Storage Access Coordination using CORBA

A. Sim, H. Nordberg , L. M. Bernardo, A. Shoshani, and D. Rotem
National Energy Research Scientific Computing Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Abstract

We describe our experience with using several CORBA products to interconnect the software modules of a fairly complex storage coordination system. In the application area of High Energy and Nuclear Physics (HENP) the volume of data reaches hundreds of terabytes per year, and therefore it is impractical to store them on disk systems. Rather they are stored on robotic tape systems that are managed by some mass storage system (MSS). The role of the Storage Access Coordination System (STACS) that we developed is to manage the caching of files from the MSS to a large disk cache that is shared by multiple HENP analysis programs. The system design involved multiple components developed by different people at different sites, and the modules could potentially be distributed as well. In this paper we describe the architecture and implementation of the system STACS, emphasizing the inter-module communication requirements. We describe the use of CORBA interfaces between system components, and our experience with using multi-threaded CORBA and moving large objects through the CORBA interfaces. STACS development was recently completed and is being incorporated in an operational environment scheduled to go on-line in the summer of 1999 [1].

1. Introduction

Many applications generate large volumes of data that need to be stored on mass storage system (typically robotic tape systems) because it is too expensive to store them on disks. Some examples are high energy and nuclear physics (HENP) experiments, climate modeling simulations, combustion modeling, and satellite data. In such applications, one of the difficulties in accessing the data stems from the need to retrieve different subsets of the data by hundreds of users at the same time. Typically, the data can be spread over many files and tapes, since the data are stored on tapes usually in the order they were generated. Another difficulty in accessing the data is that users are not interested in the API to the MSS, but rather refer to the object data structures and their attributes. We will describe in this paper the architecture of a Storage Access Coordination System (called STACS) that was built to support this application area. The system design involves multiple components developed by different people at different sites, and the communication between components needs clear interfaces to allow independent software development. STACS is designed to optimize the use of the limited resources, to simplify the interface between the client analysis codes and STACS and between STACS components, and to increase the performance of retrieving data stored on tapes. To achieve these design goals, a decision was made early on to have all the modules communicate through CORBA. We emphasize in this paper our experience of using different CORBA products and inter-ORB connection, of using CORBA multi-threading and moving large objects through the CORBA interfaces. Although in this paper we focus on High Energy and Nuclear Physics (HENP) application similar principles of architecture and the experience from using different CORBA products can be applied to other applications.

Next, we describe briefly the HENP application area as the background to the design and requirements of the system.

1.1. Background on High Energy and Nuclear Physics data

High energy and Nuclear Physics (HENP) experiments consist of accelerating sub-atomic particles to nearly the speed of light and forcing their collision. A small part of the particles collide and produce a large number of additional particles. Each such collision, called an "event" generates in the order of 1-10 MBs of raw data collected by a detector. The rate of data collected is a few event collisions per second, or about 10 MB/s on the average. This corresponds to 10^7-10^8 events per year, and

the total data volume amounts to about 300 TBs per year. This corresponds to 10,000 30 GBs tapes, which is the reason for the use of a robotic tape system. A typical experiment may run for 3 years. After the raw data is collected, they undergo a "reconstruction phase". Each event is analyzed to determine its sub-particles and to extract summary properties (such as the total energy of the event, momentum and number of particles of each type). The amount of data generated after the reconstruction phase is about a tenth of the raw data, which amounts to about 30 TBs per year. Most of the time only the reconstructed data are needed for analysis, but the raw data must still be available. Events are organized into files, normally about 1 GB each. Thus, each 1 GB file contains about 200-300 events. In [2] we have analyzed the optimal file size, and determined that a 1GB is a reasonable size. There are two conflicting considerations. If the file size is too big, too much unneeded data may be read. If the file size is too small, the overhead of accessing a large number of small files per query is too large.

A typical analysis that physicists wish to perform on the reconstructed data involves the selection of some subset of the events according to some conditions over the summary information (extracted in the reconstruction phase). This summary data is quite large. For about 100 properties and 10^8 events, the property space is about 40-80 GBs. Searching for the qualified events in the property space is another challenge. In [3], we have discussed building a specialized index to be able to search this space efficiently. Suffice here to point out that the events that qualify for a query may be spread over many files and tapes.

