Characterizing Storage Resources Performance in Accessing the SDSS Dataset

Ioan Raicu

Date: 8-17-05

Table of Contents

Table of Contents 1
Table of Figures 1
1 Overview 4
2 Experiment Description 4
2.1 Data 4
2.2 Access Methods 5
2.3 File Systems 5
2.4 Client Code 5
2.4.1 JAVA Implementation 5
2.4.2 C Implementation 6
2.5 Experiment Execution 6
2.5.1 Single Client Access 6
2.5.2 Multiple Concurrent Client Access 6
2.6 Testbed 6
3 Empirical Results 7
3.1 Single Client Access 7
3.2 Multiple Concurrent Client Access 15
3.3 Experimental Result's Conclusions 26
4 Proposed Architecture 28
4.1 Design based on TG GPFS (WAN) 28
4.2 Design based on GPFS (LAN) 29
4.3 Design based on Local FS 30
5 Bibliography 32

Table of Figures

Figure 1: Median JAVA client performance overview spanning 30 different configurations; the error bars denote the standard deviation of the data (comprising of 50~200 data points) 8

Figure 2: Median C client performance overview spanning 10 different configurations 8

Figure 3: Client speed-up: C vs. JAVA; any number above 1.0 means that the C client was faster than the JAVA client, while any number bellow 1.0 means that the JAVA client outperformed the C client 9

Figure 4: JAVA client speed-up for 1-crop vs. 10 crops 10

Figure 5: C client speed-up for 1-crop vs. 10 crops....10

Figure 6: JAVA client performance distribution via a 5-number summary box-plot for the 1 crop experiments 11

Figure 7: JAVA client performance distribution via a 5-number summary box-plot for the 10 crop experiments 11

Figure 8: C client performance distribution via a 5-number summary box-plot for the 1 crop experiments....12

Figure 9: C client performance distribution via a 5-number summary box-plot for the 10 crop experiments....12

Figure 10: JAVA client work distribution 13

Figure 11: C client work distribution....14

Figure 12: Time to complete O(100) crops from 1 client (both JAVA and C); log scale in seconds 14

Figure 13: Summary of Concurrent FIT Client (both JAVA and C) Performance (Response Time); log scale in milliseconds 16

Figure 14: Summary of Concurrent FIT Client (both JAVA and C) Performance (Throughput, queries per second); log scale 16

Figure 15: Time to complete O(100K) crops using 100 clients (both JAVA and C); log scale in seconds 17

Figure 16: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 18

Figure 17: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed....18

Figure 18: FIT C Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 19

Figure 19: FIT C Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed 20

Figure 20: FIT JAVA Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 20

Figure 21: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed....21

Figure 22: FIT C Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 21

Figure 23: FIT C Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed....22

Figure 24: FIT JAVA Client Performance over the PVFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed 23

Figure 25: FIT JAVA Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 24

Figure 26: FIT JAVA Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed....24

Figure 27: FIT C Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed 25

Figure 28: FIT C Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed 26

Figure 29: Proposed system architecture based on TG GPFS (WAN) 28

Figure 30: Proposed system architecture based on GPFS (LAN)....29

Figure 31: Proposed system architecture based on the Local FS 31

1 Overview

In this report, I will present my findings regarding my first goal:

> *"Determine what's involved in getting Szalay's code running efficiently on TeraGrid. We need to implement the basic code, get it running, and measure its performance in a few different settings in terms of data and compute location: per-node local disk, NFS-attached local disk, GPFS remote disk."*

I implemented a client in both C and JAVA that can read FIT images, crop a section of the image, and write it back to disk in FIT format. I also took about 400 compressed images (2~3MB each) summing to about 1GB of images, and made several copies on various file systems. These file systems include the local file system, NFS, PVFS, the local GPFS, and the remote GPFS. The data is stored in a GZ compressed format; in order to cover the widest range of performance tests, I created two data sets: the original compressed data (~1GB), and the decompressed data (~3GB). To quantify the effects of caching of various experiments, I decided to run two different experiments, one in which I only perform 1 crop per image, and one in which I perform 10 crops per image (in some cases this offered significant speedup since the image was already in memory). As a summary, the total number of experiments I had were 30 (5x3x2) in JAVA and 10 (5x2) in C, for a total of 40 different experiments. I have reduced all the results from these 40 experiments from the single client to 14 graphs in section 3.1.

Section 3.2 covers a similar set of experiment but for concurrent client access. For these experiments, we worked only with the uncompressed data; the file systems tested were NFS, PVFS, the local GPFS, and the remote GPFS; the local file system results were extrapolated from the results of the single client performance from section 3.1. As a summary, the total number of experiments I had were 8 (4x2) in JAVA and 8 (4x2) in C, for a total of 16 different experiments. I have reduced all the results from these 16 experiments from the concurrent client access to 19 graphs in section 3.2.

Overall, the performance of both the JAVA client and the C client running as one client at a time seemed good. The performance degradation when accessing the data in the GZ format is very large, and I believe it should be avoided at all costs. The compression ratio is only 3:1, so keeping the entire archive of 2TB in an uncompressed format would only increase the storage requirement to 6TB, but yielding magnitudes order better performance.

On the other hand, the performance difference among the different FS seems to have grown significantly as we run 100 clients in parallel. Except in just a few cases, the performance difference between the JAVA and C client seems to be almost negligible.

Based on the results in Section 3, the PVFS and NFS should be avoided if possible. Furthermore, there are significant performance gains by keeping the data available in an uncompressed format. The TG GPFS should also be avoided, unless the scalability of the system (concurrent clients accessing the data) will be larger than what can be supported by the ANL GPFS. Therefore, if performance is important, then the potential FS are the local FS and the ANL GPFS; in both of these cases, there were improvements in performance for the C client in comparison to the JAVA client, so the C client would be preferable.

One of the driving motivations for using the TeraGrid for this application was to potentially get very quick turn-around times on large number of operations that could potentially touch a large portion of the data set. Through the experimental results we performed, we conclude that 100K operations (in our case crops) could be performed on the order of 10s of seconds up to 1000s of seconds using 100 clients depending on the data access method.

2 Experiment Description

This section covers the data description, the access methods (i.e. compression, raw), file systems tested, and the client code details.

2.1 Data

The entire data set is on the order of TB, which means that there are probably on the order of 100K to 1M images in the entire database. Just to do some preliminary testing, I took about 400 compressed images (2~3MB each) summing to about 1GB of images, and made several copies on various file systems. The data is originally stored in a GZ compressed format; in order to cover the widest range of performance tests, I created two data sets: the original compressed data (~1GB), and the decompressed data (~3GB).

2.2 Access Methods

I had three different ways to access the data:

- GZ – The client read the data in GZ compressed format directly and worked on the compressed images
- GUNZIP - I first decompressed the image, and then every subsequent operation on that image took place on the uncompressed image
- FIT – The client read the data in the raw and decompressed FIT format directly

Note that only the JAVA client supports all 3 access methods, while the C client only supports the FIT access method.

2.3 File Systems

These file systems include:

- LOCAL
    - the local file system
    - $TG_NODE_SCRATCH=/scratch/local
- NFS
    - the network file system (NFS)
    - serviced by 1 server on a LAN
    - $HOME=/home
- PVFS
    - the parallel virtual file system (PVFS)
    - serviced by ? server on a ?AN
    - $TG_CLUSTER_PVFS=/scratch/pvfs
- ANL GPFS
    - the local general parallel file system (GPFS)
    - serviced by 8 stripped servers on a LAN
    - $TG_CLUSTER_GPFS=/disks/scratchgpfs1
- TG GPFS
    - the remote general parallel file system (GPFS)
    - serviced by 60+ servers on a WAN
    - $TG_GPFS=/gpfs-wan

2.4 Client Code

I implemented a client in both C and JAVA that can read FIT images, crop a section of the image, and write it back to disk in FIT format. Each client has about 1000 lines of code in each respective language and has ms accurate timing mechanisms built in to the client code for accurate measurement of performance of individual operations. The clients have also been prepared for use with DiPerF for future testing of the client performance while concurrently accessing the data along with other clients.

In the next sub-sections, I will briefly describe the implementation in each language.

2.4.1 JAVA Implementation

The JAVA implementation relies on a JAVA FITS library (http://heasarc.gsfc.nasa.gov/docs/heasarc/fits/java/v0.9/). [1] The full source code can be downloaded from http://heasarc.gsfc.nasa.gov/docs/heasarc/fits/java/v0.9/fits.jar. The Java FITS library has been developed which provides efficient I/O for FITS images and binary tables. The Java libraries support all basic FITS formats and gzip compressed files.

Currently the client is geared towards ease of performance testing. There are several arguments that allow the user to control the experiment; among these arguments are:

- Input file list (the entire data set with relative paths)
- Input path
- Output path
- Number of pictures to process (randomly chosen from the list)

- Number of crops per image (done in a sequential order, and performed at random [x,y] coordinates in each respective image)
- Size of the crop [width, height]

All the performance metrics from each task are stored in memory in order to easily compute entire run statistics. The various performance metrics are:

- copy() – the time to read the GZ compressed image into memory, decompress it, and write it back to disk in a raw decompressed FIT format
- read() – the time to read the entire image data into memory
- crop() – the time to crop out a rectangular sub-image from the original larger image that is in memory
- write() – the time to write the sub-image to disk
- TOTAL – the total time from the copy() stage to the write() stage

At the end of an experiment, the statistics that are computed for each metric are:

- Min – minimum value
- Q1 – 1st quartile value
- Aver – average value
- Med – median value
- Q3 – 3rd quartile value
- Max – maximum value
- StdDev – standard deviation for entire experiment

2.4.2 C Implementation

The C implementation relies on WCSTools libraries (http://tdc-www.cfa.harvard.edu/software/wcstools/). [2] The full source can be downloaded from http://tdc-www.harvard.edu/software/wcstools/wcstools-3.6.1.tar.gz. The WCS FITS library provides support for reading and writing primary images and reading extension data. It is quite compact, requiring only four files in the distribution. The package is particularly complete in reading header information, and of course in dealing with FITS World Coordinate System (WCS) information.

The C client has almost an identical overview, arguments, performance metrics, and statistics collected. The only exception is that although the copy() metric exists in the C client, since C does not allow the easy reading of compressed files into memory (such as was the case with the JAVA client), this feature was not implemented, and hence all values for the copy() metric always report 0.

2.5 Experiment Execution

2.5.1 Single Client Access

Outside of the fact that we had many different experiments to run, 40 in total, I used simple scripts to run the experiments in sequence. At the end of every experiment, the client saved the overall performance statistics in a file for later analysis.

2.5.2 Multiple Concurrent Client Access

To run the experiment with multiple concurrent clients accessing the data, I used the existing DiPerF framework to drive the experiments, collect the performance data, and plot the graphs. I used the TeraGrid to gain access to multiple physical machines via the command “qsub” which allowed me to make reservations for predefined number of machines and a predefined time period. During the period of the reservations, I had sole access to these machines.

2.6 Testbed

I used the TeraGrid (TG) to run the experiments. The TG has 96 Visualization Intel IA-32/Xeon nodes and 62 Compute Intel IA-64/Madison nodes. The IA-32 nodes are dual 2.4GHz Intel Xeon processor machines, with 3.5GB RAM and 1Gb/s network connectivity. Each node on the TG had access to all the different file systems through the normal Unix semantics.

3 Empirical Results

This section covers the empirical results obtained from running a single client (JAVA and C) accessing the data set over various different file systems and access methods. Most of the graphs are depicted in log scale in order to better capture the entire range of values which varies significantly depending on the testing configuration.

The following conventions will be used throughout the following graphs:

- File Systems:
  - LOCAL: local FS
  - NFS: network FS
  - PVFS: parallel virtual FS
  - ANL GPFS: local general parallel FS
  - TG GPFS: TeraGrid GPFS
- Access Methods:
  - GZ: working on GZ compressed images
  - GUNZIP: decompressing the GZ images first, and then working on uncompressed FIT images
  - FIT: working on uncompressed FIT images
- Implementation:
  - JAVA: JAVA client
  - C: C client
- Caching:
  - 50x1: test involved 50 random images with 1 crop from each image
  - 20x10: test involved 20 random images with 10 successive crops from each image
  - Other: for some of the tests, namely for the tests on the PVFS in the JAVA client, the performance was very poor and to cut down the length of time of the experiment, the experiment size was reduced to either 3x1 or 3x2

3.1 Single Client Access

Figure 1 gives the overview of the median JAVA client performance spanning the 30 different configurations on a log scale. Some of the obvious observations about the performance of the JAVA client are:

- In the GZ format, performing multiple crops per image yields very little benefit
- The PVFS file system seems to have very poor performance (100+ sec instead of 100+ ms) under the GZ and GUNZIP format
- The GUNZIP.20x10 set of results seem to have the best performance, however the results a bit deceiving since this graph represents the medians; for example, the averages between GUNZIP.20x10 and FIT.20x10 are almost identical, which implies that the techniques are actually very comparable; the better performance here is attributed to the fact that to decompress the image data, it must all be read in memory, and hence any subsequent access to any portion of the data can be served directly from memory
- The LOCAL FS offers the best performance in any access method, with the NFS and the ANL GPFS trailing closely behind; at least for the FIT case, both PVFS and TG GPFS perform a few orders of magnitude slower than the LOCAL FS; this might very well turn out to be the opposite when we run multiple clients to access the data concurrently
- For the FIT format, a single crop can be done in <40 ms locally, and about 100ms over NFS or GPFS; when multiple crops are invoked sequentially on the same image, times can be reduced to about 10ms locally, <20ms for NFS, and <50ms for GPFS.

Figure 2 gives the overview of the median C client performance spanning the 10 different configuration on a log scale. Some of the obvious observations about the performance of the C client are:

- The relative performance difference that we observed in the JAVA client hold true here as well
- It was odd that the PVFS results were significantly worse for the C client than that of the JAVA client, 2000+ ms vs. <300ms
- For the FIT format, a single crop can be done in <25 ms locally, and about 100ms over NFS or GPFS; when multiple crops are invoked sequentially on the same image, times can be reduced to about 9ms locally, <20ms for NFS, and <50ms for GPFS

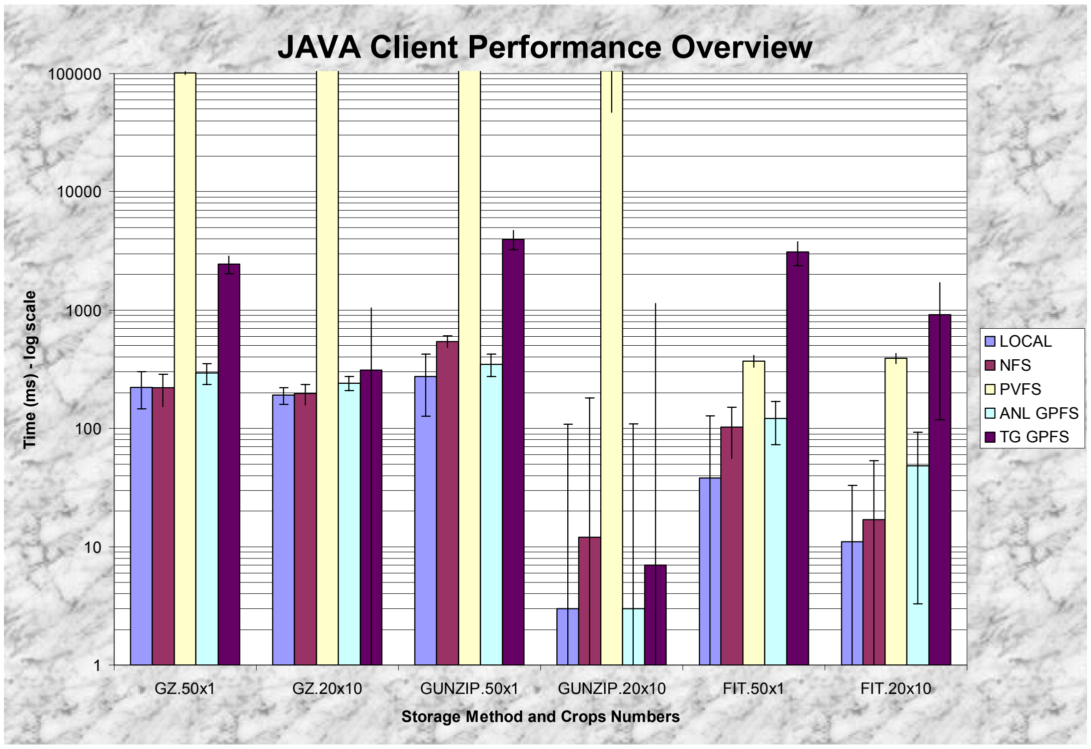


Figure 1: Median JAVA client performance overview spanning 30 different configurations; the error bars denote the standard deviation of the data (comprising of 50~200 data points)

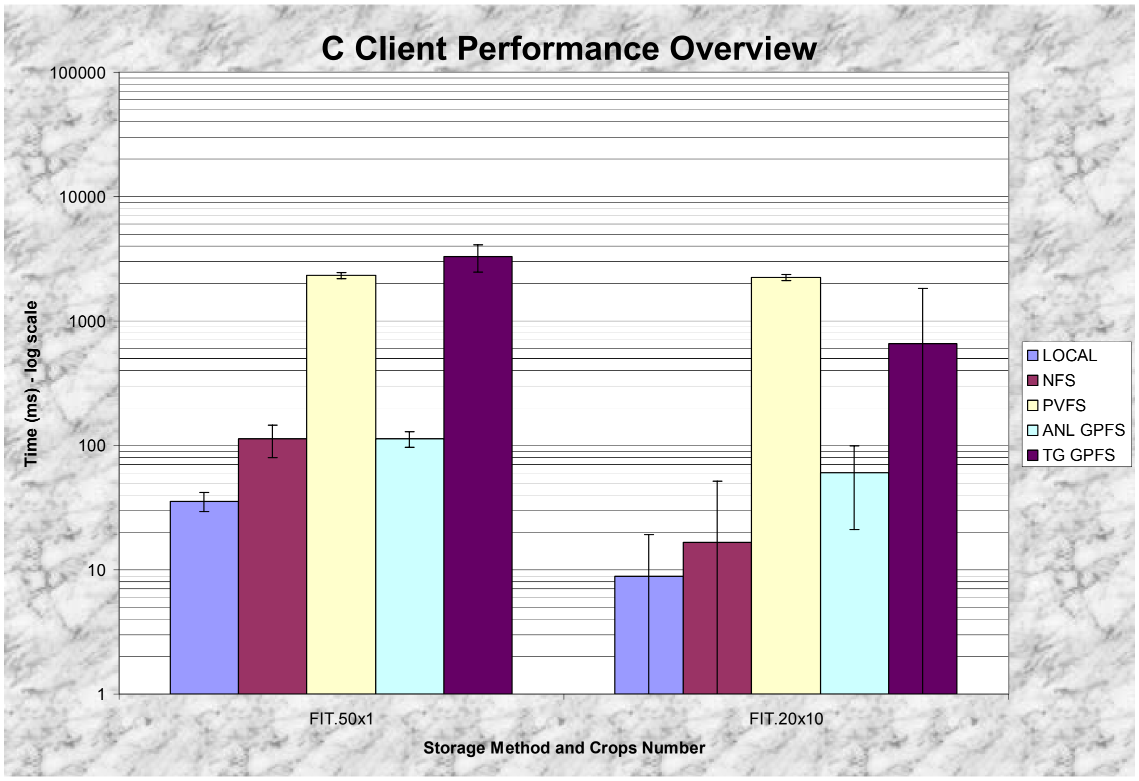


Figure 2: Median C client performance overview spanning 10 different configurations

Figure 3 shows the speed-up between the C client and the JAVA client. It was surprising that the speed-up was not more consistent across the range of experiments. With the exception of PVFS, most of the other FS performed either about the same or better in C than it did in JAVA. The greatest improvement in performance was on the LOCAL file system and the TG GPFS when performing multiple crops per image.

