55-82
53187

A Global Distributed Storage Architecture

Dr. Nemo M. Lionikis
Michael F. Shields
Department of Defense
9800 Savage Road
Fort Meade, MD 20755
nemo@romulus.ncsc.mil
mfs@romulus.ncsc.mil
Tel: 301-688-9509
Fax: 301-688-9599

I. Introduction

NSA architects and planners have come to realize that to gain the maximum benefit from, and keep pace with, emerging technologies, we must move to a radically different computing architecture. The compute complex of the future will be a distributed heterogeneous environment, where, to a much greater extent than today, network-based services are invoked to obtain resources. Among the rewards of implementing the services-based view are that it insulates the user from much of the complexity of our multi-platform, networked, computer and storage environment and hides its diverse underlying implementation details. In this paper, we will describe one of the fundamental services being built in our envisioned infrastructure; a global, distributed archive with near-real-time access characteristics. Our approach for adapting mass storage services to this infrastructure will become clear as the service is discussed.

II. High Level Architecture

As a world-wide organization, NSA's storage and retrieval services must provide for rapid, efficient, and user-driven data access from any node in our organization. Storage services must be accessible yet secure, scalable, reliable, cost effective, and manageable. The technologies used to implement storage must be commercial-off-the-shelf (COTS) wherever possible and the user interface to these services must be clear and simple. Moreover, a key requirement of the services is that they must support the notion of near-real-time access to data.

Because traditional file-based solutions, with their induced latency, are inadequate to meet the near real-time processing requirements being levied today by our users, we are developing the Byte Stream Storage and Transfer Service. The user sees the Byte Stream Storage and Transfer Service as a globally distributed archive with near-real-time access. The service is intended as a mechanism that allows a user to access and manipulate data streams. It is a critical feature of the stream service that while a producer is creating a stream at one location, a consumer, possibly at a geographically remote location, can

begin to access the producer's data. One of our design goals is that no matter where users are located, a consumer can begin accessing data within seconds of its creation.

One of the most radical aspects of the proposed stream service is the assumption of all storage management by the service. There is no concept of an "archived" stream. Once data has been written into the service, the user has one, and only one, view of it. The user sees "a stream", not "a local disk copy" or "an archived copy", each with its own interface involving different commands and even operator intervention to gain access. No knowledge of data location is required on the part of a user. No special commands to access storage are required. No special commands to transfer data to the processing system are required. No thought, beyond initial system configuration, is given to availability of space. No application code is required to handle file boundaries and file names for a stream of data. These mechanisms are created once and for all in the service and then applied consistently to every stream. Users must only know the name of the stream they wish to access and the service will find and deliver the data.

A user-level application that processes live, non-burst signals should be able to work with a data abstraction that models the "stream-oriented" nature of these signals. The notion of a Byte Stream Storage and Transfer Service was devised to support such a data abstraction. A by-product of adopting the stream data abstraction is that it supports the notion of near-real-time processing of live data quite naturally. When moving a byte stream, we do not assume that the entire stream is present or, in fact, that the entire stream even exists yet. We cannot think in terms of transferring the entire stream to a specified host and processing it. Rather, we are constantly transferring bytes of the stream as they are created. The concept of a service that moves and stores streams is not *apriori* necessary, but its advantages are huge. One cannot overstate the value of a single, universally accepted abstraction for a byte stream, captured in a stream service. Not having such a service requires producing distinct, possibly incompatible, file-based solutions for each new production data flow, with all of the attendant naming, storage, movement, administration, accounting, and maintenance issues that the new solutions would demand.

Internally, the byte-stream service is a set of geographically distributed relay/storage hubs (Figure 1), that cooperate with each other and with interface software running within a stream consumer or producer process, to accomplish the movement and storage of data. The hubs are connected via a network and control software within the hubs communicates via standard protocols (TCP/IP or UDP/IP). Hubs are logical entities that may consist of several systems. A stream might reside within a single hub or be distributed among multiple hubs. Multiple copies of pieces of the stream may exist in different hubs. A consumer will receive a copy from the nearest hub. There are no coherency issues because a stream can be written only once (archive semantics). There are, however, issues of deleting extra copies when they become old or inactive and the

stream service must institute policies to manage this. In general, one copy will be labeled for retention and all other copies will be considered cached and can be deleted.

