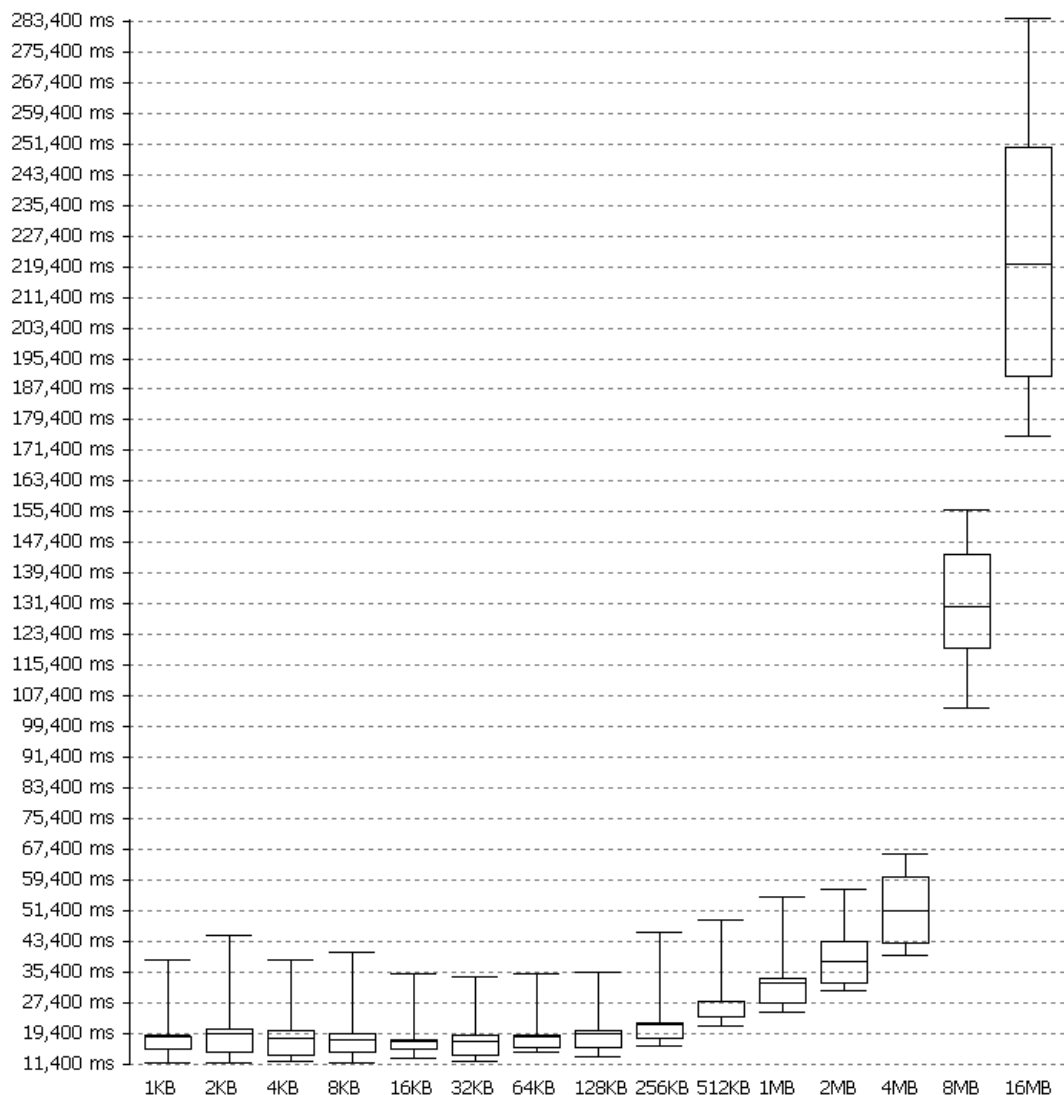


Figure 5: Data register measures obtained for uploading data to the SRB server using the bulk upload mechanism. Note that 32MB file transfers have not been included on this plot to increase resolution of other results due to exponential growth although values are presented in Appendix B.2.