When hundreds of users execute different queries, the number of files to cache from the MSS (which in our case is HPSS – High Performance Storage System developed by IBM) can be very large; hundreds of files may be requested at the same time. Direct access to HPSS does not guarantee coordination of file caching or file sharing since no advance knowledge of the requested files for queries is used. Our goal was to take advantage of such knowledge and to develop an efficient way to coordinate the caching of files so that analysis programs share files in cache whenever possible, and all queries are treated fairly. We achieve this by providing a simple API to the STACS instead of having users access HPSS directly. Another problem we had to face is the inter-communication between the modules that required passing large amount of data through interfaces for large queries. This problem exists because queries that cover a large property range, the number of qualified events is very large (several 100,000s), and the event lists need to be passed between some of the modules. We have tested a few cases, and discuss them in the section on test runs. In a previous paper [3], we discussed an efficient way of coordinating the queries. Another major problem we had to face was the inter-communication between modules developed by different people on different sites. We will focus here on the problems of simplifying the interfaces, passing large amount of data, and our experience using different ORBs and inter-ORBs.

This paper is organized as follows: In section 2 we describe the system architecture we developed. In section 3 we describe results from the test runs, and discuss our experience with different ORBs and inter-ORB implementation. In section 4 we summarize the paper.

2. The Storage Access Coordination System Architecture

The architecture of STACS was designed to simplify the interface between client user codes from many different machines and platforms, and between the STACS components. Another aspect of the design is in the use of different ORBs between client user codes and STACS.

STACS has three main components [3] which interface with each other through CORBA (see Figure 1 below).

1) The Query Estimator (QE) uses its index to estimate the total number of events that will result from a query, the number of files involved, and how long it will take to process the query. The query estimation is passed to the HENP analysis component via the Query Object, so that it can decide whether to proceed with the query. The user may choose to abandon or modify the query if the number of files (and therefore its processing time) is too long. If he/she decides to proceed, the QE determines the list of files needed by the query and the set of events in each. This result, which can have thousands to millions of events, is passed to the Query Monitor (QM) component. This represents a large transfer across a CORBA interface, in the order of several megabytes. We also support another functionality: returning the entire event ID list to the HENP application. Again, this requires the transfer of several megabytes across the CORBA interface. In the QE, we have experimented with three different ORBs: Orbix [7], Orbacus [8] and TAO [9]. The HENP application always uses Orbacus, and the QM always uses Orbix.

2) The Query Monitor (QM) keeps track of what queries are executing at any time, what files are cached to disk for each query, what files are not in use but are still in cache, what files still need to be cached, and what files need to be purged to make more cache space available for other

