Network Livermore Time Sharing System (NLTSS)

S Terry Brugger
Madhavi Gandhi
Greg Streletz
Department of Computer Science
University of California, Davis

March 7, 2001

Abstract

NLTSS was a distributed, object-oriented operating system based on a pure message passing kernel and capabilities. This paper will describe the goals of its development, an overview of the system as well as a detailed examination of some of its subsystems and what we learned from its development.

1 Introduction

The history of Lawrence Livermore National Lab is closely intertwined with the history of supercomputing. Ever since the lab's founding in the early 1950's, it has used state of the art machines to solve difficult scientific problems. In the 1960's it started looking at ways that it could better utilize the resources of these huge machines and better serve the scientists that used them. From this, the Livermore Time Sharing System (LTSS) was born. By the late 1970's with the advent of the microprocessor, local area networks and advances in operating system technology the Livermore Computing Center set forth to design a new operating system. This 4th generation operating system was "[e]xplicitly designed to support multiprocessing, distribution, extendable object oriented, uniform local & remote IPC, [and] least privilege access control." [21] This successor to LTSS was a specific implementation of a larger framework called LINCS (Livermore Integrated Network Computing System) which allowed distributed access to abstract object interfaces [20]. Designed with this network functionality in mind, the new operating system was called the Network Livermore Time Sharing System, or NLTSS.

The first three years of the project concentrated on research and development of prototypes examining various aspects of multicomputing, network protocols and servers. This period was followed by four years of intense development on the system. Beginning around 1987, NLTSS entered into service on the Cray XMP running alongside LTSS through the use of a virtual hardware abstraction layer [20]. In 1988 NLTSS entered full production on the Cray XMP and later the YMP. It remained in service until shortly after Noon on March 22, 1993 when the last NLTSS installation on the Cray YMP was shutdown [9] and replaced with Cray's version of UNIX (UNICOS).

This paper will examine the goals of NLTSS, take an overview look at the system from the contemporary viewpoint of distributed and object based systems, then it will look in detail at NLTSS's use of capabilities, it's CPU and file servers and it's message passing system.

2 Goals

The primary goal of NLTSS was to design an operating system which allowed the application developer to take advantage of both the multiprocessing capabilities of supercomputer class machines and the distributed computing resources of a network of computers. The NLTSS designers referred to this as a Multicomputing environment[21, 24]. The general goals of the system were to facilitate extendibility, be backwards compatible with LTSS and use standard interfaces [21, 24]. Contrasting these goals with operating system technology at the time, they decided that stage 3 of Kuhn's Model of Scientific Progress had been reached and rather than try to patch existing systems such as LTSS or UNIX, they decided to propose an alternative model [21]. They considered UNIX unworkable as it had problems (at that time) with extendibility, security, distributed computing, multiprocessing, and its implementation had numerous problems such as a lack of clean modularization [21].

The designers established numerous goals in order to achieve the general goal of supporting a multicomputing environment. For instance, the system should do symmetric multiprocessing of the kernel, servers and other user level applications. This symmetrical architecture should be general and allow for all tasks to compute and block for I/O. Any task synchronization and switching should be low cost. The system libraries and scheduler should allow for asynchronous system calls and the ability to wait on an arbitrary number of processes. Finally, the system should perform deadlock prevention and detection [21, 20]. There were additional requirements to support the distributed environment in a transparent manner. For instance, the syntax to access local and remote resources should be the same and, in return, the resource should behave the same way regardless of where it was accessed from. The performance difference between a local or remote access should be as minimal as possible. There should be a separation of machine names from the human-readable names which allows the human-readable names to be completely independent of resource and access location. An object should have the same access rights to a resource regardless of where it's accessing the resource from. All accounting and allocation within the system should be uniform. Finally, the implementation should be such that access is only allowed to objects via well defined interfaces [21, 24, 20].