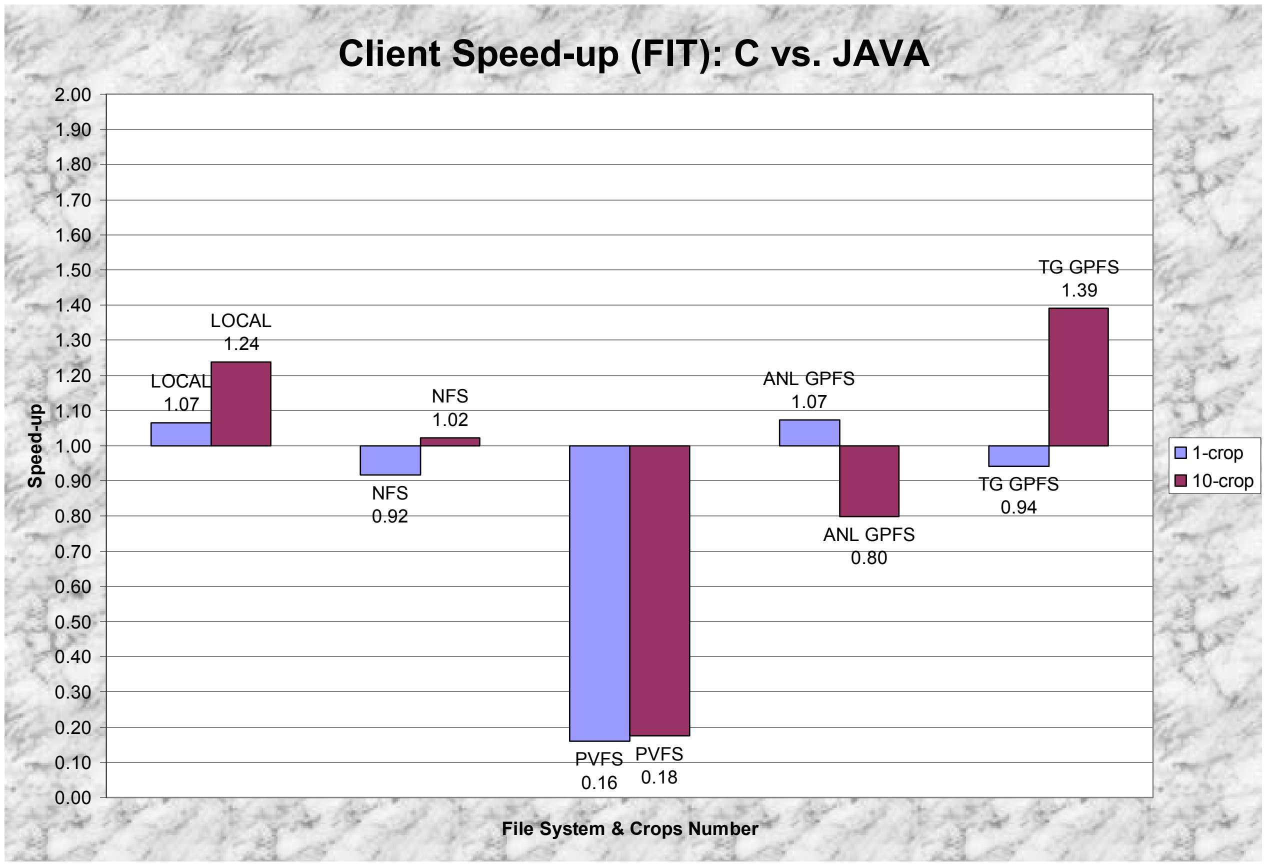


Figure 3: Client speed-up: C vs. JAVA; any number above 1.0 means that the C client was faster than the JAVA client, while any number bellow 1.0 means that the JAVA client outperformed the C client

Figure 4 and Figure 5 shows the speed-up due to the caching effect when performing multiple crops on the same image. With the exception of some of the PVFS experiments, all the other experiments regardless of FS and access methods showed improvement in performance. Both the C client and the JAVA client experience similar speed-ups when working in the FIT access method.

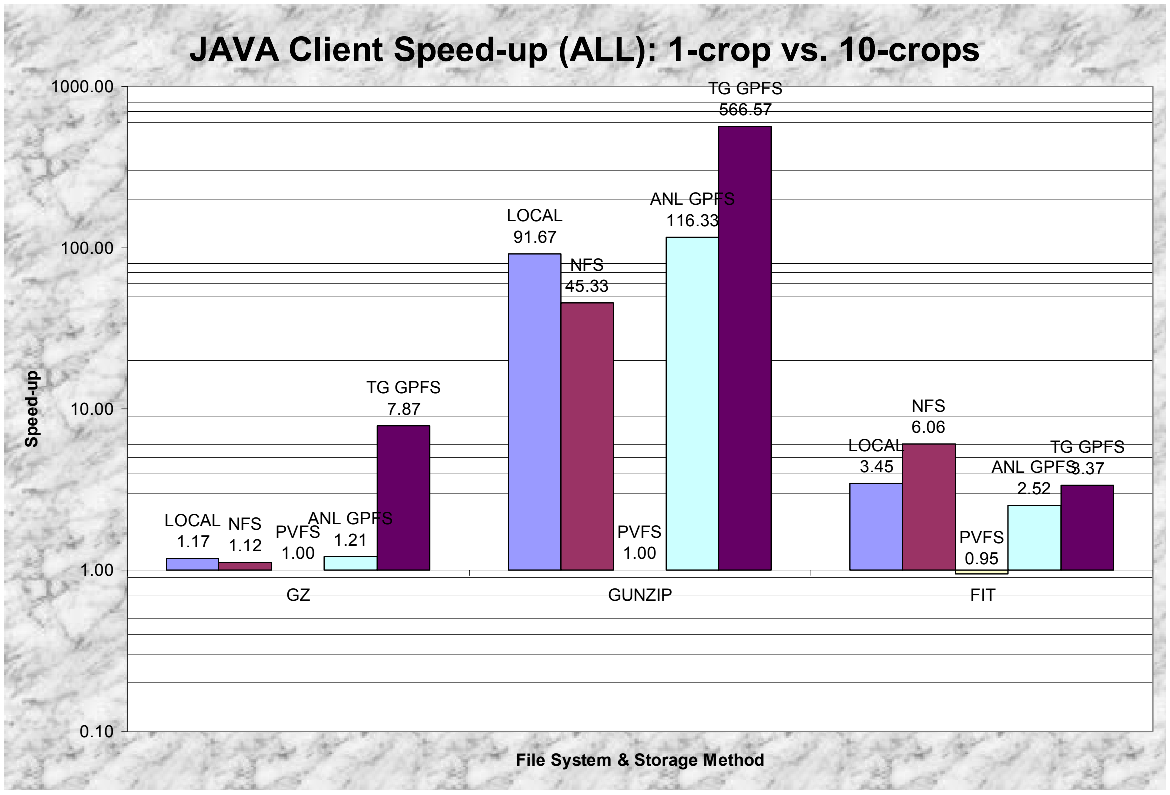


Figure 4: JAVA client speed-up for 1-crop vs. 10 crops

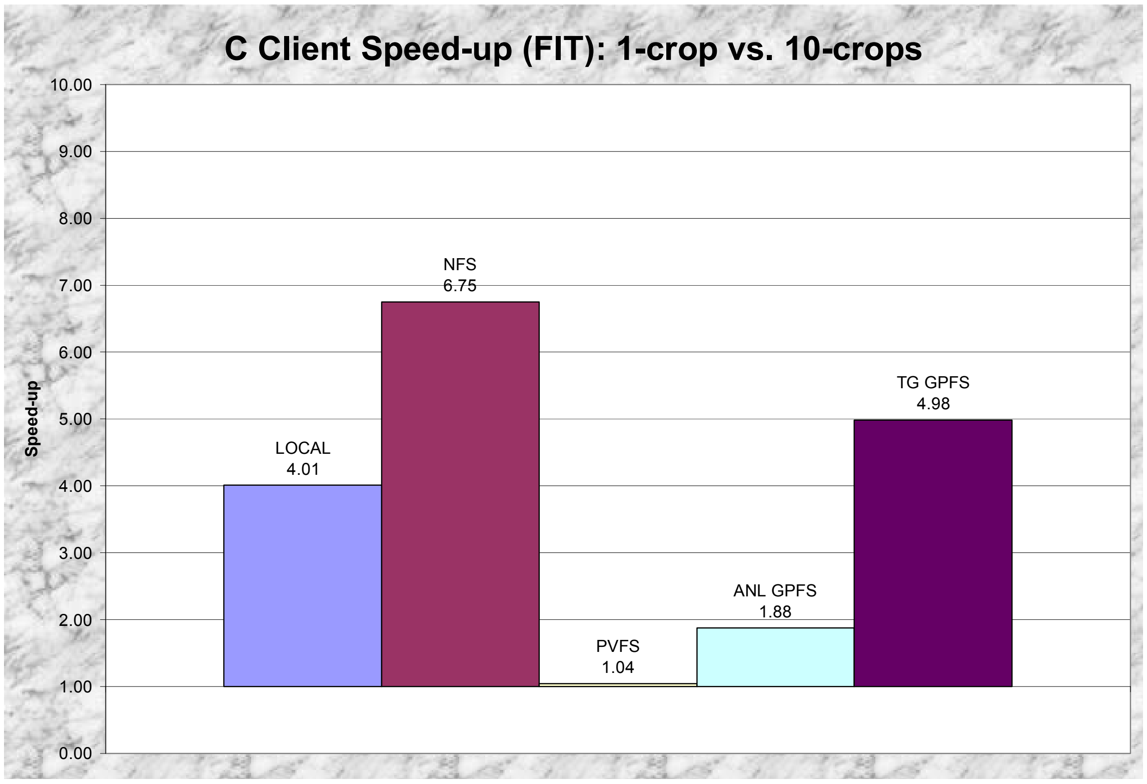


Figure 5: C client speed-up for 1-crop vs. 10 crops

Figure 6 and Figure 7 represent the 5-number summary depicted via box-plots for both the 1-crop per image and 10-crop per image experiments for the JAVA client. It is interesting to note that Q1 and Q3 are very close in value to the median in Figure 6 (which is depicted by the size of each corresponding box), which indicates that the majority of the performance metrics collected were close to the median values. On the other hand, in Figure 7 the boxes are significantly larger, and hence we see the much larger variance in performance due to caching and the randomness that was built in the client.

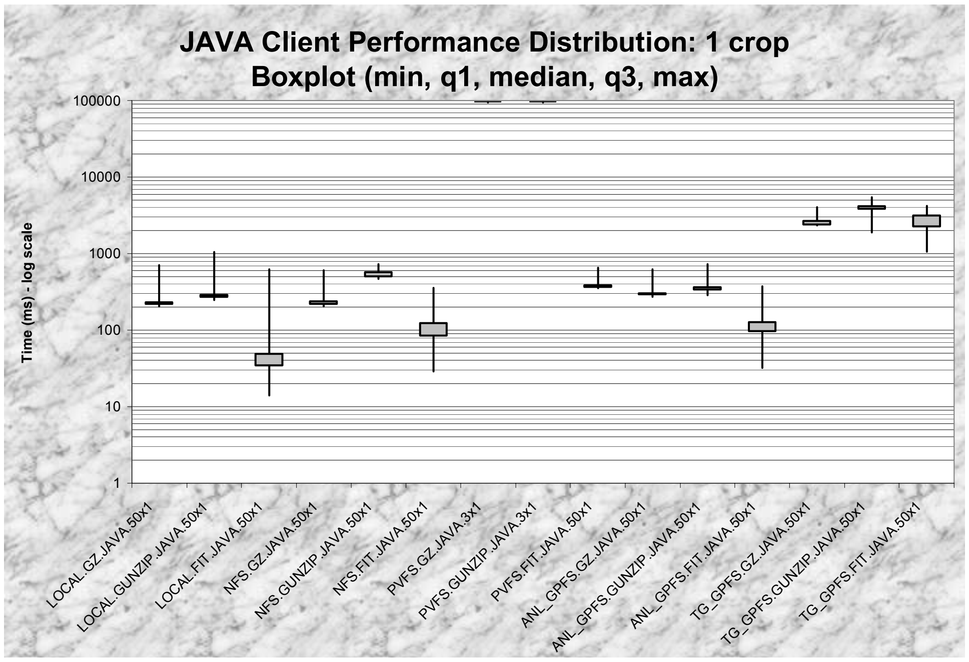


Figure 6: JAVA client performance distribution via a 5-number summary box-plot for the 1 crop experiments

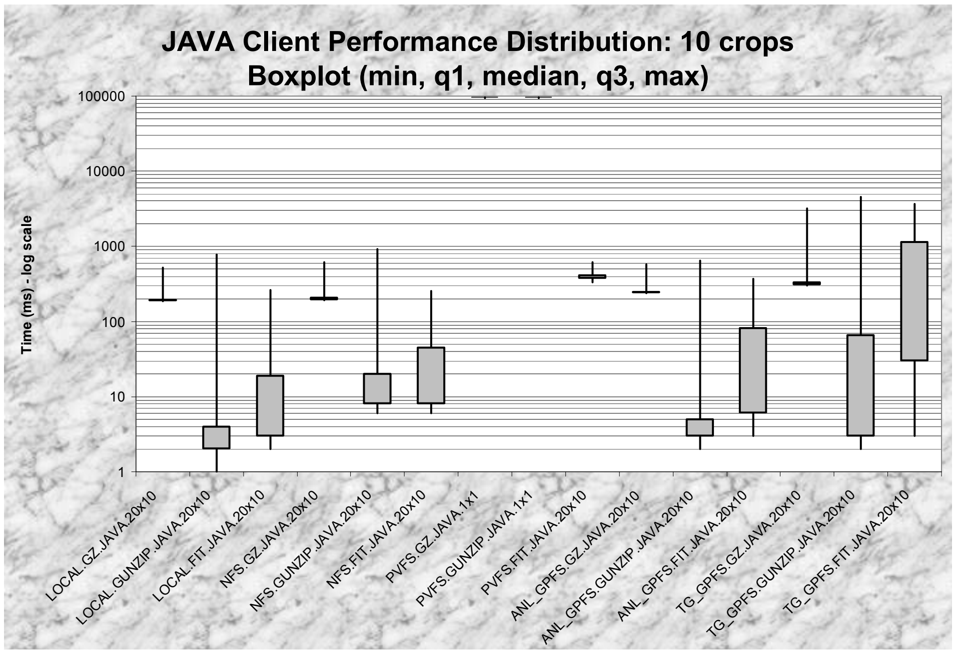


Figure 7: JAVA client performance distribution via a 5-number summary box-plot for the 10 crop experiments

Figure 8 and Figure 9 represent the 5-number summary depicted via box-plots for both the 1-crop per image and 10-crop per image experiments for the C client. Here we see very similar (as in the JAVA client) differences between the two figures. All these 4 graphs are excellent at seeing the performance distribution of a particular test case and the difference among the various configurations.