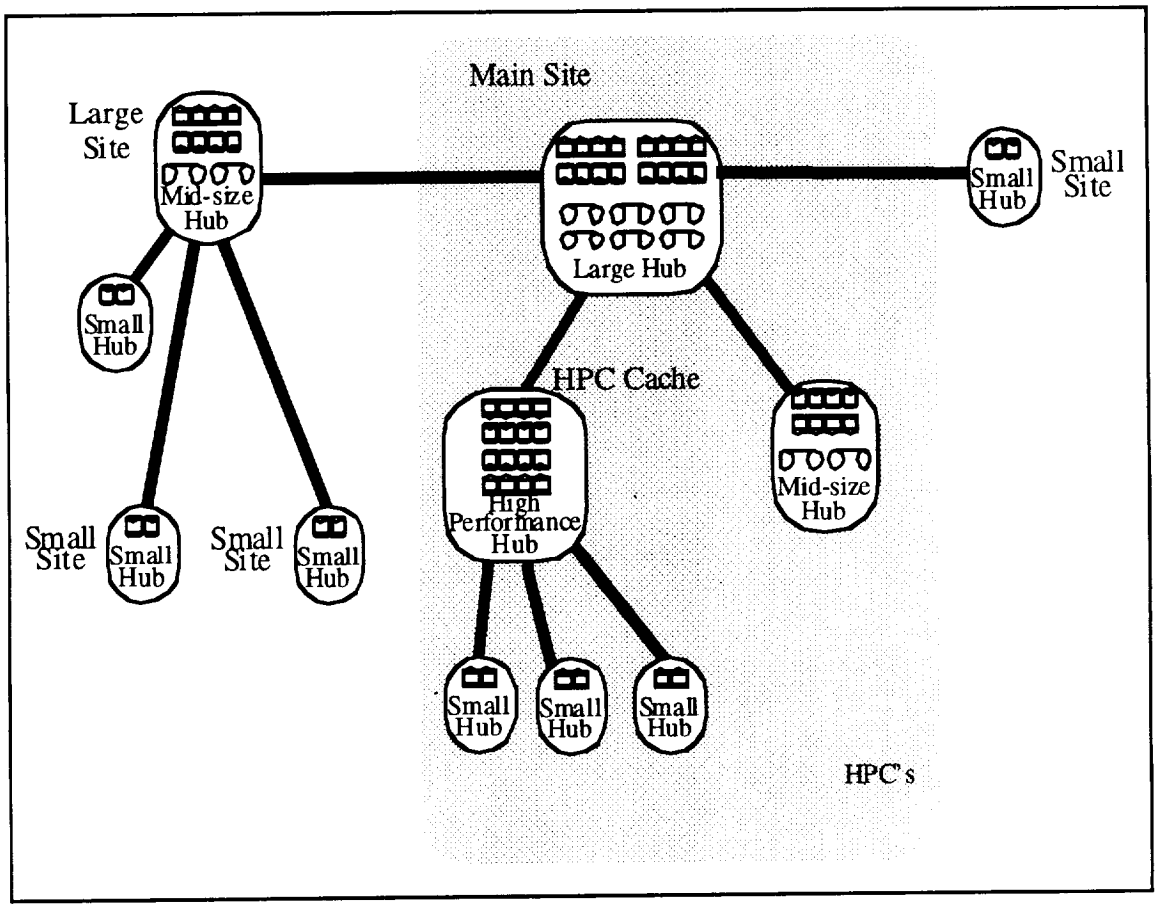

Figure 1: High Level Architecture of the Global Archive

The stream service interface will mimic the POSIX system call I/O interface, with common UNIX extension, using the C-language binding. There are at least two compelling reasons for doing this. First, the POSIX system call interface has been used in countless settings and has proven its versatility. It is safe to assume that the interface will support both current and future requirements. Second, developers are already familiar with the interface, so, learning to use the stream service should not be an onerous task. Having said this, it must be pointed out that the stream service interface will be a mixture of file and network semantics. This is because it is desirable to allow a developer to use such calls as creat(), open(), close(), read(), write(), and lseek() from the file domain. It is also necessary, though, to provide the capabilities of select() from the network domain in order to support the abstraction of a near-real-time stream. The actual interface will consist of a library of subroutines containing at least yopen(), ycreat(), yclose(), yread(), ywrite(), ylseek(), and yselect().