3.3 Individual upload vs. bulk upload

With the exception of 1KB file sizes, the results from sections 3.1 and 3.2 have shown consistent performance trends with a cross-over in terms of efficiency between uploading files individually as opposed to in a bulk collection due to their respective pros and cons. This relationship is plotted in Figure 6 which shows the total time it would take to upload 200 files using the different mechanisms across the different file sizes based on the results obtained.

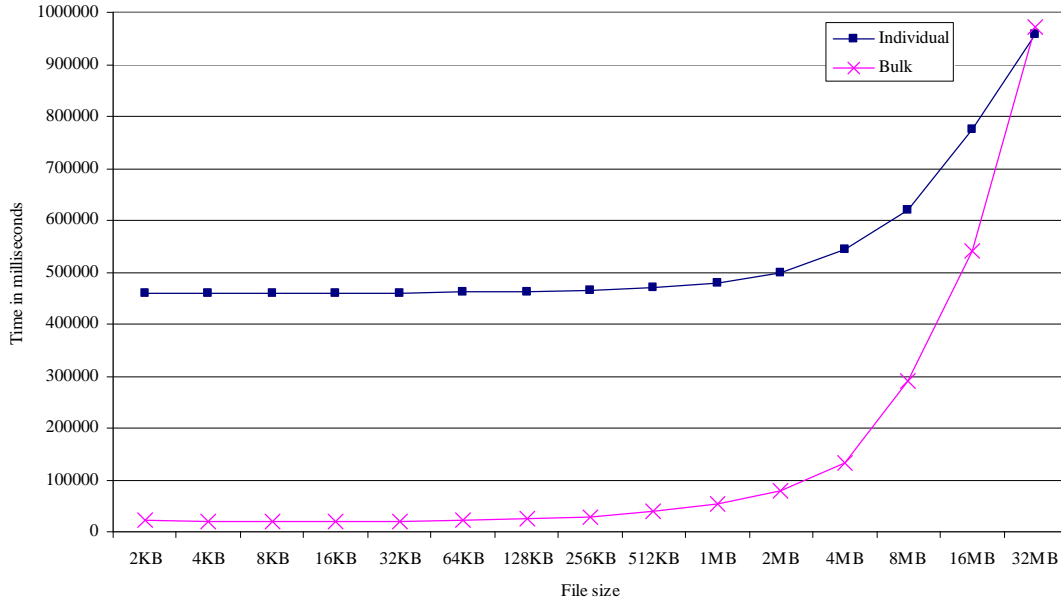


Figure 6: Plotting the total transfer times required to copy 200 files individually and using the bulk mechanism over a range of different file sizes

As figure 6 illustrates, somewhere between 16MB and 32MB files it becomes more efficient to upload files individually than using the bulk upload approach; effectively the multiple client-side SRB file open and closing operation overhead is exceeded by the penalty of copying the data twice on the server. Recall from section 3.2 that the Jargon API has a default threshold of 2MB which is far below the empirical threshold obtained.

Using the component results obtained from sections 3.1 and 3.2 the following mathematical relationships can be derived:

Individual Upload:

$$t = n \left(o_t + \frac{s}{r} + c_t \right) \tag{1}$$

where:

- t is the time to complete the operation in milliseconds
- o_t is the time to open a new SRB file in milliseconds
- s is the file size in bytes
- r is the data transfer rate in bytes per millisecond
- c_t is the time to close the SRB file
- n is the number of files to upload

From the empirical results o_t can be set to 2,083.6 ms and c_t equal to 212.5 ms. The data transfer rate, r , can be determined by dividing the transfer times in Figure 2 by the number of bytes transferred. A plot of these is given in Figure 7. Data transfers of 256KB and below are not utilising the full network bandwidth available, although above this value the transfer rates can be seen to plateau to an average rate of 10,266 bytes/ms (this equates to approximately 82Mbps, which is on a par with that expected due to the limiting network transport bandwidth of 100Mbps). While this constant ignores the lower data transfer rates observed with smaller files, it does represent the best typical transfer rate. Furthermore, from Figure 6, the performance crossover between individual and bulk upload happens above 512KB and thus this constant is applicable for this domain of values.