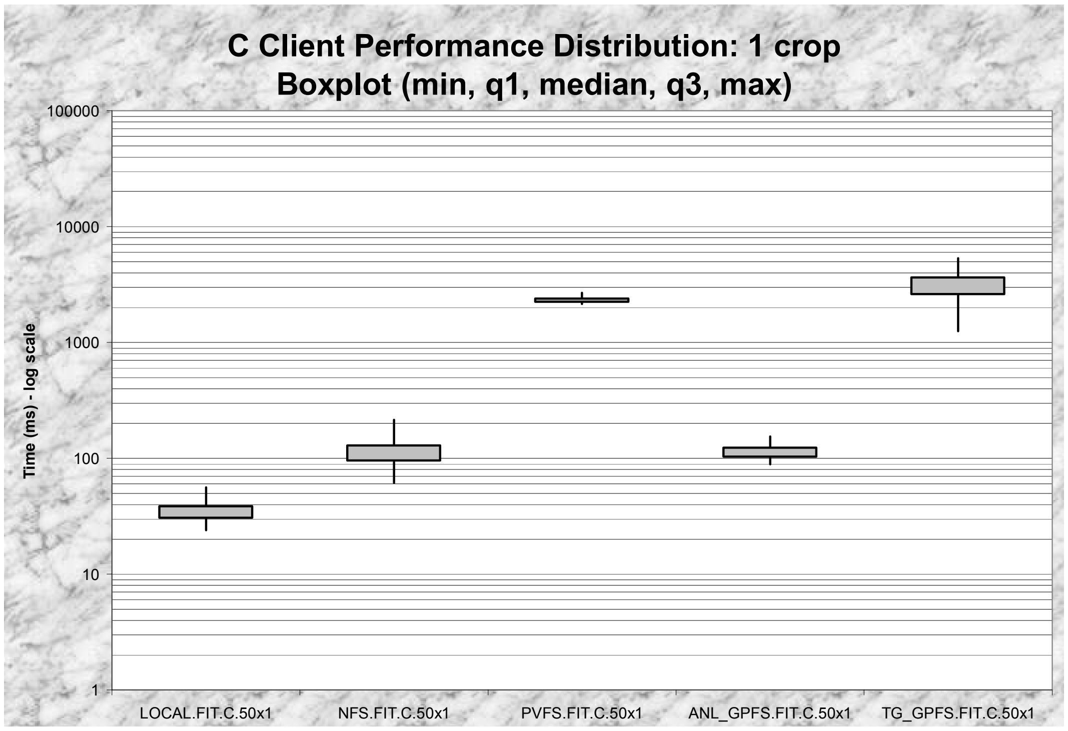


Figure 8: C client performance distribution via a 5-number summary box-plot for the 1 crop experiments

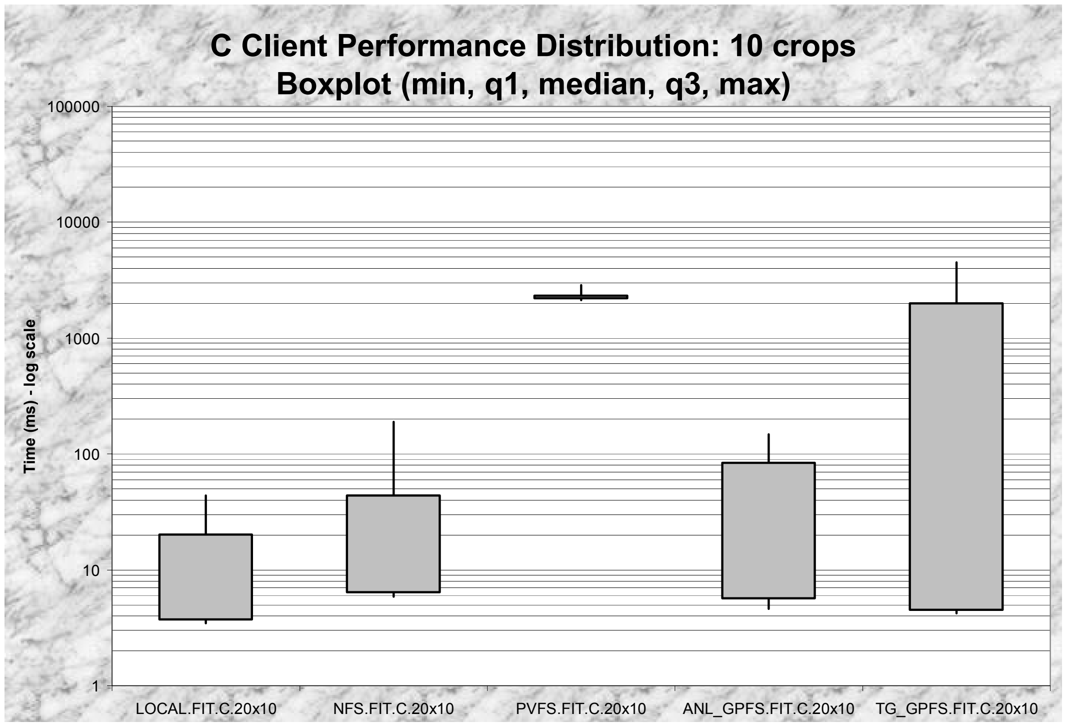


Figure 9: C client performance distribution via a 5-number summary box-plot for the 10 crop experiments

Both Figure 10 and Figure 11 show the distribution of the work for each respective client.

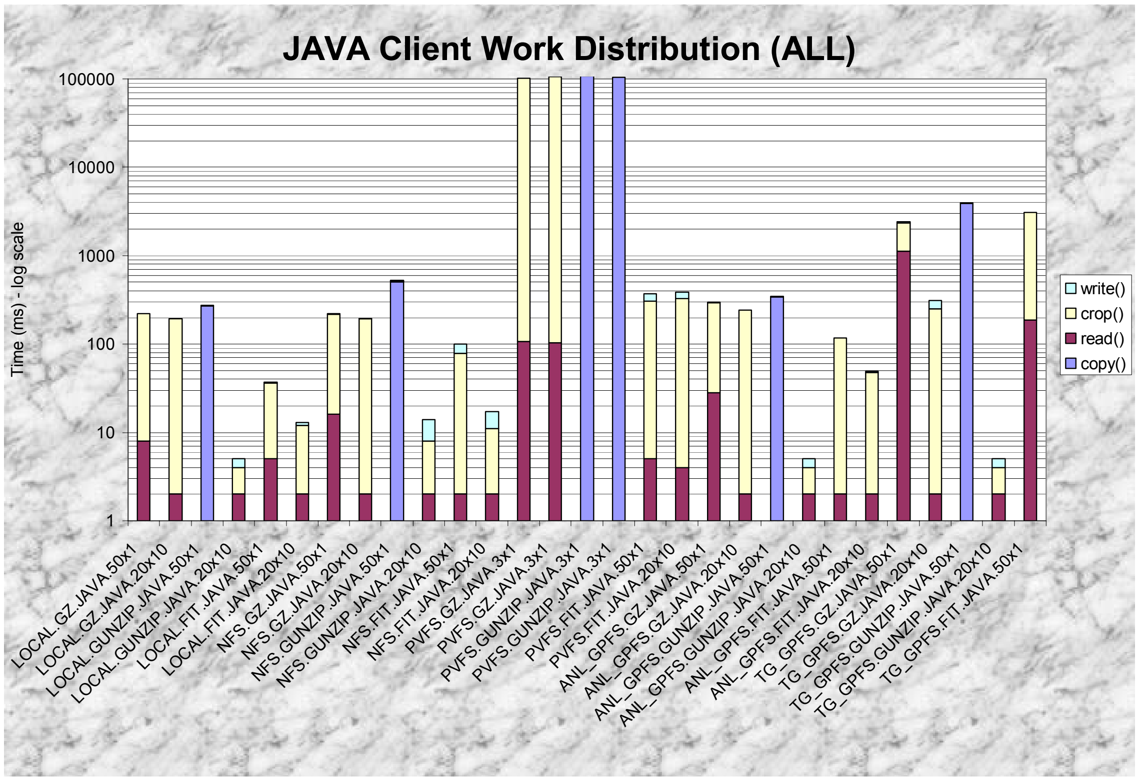


Figure 10: JAVA client work distribution

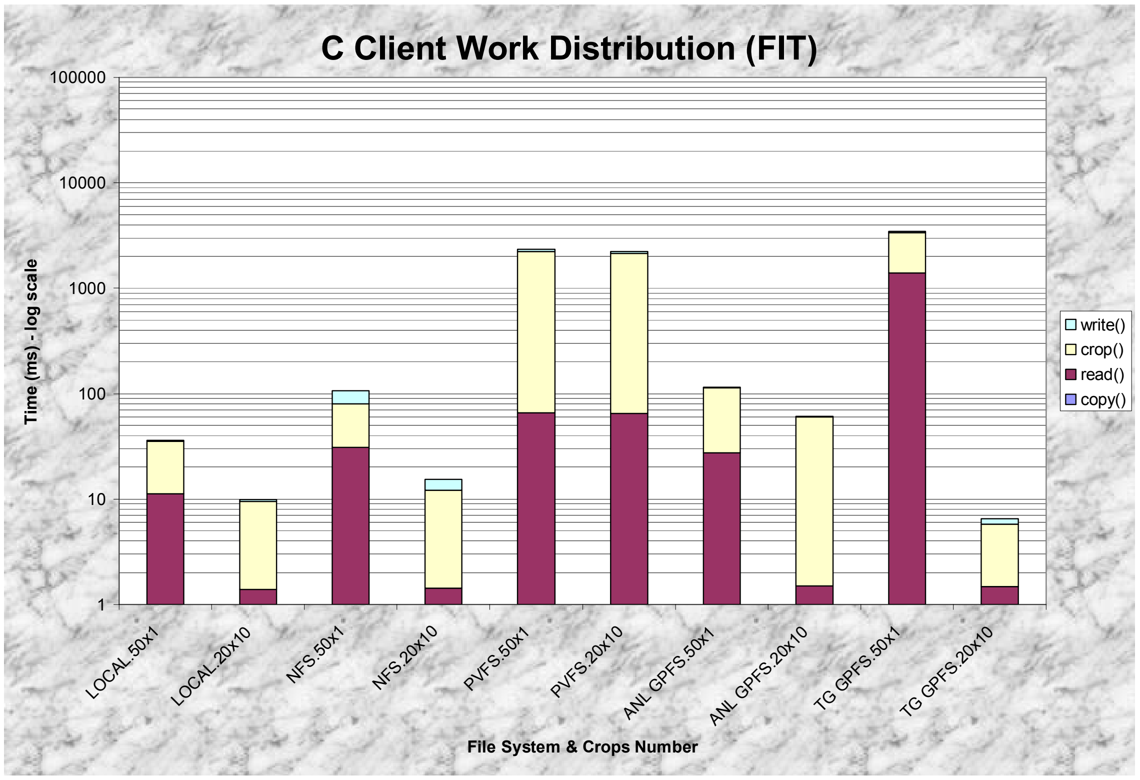


Figure 11: C client work distribution

The client normally has 3 stages, with a possible 4th in some experiments:

- copy() – read the GZ compressed image into memory, decompress it, and write it back to disk in a raw decompressed FIT format; this is an optional stage, depending on the particular experiment
- read() – read the image data into memory
- crop() – crop out a rectangular sub-image from the original larger image
- write() – write the sub-image to disk

The most interesting result of Figure 10 and Figure 11 is that the client spends the majority of its time in either the copy() stage (if it has one), or in the crop() stage.

Finally, Figure 12 shows an estimate of the time to complete about 100 crops on a single client for both JAVA and C. This graph was motivated by an earlier email discussion:

> *"He wants to run a service that supports requests of the form "extract and return a small piece out of a specified set of O(a few hundred at most) files from a large collection of O(10 TB)." The actual computation is probably 10 secs if there are a lot of processors, so the key issue is being able to have the data accessible and schedule the computation quickly on enough processors."*

It is very interesting to see that there are several configurations that can already satisfy the requirements from above on a single client. In general, the best performance was when the data was accessed in its raw FIT format; LOCAL, NFS, and ANL GPFS all are able to complete 100 crops in about 10 seconds or less, even without performing multiple crops on images. It will be interesting to see how this graph changes as we start performing crops in parallel, and how the time to complete 100 crops drops as we range the number of concurrent clients from 1 to 100 clients.

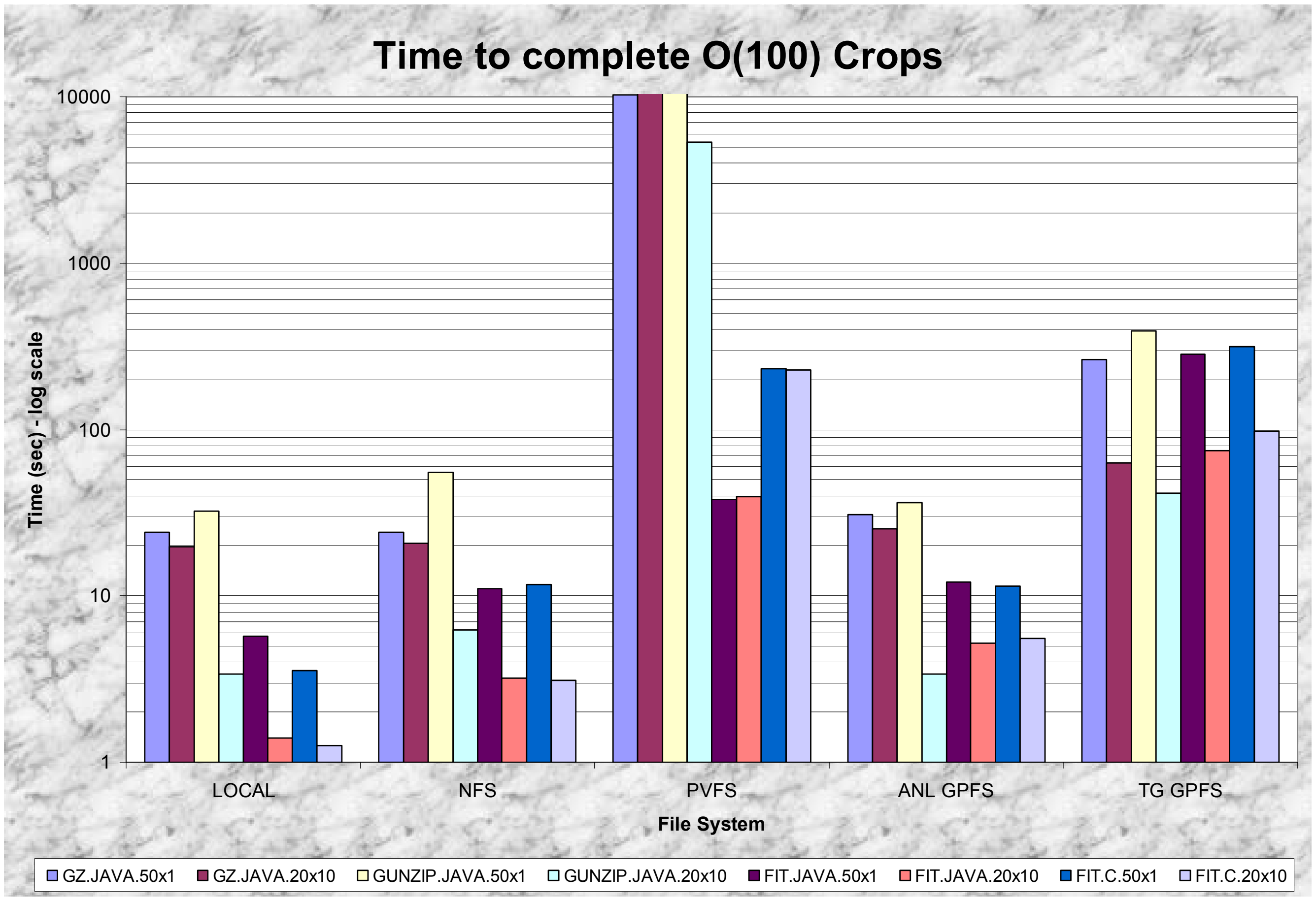


Figure 12: Time to complete O(100) crops from 1 client (both JAVA and C); log scale in seconds

3.2 Multiple Concurrent Client Access

To run the experiment with multiple concurrent clients accessing the data, I used the existing DiPerF framework to drive the experiments, collect the performance data, and plot the graphs. I used the TeraGrid to gain access to multiple physical machines via the command "qsub" which allowed me to make reservations for predefined number of machines and a predefined time period. During the period of the reservations, I had sole access to these machines.

Figure 16 through Figure 28 cover the individual performance runs for both the JAVA and C FIT client as the load varied from 1 to 100 clients running on 50 dual processor nodes; the performance runs varied in the file systems used (data access method), and the number of crops performed per image. The 1 crop per image represents the worst case scenario, while the 10 crops per image represents the improvement we might expect due to caching if multiple crops are performed on each image. The experiments on three file systems (ANL GPFS, TG GPFS, and NFS) all went smoothly, but PVFS had reliability issues and has been down most of the previous past few weeks, and hence I was only able to complete one of the four different performance runs on PVFS. Furthermore, the local file system performance numbers presented in this section have been extrapolated from section 3.1 results based on a single client running in series. A similar test as I performed with the other FS, but on the local FS would have been difficult due to the requirement to replicate the data set (3GB data) over 50 physical nodes. Although this would not have been impossible, but since in the actual experiment, each of the 50 nodes would have acted completely independent of the other nodes, it would have been expected that the performance from 1 to 50 nodes would have scaled linearly.

Figure 13 and Figure 14 attempts to summarize Figure 16 through Figure 28 by depicting the peak response time and throughput achieved for 100 clients running concurrently, along with the standard deviation to better visualize the spread of the performance results. Some interesting observations:

- Caching improves performance in the Local FS, TG GPFS, ANL GPFS, but in NFS it does not seem to make much difference, and in PVFS we did not have all the results to be able to clearly say
- The Local FS clearly has the best performance, but any implementation based on this would be inherently more complex and would most likely require more storage resources, but the results would be more predictable since the performance would only be dependent on the node itself, and not on a larger infrastructure that is affected in many ways
- The best next viable alternative which would make the implementation rather simple is the ANL GPFS; even with 100 clients running concurrently, each crop can be served in 60 ~ 200 ms depending on how many crops are performed per image; the achieved aggregate throughput is between 500 and 1500 crops per second
- TG GPFS is the worst performed, mostly due to its high latencies incurred since the data physically resides at another site, so the data is brought into ANL over a WAN link
- There is a very wide range of performance: response times range from 9 ms to over 3000 ms, and throughput ranges from 25 crops per second to over 10,000 crops per second
- There are some performance differences between JAVA and C, but at this level, it is not very evident; the biggest difference seems to be the standard deviation of the results

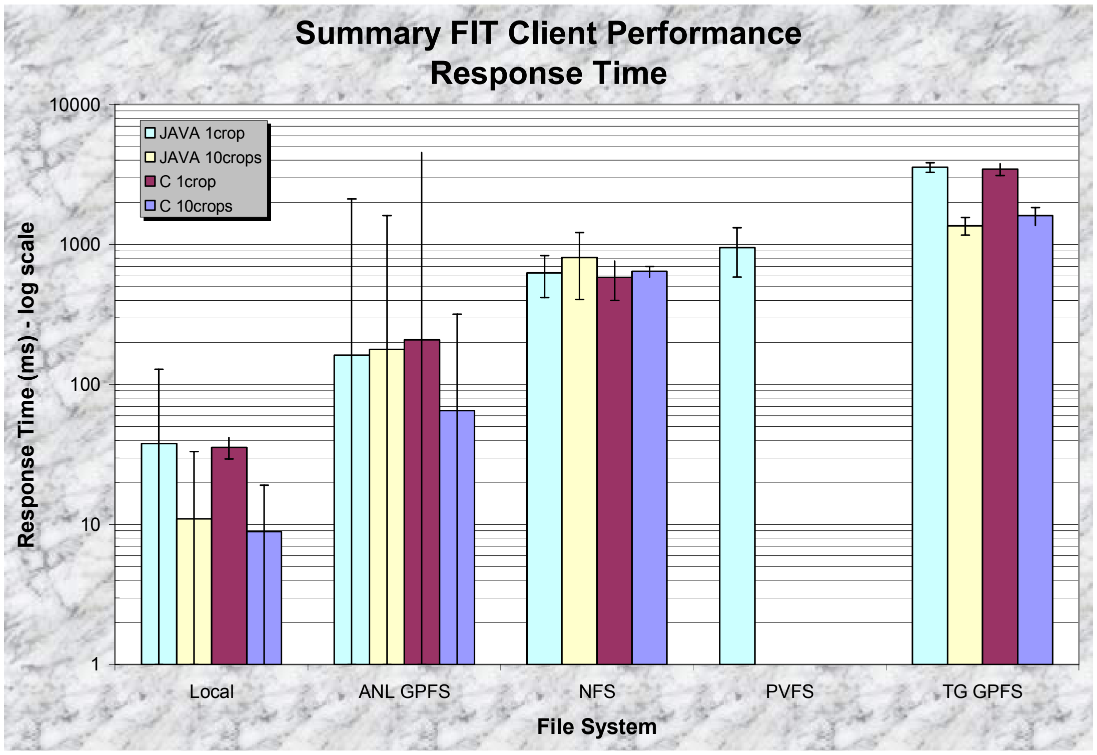


Figure 13: Summary of Concurrent FIT Client (both JAVA and C) Performance (Response Time); log scale in milliseconds