In order to achieve a highly scalable system, the designers decided early to use a pure message passing architecture with built-in support for the client-server model. For that reason, numerous goals were made for the interprocess communication (IPC), first and foremost being network transparency: IPC calls should look the same whether the client and server are on the same or different machines. To that end, the protocols should be low latency and high throughput. What latency did exist should be hidden as much as possible and the message system may be accelerated through the use of specialized hardware. Processes should have a great deal of latitude in how they use IPC. This latitude included the ability to use arbitrary communications structures and message lengths, the ability to communicate with any other process (subject to the security policy and mechanism), the ability to use multiple concurrent message streams, and the ability to use both synchronous and asynchronous communication. The message system also needed to be clean in its design and implementation through the use of standard request & reply syntax & encodings, separation of data & control, and separation of requests & replies. Finally, the message system would provide flow and error control and protect the data streams [21, 24, 20].

While NLTSS was a revolutionary advance in operating system architecture at the time, it also needed to support the legacy LTSS environment that its user base was used to, and which existing applications were designed for. To that end, NLTSS needed to support existing user interfaces (specifically the terminal interface) and network services. It also needed to emulate the LTSS at the system interface level so that existing application source code could be moved to NLTSS with little to no effort. Finally, it should support incremental evolution by allowing applications to use some of the new functionality while other parts of the code continued to use the legacy interfaces [24, 20].

With the Livermore Computing Center's long history of utilizing and operating supercomputers they added some additional considerations to the goals of the system which came more from practical experience than purely academic operating system design. For example, one goal was to minimize the human cost involved in learning, maintaining, developing for, or in any other way using the system. The system should be designed under the principle that high performance (and, by extension, high cost) computing resources should be available to as large a user base as possible (not just the users in one building for instance). Finally, the system should make it easy to integrate new technology - it needed to be flexible and expandable. This implied a modular or object-oriented design [24, 20].

Finally, the designers recognized the issues common to all distributed and object-based systems as discussed in the overview, below, so there were commensurate goals for the system to deal with those issues [21, 20].

3 Overview

3.1 Issues in Distributed operating systems

NLTSS was designed from the beginning to be a distributed operating system. As such, it has addressed most of the issues inherent in distributed operating systems. The common issues in distributed operating systems [18] and how these are handled in NLTSS are discussed below.

3.1.1 Global Knowledge

Due to the lack of a global clock and global memory as well as unpredictable message delays, determining the entire state of the system at any given moment in time is impossible. Thus, it is necessary to use efficient algorithms to approximate this state. NLTSS had no built-in support to achieve global consensus. The NLTSS designers saw this functionality as being under the purview of the application programmer [15].

3.1.2 Naming

The various objects to which a system refers need to be named in some way. The issue of naming is especially important in a distributed system because the various entities of the system need to understand a common naming convention, such that there must be a global name space. NLTSS had such a global name space implemented with a Directory Server that maps human readable names to machine oriented names. This separation is important as it frees users from the need to follow any naming convention just so the system can locate a given object. It also allows for multiple human readable names for any single object. NLTSS also allows for multiple copies of the same object. Objects in NLTSS can be extended to build new types. Names are ASCII strings up to 16 characters in length. This was seen as a major improvement over NLTSS's predecessor, LTSS, which only allowed 8-10 character names. The namespace in NLTSS did not need to support any type of automatic object replication[21, 24, 20].

3.1.3 Scalability

A distributed operating system should guarantee that the system performance does not degrade as more system resources are added. Certain operating system techniques that work well for a small number of resources may not work well when the system becomes substantially larger. NLTSS was designed such that all the aspects of its distributed nature were abstracted by the message system, which did not pose any limitations on the scalability of the system [13].

3.1.4 Compatibility

A distributed system may be composed of heterogeneous resources. Compatibility refers to the ability of the operating system to provide portability across these various resources. NLTSS was an implementation of the LINCS architecture, which specified a standard architecture (including network and transport layer protocols) for accessing abstract object interfaces [20]. The NLTSS kernel and many of its lower level components were designed exclusively for the Cray massive vector architecture. In particular, it was designed to accommodate many of the irregularities in the Cray memory and process model [15, 13]. Other components such as the file server were compatible across various architectures such as the VAX. NLTSS provided no execution-level compatibility or binary compatibility as it would have been prohibitively expensive.