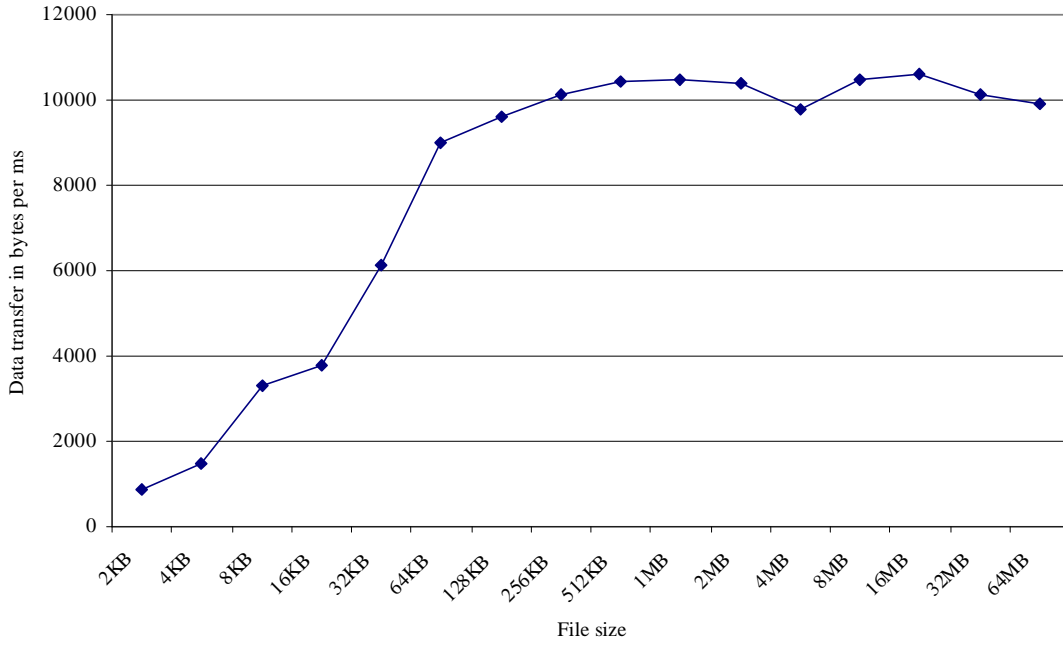


Figure 7: Data transfer rates achieved while uploading individual files

Bulk Upload:

$$t = o_t + \frac{(n * s)}{r} + c_t + n \left(a + \frac{s}{b} \right) \quad (2)$$

where

a is the server-side overhead to open, close and register a file

b is the server-side data transfer rate for the second copy operation (local disk to disk)

The data transfer rates observed when performing a bulk upload operation quickly plateau to the maximum value determined above. The constants a and b can be derived using the empirical data and the registration time component of Equation 2:

Using 2MB and 8MB data values:

$$t_2 = 200 \left(a + \frac{s_2}{b} \right) \quad \text{and} \quad t_8 = 200 \left(a + \frac{s_8}{b} \right)$$

$$\Rightarrow t_8 - t_2 = 200 \left(\frac{s_8 - s_2}{b} \right)$$

$$\Rightarrow 130569.98 - 38235.34 = 200 \left(\frac{2^{23} - 2^{21}}{b} \right)$$

$$\Rightarrow 92334.64 = 200 \left(\frac{3 * 2^{21}}{b} \right)$$

$$\Rightarrow b = \frac{200 * 3 * 2^{21}}{92334.64}$$

$$\Rightarrow b = 13627.50968 \text{ bytes per millisecond } (\sim 13\text{MB/s})$$

$$\begin{aligned}
&\text{from } t_8 = 200 \left(a + \frac{s_8}{b} \right) \\
\Rightarrow 130569.98 &= 200 \left(a + \frac{2^{23}}{13627.50968} \right) \\
\Rightarrow a &= \frac{130569.98}{200} - \frac{2^{23}}{13627.50968} \\
\Rightarrow a &= 37.2856 \text{ milliseconds}
\end{aligned}$$