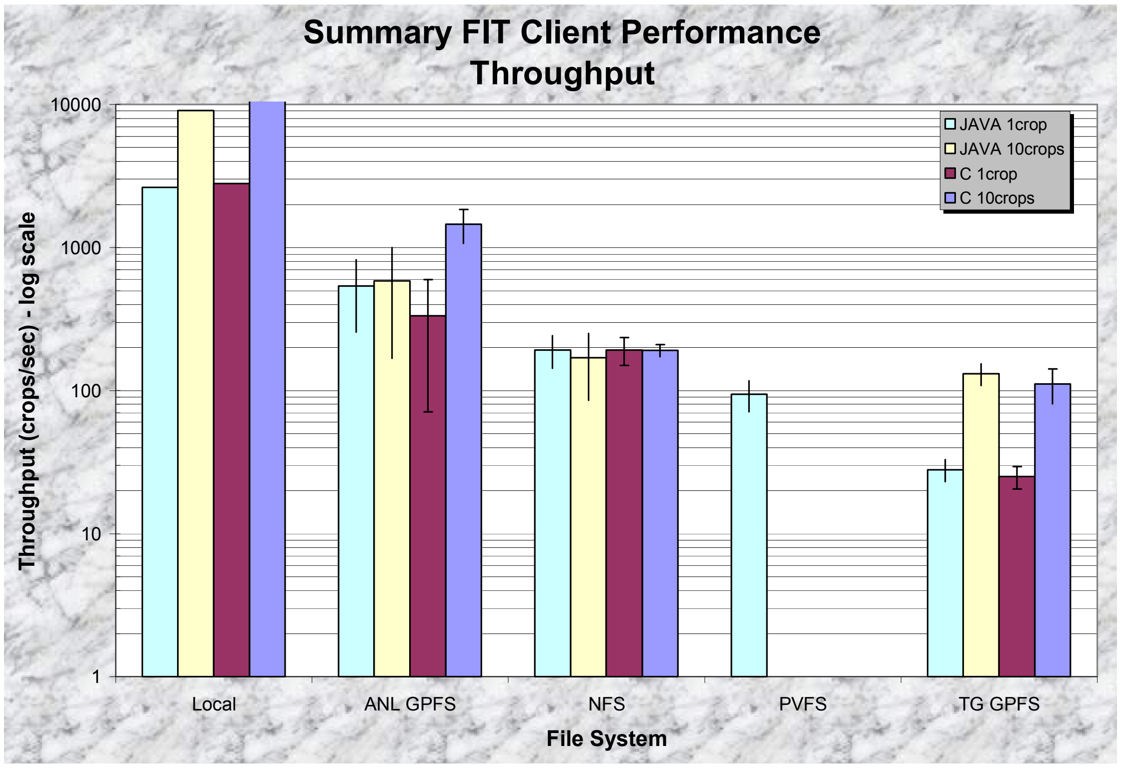


Figure 14: Summary of Concurrent FIT Client (both JAVA and C) Performance (Throughput, queries per second); log scale

Back in Figure 12, I showed the estimated time to complete 100 crops on 1 node based on the results from section 3.1 (the performance of a single client); the performance of the better FS ranged in the 1 to 10 seconds while more average results ranged in the 100 seconds area. Based on the results in Figure 13 and Figure 14 in which we had 100 clients accessing the data concurrently, Figure 15 shows the estimated time to complete 100K crops on 100 nodes; we see that in the best case scenario, we have the local FS which would take 10 to 30 seconds, depending on how many crops per image there were on average, with more mediocre but practical results from the ANL GPFS taking between 70 to 200 seconds. The worst was the TG GPFS with times as high as 3000 seconds (almost an hour). Depending on the desired complexity of the system, we can conclude that 100 clients could potentially finish 100K crops on the order of 10~200 seconds utilizing 50 concurent nodes, which seems to be a vast improvement over the sequential time it would take (between 0.3 and 30 hours) depending on the FS used.

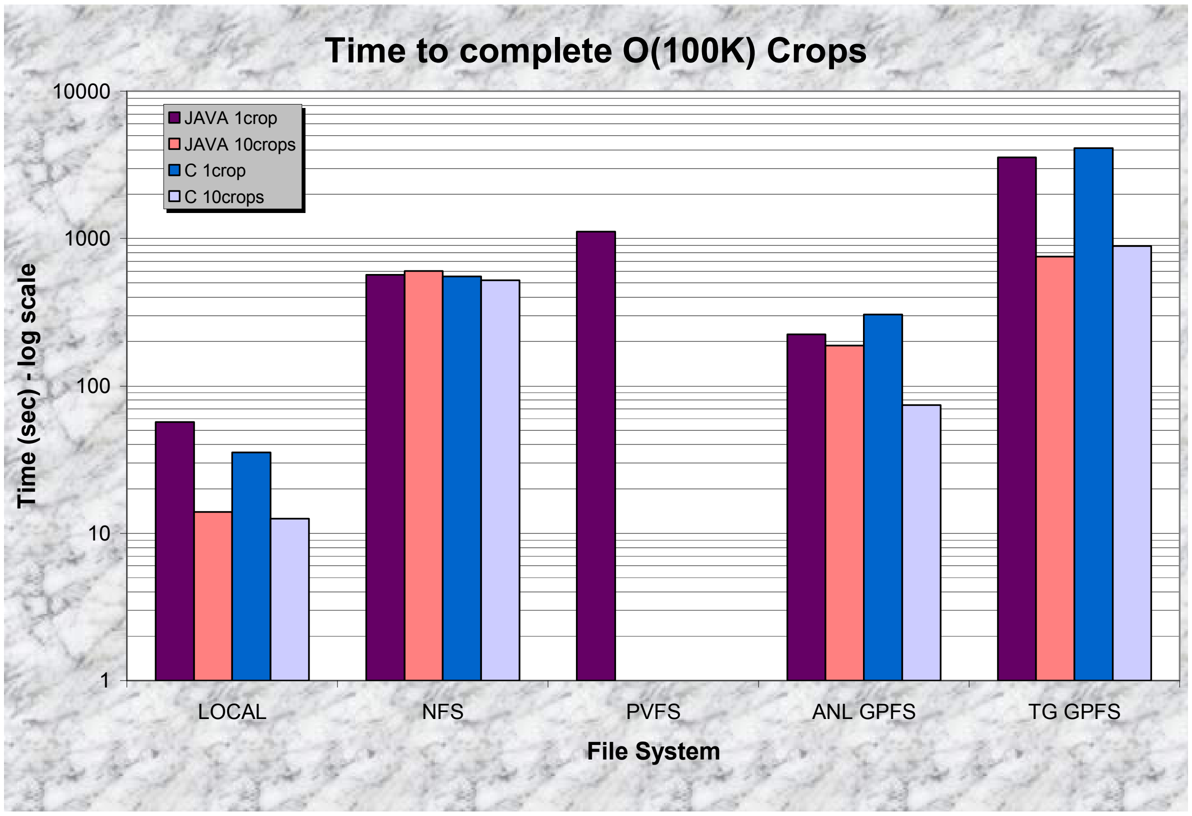
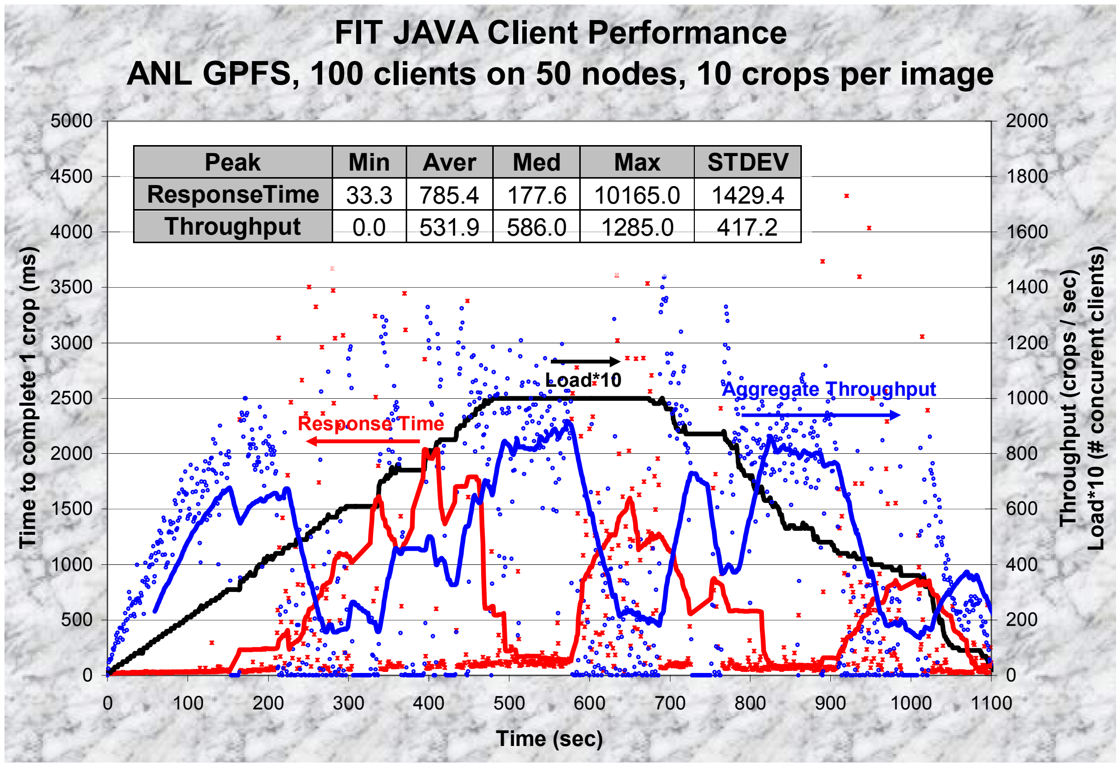


Figure 15: Time to complete O(100K) crops using 100 clients (both JAVA and C); log scale in seconds

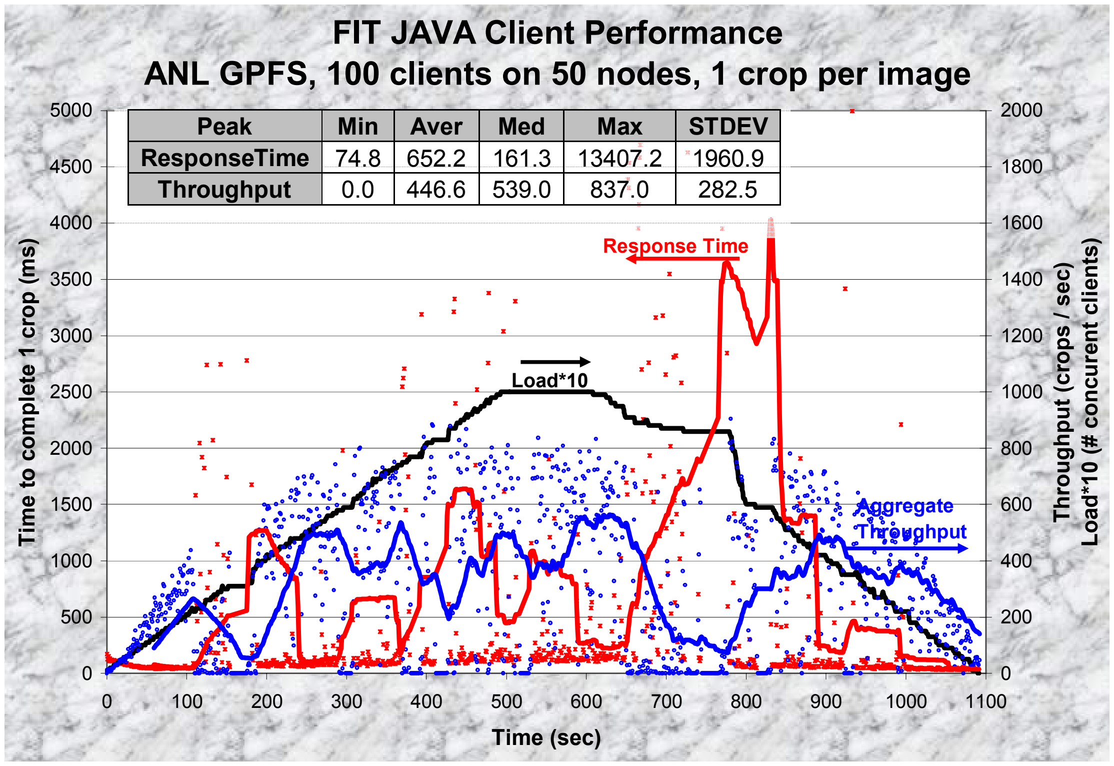
Figure 16 through Figure 28 cover the individual performance runs that were depicted in Figure 13, Figure 14, and Figure 15.

The most notable observation for Figure 16 and Figure 17, which represent the JAVA client performance on ANL GPFS, is that the performance varied quite a bit; this could have been caused by much activity on ANL GPFS due to other work at ANL and across the TG, or it could have been a results of the JAVA semantics on reading from files. Only further experimentation can confirm which of the two scenarios is more likely. Furthermore, we saw very minor improvements due to caching (multiple crops per image). We also saw the throughput flatten out at around 50 clients, although it is hard to really tell because of the large variations in the performance. If this is the case, based on later experiments on the ANL GPFS, we can conclude that the JAVA clients are very CPU resource intensive and that adding a second client on the same node does not yield better performance, despite the fact that all the nodes were dual processor machines. These two tests would have to be redone to solidify any conclusion we arrive about the performance of the JAVA FIT client over ANL GPFS.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	33.3	785.4	177.6	10165.0	1429.4
Throughput	0.0	531.9	586.0	1285.0	417.2

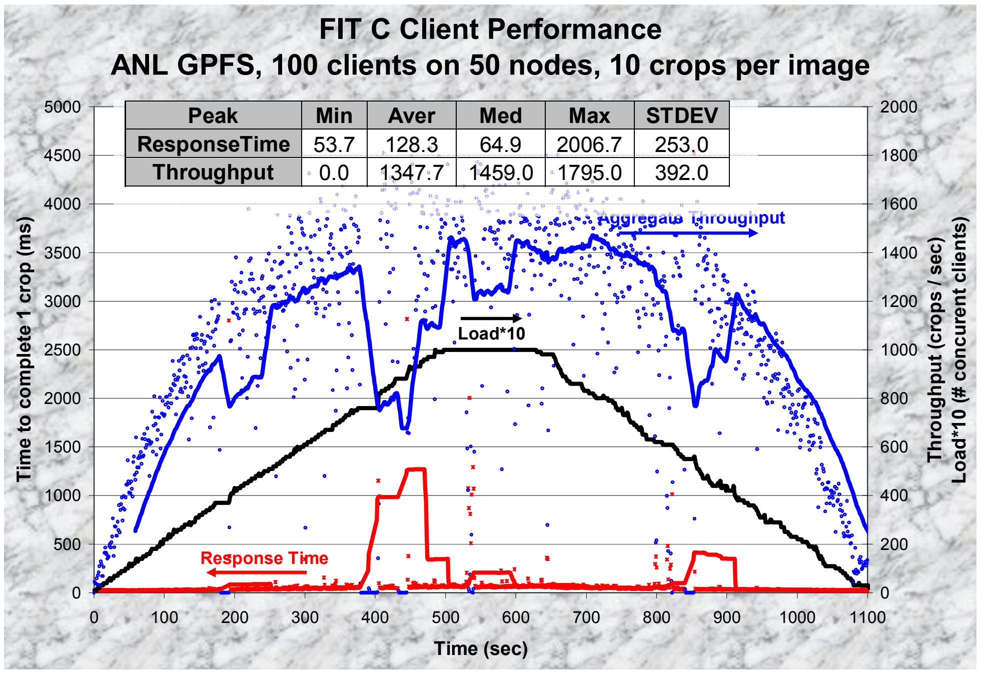
Figure 16: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	74.8	652.2	161.3	13407.2	1960.9
Throughput	0.0	446.6	539.0	837.0	282.5

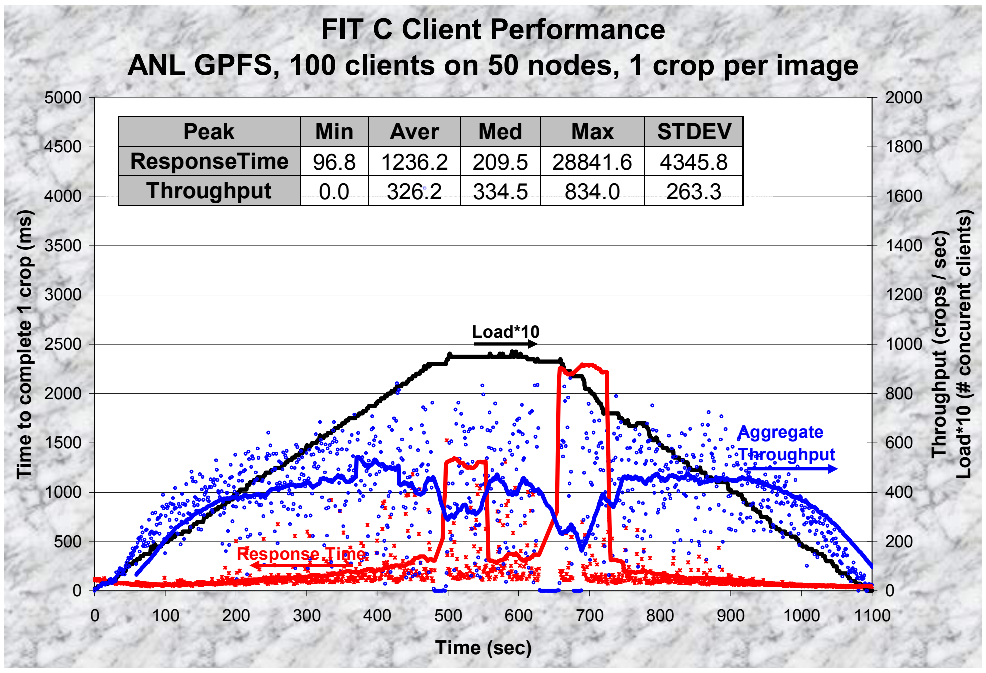
Figure 17: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