3.1.5 Process Synchronization

When multiple processes can access a shared resource, the operating system must synchronize access to the resource in order to ensure integrity. Typically, this is accomplished through mutual exclusion. However, due to the lack of shared memory in a distributed system, providing mutual exclusion is a challenge. As such, NLTSS did not provide a mechanism to detect and recover from deadlock, nor to provide mutual exclusion. These were seen as being under the purview of the application developer. A simple mechanism that application developers could use to achieve mutual exclusion was through the use of file locking. Alternatively, they could implement their own system using the flexibility of NLTSS's message system which allowed both synchronous (blocking) and asynchronous (non-blocking) communication between processes [21, 20]. There were plans to add a semaphore server, however it was never implemented due to lack of user demand. There were numerous times throughout the development of NLTSS where the servers themselves would go into deadlock. The solution to this was fixing the servers such that the chain of dependencies that caused the deadlock would not occur (deadlock avoidance) [15].

3.1.6 Resource Management

In a distributed system, the resources available to a process can be both local and remote. The operating system must provide a common method for accessing these resources. NLTSS is designed to deal with this issue by considering all resources to be remote [21]. Typically, NLTSS migrates the data to the computation. It does, however, have the ability to perform computation migration through the use of remote servers and message passing. NLTSS also provides distributed scheduling; it can distribute the computational load amongst different nodes through its process scheduling mechanism [21, 24]. Other considerations were given in NLTSS's design with respect to resource management, such as support for accounting of all resources, administrative allocation of resources and an avoidance on any restrictions on the number or size of resources that an object can use [21, 24, 20]. NLTSS separated the low-level hardware access driver (which had to reside on the machine with the hardware that it controlled) from the policy implementation on the server (such as the File Server or the Process Server which, could be located anywhere on the network) [5].

3.1.7 Security

The two fundamental aspects of security that a distributed system should address are authentication and authorization. Authentication is the means by which a user is verified to be who they say they are and authorization is the means by which the use of resources is restricted to certain users. In NLTSS, authentication is provided through the use of an Account Server [17]. Authorization is provided through the use of capabilities[16]. Numerous other considerations were given to security in NLTSS's design. For example it has a consistent multilevel mandatory access control and security policy for all resources, least privilege discretionary access control to all resources with rights passing, and it allows multiple access rights for a given resource. Finally, its security is based on the principle of mutual suspicion between nodes [21, 24, 20].

3.1.8 Structuring

The structuring of an operating system refers to the method by which the operating system components are organized. The major design decision in an operating system is if it is going to be monolithic or use a microkernel with services separated into individual processes. The microkernel approach lends itself much better to an object-oriented design (where each component is an object with a well defined interface) and the client-server model (where some processes or objects are dedicated servers that handle requests from other processes or servers). NLTSS was designed as a pure microkernel with separate drivers for low level components and user level servers to provide system services. Everything in NLTSS, including the user level servers and all other processes, was considered an object, which places NLTSS in the realm of an object-oriented operating system. Another design technique that NLTSS leveraged was the principle of separation of policy and mechanism which allowed for maximum flexibility of the system.

3.2 Issues with Distributed Object-Based Programming Systems

The use of the object paradigm for creating distributed systems can result in a much more extendable, flexible, programmer friendly architecture. These systems are called Distributed Object-Based Programming Systems (DOBPS) [4]. In the creation of a DOBPS, issues, beyond those of a generic distributed system, need to be addressed. We will briefly explain what these issues (based on the ontology given in [4]) are and how NLTSS addresses them.

3.2.1 Object Structure

As objects are central to the design of a DOBPS, their structure greatly influences the design of the entire system. The two main attributes of object structure are the granularity and composition of the objects.

Granularity The granularity of objects refers to their relative size. There is a tradeoff between how much control the system has over the objects and how many of the system resources are needed to manage those objects. NLTSS considered most everything that the operating system dealt with, such as processes, files, terminals and text strings, to be an object. Hence, NLTSS supported large, medium and small grained objects. It should be noted that not all objects in NLTSS were visible to, and hence managed by, the system. For example, linked lists that existed solely within a user application would not be managed directly by NLTSS [7].

Composition The object composition is the relationship between objects & processes in the system. If processes are part of the object, it's an active composition. If processes exist outside of objects, the system has a passive object composition. Active objects are more resource intensive (as multiple processes may be required to perform a task - one for each object), but they are more amenable to client-server programming. Although processes are objects in NLTSS, not all objects have processes associated with them in NLTSS, hence NLTSS has a passive object composition.