To determine the crossover point, equation 1 and equation 2 are equated:

$$\begin{aligned}
n \left(o_t + \frac{s}{r} + c_t \right) &= o_t + \frac{(n*s)}{r} + c_t + n \left(a + \frac{s}{b} \right) \\
\Rightarrow n*o_t + \frac{n*s}{r} + n*c_t &= o_t + \frac{(n*s)}{r} + c_t + n \left(a + \frac{s}{b} \right) \\
\Rightarrow \frac{(n-1)}{n} (o_t + c_t) &= a + \frac{s}{b} \tag{3}
\end{aligned}$$

Equation 3 illustrates that the crossover point between using individual and bulk upload would appear to depend on two factors: the file size and the number of files to upload. However as the number of files, n , grows, its contribution in Equation 3, $(n-1)/n$, tends to 1. Equation 3 can therefore be evaluated to determine the file size at which the upload approaches crossover:

$$\begin{aligned}
o_t + c_t &= a + \frac{s}{b} && \text{(from Equation 3 and taking } n \text{ to its limit)} \\
\Rightarrow s &= b(o_t + c_t - a) \\
\Rightarrow s &= (2083.6 + 212.5 - 37.2856) * 13627.50968 \\
\Rightarrow s &= 30782015 \text{ bytes} \\
\Rightarrow s &= 29.356 \text{ MB}
\end{aligned}$$

This result is further demonstrated in Figure 8 where the number of files is plotted against the crossover file size (Equation 3).

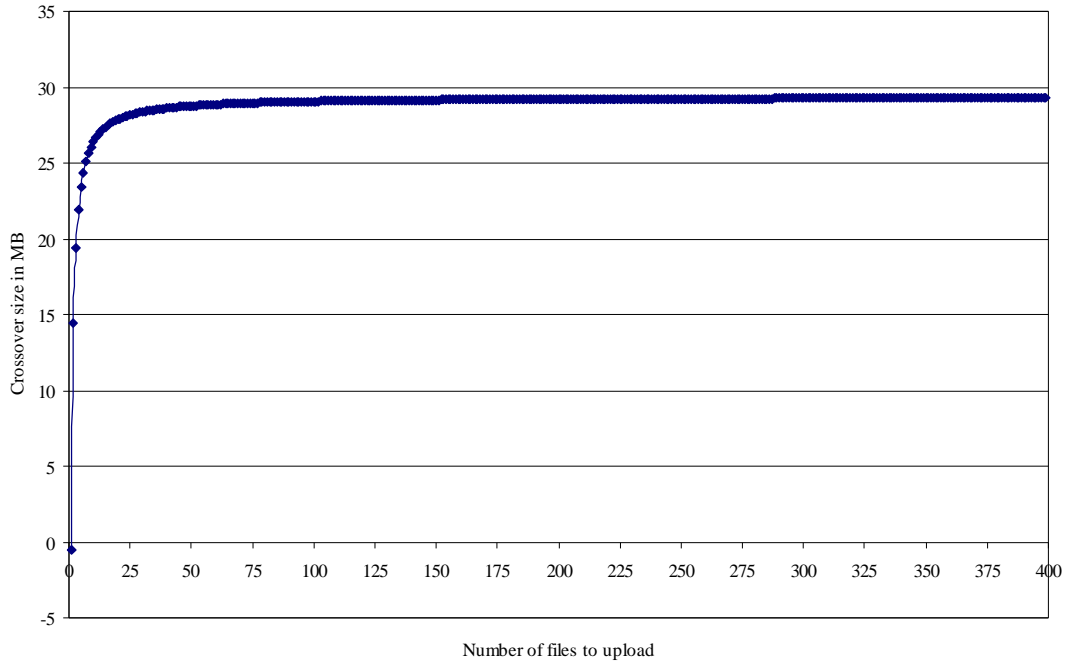


Figure 8: Plotting Equation 3 to illustrate where the crossover point occurs for any given number of files to be uploaded

Therefore the empirical results and analytical study suggest that when dealing with more than 10 files, file sizes below 29MB are best suited to the bulk upload mechanism, whereas larger files should be dealt with by uploading them individually. This theoretical boundary defines the crossover point seen in Figure 6.