Figures 2 and 3 provide simple examples of how these routines can be used to read or write a stream. To read, a user application will open a stream, referring to it by a name. The application then seeks to the position of interest and repeatedly reads and processes data. When done, the application will close the stream. Writing a stream will be a similar sequence of calls (open, repeated writes, and close). Both of the code segments are extremely simple and dramatically illustrate the virtues of the stream service. Note that there is no reference to location, no concern about file boundary conditions, no concern about storage. There is also no notion of whether the data is being obtained from storage or from a live source. The program only requires a name to access the stream. All of the general problems of movement and storage are handled transparently. It should be noted, again, that one assumption of the stream service is that, once created, a stream cannot be edited. In order to modify a stream, it must be read, processed, and a new stream created for the resultant output.

```
stream_id           sid;
char*               buffer[MAX];
int                 bytes_read;

sid = yopen("stream_name", YO_RDONLY);
ylseek(sid, POSITION, YSEEK_SET);
while ( (bytes_read = yread(sid, buffer, MAX)) != ERROR) {
        /* Process the bytes read from the stream  .....          */
}
yclose(sid)
```

Figure 2: Reading a byte stream

```
stream_id           sid;
char*               buffer[MAX];
int                 bytes_written;

sid = yopen("stream_name", YO_CREAT | YO_WRONLY);
while ( NOTDONE ) {
        /* Get data and perform processing .....              */
        bytes_written = ywrite(sid, buffer, MAX);
}
yclose(sid)
```

Figure 3: Creating and writing a byte stream

When a user process wishes to write a stream, it begins by calling yopen. Internally, the service interface software establishes communications with its local hub (Figure 4). When writing begins, an agent is started on the hub and is connected to the interface software in the user process. Data then passes through the interface to the agent on the hub which caches the data on disk. As the cache fills, data may be moved by the service to storage systems within the hub for short-term retention. At this point a stream (potentially, but not necessarily, live) is being captured and stored. Note that storage is not a direct concern of the user process.