On the other hand, the C client over the same FS (ANL GPFS) performed significantly better, almost tripling the median throughput achieved. We see that the throughput increases significantly in the first 50 clients (when there are only 1 client per node), but the performance keeps increasing at a slower pace all the way up to 100 clients. This shows that the ANL GPFS is not saturated with 100 clients generating an aggregated 1500 crops per second. Note that caching made a huge difference for the C client, as the performance increased 4 to 5 times. What is interesting is that the response time remained relatively constant in both scenarios, with a median between 100 and 200 ms.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	53.7	128.3	64.9	2006.7	253.0
Throughput	0.0	1347.7	1459.0	1795.0	392.0

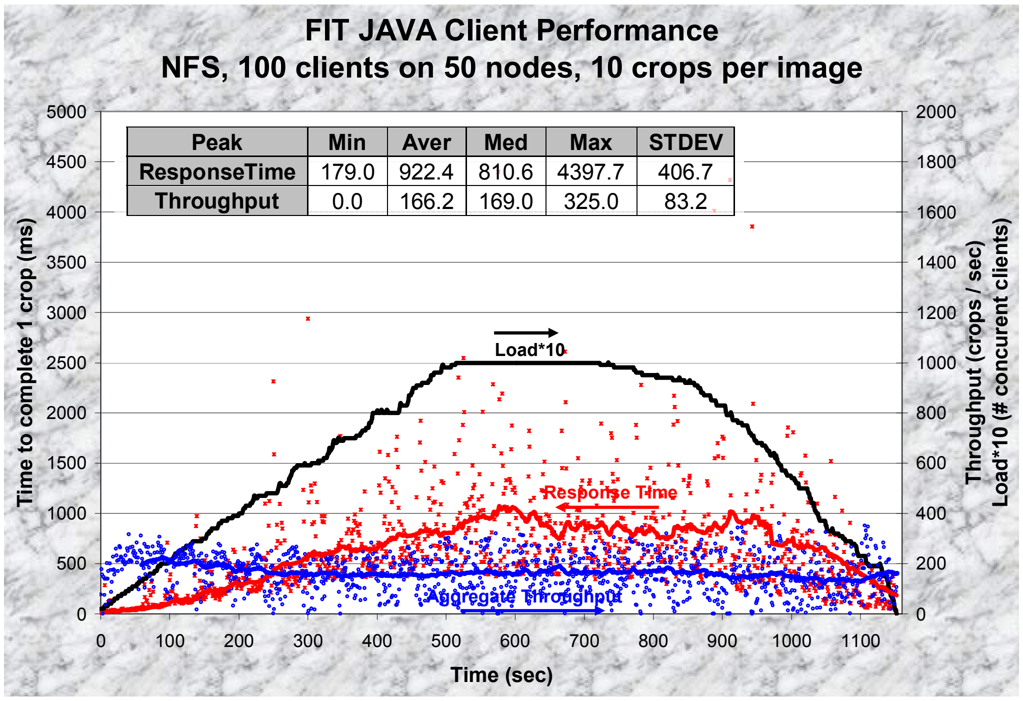
Figure 18: FIT C Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	96.8	1236.2	209.5	28841.6	4345.8
Throughput	0.0	326.2	334.5	834.0	263.3

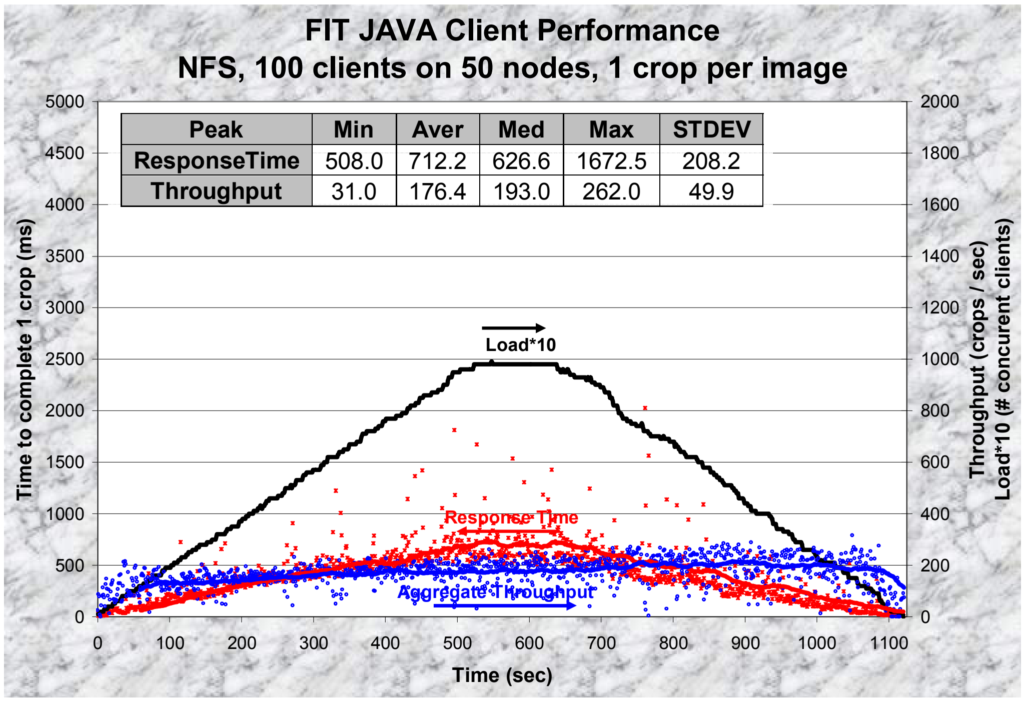
Figure 19: FIT C Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

The NFS depicts very different characteristics when compared to the ANL GPFS in the previous 2 figures. Figure 20 and Figure 21 shows that the NFS gets saturated relatively early with only a few concurrent clients, and thereafter maintains its performance throughout despite the increasing number of clients. Once again, the caching did not seem to make any difference for the JAVA client.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	179.0	922.4	810.6	4397.7	406.7
Throughput	0.0	166.2	169.0	325.0	83.2

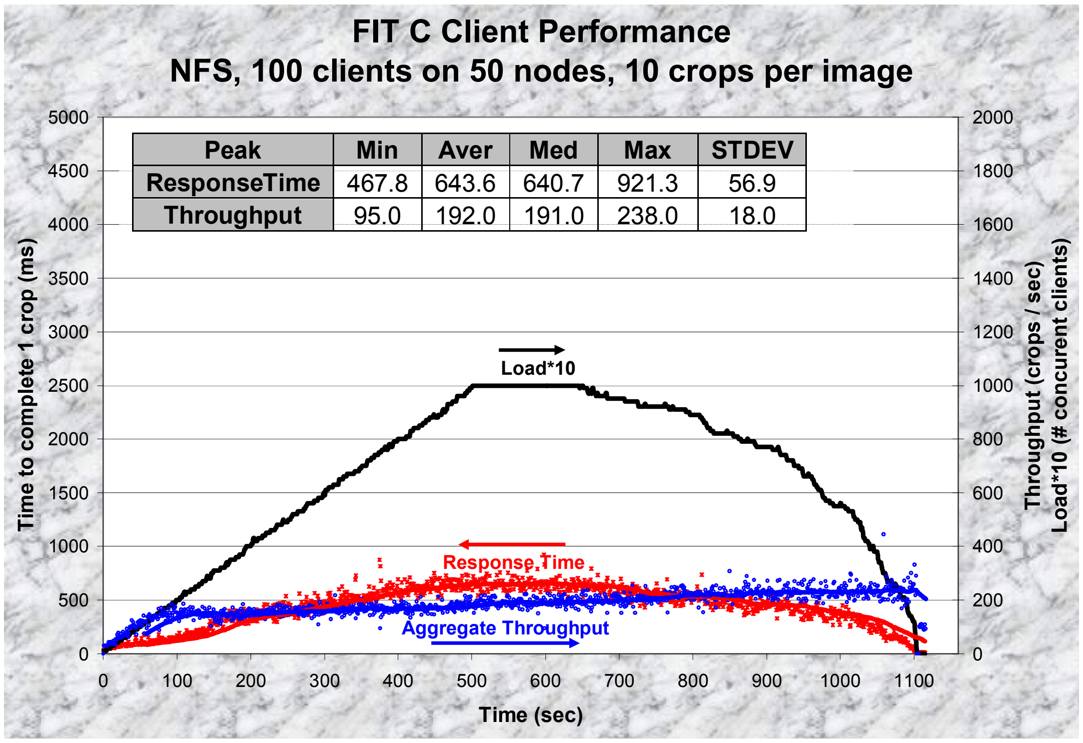
Figure 20: FIT JAVA Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	508.0	712.2	626.6	1672.5	208.2
Throughput	31.0	176.4	193.0	262.0	49.9

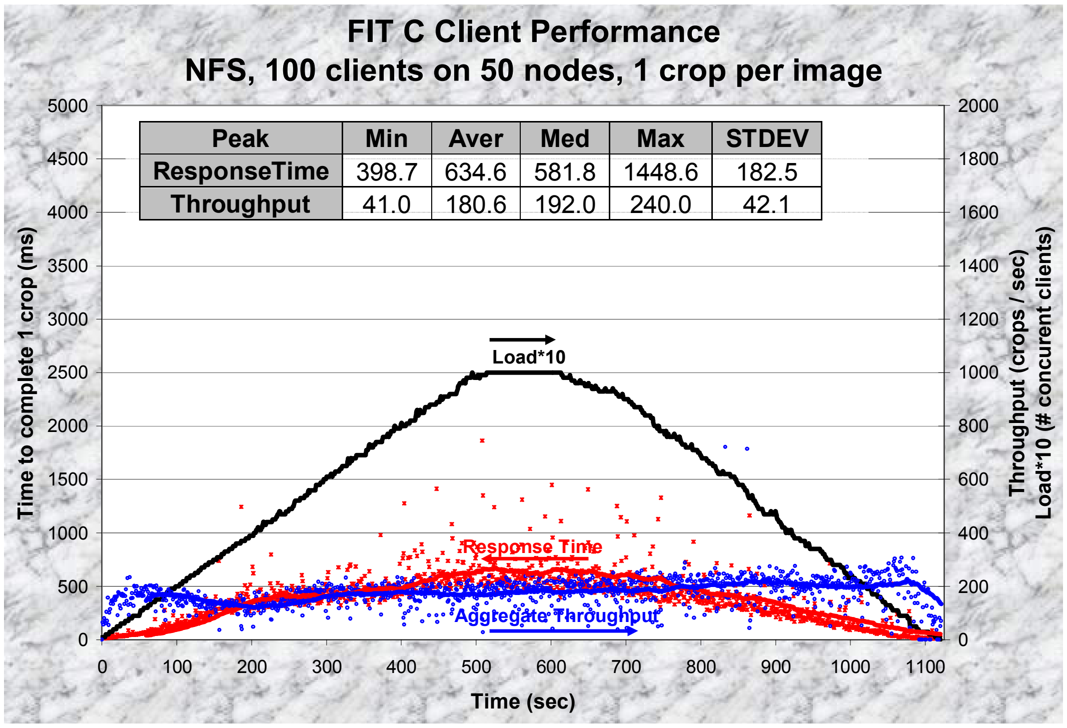
Figure 21: FIT JAVA Client Performance over the ANL GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

In the case of the C client performance over NFS, there was no significant difference between the C client and the JAVA client. Furthermore, there was no significant improvement based on the caching.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	467.8	643.6	640.7	921.3	56.9
Throughput	95.0	192.0	191.0	238.0	18.0

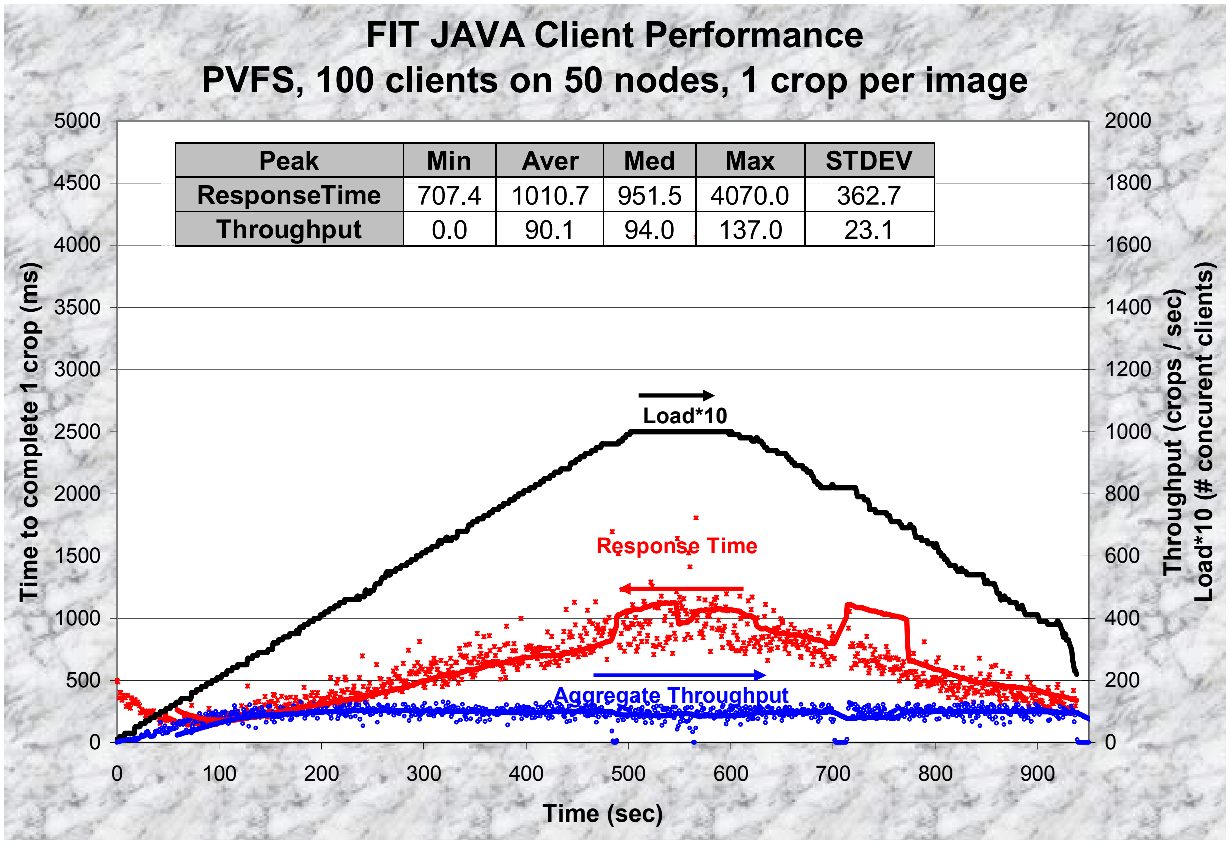
Figure 22: FIT C Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	398.7	634.6	581.8	1448.6	182.5
Throughput	41.0	180.6	192.0	240.0	42.1

Figure 23: FIT C Client Performance over the NFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

Due to technical difficulties with the PVFS, I was not able to run all four experiments (JAVA client with 10 crops / image, JAVA client with 1 crop / image, C client with 10 crops / image, and C client with 1 crop per image) as I did for the other FS. However, I did manage to complete one test, namely the JAVA client with 1 crop per image; the results are shown in Figure 24. We see that PVFS got saturated with just a few clients (similar to NFS), and reached a peak throughput of less than 100 crops per second, which only half the performance of the same client running over NFS. Out of all the alternatives, only the TG GPFS performed worse, but even TG GPFS could have outperformed PVFS given enough clients.