3.4 Upload Conclusions

The results in this section demonstrate that the bulk file upload mechanism utilises the network bandwidth more efficiently than the individual file upload approach which is due to the high overhead in opening and closing files on the SRB system. However, the bulk upload approach also requires the data to be copied twice: once by the client to upload the data and once by the SRB server to put it into the correct structure during the registration phase. The results further indicated that the influencing factor for the registration phase comes from the size of file as opposed to the number of files. An important question is therefore at what point bulk uploading files becomes less efficient than uploading them individually, i.e. when does the file open/close overhead become less than the double copy penalty? The threshold used within the Jargon API is 2MB although the results obtained suggested a file size that is significantly larger at 29MB.

Another noteworthy result from the analysis is the time penalty incurred when opening a new SRB file to write to. Such a high penalty means that the SRB system handles small files inefficiently when creating them. However, if much of the data uploaded to the SRB system is write once, read many, the overhead can be viewed as more of an inconvenience, assuming it can be more quickly retrieved. The access of existing data files is considered in the following section.

4 Access Data

There is only one mechanism to access and retrieve data held on an SRB system which follows the traditional approach of opening one file at a time as required. In this section the performance of this mechanism is analysed and discussed in particular connection to the Jargon APIs.

4.1 Opening an existing data file

Before data can be retrieved from the SRB system, a file connection needs to be established. The performance timings obtained for creating this connection are given in Figure 9 and do not include opening a connection to the SRB file system itself, which is assumed to be already present. Figure 9 illustrates the timings for accessing 1,000 randomly sized files that exist on the SRB system.

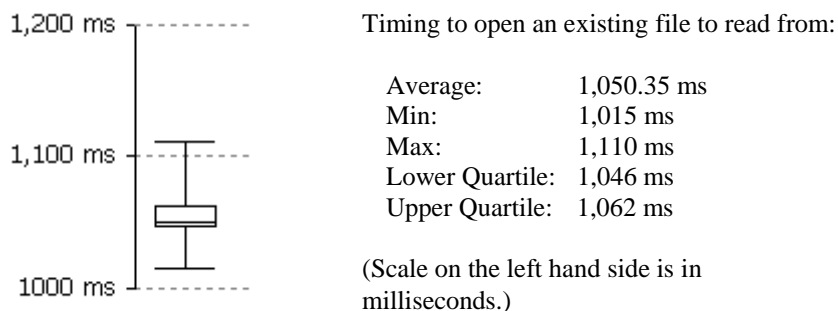


Figure 9: Timings obtained for opening an existing file to read from using a sample size of 1,000 files

The empirical data illustrates that the performance for opening a file connection to a SRB file is independent of file size. Furthermore, the time to open an existing file is less than that for opening a new file (Figure 1). However, the average connection time of over 1 second has practical implications when delivering content to an end-user, especially when many small files are needed to construct an interface, as in the case of our virtual exhibitions (section 1.2). The relatively high delay in opening a data file uncovered several other cravats of the Jargon API as workarounds were pursued.

4.1.1 Simultaneously file access

Opening multiple files simultaneously would appear to offer one solution to this problem, however in practice this causes further problems and fails to improve performance. The first issue associated with simultaneous access is that each SRB file that is opened requires an “unused” SRB file system connection. SRB file systems can be recycled when open files are closed, but attempting to open two or more SRB files at the same time using a single SRB file system handle causes the API to deadlock and neither data file can be accessed. Multiple SRB file systems therefore need to be opened, which introduces further overhead. This new overhead can be managed by maintaining a stack of SRB file system handles that are popped when required and pushed back on when the SRB file is closed.

Unfortunately, multiple SRB file system handles still do not permit better simultaneous access because the SRB file open method takes longer to complete when multiple files are open and/or being opened simultaneously. Furthermore, SRB file open requests also have a detrimental effect on transfer rates; even with multiple SRB file system handles, calls to open SRB files appear to be queued and processed synchronically and no other data transfer is permitted while the operation is still pending.