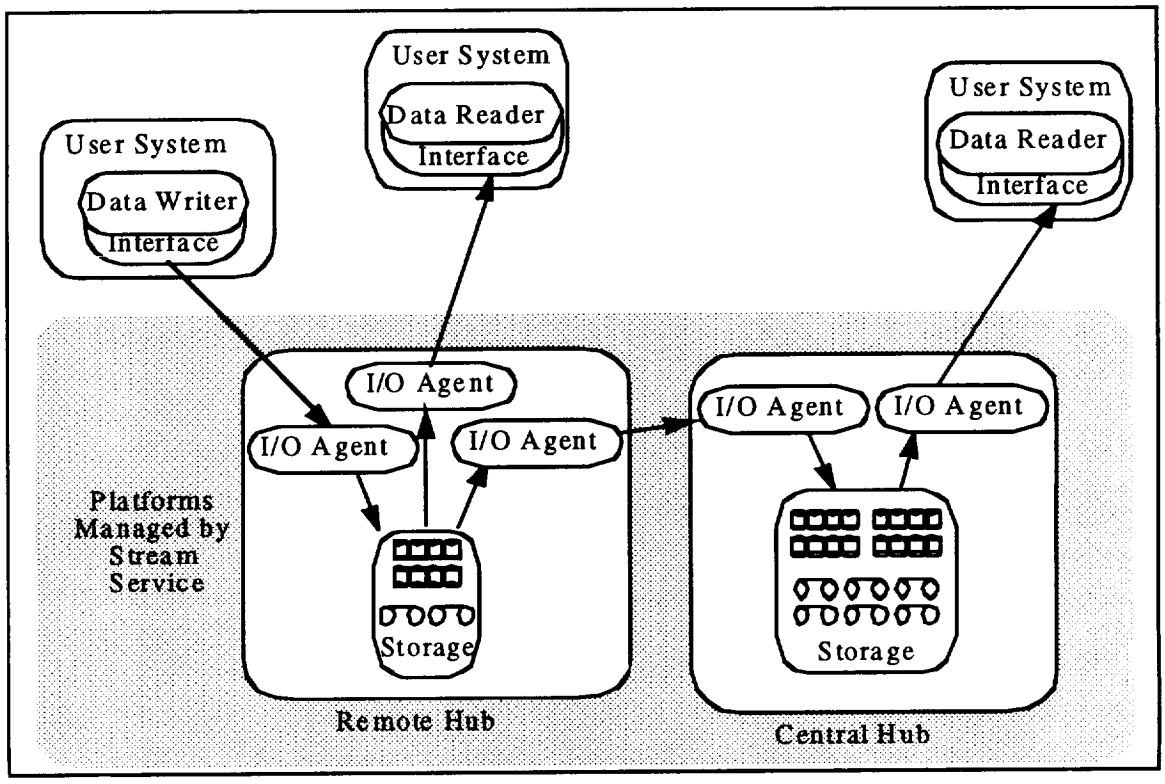

Figure 4: Data Flow Examples

When a user process wishes to read a stream, it begins by calling yopen, and, once again, the service interface software establishes communication with its local hub. When yread is called, the local hub determines if the desired data is present. If not, the hub finds a remote hub that has the desired data, and requests a transfer from the remote hub to the local hub. Now, with data present in the local hub, a connection is established from an agent in the hub to the interface software in the user process, and data is forwarded. As the data arrives from a remote hub, it is cached in the local hub. As the cache fills, data moves to a storage system for retention. Note that the reading process may or may not be receiving live data, and is unaware and unconcerned as to whether the data originated at a local or a remote location. In fact, all storage details are hidden from the user.

A near-real-time flow is established when a stream is being produced at the same time that it is being consumed. Should a communication outage occur between hubs, data will not be lost because the hub that is local to the stream producer will continue to cache and store data. Of course, a network outage between the producing system and its local hub will cause a data loss if the buffering capability of the producing system is exceeded. Consumers and producers can run on the storage platform. In this case, the network will be circumvented and we expect to observe reading and writing at very near disk speeds.

One of the great advantages of the service described here is that accessing stored data is exactly the same as accessing live data. It is the responsibility of the local hub to discover where pieces of a stream are stored. If a stream has been moved from cache to storage, the hub will ensure that it is drawn into cache again with forwarding identical to the near-real-time case.

It is common to associate related information with a byte stream. A follow-on development, the Annotated Stream Service, built on top of the Byte Stream Storage and Transfer Service, supports this notion. An annotated stream consists of several byte streams, one being a data stream and the remainder being annotation streams. The Annotated Stream Service provides a mechanism for a stream writer to associate annotations with specific points in a data stream. For a stream reader, the service synchronizes the reading of the annotations with the reading of the data. The internal form of an annotation is chosen by the application developer. The service merely provides a framework for the association, storage, and synchronized delivery of the data and the annotations. As with the byte stream service, all of this is done while still preserving a simple "open, close, read, write" interface.

III. Storage Strategy

Guiding Principles: NSA has adopted a COTS, to the maximum extent possible, approach to any Mass Storage requirement. As a direct result of this policy, we have carefully approached the global distributed storage architecture steered by previous work in developing a scalable set of disk and tape components, subsystems and systems matched to specific requirements. Significant consideration is given to performance, functionality, and cost, with a keen eye on system level reliability. To the maximum extent possible, we strive to achieve vendor independence and network connectivity; wherever possible, we desire data sharing and products which facilitate technology insertion. Finally, remote monitoring is key to overall system viability.

Product Considerations:

Disk Storage: For both large and small nodes the disk subsystems are almost always specified to be RAID devices. While the majority of the current set of disk subsystems are SCSI-2 F/W, our high-end nodes will require fibrechannel speeds. The ability to remote the arrays beyond today's cable limits greatly enhances our physical layout potential. In addition, the ability to connect large arrays to multiple servers enhances our reliability, shareability, and control. NSA has relied heavily on shared network disk arrays within our supercomputer complexes and has urged industry to develop products of this class. To achieve the desired performance and flexibility for the individual nodes of this architecture, extremely large network RAID arrays are a must. The disk arrays must be platform independent, reducing reliance on any single vendor.