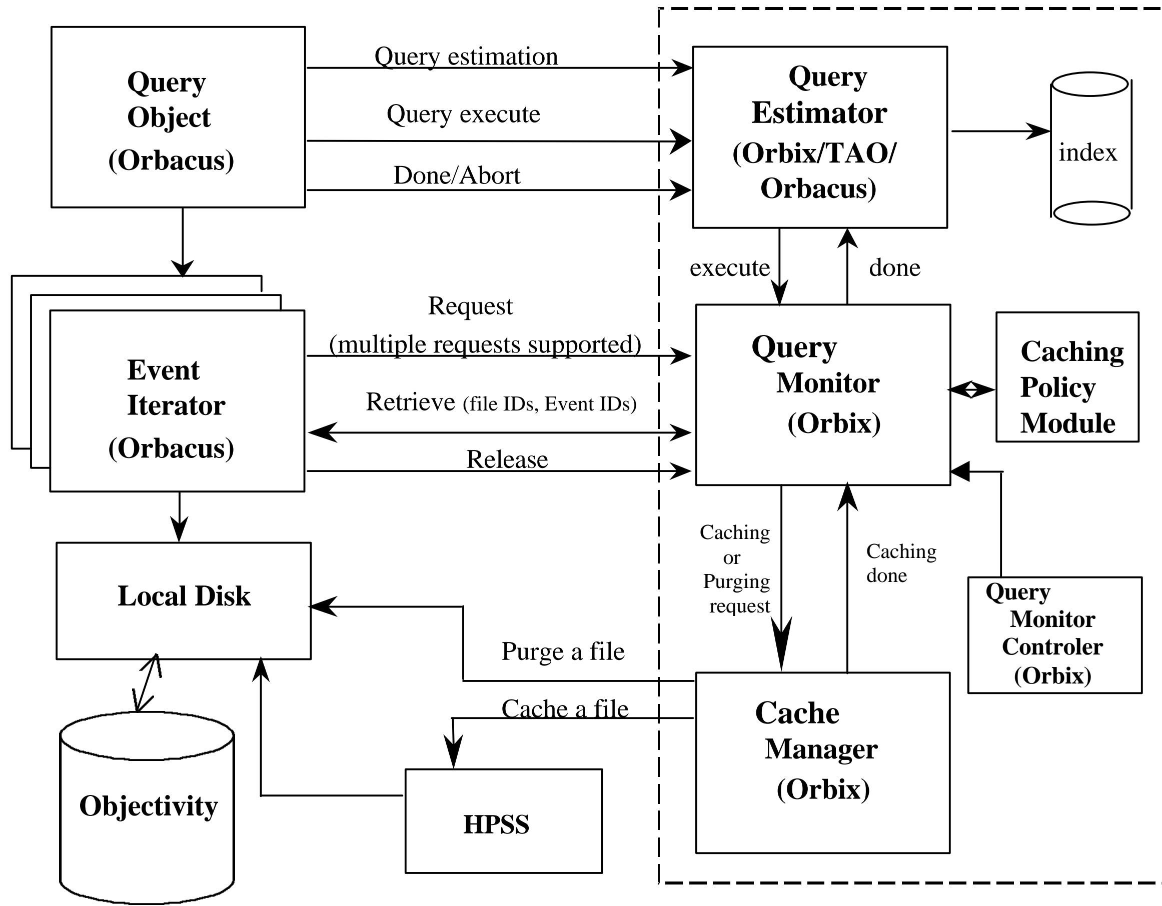

Figure 1. The architecture of STACS and its interaction with other system components

queries. The QM consults an additional module, the Caching Policy, that determines what files to cache next according to the policies selected by the system administrator. The QM performs the scheduling of file caching and the coordination between files needed by multiple queries. Because it deals with multiple query requests simultaneously, and with multiple file caching requests simultaneously it uses multi-threading extensively. The QM uses Orbix, but interfaces with the QE that uses Orbix/Orbacus/Tao, with the Query Object that use Orbacus, and the Cache Manager (CM) that uses Orbix.

3) The Cache Manager interfaces to HPSS to perform all the actions of staging files from HPSS to local cache and purging files from the local cache. The CM interfaces to the QM with Orbix. It, too, needs to use multi-threading.

These three components use multi-threaded ORBs [7, 8, 9] to communicate with each other, and each component can reside on a different machine. The reason that we chose multi-threaded ORBs within STACS is that each of these components has to deal with multiple requests concurrently. The reason for choosing Orbix [7] initially was its popularity, the product compatibility, and their support. We will discuss our experience with using different ORBs and inter-ORBs in the following section.

STACS also interfaces to other components of the HENP system with CORBA. While we only focus on STACS in this paper, it is important to understand how it interfaces to the rest of the HENP system. Figure 1 shows the HENP system components and the interaction between them, including STACS. On the right are the three components of STACS. On the left there are two components: 1) the Query Object, which is the component

that initiates the query, and 2) the Event Iterator, which is the component that passes the events one by one to the analysis program, and reads the events one by one from the file system. The client user codes interact with the Query Object and Event Iterators. Multiple Event Iterators are supported for parallel computing. Orbacus [11] (formerly OmniBroker) is used in these components. The user codes use Orbacus also.

The labels on the arrows show the type of messages sent between the components. We discuss below the CORBA interfaces used to process a query.