4.1.2 Multiple file concatenation

An alternative approach is therefore to concatenate multiple files into a single one, e.g. a ZIP file. This approach is not directly support by SRB so files need to be specially prepared. Furthermore an extra layer of data handling needs to be written into a client application to access files that are combined in this manner. Fortunately the SRB file object provides a skip method that permits specific regions of a concatenated file to be quickly accessed and retrieved; the skip method completes significantly more quickly than an open operation. However, the code within the Jargon SRBFileInputStream.skip(long n) method is incorrect³ and needs modification. The original skip method is given below:

```
public long skip(long n) throws IOException
{
    long length = available();
    if (length <= 0)
        return 0;
}
```

³ This relates to version 1.4 of the Jargon API

```

    long abspos=filePointer+n;
    if (abspos<0)
        n=-filePointer;
    else if (abspos>length)
        n=length-filePointer;

    fileSystem.srbObjSeek(fd, n, GeneralRandomAccessFile.SEEK_CURRENT);
    filePointer+=n;
    return n;
}

```

The problem lies with the use of the `available()` call which this method expects to return the length of the file, but in fact returns the length of the file minus the current position in the file. This is illustrated by the line in bold in the code below for the `available()` function:

```

public int available() throws IOException
{
    MetadataRecordList[] rl = null;
    MetadataCondition[] conditions = {
        MetadataSet.newCondition( SRBMetadataSet.DIRECTORY_NAME,
            MetadataCondition.EQUAL, file.getParent() ),
        MetadataSet.newCondition( SRBMetadataSet.FILE_NAME,
            MetadataCondition.EQUAL, file.getName() ) };

    MetadataSelect[] selects = {
        MetadataSet.newSelection( SRBMetadataSet.SIZE ) };

    int available = 0;
    try {
        rl = fileSystem.query( conditions, selects, 3 );
        if( rl != null ) {
            available = (int) (Long.parseLong(rl[0].getValue(
                SRBMetadataSet.SIZE ).toString()) - filePointer);
        }
    } catch ( IOException e ) {}

    if (available <= 0)
        return 0;
    else
        return available;
}

```

The net effect of this is that the current position within the file is subtracted from the total length twice within the skip function and when the `abspos` is tested against the `length` value, incorrect seek values can be generated. To fix this issue and improve performance of the skip function, we have recoded the skip method as follows:

```

public long skip(long n) throws IOException
{
    if (filesize==-1)
        filesize=available()+filePointer;

    long abspos=filePointer+n;
    if (abspos<0)
        n=-filePointer;
    else if (abspos>filesize)
        n=filesize-filePointer;

    fileSystem.srbObjSeek(fd, n, GeneralRandomAccessFile.SEEK_CURRENT );
    filePointer+=n;
    return n;
}

```

As opposed to keep calling the `available()` method each time the skip method is called, which incurs an SRB communication overhead, we assume that the file size never changes and therefore can

be determined once and stored for future use⁴. The `filesize` attribute is a non-static class variable and initially set to -1. When the revised `skip()` method is called for the first time it calls the `available()` method and adds back on the `filePointer` value from that returned. Thereafter the `filesize` is known for future calls. Although this is perhaps not the most elegant solution it localises the impact of the changes.

The combination of fixing the `skip` method and the concatenated file structure greatly improves file access. This approach does however decrease the transparency of the files held on the SRB system and requires the introduction of sub-level file processing when uploading and retrieving data.

4.2 Retrieving data

Through the practical use of the Jargon API, one further aspect merits discussion in relation to retrieving the data from an SRB file. Unlike local disk, HTTP and FTP data access, considerably larger read buffers are required to optimise the available network bandwidth. Whereas a 64KB read buffer is sufficient to achieve a high bandwidth utilisation over HTTP and FTP, retrieving data via the SRB system using this buffer size yields poorer results. This is illustrate in Figure 10, which plots read buffer size against overall data rate achieved for the SRB system.