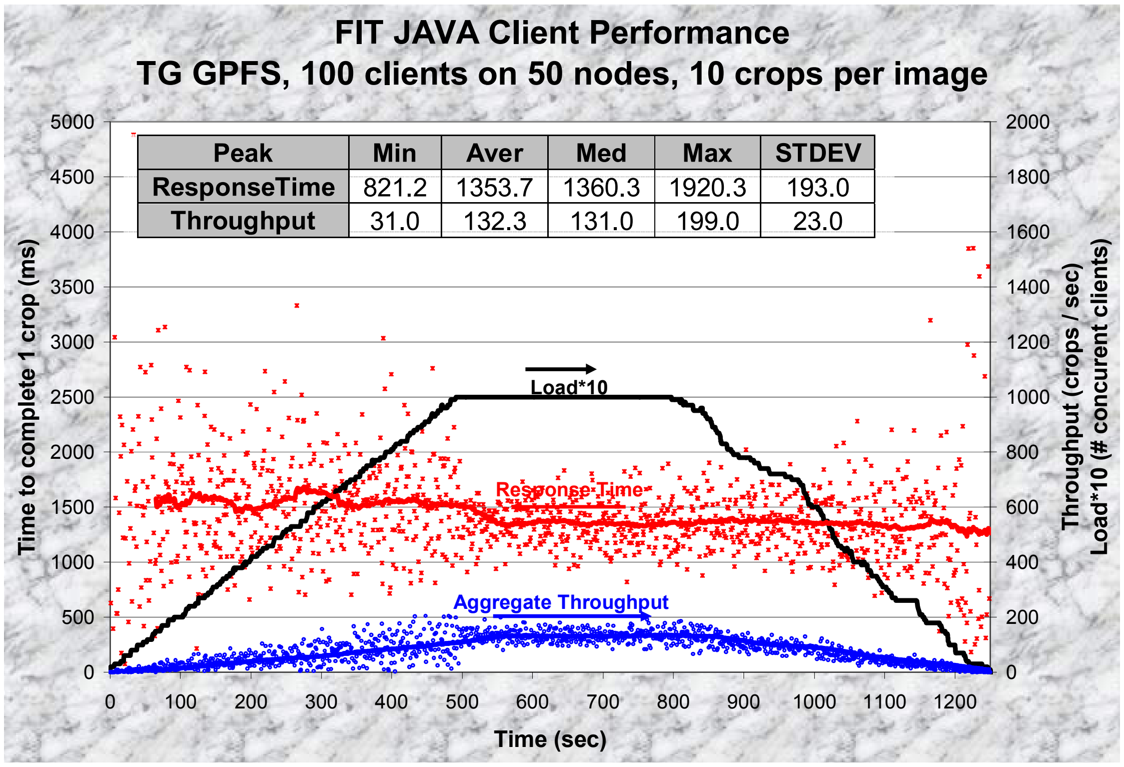


Peak	Min	Aver	Med	Max	STDEV
ResponseTime	707.4	1010.7	951.5	4070.0	362.7
Throughput	0.0	90.1	94.0	137.0	23.1

Figure 24: FIT JAVA Client Performance over the PVFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

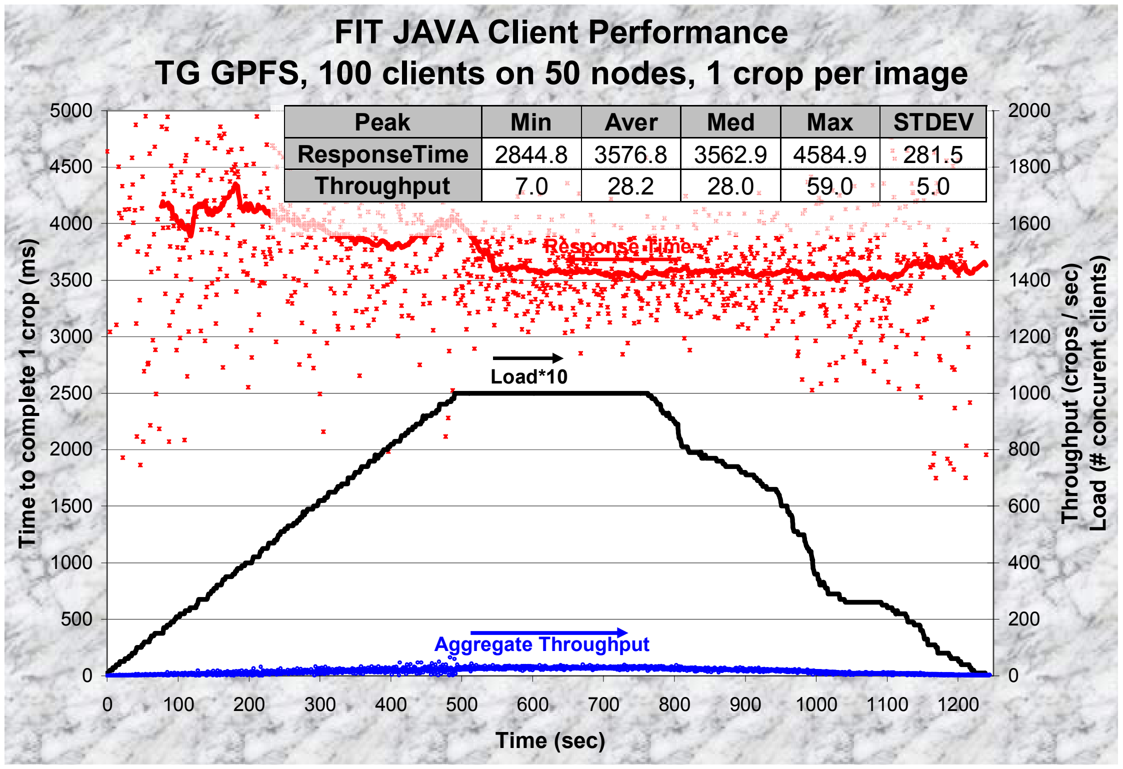
The final sets of experiments were over the TG GPFS (which is over a WAN, as compared to all the previous experiments that were over a LAN). The advantage to the TG GPFS is that it is probably the most scalable due to the many servers that serve TG GPFS requests, but the increased latency to access the data causes the performance of the TG GPFS to suffer for a small number of clients; the more concurrent clients, the better the performance of the TG GPFS will be, up to a certain point (which is surely higher than any of the other FS tested here, i.e. ANL GPFS, NFS, PVFS).

Figure 25 and Figure 26 shows the performance of the JAVA client over TG GPFS. We see a huge improvement with caching, and a solid increasing throughput as the number of clients increase. We certainly did not saturate the TG GPFS with only 100 clients. The biggest drawback is that the peak achieved throughput ranged from 28 to 130 crops / sec, which is considerably lower than what we could achieve with the ANL GPFS, or even NFS. Also note that it takes on the order of 1.5 seconds to perform one crop, which is almost 10 times larger than what we observed for the ANL GPFS.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	821.2	1353.7	1360.3	1920.3	193.0
Throughput	31.0	132.3	131.0	199.0	23.0

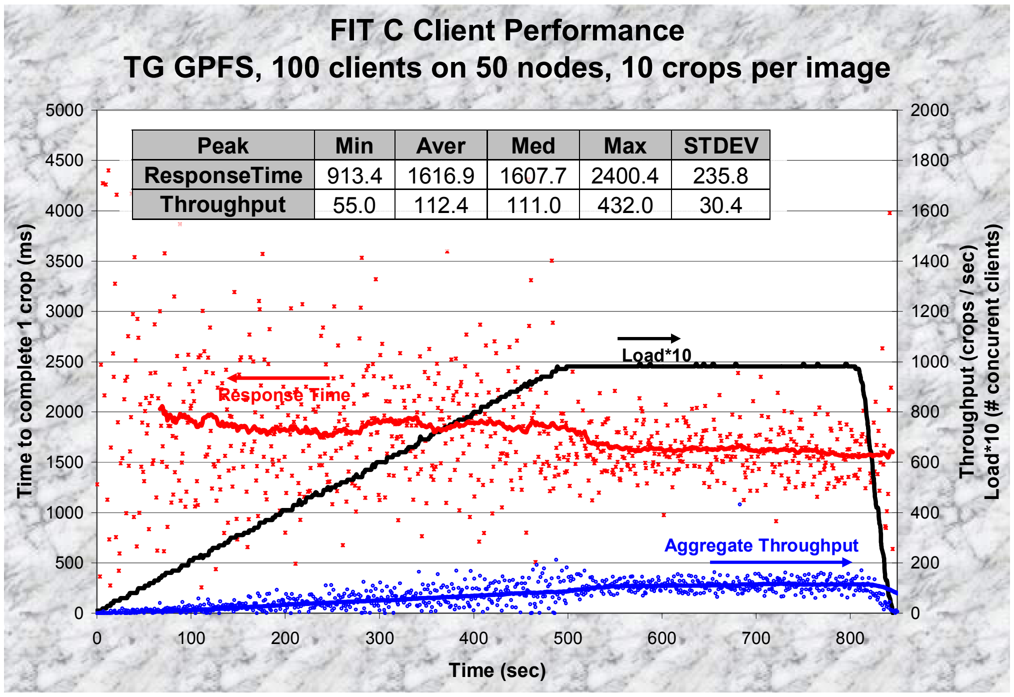
Figure 25: FIT JAVA Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	2844.8	3576.8	3562.9	4584.9	281.5
Throughput	7.0	28.2	28.0	59.0	5.0

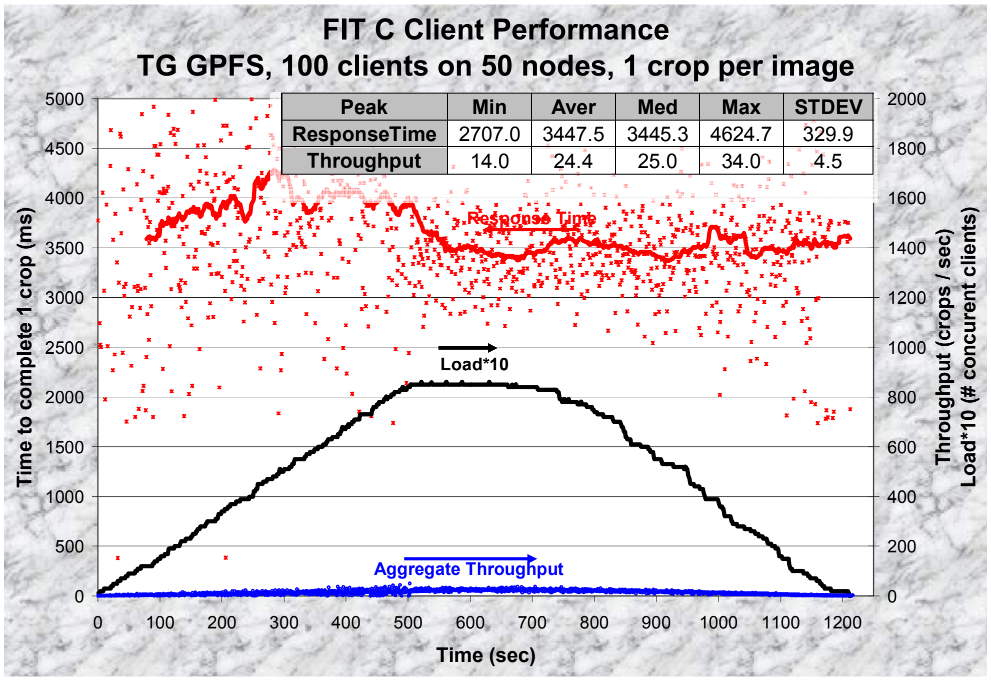
Figure 26: FIT JAVA Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

Figure 27 and Figure 28 shows the performance of the C client over TG GPFS. It is interesting that the performance of the C client came out about 10% lower than that of the JAVA client for the same FS, but since the TG GPFS is a production system, the difference in performance could have been from other usage of the TG GPFS. It would be interesting to redo these two sets of experiments to get a firmer confirmation whether or not the JAVA client is indeed faster over TG GPFS.



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	913.4	1616.9	1607.7	2400.4	235.8
Throughput	55.0	112.4	111.0	432.0	30.4

Figure 27: FIT C Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 10 crops performed



Peak	Min	Aver	Med	Max	STDEV
ResponseTime	2707.0	3447.5	3445.3	4624.7	329.9
Throughput	14.0	24.4	25.0	34.0	4.5

Figure 28: FIT C Client Performance over the TG GPFS with 100 clients running on 50 dual processor nodes; each image had 1 crop performed

3.3 Client Performance Profiling

In order to see the efficiency of the code that reads the FIT file from the various file systems (LAN GPFS, WAN GPFS, and NFS), I performed some basic experiments that looked at the CPU utilization, observed network throughput, and theoretical throughput (lower bound based on ROI size, and upper bound based on image size). The numbers presented in the following figures are just a rough estimate, and if they seem interesting, a more detailed analysis should be done.

The tables follow the following conventions:

- ROI size (maximum size is 1536x2048
    - 10x10: 10 pixel by 10 pixel region of interest (ROI), where each pixel is a 16 bit short value
    - 100x100: 100 by 100 pixel ROI; this is the size that I used in all my previous experiments
    - 1000x1000: 1000 by 1000 pixel ROI
- ROI TP (ROI/s)
    - ROI (of the corresponding size) cropped per second by a single client from the corresponding file system (i.e. ANL GPFS, TG GPFS, or NFS)
        - ANL GPFS: GPFS at ANL which is accessed over a LAN
        - TG GPFS: GPFS at SDSC which is accessed over a WAN
        - NFS: NFS at ANL which is accessed over a LAN
- CPU (%)
    - CPU utilization (2 processor system) at the client side by having 1 client cropping out the needed ROI from the corresponding FS
- Theoretical TP MIN (Mb/s)
    - The minimum amount of data in Mb/s that would be needed to transfer in order to sustain the achieved ROI TP according to the ROI size and the corresponding FS; for example, if the ROI TP was 1 per second, and the ROI size was 10x10, then the minimum needed data needed to perform the crop of the ROI would be: 10 pixels x 10 pixels x (16 bits / (8 bits/byte)) x 1 ROI/s = 200 bytes / sec; the observed network usage should never be less that this theoretical lower minimum
- Theoretical TP MAX (Mb/s)
    - The maximum amount of data in Mb/s that should be needed to transfer (if all image data was transferred regardless of the ROI size) in order to sustain the achieved ROI TP and the corresponding FS; for example, if the ROI TP was 1 per second (for any ROI size), then the maximum needed data needed to perform the crop of the ROI would be: 1536 pixels x 2048 pixels x (16 bits / (8 bits/byte)) x 1 ROI/s = 6 MB / sec; the observed network usage should never exceed this theoretical upper maximum
- NET TP (Mb/s)
- COPY TP MAX (Mb/s)
    - The maximum observable network throughput in Mb/s achievable on the corresponding FS by copying a large (1000 files x 6.1 MB each = 6.1GB) set of files from/to the corresponding FS in a single sequential thread; due to overhead in performing the ROI crop, the observed network throughput (NET TP) should not exceed this value for a single client

In the ANL GPFS, we notice that the observed network throughput (NET TP) increases with larger ROI sizes, which indicates that the read operation does not read the entire file, but only a subset. Further proof of this is the fact that for ROI of 10x10 and 100x100, the NET TP is much lower (123 ~ 167 Mb/s) than the theoretical maximum throughput (Theoretical TP MAX) of 567 ~ 841 Mb/s. We notice that for a 1000x1000 ROI size, the achieved network throughput almost reaches the theoretical maximum throughput, indicating that in order to obtain a 1000x1000 ROI from a 1536x2048 image, almost the entire image is read into memory. We also see that the CPU utilization is minimal (11%) when getting ROI of 100x100 or smaller, so we can actually have multiple concurrent clients on the same physical machine without saturating the processor. It would be interesting to get some results via DiPerF to see the performance increase with increasing concurrent clients on the same physical machine on the various different file systems.

LAN GPFS						
ROI size	ROI TP (ROI/s)	CPU (%)	Theoretical TP MIN (Mb/s)	NET TP (Mb/s)	Theoretical TP MAX (Mb/s)	COPY TP MAX (Mb/s)
10x10	17.2	11%	0.0	123.3	841.4	600
100x100	11.6	11%	1.8	167.5	567.4	600
1000x1000	6.1	39%	78.3	290.2	299.1	600

Figure 29: ANL (LAN) GPFS Performance

In the WAN GPFS, we see that the CPU % utilization was constant throughout the various ROI sizes; at the same time, the observed network throughput (NET TP) also seemed to be relatively constant. This would indicate that the high latency (~56 ms) is limiting the network throughput (due to the bandwidth delay product), and hence the achieved throughput in terms of ROI/sec. The low CPU utilization indicates that dozens of concurrent clients could possibly be run on a single machine, effectively getting throughput numbers similar to those observed on the LAN GPFS. This needs to be investigated further.

WAN GPFS						
ROI size	ROI TP (ROI/s)	CPU (%)	Theoretical TP MIN (Mb/s)	NET TP (Mb/s)	Theoretical TP MAX (Mb/s)	COPY TP MAX (Mb/s)
10x10	1.1	3%	0.0	34.4	53.8	160
100x100	1.1	3%	0.2	33.7	53.1	160
1000x1000	0.8	3%	8.4	33.0	36.7	160

Figure 30: WAN GPFS Performance

Finally, the NFS results are similar to those of the ANL GPFS.

NFS						
ROI size	ROI TP (ROI/s)	CPU (%)	Theoretical TP MIN (Mb/s)	NET TP (Mb/s)	Theoretical TP MAX (Mb/s)	COPY TP MAX (Mb/s)
10x10	14.7	5%	0.0	31.7	717.6	320
100x100	12.0	12%	1.8	136.5	588.0	320
1000x1000	3.1	28%	46.1	151.2	152.3	320

Figure 31: ANL NFS

3.4 Experimental Result's Conclusions

Overall, the performance of both the JAVA client and the C client running as one client at a time seemed good. The performance degradation when accessing the data in the GZ format is very large, and I believe it should be avoided at all costs. The compression ratio is only 3:1, so keeping the entire archive of 2TB in an uncompressed format would only increase the storage requirement to 6TB, but yielding magnitudes order better performance.

On the other hand, the performance difference among the different FS seems to have grown significantly as we run 100 clients in parallel. Except in just a few cases, the performance difference between the JAVA and C client seems to be almost negligible.

Based on the results in Section 3, the PVFS and NFS should be avoided if possible. Furthermore, there are significant performance gains by keeping the data available in an uncompressed format. The TG GPFS should also be avoided, unless the scalability of the system (concurrent clients accessing the data) will be larger than what can be supported by the ANL GPFS. Therefore, if performance is important, then the potential FS are the local FS and the ANL GPFS; in both of these cases, there were improvements in performance for the C client in comparison to the JAVA client, so the C client would be preferable. The data should also be stored in a raw and uncompressed format.

Regarding the conclusions we can draw from the client performance profiling, there seems to be quite a bit of variation in the performance (perhaps due to the shared nature of the file systems); the good part is that with ROI of 100x100 in size, only 30~60% of the data is being transmitted in order to effectively perform the crop on the ROI. This is much higher than what would be optimal (less than 1%), but at least we do not need to transfer the entire data (100%) to obtain the ROI. One possible explanation is that the ROI does not form a contiguous region on the remote file system storage, and hence is read in smaller chunks; these smaller chunks are usually relatively large (several KB at least), and hence much unneeded data is often read when needing ROI of such small sizes (i.e. 100x100). Also, it seems promising that due to the low CPU utilization during a single client accessing the data, it would seem worthwhile to investigate (via DiPerF) the upper limits of saturating the CPU and network resources at a single physical machine.

4 Proposed Architecture

Due to my conclusions from the previous section, I have come up with three potential architectures, each with their own merits on three axes: 1) simplicity of design 2) performance, and 3) scalability.

4.1 Design based on TG GPFS (WAN)

The upper bound of the performance of this system based on TG GPFS (WAN) would be between 25 to 100 crops per second (aggregate throughput), which is the lowest performance of the three proposed architectures. It would take on the order of low 1000s of seconds to complete 100K crops on 100 concurrent clients. The scalability of this architecture will be dependent on how many servers are available to serve the TG GPFS (currently about 60 servers); the scalability would be better than the ANL GPFS, but not as good as using the Local FS. Out of the three architectures, this one would be the simplest design due to the shared and global file system.