1) A query estimation request is sent from the Query Object (QO) to the Query Estimator (QE) through Orbacus-to-Orbix interface. The QO only passes the user ID and gets the query ID back. The QE uses its index to estimate the total number of events that will result from such a query, the number of files involved, and how long it will take to process this query. Even small queries involving several 10s of files may take hours to download from tape, so users are given the estimated processing time to decide if to proceed. Query estimation can also be used by the system to prevent users from proceeding with large queries if they do not have proper user permissions.
2) If the user decides to proceed, he/she issues an "execute" request to the QE with the query ID. The QE uses its index to generate the list of files that the query needs to access, as well as the set of event IDs for the events that qualify for the query.
3) The user code then starts an Event Iterator that issues one or more file requests to the Query Monitor through Orbacus-to-Orbix interface. Multiple Event Iterators can be started for parallel analysis.
4) The QE passes the (file IDs, event IDs) request in a CORBA sequence form to the Query Monitor (QM) through an Orbix-to-Orbix interface. The set of (file IDs, event IDs) can be as big as a few 10s of MBs. Passing these large data through CORBA was tested and will be discussed in the following section.
5) The QM adds the query to its query queue. When it is the query's turn to be serviced, the QM consults the Policy Module as to which file to give the query next.
6) The QM checks what files are in the cache. If a file for the query is found in the cache, it is locked into the cache. If no file is found in cache, the Policy Module selects a file to be cached from tape according to the policy that the STACS administrator has set. If necessary, it also selects which unlocked and unused files to remove from cache to make more cache space available for the new file. Note that the events needed by a given query may be spread over many files and tapes. In such a case, multiple files need to be cached.
7) If a file has to be read from tape, the QM requests the Cache Manager (CM) to cache that file through an Orbix-to-Orbix interface, passing the file ID. The CM requests the caching of the file from the HPSS (currently via parallel FTP), and monitors its progress. When the file is cached, the QM is notified.
8) The QM passes to the Event Iterator the set of file IDs along with the set of event IDs that qualified in each file for the query in CORBA sequences through an Orbix-to-Orbacus interface. This interface passes relatively small chunks of data.
9) When the Event Iterator finishes processing all the events in a file, it issues a "release" message to the QM through an Orbacus-to-Orbix interface. The QM then makes the query "active" and processes it for the next file in its turn.
10) When all the requested files by a query are processed, the QM notifies the Event Iterator that there are no more files, through an Orbix-to-Orbacus interface. This causes the Query Object to terminate the query, by issuing the "done" message to the QE through an Orbacus-to-Orbix interface. The QM also notifies the QE that the query is finished. The whole process can also be stopped with an "abort" message from the Query Object.

3. Test Runs and Our Experiences

3.1 Experience with ORBs

The system has been tested in a real environment. While running the tests, we log various events, such as when a request is made for caching, and when a file is passed to the analysis code. By plotting the behaviors of different tests we could confirm the correct behavior of STACS. We managed CORBA interfaces between different ORBs (Orbix, Orbacus and TAO), between different machines and between different platforms (Solaris and Linux, Solaris and Windows NT).

Another important aspect that we have tested was passing very large objects sets with CORBA. We have passed a set of file IDs and a set of one million event IDs in a CORBA sequence of structs between the STACS components (approx. 10 Mbytes), and the CORBA interfaces handled it very well. However, it slowed down the query execution process for that query. To check if one large query affected the execution of smaller queries, we tested running the same big query with two other small queries. STACS with multi-threaded CORBA interfaces handled the execution very well without having much delays on smaller queries because of the larger query.