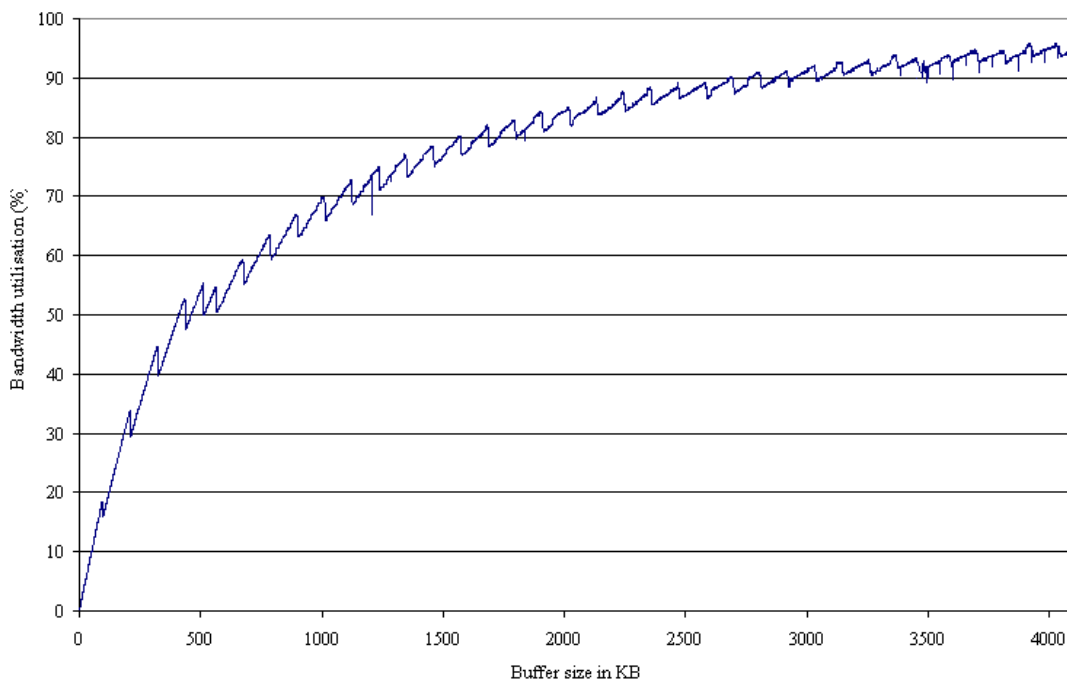


Figure 10: SRB bandwidth utilisation against read buffer sizes

The results in Figure 10 are obtained by downloading a 50MB file by continuous reading it into a fixed sized buffer until the complete file has been retrieved. The bandwidth utilisation is therefore the average obtained over the whole file as opposed to instantaneous burst levels. As this figure illustrates, bandwidth utilisation is best achieved with fewer read calls to the SRB system and larger buffer sizes.

Which such a large buffer size required to achieve a high level of network bandwidth utilisation, the developer is required to balance this with available system resources and end-user responsiveness; larger buffers will take longer to fill than smaller ones regardless of the overall bandwidth utilisation. This is especially true if the data that is being retrieved can be streamed and manipulated while more data is still being downloaded.

⁴ This assumption works well for our dataset but would be unsuitable for data files that can change in size as they are being read. In these cases you would need to call the `available()` method each time and add back on the `filePointer` variable.

5 Access Permissions

Setting and resetting file permissions within the SRB system is the final type of file operation considered in this paper. File permissions can only be set at the individual user level as there is no concept of setting file permissions for a group of users. Furthermore there is no concept of inherited file permissions and thus when new files are created only the user creating the file has permission to do anything with it. The exception to this rule is when replacing an existing file where any permissions already set are preserved.

The consequence of the way SRB deals with file permissions is that for each file you wish to change there needs to be a permission change call for each user. For example, to add 3 new users to 4 files, 12 permission change calls to the underlying API are needed. This multiplicity is further compounded when considering the performance of a single permission change call, which can take between 400 and 500ms.

6 Conclusions

While the SRB system provides a very workable front-end to distributed file systems, developers wishing to interface with it need to consider its artefacts to get the most benefit from it; this is significantly more true than accessing data files locally, or via HTTP and/or FTP for example.

From the empirical data collected in this paper, it is apparent that SRB file systems do not handle small files efficiently. This is primarily due to the relatively high overhead incurred when opening a SRB file for reading or writing. This overhead can be counterbalanced while uploading data using the bulk upload mechanism which reduces the number of SRB file open and close operations. Performance plots and mathematical formulations have indicated that file sizes up to approximately 30MB are best handled with the bulk upload operation as opposed to uploading the files individually when dealing with multiple files.