Robotic Tape Storage: Our larger nodes require robotic tape libraries which range from tens of terabytes up to multiple petabytes. They are sized to match specific user needs from a performance, capacity, and user access perspective. We envision each node to have multiple tape libraries, matched to the specific type of stream data. Our goal is to make our distributed library infrastructure transparent to the user. While certain data types lend themselves to very large capacity libraries, others do not. As such, our experience with the current set of storage management software offerings forces us to adopt a multiple library strategy. The majority of today's products use commercial relational database management system (RDBMS) products to manage the files stored in the libraries and this artifact must be accommodated in the overall architecture. Most of the products evaluated to date are limited to the tens of millions of files. Large files (>150 MB) are ideally suited to high performance helical drives which can deliver petabyte class individual libraries. However, small file (15 MB) mass storage libraries will outpace the RDBMS' ability to scale to the 100 million file mark. Because the stream service controls file creation, large files should be the norm. While these numbers are not exact for today's storage software offerings, they are representative of the challenge that system architects face in designing a multi-node hub. The vendor community can deliver hardware that easily scales into the multiple petabyte range today; however, the storage software lacks the maturity, performance, and ability to service this class of system. Although the smaller nodes are disk only, they will still require high-performance robotic tape backup systems.

Storage Software: There are two common cross vendor categories of storage software in wide use today at NSA, Hierarchical Storage Management (HSM) and Virtual File System (VFS) software. Of the two, the latter is most widely used. HSMs classically are major computer systems (processors, disk, robotic tape) that are network connected to multiple client systems. Both VFS and HSM are primarily skewed towards the operational paradigm of store with infrequent retrieves. While performance is dependent on multiple factors and is highly dependent upon the network connectivity, VFS systems generally deliver higher performance than HSMs. VFSs today use large UNIX servers with RAID arrays and manage 7-40 TB robotic tape libraries. They too are network

connected to multiple client systems, but do not possess the full range of archive functionality of the HSM. However, they interact with almost any client and provide a file system view to that client; hence they are very easy to install and are widely used by a diverse population due to their simple interface. To support the distributed storage architecture, our large node will be based on multiple VFS storage systems. Emerging multimedia software products easily embrace this technology which further enhances its role in our infrastructure. Finally, the multiple library approach facilitates technology insertion for the physical components that make up the storage library allowing for the migration of data to be performed as a background job as older drive technology is retired.

MetaData: The most difficult element of the storage system is the metadata system. With multiple, disparate libraries connected to the large node, and several nodes in a hub within the archive, transparent access by a diverse population is facilitated by this critical element. Its importance has been recognized by the Mass Storage Community as evidenced by the IEEE sponsoring a yearly MetaData Conference. The ability to manage hundreds of millions to billions of files can only be done by a carefully designed metadata system. NSA has taken the approach of a distributed metadata system for its scientific processing complex; however, to scale to the numbers of files needed for the future, significant breakthroughs are needed. Suffice it to say, that the integration and use of metadata and its storage will need to be accomplished. Scalability here is fundamental to the success of this endeavor. This paper will not address metadata.

IV. Initial Development Plans

The architecture discussed above will be implemented incrementally. The intent of the initial configuration is to present users with the first view of the stream service/ distributed archive and validate the concepts contained in the architecture.

Near Term Plans: Initially, a single-system, mid-sized hub will be built. The hub will employ a medium performance UNIX server with tens to hundreds of gigabytes of disk cache and a single robotic tape library. The storage software will be Virtual File System based. The hub will run early increments of the stream service software and will be used to validate many of the concepts of the architecture. After the first hub has been built and tested, a second, large, two-system hub will be built. The hub will consist of two identical high-end UNIX servers, each with a large size RAID disk array and one or more robotic tape libraries. Both high-end and medium performance/capacity libraries may be employed and, again, the storage software will be Virtual File System based. The systems will have multiple network connections of differing performance levels (FDDI, ATM, and Ethernet). The two-system hub will run a follow-on increment of the stream service that manages the multiple storage platforms. Inter-hub data transfers will be based on a static policy. A mix of user workstations which mirror the current

infrastructure will complete the near term test configuration (Figure 5). Using this configuration, we intend to evaluate the user interface, desired functionality, initial scalability, overall reliability, as well as subsystem, software, and system reliability. We will focus on the adequacy of the specific technologies chosen, calibrate performance choke points and scalability considerations. As a result of our analyses, the overall architecture will be modified, if necessary, and the lessons learned will be incoporated into our long term plans.