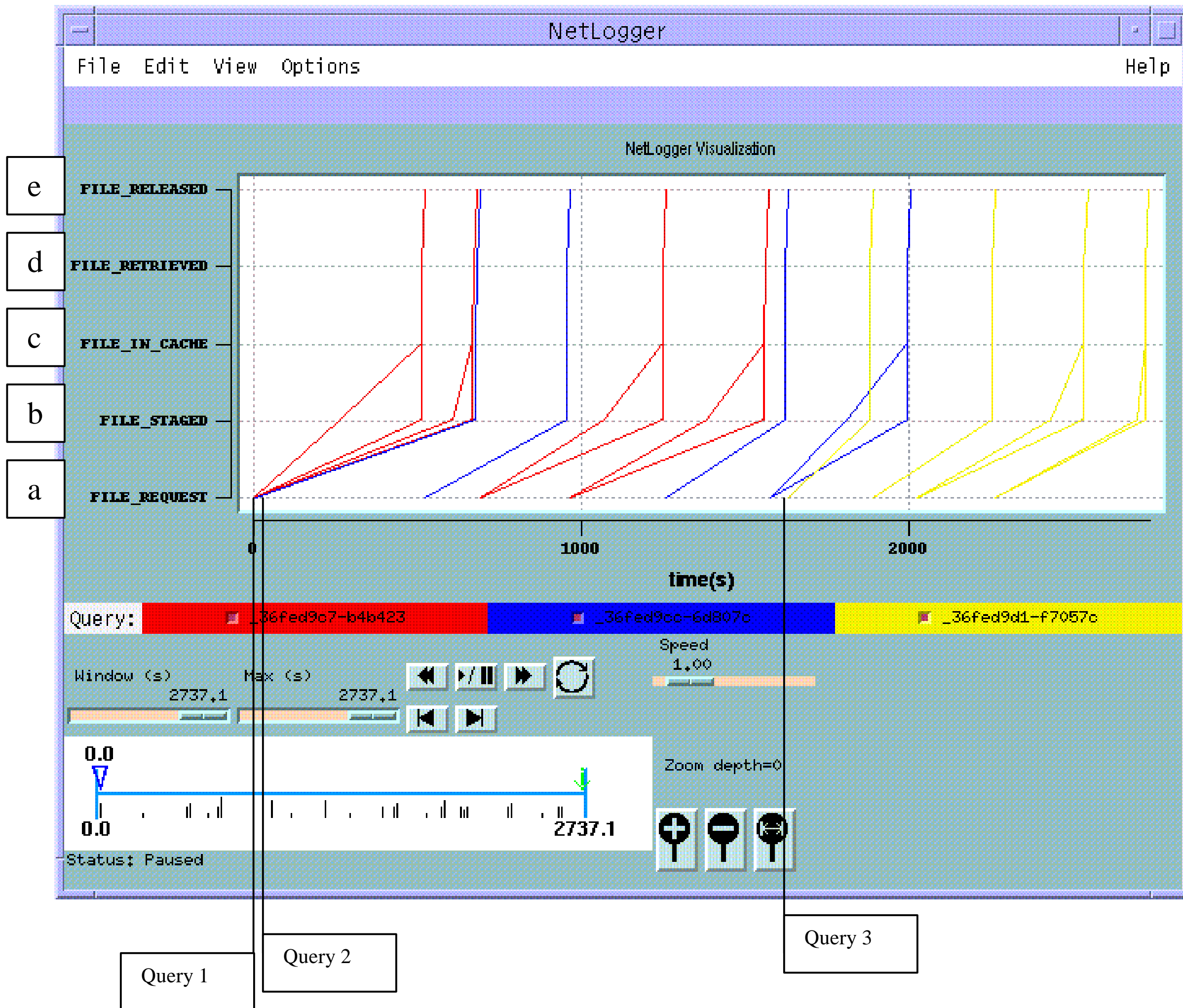

Figure 2. Display of a test run showing file sharing in cache

Our experience with mixed ORBs is that ORBs seem to talk to each other flawlessly. However, we could benefit from ORBs that support a daemon feature, in order to establish the CORBA connections easily, and to perform crash recovery by restarting the crashed module and re-establishing the connections. Unfortunately, the Orbix daemon (for Orbix 2.3c MT) crashed when we had many inter-ORB connections. We have worked around the problem of re-establishing connections with a transient port method without the Orbix daemon, and have the client user codes connect to STACS directly. However, we are still trying to solve the problem of restarting crashed modules without an ORB daemon.

We have tested Orbacus and TAO. They come with source code and are free. (Orbacus is free for non-commercial use.) The reason that we tested TAO is because of its reputation. TAO, indeed, seems very efficient, but we have still to do rigorous testing. TAO and Orbacus support several threading models. It is very easy to switch between models (thread per client, thread per request, single threaded, thread pool). These models are also possible with Orbix, but the programmer needs to do the coding himself (by adding Orbix specific calls). It was relatively easy to write codes in such a way that switching ORBs is a matter of recompiling with different compilation flags. Switching between ORBs while developing, is useful because the user must follow the standard CORBA specifications. When the standard CORBA calls are followed, it is easy to switch ORBs if the need arises. One can be forced to switch ORBs due to budget constraints or because part or all of the developed code needs to be run with a framework that does not work with a certain ORB.