3.2.2 Object Management

Object management deals with the aspects that a DOBPS must deal with that are orthogonal to the types of issued described above (reliability, integrity, security, etc), but unique only insofar as the object paradigm differs from the traditional view of systems as procedural and monolithic.

Action Management Action management is concerned with issues such as serializability, atomicity and permanence. In particular, it is characterized by the type of commit system used by the system to insure the integrity of the objects. In the request scheme, commits are requested by the objects. In the transition scheme, commits are automatically performed by the system as actions complete. NLTSS was designed for the supercomputer environment where system failures could mean the loss of weeks worth of computation. It did not do any type of explicit checkpointing however. Instead, it relied on a feature of the hardware it ran on to dump out a copy of memory to disk in the event of a failure. NLTSS would use the process swap file as a backup to that hardware feature [15].

Synchronization Synchronization is the method by which serializability is achieved. Systems may use a pessimistic (only one action can access an object at any one time) or an optimistic (multiple invocations but no commits until integrity is ensured) scheme.

As noted previously, NLTSS left synchronization up to the application developer.

Security Security ensures that only authorized clients can invoke object methods. The most common schemes are capabilities (which are little tickets that verify the bearer is authorized) or control procedures (where incoming invocation requests are processed by a "guard" that verifies authorization). NLTSS is a pure capabilities based system. This will be explained in greater detail below. Other aspects of NLTSS's security model were discussed above.

Reliability Reliability is the ability of the system to continue to operate when an object fails. It is commonly handled through either object recovery or object replication. In an object recovery scheme, when an object failure is detected it is recreated using either its last committed state (rollback) or its last committed state with all in progress actions reapplied to the object (roll-forward). In an object replication scheme, multiple copies of the object are kept such that if one fails, another will service its requests. This scheme is complicated by the need to have all copies of an object consistent. NLTSS uses a roll-forward object recovery scheme that supported multiple levels of failures. NLTSS is designed to do a deadstart with a minimal loss of state [21, 24, 20]. The most frequent type of failure was when the system got into some type of inconsistent state. This would happen anywhere between every few minutes (usually due to some type of hardware or software bug) to every couple weeks. In this event, a hotstart "DS" would be performed that would simply scan the memory of the machine and restore all processes to a known, good state. Every few weeks to months the errors would be more severe and require a warmstart "DSU" where the objects would be restored from the memory image and process swap files mentioned above. The most catastrophic failures, the time between which was measured in years, required a full coldstart "DSB". As with a warmstart, as much object and process information that could be restored, would be [15].

While Chin and Chanson[4] do not discuss it, it should be noted that NLTSS was designed with robust communication protocols and distributed exception throwing and handling mechanisms [21, 24, 20].

3.2.3 Object Interaction Management

One of the most important functions of a distributed object-based programming system is managing the interactions between objects through location transparency, invocation handling and handling failures.

Location of an Object DOBPS are designed to maintain location transparency of all of its objects. In doing so, it needs a way to locate objects when it is invoked. One scheme is to embed the location of the object in that object's name. Another is to use a name server that maps object names to locations. The third scheme is to keep a small cache of mappings and if the decided object isn't found, broadcast a request for its location. NLTSS uses a combination of the encoding and directory server schemes. NLTSS supports numerous human readable names for any one object [21, 24]. Each object has a single canonical machine readable name[24] called a capability [16]. This capability has embedded into it the location of the object [16]. NLTSS uses a directory server to map the human readable names to capabilities. When the directory server is presented with a human readable name, it finds the corresponding capability, verifies that the requester is allowed to access that object and, if so, returns the capability [12]. Capabilities are discussed in more detail below.

System-level Invocation Handling When one object invokes the method of another object, it's the responsibility of the DOBPS to marshal that invocation to the proper object. The two primary schemes used to achieve this are message passing and direct invocation. NLTSS was designed to be a pure message passing system. Any and all object invocations (even those local to the machine) would be executed through a message that was passed to the kernel, then to the kernel of another node if necessary, and finally delivered to the desired recipient [21]. It was discovered that the overhead of such a system (especially the expense of so many context switches) was too high when accessing some system objects, particularly the file server. These select objects were henceforth incorporated into the kernel. The message passing protocol remained the same, the savings was solely in the lack of context switches between the kernel and the services as the messages were passed between them.