Unfortunately there is no similar bulk download mechanism within SRB to alleviate the overhead incurred while opening smaller files to retrieve data from. The solution we have presented in this paper introduces an intermediate step that concatenates many small files into one larger file, similar to a ZIP file. This uncovered a coding error in the current version of the Jargon API within the skip method; the skip method allowing the smaller files to be indexed within the larger one and retrieved in a more efficient manner. Section 4.1 outlined a possible fix to this code.

In terms of retrieving data from the SRB system, practical use revealed that large client-side buffers are required to optimise the available network bandwidth, typically greater than 1MB for 70% network utilisation on a 100Mbps LAN. This however impacts the user-apparent responsiveness when downloading data over a slower bandwidth connection, such as broadband. Therefore when determining a suitable read buffer size, the developer must balance available system resources with responsiveness and optimal network utilisation.

To determine who can actually access files, SRB provides a user-based file access system. However, this system is somewhat rudimentary and not really suitable for handling a large number of files in a timely manner. Group permissions also appear to be absent.

Although smaller files were handled with less efficiency than the authors would have liked, the results have shown that larger files are comparatively much better dealt with. Furthermore, throughout the collection of the empirical data, the SRB system remained stable. However, it was noted that long-term subjection to large data manipulations had an increasingly detrimental effect on responsiveness of the system. During our tests, we were able to eliminate this degradation in performance by emptying the trash for all users, vacuuming the database and re-indexing it; these operations are recommended for SRB systems under heavy load.

SRB does therefore provide a convenient distributed file system mechanism but care needs to be taken when considering how best to interface with it to maximise its use and avoid potential pitfalls.

Appendix A: Single file upload transfer times (in milliseconds)

File size	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB
Min Value	31	0	0	0	0	0	0	0	15	46	93	187	406	781	1547	3094	6156
Lower Quartile	47	0	0	0	0	0	0	15	16	47	94	203	422	797	1578	3234	6672
Mean Value	47.291	2.364	2.731	2.489	4.352	5.355	7.288	13.668	25.841	50.185	100.084	201.852	428.151	801.187	1580.636	3311.992	6758.27
Upper Quartile	47	0	0	0	0	15	16	16	31	47	109	203	437	812	1579	3391	6860
Max Value	63	562	515	438	594	516	16	32	47	94	125	235	438	813	1656	3579	7172

Appendix B.1: Bulk file upload transfer times (in milliseconds)

File size	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB
Min Value	9984	265	312	375	516	828	1422	2609	4953	9734	19484	38937	78312	156703	316485	634578
Lower Quartile	10000	281	312	390	531	828	1437	2625	5000	9796	19500	38957.5	78406	156863	316980.5	635051.3
Mean Value	9999.96	279.38	314.36	387.22	534.5	838.18	1442.44	2633.64	5001.26	9795.64	19520.3	39001.86	78442.56	156945.1	318931.9	637653.2
Upper Quartile	10000	281	313	391	532	844	1453	2641	5015.75	9797	19531	39015	78465.25	156984	321496	636180.5
Max Value	10016	297	328	391	547	844	1454	2641	5016	9922	19656	39234	78750	157375	323188	668782

Appendix B.2: Bulk file upload register times (in milliseconds)

File size	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB
Min Value	11469	11641	12125	11500	12812	11907	14265	13390	16188	21110	24812	30391	39797	104063	175328	303047
Lower Quartile	15062.25	14472.25	13472.75	14316.75	15121.5	13562.75	15621.5	15715.25	18050.5	23742.5	27375	32371.25	42699.25	119890.3	190988	321617
Mean Value	18628.04	19034.96	17925.96	17717.2	17312.9	17368.18	18492.16	19209.66	21589.64	27705.04	32311.5	38235.34	51158.96	130570	220207.1	331564.1
Upper Quartile	18453.25	20292.5	19988	19234	17515.5	18734.5	18699.5	19968.75	21913.75	27629.25	33484.25	43364	60289	144281	250437.8	341207
Max Value	38375	44844	38375	40375	34984	34203	34969	35266	45843	48828	54843	56829	66141	155719	284375	357062