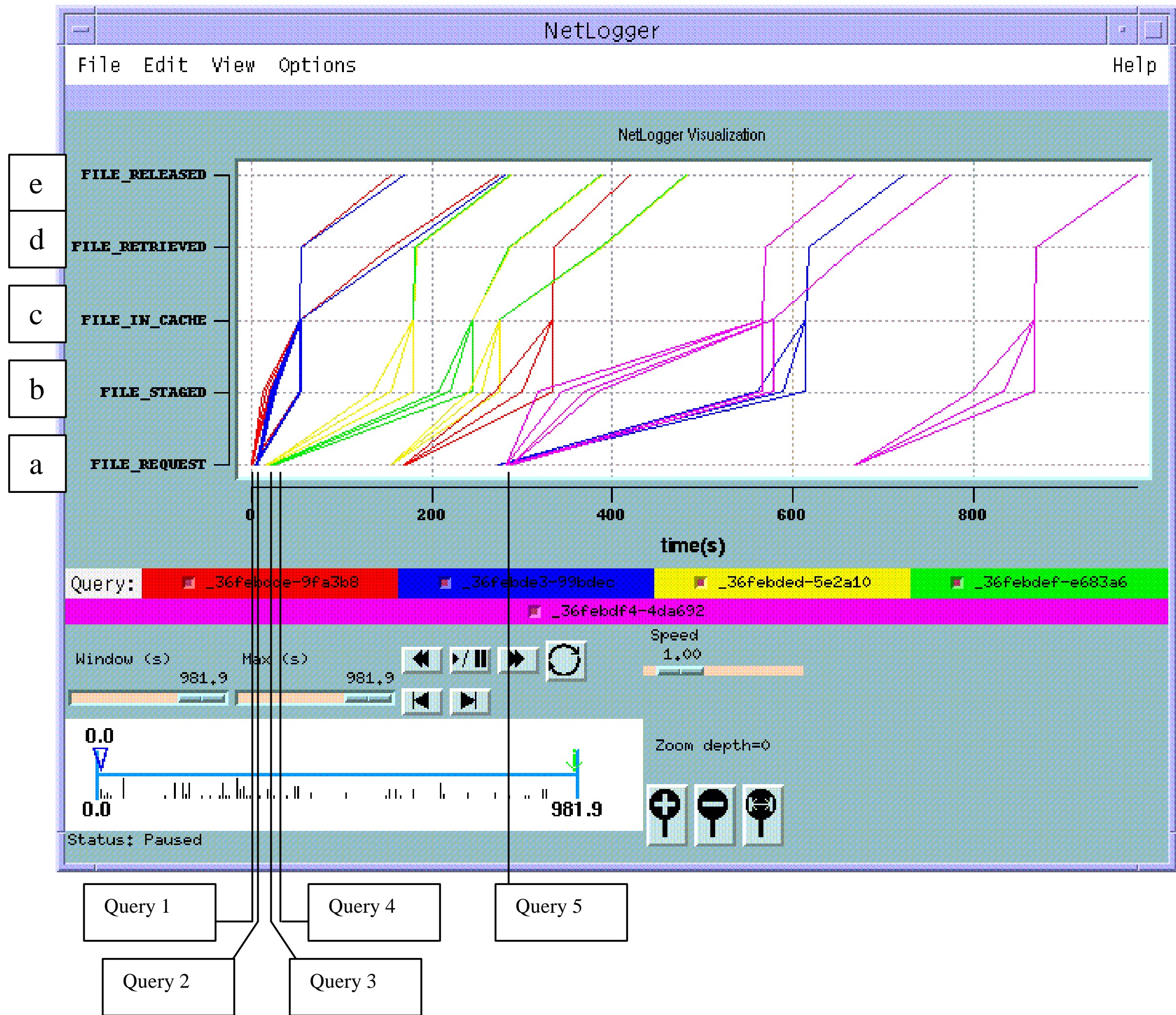

Figure 3. Display of a test run with 5 client user codes without over-lapping files

3.2 Test runs

Figure 2, figure 3, and figure 4 show graphs drawn from the actual logging of the test runs. A visualization tool, NetLogger [4], developed at LBNL was used to show the progression of events in real time.

The graph shown in Figure 2 illustrates the value of file sharing in a very limited cache which can hold only four files at one time. We started with an empty cache. We ran three queries: each query needs 8 large files from tape to be cached from tape to disk. The entire test ran for about 50 minutes.