Detecting Invocation Failures It is easy to detect failures that occur before object invocation is made (existing faults). It is much more difficult to detect failures that occur during invocation (transient faults). It is necessary to do this however lest objects block indefinitely on an invocation that will never return. Numerous schemes for handling transient faults exist from time outs to invocation probes. NLTSS used a basic time out scheme to detect faults and killed processes that were blocked waiting for a response that would never arrive. This may have had catastrophic consequences on the system if the transient fault was located in a critical server such as a file server or the process server. On the occasions that such faults happened, the system typically had to be restarted.

3.2.4 Resource Management

A DOBPS is particularly concerned with the interaction between objects and system resources, such as object representation in memory and secondary store, or object scheduling.

Representation of Objects in Memory and Secondary Store Objects are persistent if a copy is kept in secondary store and the object can survive a machine failure, otherwise the object is volatile. Persistent objects that reside both in memory and secondary store are considered active, otherwise the object is only found on disk and is considered inactive. A DOBPS may keep either a single or multiple copies of an object in memory depending on the synchronization scheme. When a persistent object is updated, either the entire object can be rewritten to disk (a checkpoint scheme) or only the changes since the last time the whole object was saved can be stored (a log scheme). All objects in NLTSS are persistent. This was accomplished by virtue of the capabilities: as long as a capability to an object existed, the object could be accessed (in this respect, the capabilities are like references in a programming language). As explained in the section on action management, NLTSS uses a checkpoint type scheme to backup its process objects.

Object Scheduling and Mobility Object scheduling is concerned with what processor a given object will reside on. This may be either explicit (given by the user) or implicit (determined by the system based on metrics such as the relative load of machines). Object mobility is concerned with how objects are moved from one machine (such as one with a high load) to a different machine. NLTSS was designed for the supercomputer environment, so its scheduler is a distributed batch scheduler. NLTSS uses a combination of explicit and implicit scheduling: the system will schedule the object to reside on whichever machine fulfills the constraints given by the user. The user may specify such metrics as the amount of processing time and memory the object will require, as well as the machine architecture the object should run on. The user has the option of specifying which machine the object will reside on. NLTSS also has the notion of a priority based on the identity of the user that submitted the job (so the users in departments that paid for more of the machine get preference in scheduling). NLTSS did not have explicit support for object mobility. In particular, process objects were tied to one machine once scheduled and the mobility of other objects (such as files) had to be handled at the application level. Little work was done with object mobility in NLTSS as there was no support for such a concept in NLTSS's predecessor, LTSS and the NLTSS user community was more interested in backwards compatibility than new features [7].

3.3 Other Issues Considered

During the design of NLTSS, other issues were considered that were not found in the common literature. They stemmed mostly from Livermore's experience in building and operating large computer operating systems. They included features to ease the operation of the system such as the ability to put the system into operator/maintenance mode, the ability to relocate, backup and purge objects, status reporting and performance monitoring. The designers made a conscience effort to separate policy and mechanism. Finally, the designers realized that system implementations are never perfect, so the system had to be easy to debug through the use of tracing, event logs and similar features [21, 20].

4 Design & Implementation

Numerous parts of the design and implementation of NLTSS were discussed in the overview above, particularly those parts than can be compared and contrasted with other distributed systems. Here we will look at some of the components of NLTSS in detail.

4.1 Capabilities / Directory Server

Capabilities are tickets that establish that the bearer has permission to access some object. They are a generalization of capability-list (C-list) operating systems. In C-list systems, the kernel handles all capability granting and tracking. This doesn't scale well for a distributed system [8], so the general capabilities mechanism allows processes to manage their own capabilities. In NLTSS, capabilities are synonymous with an object's machine-oriented name [16].

4.1.1 Capability Creation and Use

The capability representing some object is created at the time that object is created by the server that creates the object, so the file server creates a capability for every file that is created, the process server creates a capability for every process that is created and so forth. This capability is passed back to the object that requested the creation. The capability is then sent with every access request to show that the requesting object is authorized to access the given object [16].