Figure 32 shows the proposed architecture that is based on the TG GPFS, a file system that runs over a WAN. In this system, the Astro Portal (AP) is the gateway to access the compute nodes. For simplicity, the compute nodes (irrespective of location) access the needed data from a common global file system, namely the TG GPFS; depending where the compute node physically resides, the connection could either be over a LAN or a WAN. The AP Manager's duty is to keep the data set consistent between the TG GPFS and where the original data set resides via protocols such as GridFTP, RFT, etc. It is the AP's responsibility to send query requests to the nodes that have the least expensive access to the TG GPFS; with only 8 sites, it would probably not be hard to build some static models which then get updated based on real-time observed performance.

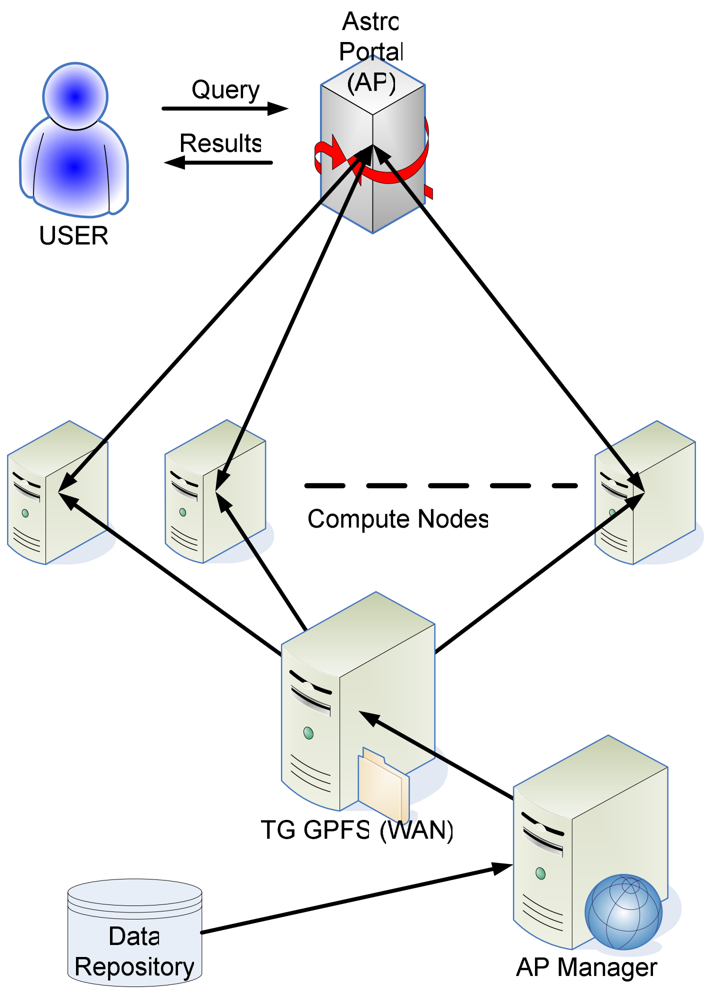


Figure 32: Proposed system architecture based on TG GPFS (WAN)

4.2 Design based on GPFS (LAN)

The upper bound of the performance of this system based on GPFS (LAN) would be between 400 to 1500 crops per second (aggregate throughput), which offers the best performance while still maintaining a relatively low complexity in the system architecture. It would take on the order of low 100s of seconds to complete 100K crops on 100 concurrent clients. The scalability of this architecture will be dependent on how many servers are available to serve the local GPFS (currently about 8 servers at ANL, and probably something similar at each of the 8 sites); the aggregate scalability if all sites would be used is probably similar to that of the TG GPFS, but not as good as using the Local FS. Out of the three architectures, this one would be the middle one in terms of design complexity mainly due to the shared site file system. The entire data set would be replicated (by the AP manager) across all participating sites (about 8 of them), and hence the AP would be almost identical as to the one for the TG GPFS. The AP's responsibility in this case would be to make sure it is load balancing the queries uniformly across the various sites.

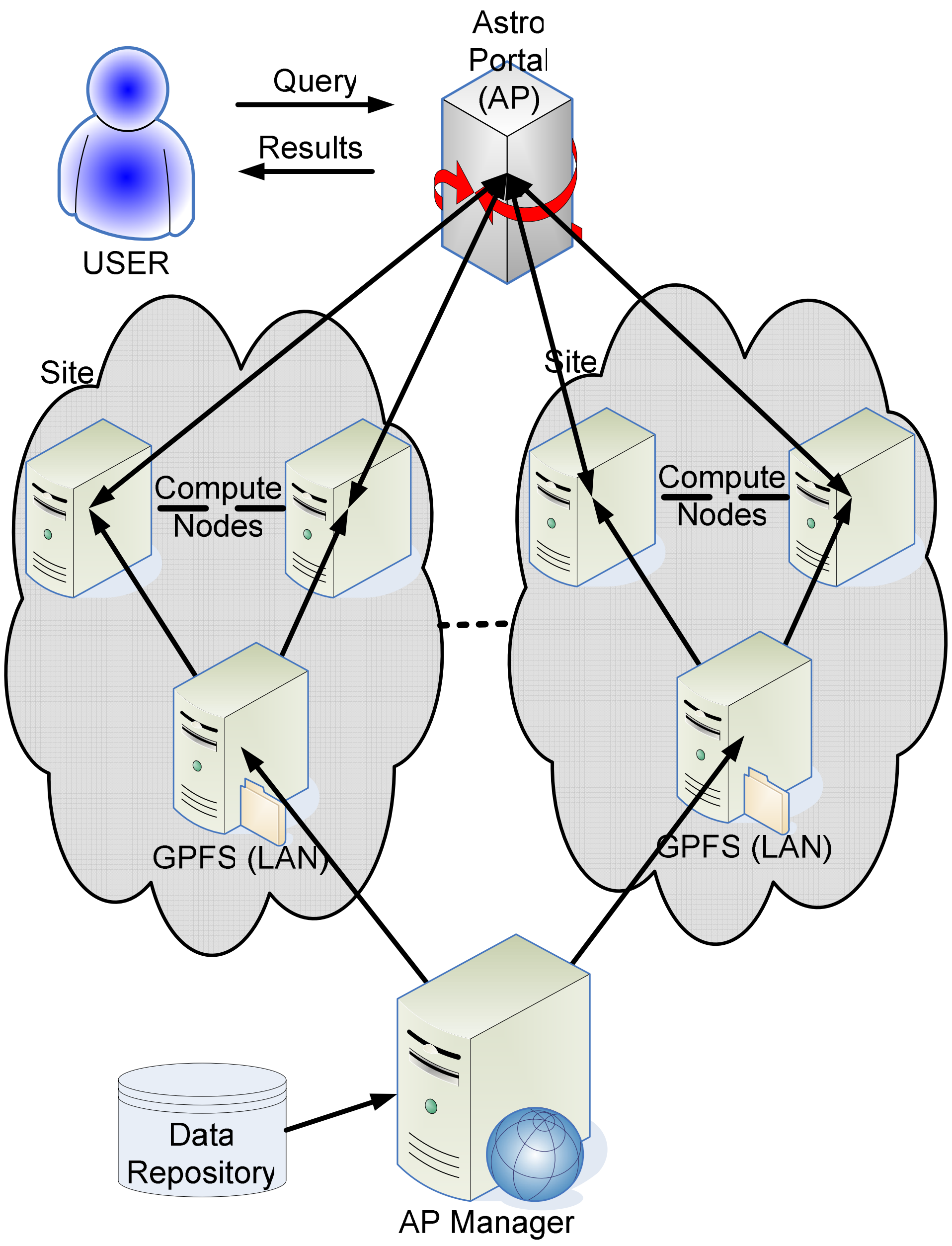


Figure 33: Proposed system architecture based on GPFS (LAN)

4.3 Design based on Local FS

The upper bound of the performance of this system based on the Local FS would be between 1,500 to over 10,000 crops per second (aggregate throughput), which offers the absolute best performance, but increases the complexity of the system architecture considerably. It would take on the order of low 10s of seconds to complete 100K crops on 100 concurrent clients. In theory, the scalability of this architecture should be linear with each increasing client with its biggest limitation coming from the AP Manager keeping all the data set replicas synchronized. This architecture is by far the most complex due to the fact that partial data sets must reside at each node on local storage; the entire data set on the order of TB is not feasible to store at each local node, but a subset of that on the order of 10s GB is certainly feasible with disk sizes commonly exceeding 100s of GB. Note that the AP Manager has an index as well; this index keeps track of what node has what data, so that queries can then be directed directly at the nodes that have the correct data. The AP's responsibility in this architecture is to make sure it keeps the main index synchronized and sending queries to the nodes that have the necessary data in their local FS. Outside of the fact that the complexity of this architecture is significantly higher, there are two main drawbacks to this architecture: 1) a data set that changes very frequently would cause a big overhead to keep the replicas synchronized, and 2) this would require more aggregate storage since it is feasible that each piece of data would have to have multiple copies for redundancy and robustness of the system.

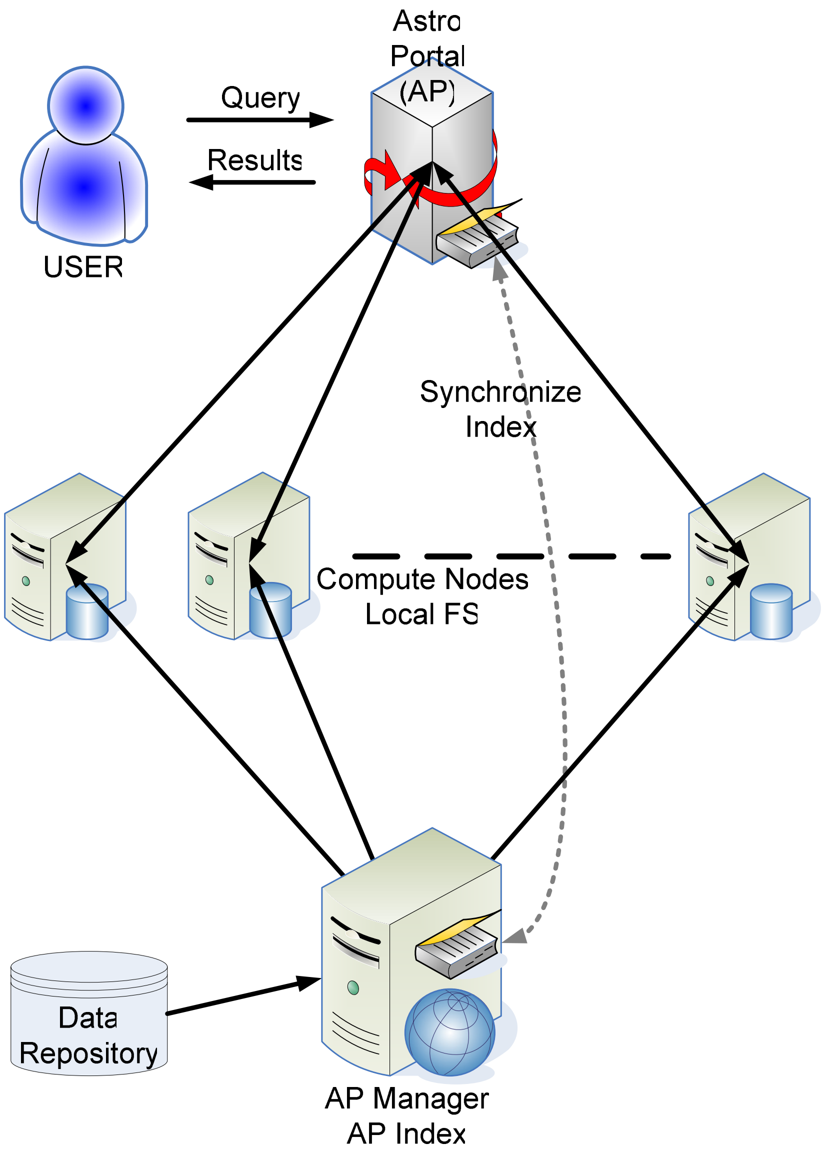


Figure 34: Proposed system architecture based on the Local FS

4.4 Architecture Building Blocks

The target environment for this architecture is the TeraGrid, which gives us a rich array of both software and hardware infrastructure in order to come up with an efficient and good implementation of the proposed architecture without "re-inventing the wheel". Specifically, the TeraGrid has the Globus Toolkit (GT) installed through the eight different sites, giving us extra incentives to leverage components of the GT in the actual design of the system.

It is envisioned that the user would submit the queries via a web interface, and receive the results via the same web interface. The web interface could be a wrapper around a web service client that would communicate with another web service (WS), namely the Astro Portal (AP) WS. Advanced users could access the web service directly bypassing the web interface in order to make automating the use of the AP system easier. The first two proposed systems (Figure 32 and Figure 33) have quite a simple data location mechanism, but the third system (Figure 34) requires a more sophisticated data location component due to the storage of the data across many local disks, rather than some common globally accessible file system, such as GPFS. In this more complex system, the AP could use RLS (Replica Location Service) to locate the data it needs to process the query. Once the data has been found (via RLS or some other very simple mechanism), the AP could use GRAM (Grid Resource Allocation and Management) to submit jobs in the TG across the compute nodes in order to process the queries. If GRAM is not used, then a web service would have to be run at the compute nodes. The easy solution is to start up a pool of web services, and let them sit idle until queries arrive. The problem with this approach is that it works great in an environment where machine resources are shared among many users at the same time, and hence it could only offer best effort service. On the other hand, the TG has local schedulers that are configured to allow only a certain number of concurrent jobs on the same physical machine on the order of number of processors. This means that the machines where the web service would be running would be significantly underutilized whenever the web service was idle. There would be a need for a mechanisms to allow other jobs to be processed by these nodes, which could be suspended while the web service had useful work to do. On the other hand, the solution involving GRAM to submit jobs to a pool of nodes has its own limitations as GRAM is quite expensive and would slow down the performance of the system considerably if the query size would not be large enough.

4.5 Single Site AstroPortal Architecture

In order to make Figure 35 and Figure 36 easier to understand, let me define the basic components and their abbreviations:

- **Site:** A TeraGrid site, such as UC/ANL, SDSC, NCSA, PSC, ORNL, TACC, etc…
- **User:** user from the astronomy domain who wants to query the data set with a 5-tupple (path & file name, x-coordinate, y-coordinate, height, and width)
- **AstroPortal Web Service (AP WS):** A WS that gives users an entry point into accessing TG resources to process the user's queries
- **MDS4 Index:** A standard MDS4 Index used for resource (AP WS) discovery by the users
- **Compute Nodes - AstroClient (AC):** dedicated nodes in TG that are reserved in advance to be used for processing queries from the AP WS
- **Data Repository:** the original data set in compressed format that can be accessed via GridFTP
- **AstroData (AD) Manager:** A data resource manager that keeps the data set up to date between the data repository, and the corresponding file systems (Local GPFS, TG GPFS, etc…); in the distributed version, the AD Manager could also use RLS to manage data replication; the AD Manager also communicates with the AP WS in order to keep the AP WS data set index updated with the latest data set location
- **Local GPFS:** Refers to site local GPFS accessed over a LAN
- **TG GPFS:** TeraGrid wide GPFS accessed over a WAN
- **RFT:** Used to update the working data set on GPFS from the data repository
- **GRAM:** Used to make advanced reservations of AC compute nodes by being scheduler independent
- **RLS:** used to keep track of the data replicas in the distributed AP architecture

The entire AP system would be based on various components (GRAM, MDS4, RFT, GridFTP, RLS, and WS) of the GT4. In addition, some internal communication between various components of the system could be achieved using network services in order to keep the communication as light weight as possible.

Once the AP WS is up through a basic bootstrapping mechanism (possibly via GRAM by the administrator of the AP), the AP WS would register itself with a well known MDS4 Index. Users could later find where to access the AP WS via the well known MDS4 Index.

The user would make a query against a database containing the meta-data information (i.e. give me all the quasars in the solar system Omega), and would get back a list of tuples each indicating a quasar; this tuple would consist of: 1) file path and name, 2) center of region of interest (in terms of x and y coordinate), and 3) the size of the region of interest (in terms of height and width). Ideally, users could send an entire job (made up of multiple tuples, possibly 10s to 1000s of tuples) in one WS call in order to amortize the cost of the WS call itself.

Initially, the AP WS would make some advanced reservations (via GRAM) of some predefined set of resources within the local site for a predefined duration; the set of resources and the length of the reservation should be kept relatively small due to these resources being dedicated to the AP WS regardless if there is work to do or not.

Via the GRAM API, the AP WS would start up the AstroClients (AC) on the reserved set of resources; the AC would all be idle until queries would be sent by the AP WS. The AP WS would maintain a queue of queries being sent by users; queries would be taken from the queue and forwarded to the appropriate resource to process it. When all the queries from one job (1 job could contain multiple queries from 1 user) are complete, the AP WS would package the results (returned by the AC) into an archive, and send it back to the user.

In the meantime, the AP WS would have to maintain the set of available resources by dynamically increasing and decreasing the advanced reservations; ideally, there would always be idle AC ready to process any query almost instantaneous, yet there would never be more than a few idle AC for a prolonged period of time since that would consume resources that would otherwise have been available to solve some other problems.