The graph represents the occurrence of logged events over time (the x-axis), but spreads out various logged events in the y-axis. In this graph, five logged events are shown from bottom to top: a) caching request arrived to HPSS, b) stage finished, d) file pushed (i.e. file is available for the Event Iterator), e) file retrieved (by the Event Iterator), and f) file released. We note that the time between a) and b) is the time to get the file from tape to local cache. The time between b) and c) is the time to pass the file to the EI queue after it was cached. This should normally be done immediately. The time between c) and d) is the time between making a file available to an EI and the time the EI actually reads it. The time between d) and e) is the time that it took the user code to process all the events from the files. Thus, a vertically connected line represents a single file and chronicles how it was processed.

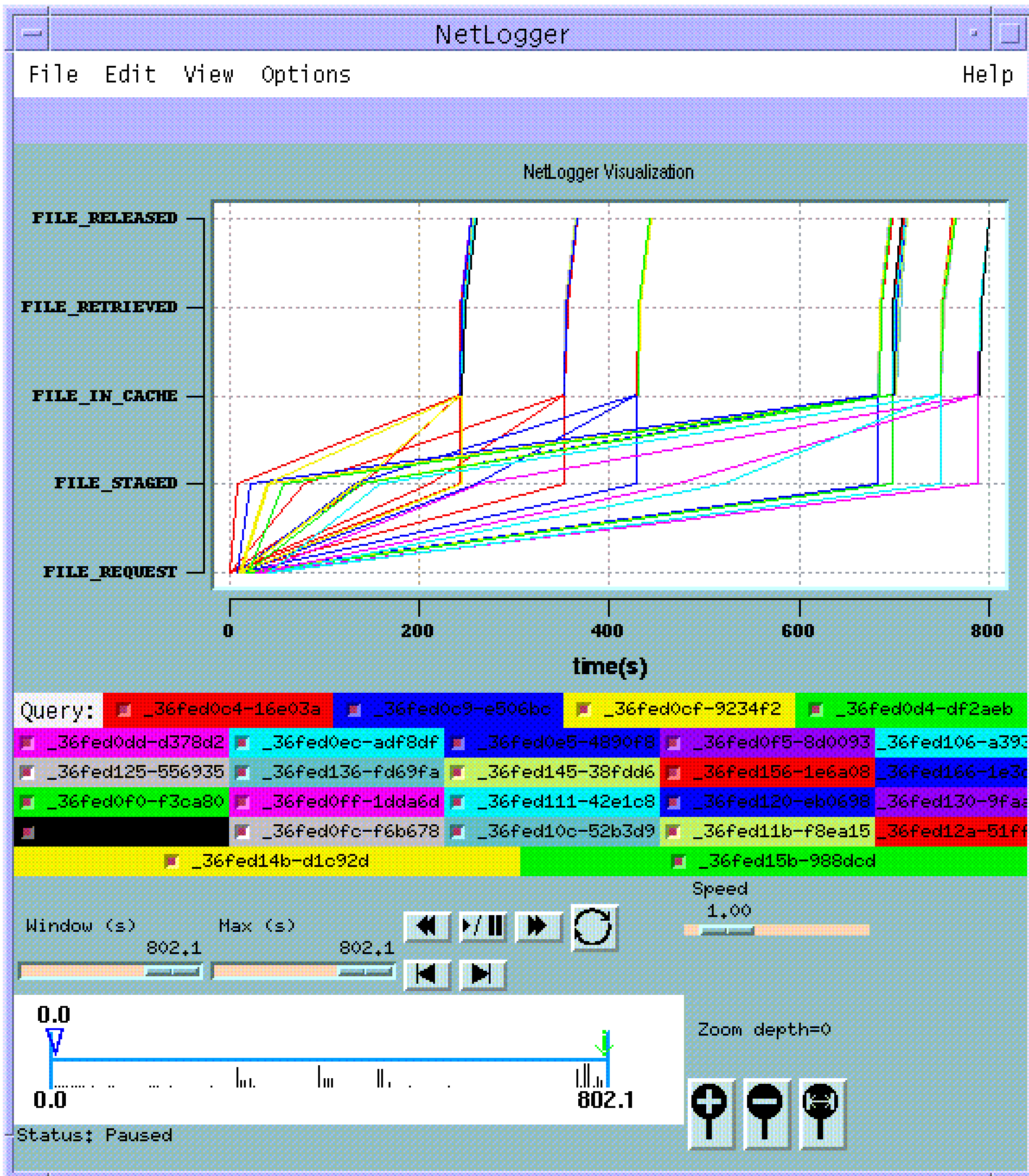

Figure 4. Display of a test run with 26 client user codes showing file sharing in cache