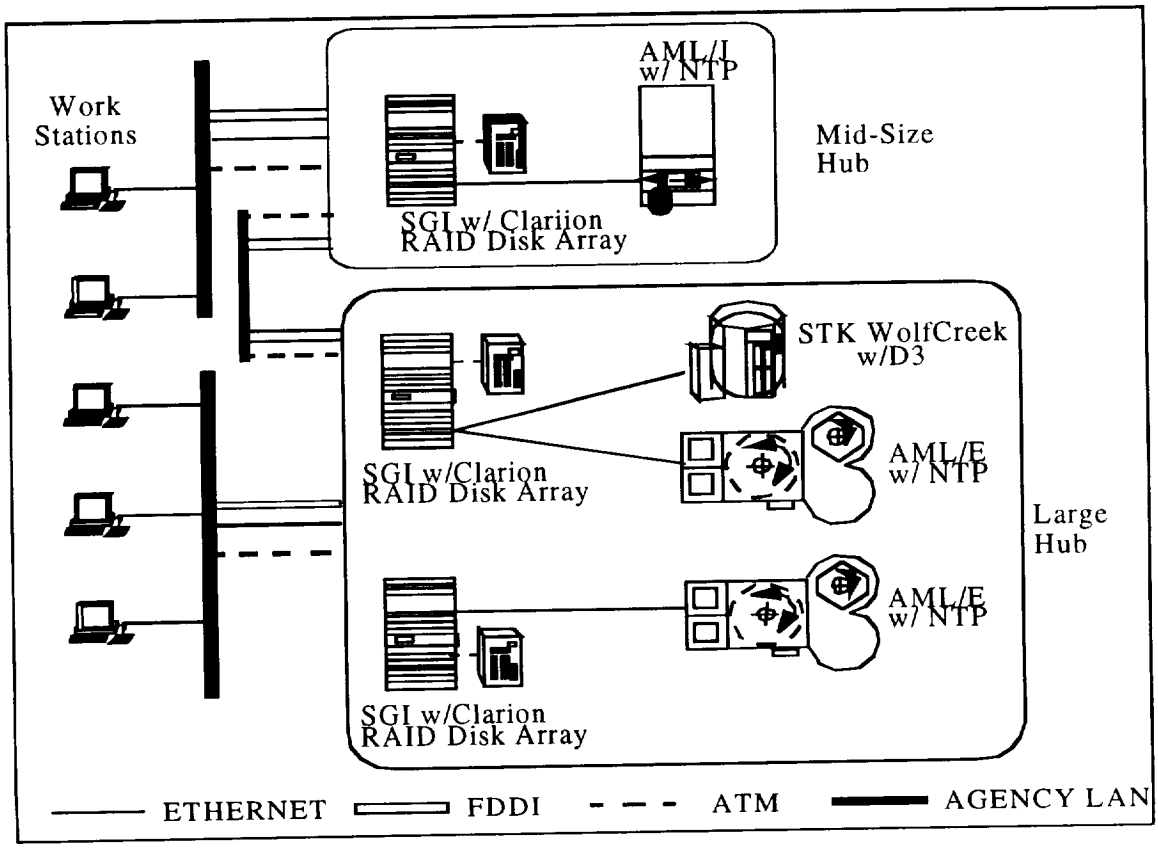

Figure 5: Complete Near Term Test

Longer Term Plans: While this area is highly dependent upon the prior phase and its success, several features are already slated for implemention in the longer term. Among these are:

- Inclusion of a wide area network (WAN) connected hub
- Inclusion of bandwidth management and flow control between hubs across the WAN
- Increasing the numbers/scales of hubs and MSS libraries
- Evaluation of metadata system approaches and their scalability

Other areas under consideration, even though they are merely "on the drawing board", include:

- Expansion of the server area to include Massively Parallel Processors (MPPs)
- Inclusion of Web-based user access
- Inclusion of MultiMedia into the test set

V. Conclusions

In summary, this paper has been an attempt to present a brief overview of the architecture for a global distributed archive with near-real-time access characteristics and the strategy for use of mass storage systems within that architecture. The instantiation of the architecture is clearly a long term project that must be approached incrementally. As such, it is vital that the interface to the archive be implemented early on and that the archive be expanded and improved transparently to early users, behind this interface. Although we would not minimize the challenge of the long term development, we hope that the tremendous benefits to be gained by building such an archive are evident from this brief exposition.