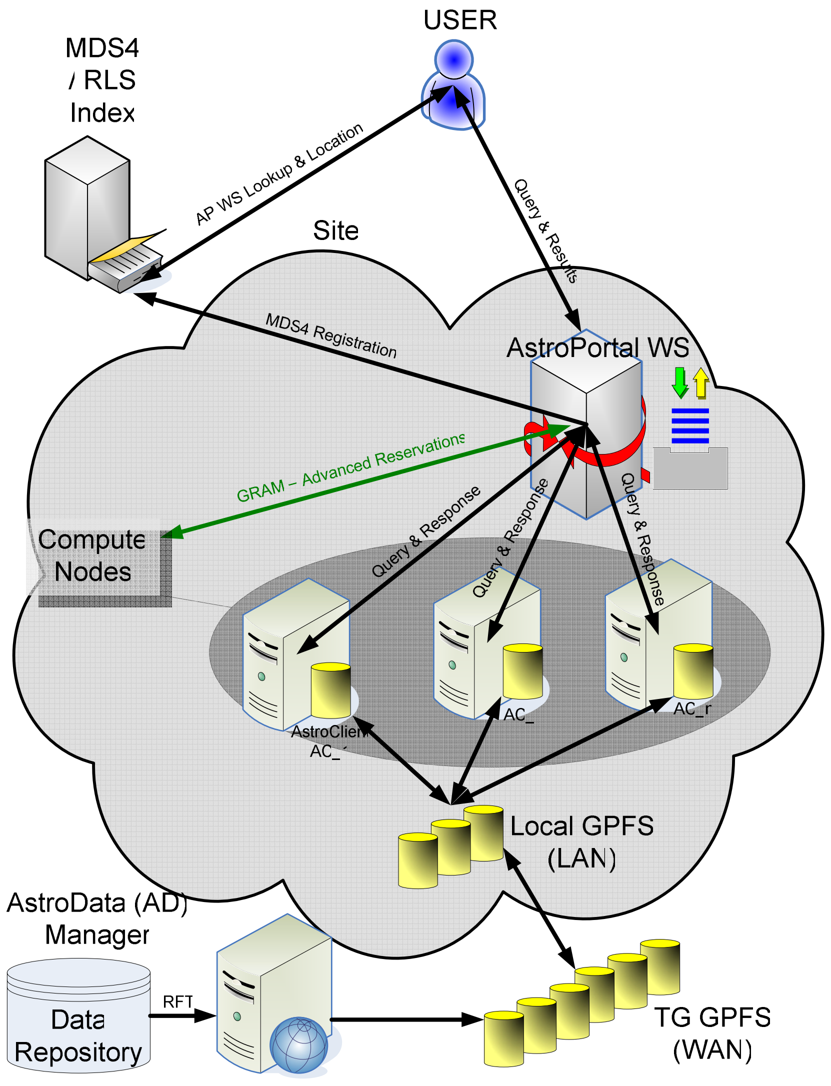


Figure 35: Single Site AstroPortal Architecture

The AP WS could use RLS to maintain a coherent state between the replica location among the different layers (LOCAL, LAN, WAN). Ideally, as the data gradually flows in (from WAN, to LAN, to LOCAL) as AC access the data, queries would run faster and faster over time. This would be true if we were able to keep the reservations of the worker resources indefinitely, which would mean that the data would be accessed more and more over the local disk as the system was used more and more. The TG GPFS (WAN) and local GPFS (LAN) will be persistent storage, but the LOCAL disk storage will be fairly dynamic (worker resources will start-up and terminate regularly due to advanced reservation policies and the fact that the system should not waste resources). In a simple implementation, if the advanced reservations are not long enough, then the LOCAL disk will never get to be populated enough that the system gets a query that actually re-uses the LOCAL data, and hence the system would read most of the data from the local GPFS (LAN).

In order to truly reach that best performance, there would be a need for a worker resource to transfer its state (work queues and locally cached data) from one resource (i.e. node) to another. As time progresses, we could see this transferring of state take longer and longer due to a growing local cache of data. For example, if we had 10TB of data, and 100 clients (distributed over 100 nodes), then each client could potentially have up to 100GB of local cache data, which would take around 15 minutes to transfer from one node to another over a 1Gb/s link. This is high cost of transferring state is OK as long as it does not occur too often, but that means that the system will not be very dynamic and will not be able to respond to "hot-spots" of large number of queries for a short period of time without wasting many resources.

Furthermore, when the system is idle because of no new incoming work, the AP WS will eventually de-allocate all the reserved resources except for a few. What happens to the state of all the resources that get de-allocated? If the system actually purges the state as well, then we are back to the simple implementation that I started with, in which most of the cached data will be deleted when the system is idle, and the performance of the entire site will really be limited to the performance of the local GPFS (LAN).

Finally, the AD Manager would ensure (via RFT) that the data on the TG GPFS is up-to-date based on the data repository. This is not a critical component since we can assume that the data does not change frequently.

4.6 Distributed (Multiple Site) AstroPortal Architecture

The distributed version of the AP WS becomes much more interesting with its added complexity, but also offers a much more scalable solution! The majority of the intra-site communication remains unchanged, with the exception that the MDS4 Index need not be specifically associated with any particular site. Each AP WS from each site would register with the MDS4 Index; when users would query the MDS4 Index, users could pick a possible AP WS at random, or based on some simple metrics (i.e. AP WS load, latency from the AP WS to the user, etc) that MDS4 exposes to the users about each AP WS.

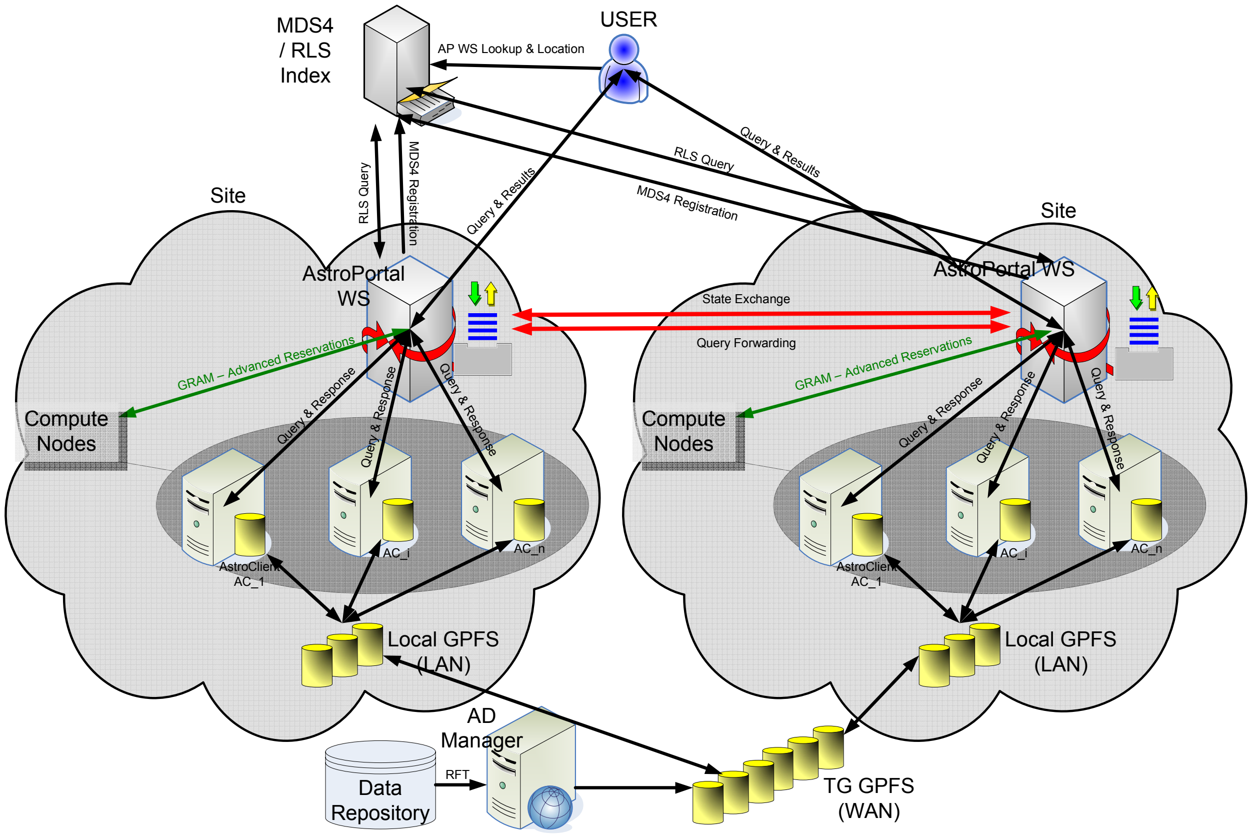


Figure 36: Distributed (Multiple Site) AstroPortal Architecture

The distributed architecture gives us a more scalable system with higher performance. The key to the enhanced performance is the ability to harness all the resources across various sites in the TG; the interaction between the various AP WS from each of the various sites is critical. Each AP WS would get its share of queries, but depending on what data is needed, and the load at the various sites, queries could be handled locally within the site, or they could be forwarded to another site that could offer faster performance due to data locality, more available resources, faster hardware, etc…

Finally, the AD Manager remains unchanged in the distributed architecture.

4.7 Open Research Problems

I believe that there are three main areas with open research problems that the architecture design in Figure 35 and Figure 36 exposes:

1. cluster level advanced reservations, resource allocation, resource de-allocation (Figure 35)
    - A TG site can be considered as a cluster, so we might be able to leverage from techniques used in large clusters; depending on the problem we are addressing (workload characteristics, number of users, data set size and distribution, computational intensive clients, I/O intensive clients, etc…), different techniques and heuristics will apply for managing efficiently the set of resources
2. data management, location, and replication (Figure 36)
    - There are some very interesting problems around data management, in which we have a very large data set that we want to break up among various sites, but also do some level of replication among the sites for improved performance; furthermore, doing data movement based on past workloads and access patterns might prove to offer significant performance gains; a significant challenge will be how to perform efficient state transfer among worker resources while maintaining a dynamic system
3. Distributed resource management between various sites (Figure 36)
    - The inter-site communication among the AP WS and its effects on the overall system performance is very interesting; work can be performed at the local site, or it could be delegated to another site that in theory could complete the work faster; the algorithms, the amount of state information, and the frequency of state information exchanges all contribute to how well and evenly the workload is spread across the various AP WS, which ultimately decides the response time that the user observes

In my opinion, all three areas mentioned above seem to fit in the larger context of resource management. The successful implementation the distributed AP WS and the optimization of the use of both the data storage and the computational resources could lead to a scalable system supporting large numbers of concurrent users while providing very fast response times in comparison to traditional single server implementations. The use of the GT4 throughout the architecture will allow the system to interoperate with other system easily, and provide a standard method of accessing the system.

Although this system is geared towards an astronomy application, I believe that the exact same architecture and optimization algorithms would be suitable for a wide range of applications that have the following characteristics: 1) large data sets, 2) large number of users, and 3) large workloads that can be easily broken up into smaller workloads. I believe that there are many applications that fall in this category; one such example is Computer Aided Diagnosis (CAD) medical systems that are used to screen large number of patient images for cancers; the data set is very large (millions of images of the human body), and each image can be processed independently of other images in the data set (hence the ease of braking up large workloads into smaller ones). If analyzing a single image took on the order of seconds, processing the entire data set on a single machine could take on the order of days, but parallelizing the entire process to hundreds of worker nodes could provide speedups equal to the number of resources used. Furthermore, there are medical centers throughout the whole country that perform these early screening for cancer, and hence there are probably a large user base that would use such a system if it gave the doctors a faster way of screening their patients. This problem from the medical field is just one example of another domain that could benefit from such a system as the one described here.

5 Implemented Architecture

The current implementation is shown in Figure 37.

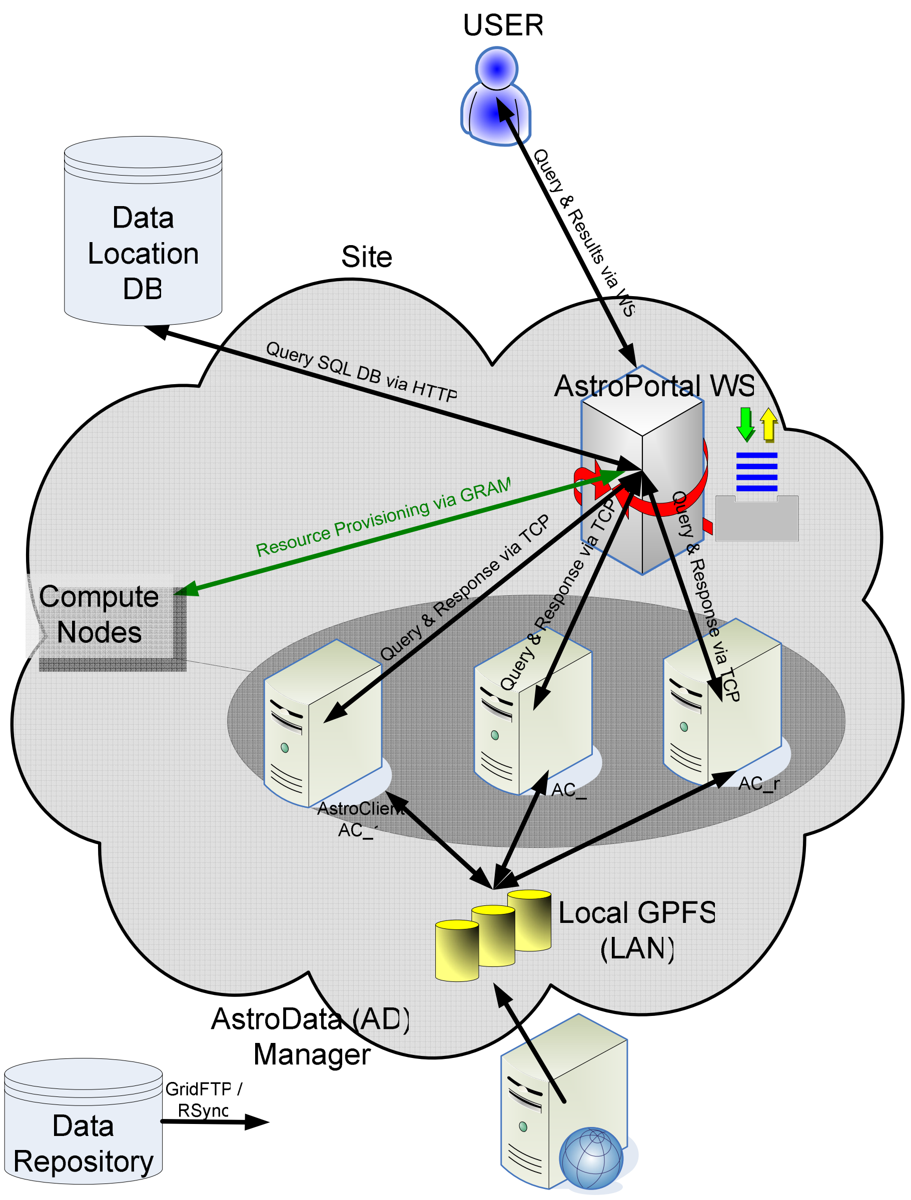


Figure 37: Implemented AstroPortal Architecture

The key missing functional components are:

- AP WS location via MDS Index: User would use the MDS Index to find the AP WS
- Storage Hierarchy:
    - L1: Local
    - L2: GPFS (LAN)
    - L3: GPFS (WAN)
- Automation of resource provisioning via GRAM

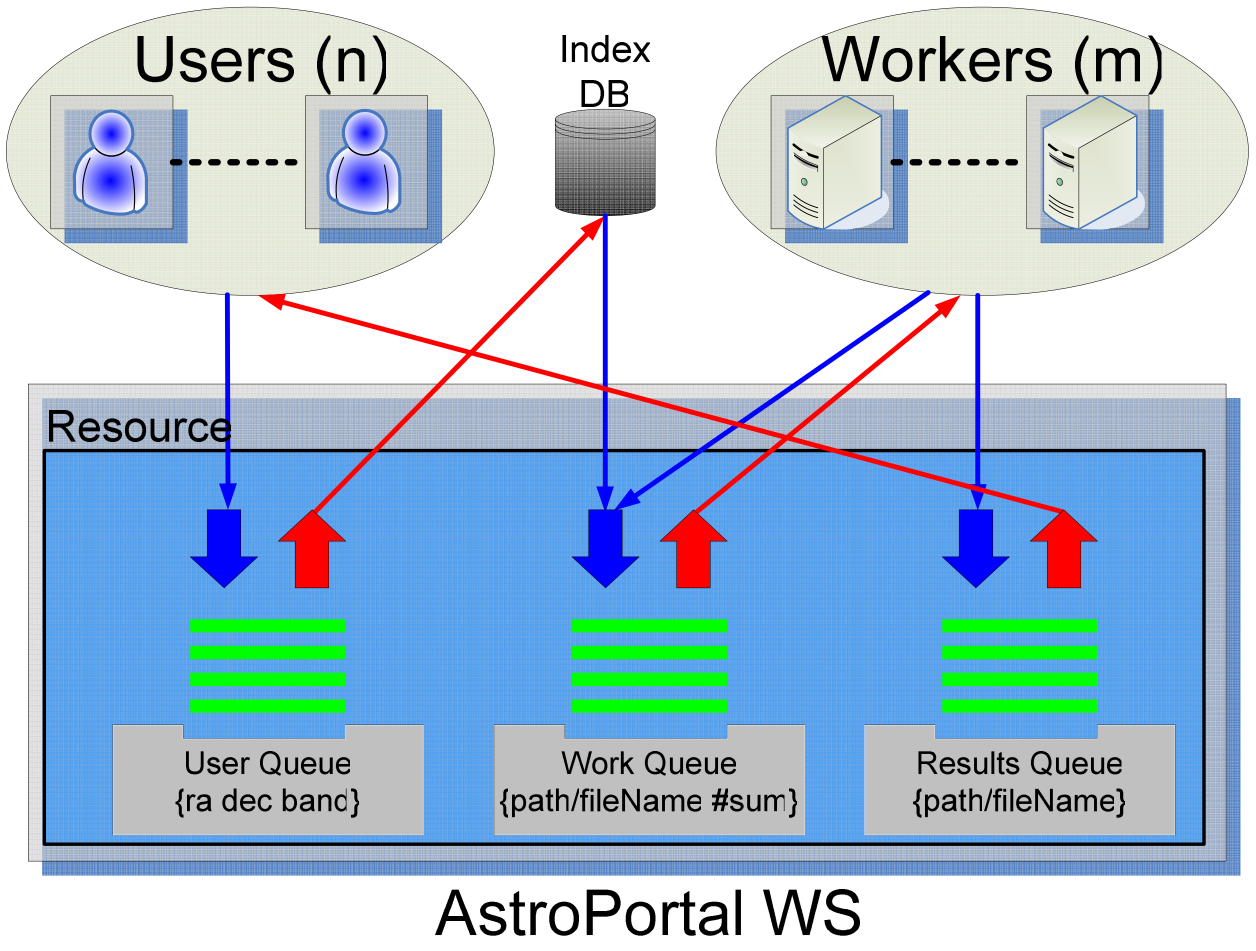


Figure 38: AstroPortal WS Implementation Overview

6 Bibliography

[1] JAVA FITS library: http://heasarc.gsfc.nasa.gov/docs/heasarc/fits/java/v0.9/

[2] WCSTools libraries: http://tdc-www.cfa.harvard.edu/software/wcstools/