Figures 2, 3, and 4 show increasing number of queries processed in parallel. In figure 2, we had 3 parallel queries, in figure 3 we had 5 parallel queries, and in figure 4 we ran 39 parallel queries. We explain briefly below each figure. Figure 2 shows that when Query 1 and Query 2 started at the same time, four file requests were made, two for each query. The four files took several minutes to cache, and two files at a time were passed to the queries for processing. Because the needed data were spread over files, those two files had to be passed together to the EI. The files were processed for a few seconds. We intentionally made the processing time short to make the caching time dominate the total time. Only when each query finished and released the files, a new request was made, reflecting the pre- fetching policy. This process continued and files in cache are shared with other queries whenever possible.

This test validated the correctness of the performance of the software, as well as the ability of the multi-threaded CORBA calls to handle the multiple requests properly.

Figure 3 shows a test with five queries launched simultaneously, each requesting non-overlapping files. We used a limited cache space, which is the reason that Query 5 was postponed until cache space became available. Figure 3 also shows how STACS coordinates the queries to optimize the limited system resources so that no one query can dominate the cache and HPSS.

Another test we ran was with 39 queries running all at the same time while sharing cached files, and the test ran very well. This test is shown in figure 4. This test took only about 800 seconds, because we designed the test so that all these queries shared nine files. If those queries contacted HPSS individually, and cached files for their analysis, it would have taken hours. Note that some files took a long time to be cached because file caching requests queued. This test shows the robustness of STACS, as well as the benefits of coordination by STACS.

4. Summary

In this paper, we presented the architecture of the Storage Access Coordination System. We found that CORBA simplifies the interfaces between the user code and STACS and between the STACS components. CORBA also simplifies the communication between different machines and between different platforms. CORBA IDL was used to define the interfaces clearly between the system components to allow independent code development by different people at different sites.

The mixed-ORB between the user codes and STACS worked well. Using only the standard features of ORBs permits users to select an ORB that they are already familiar with.

Passing large amounts of data through CORBA connections was tested, and worked well. This experience gave us the confidence to continue to scale up to even larger number of users and larger number of files requested per query.

One missing part in our experience is the ability to recover from crashes of system components, using an ORB daemon. Since Orbacus and TAO did not have a daemon feature at the time of the system development, we wanted to take advantage of this feature in Orbix. However, so far we are not able to use the Orbix daemon feature as it crashes often. If this feature is stable, it would be very useful for crash recovery. This recovery issue is currently under development.

Acknowledgement

This project is funded by the Grand Challenge program at the Department of Energy, in the Office of Energy Research, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

We thank all the collaborators of this project who provided the realistic requirements for the storage management component of this project. Special thanks are due to Dave Malon, Jeff Porter and Dave Zimmerman who developed and implemented the other non-STACS components of the system.

References

[1] STAR Computing Software, http://www.rhic.bnl.gov/STAR/html/star_computing.html

[2] L. Bernardo, H. Nordberg, D. Rotem, and A. Shoshani, Determining the Optimal File Size on Tertiary Storage Systems Based on the Distribution of Query Sizes, Tenth International Conference on Scientific and Statistical Database Management, 1998. (http://www.lbl.gov/~arie/papers/file.size.ssdbm.ps).

[3] A. Shoshani, L. Bernardo, H. Nordberg, D. Rotem, and A. Sim, Multidimensional Indexing and Query Coordination for Tertiary Storage Management, the 11th International Conference on Scientific and Statistical Database Management, 1999.

[4] NetLogger: A Methodology for Monitoring and Analysis of Distributed Systems, http://www-itg.lbl.gov/DPSS/logging/

[5] R. Orfali, D. Harkey, J. Edwards, *Instant CORBA*, (John Wiley & Sons), 1997

[6] Sean Baker, *CORBA Distributed Objects using Orbix*, (Addison-Wesley), 1997

[7] IONA Technology, Orbix 2.3c MT, http://www.iona.com

[8] Object Oriented Concepts, Inc., Orbacus 3.1.2, http://www.ooc.com

[9] D. C. Schmidt, The ACE ORB (TAO), http://www.cs.wustl.edu/~schmidt/TAO.html