Any capability can be passed to another user or object so that the other user or object can access the object represented by the capability. This is the mechanism by which Discretionary Access Control (DAC) is achieved in NLTSS. The designers of NLTSS believed that there was an inalienable right to pass capabilities. If the system tried to disallow capability passing, the object that was granted a capability could always act as a broker between the object that was being accessed and the object that it thought should be allowed access. This capability passing can be done without any server intervention, so it should be done over a secure channel such that it can't be snooped and used by a third party [8].

NLTSS also implemented Mandatory Access Control (MAC) by virtue of requiring all objects to have an associated capability and all capabilities to have a declared protection level (Secret, PARD, Unclassified, etc.). NLTSS would not allow a user or process to access any object with a higher protection level. This was done by means of flow control on messages at the kernel level. Each message would have a certain protection level. The kernel would verify that messages were only delivered to processes with an equal or higher protection level. The servers would then verify that the protection level of the contents of every message was equal to or lower than the message, otherwise the message was discarded. As all data flow took place via message passing, this mechanism was sufficient to provide flow control [7].

A user or object can assign arbitrary human-readable names to the objects that they hold capabilities for. The capabilities can then be stored in a directory, allowing a user to log out or an object to be removed from memory. The user can then login, or the object can be reconstructed later and the capabilities can then be retrieved from the directory server. Directories can store any number of capabilities. These capabilities may represent any type of object [8, 12]. Hence, NLTSS allows users to store anything in a directory, not just files. This allows for a natural structure where a process object, numerous file objects, a network I/O object and a windows interface I/O object all representing a single logical job can be stored together in a directory. Users can then do things such as easily pass the ability to check and/or modify the job to others. It is important to note that directories themselves are objects and are hence accessed with capabilities [8]. This allows directories to be nested inside other directories, hence creating a resource graph. This is properly a directed graph and not a hierarchical tree like the file system in UNIX as a capability may be stored in any number of directories by just as many different names. The graph represents the global namespace and is distributed among all the Directory Servers in the system [20]. A directory object can even be stored in one of its subdirectories, hence the graph is not acyclic. (This can be simulated in the UNIX filesystem with links, however those are either limited to a single disk partition or are symbolic and can be ignored.) Each user has their own root directory which contains the system directories they need to have access to underneath it (as opposed to having the user's root be a subdirectory of some global root).

NLTSS provides give and take directories for each user to facilitate resource sharing. Any user has the ability to put capabilities into another user's give directory. That user, who has sole read access to the directory, then has access to whatever capabilities were given. Similarly, every user has a take directory where they have sole write access and all other users have read access: this allows that user to publish a resource for all other users to access [8]. This mechanism is similar to, but not as general as the formal take-grant model, but it achieves most of the same functionality.

Directories themselves are comprised of numerous fields in addition to the list of capabilities that they store. Directories are labeled with a certain protection level and access rights, they contain accounting information such as usage counts, times of creation and access, and account to charge for storage space. Additionally, the user can give each directory a comment up to 48 characters in length. Directories support functionality similar to disk directories such as create, delete or list, as well as some functionality unique to their generalized nature, such as reduce access, restore or change header [12].

4.1.2 Capability Structure

A capability in NLTSS is between 68 and 256 bits in length. It is composed of one 64 bit Network Address and between 4 and 192 bits of additional information. This additional information includes, at a minimum, 4 bits of protection level information (such as Secret, Confidential, etc.). As NLTSS was designed for an environment where classified work was performed, this information was considered essential. There were 8 1 bit fields that defined the type of access allowed: Destroy, Modify, Add, Read, Owner, Free Access and 2 reserved for future use. The free access bit, also called the inheritance bit, is particularly interesting. It could be set on a directory capability and given to another process. The receiving process would then have access to the capabilities in that directory, but at no higher level than specified for the directory that was given. For example, if User A has a directory with a bunch of working documents in it (that is, they have Read and Modify permission), they can give a capability to that directory to User B with only the read and Free Access bits set. User B then can read all the documents in that directory, but can not modify them. Capabilities also contained 2 bits of information about the server (whether it supports some, all or more features than a standard server), 8 bits of information on the resource type (File, Account, Process, Terminal, etc.), 10 bits of reserved space and up to 160 bits of server dependent info [16].

4.1.3 Capability Protection

Protection of capabilities through encryption is optional. Capabilities are encrypted with a secret key known only to the server. This is done by encrypting word 1 and word 3 with the server's secret key using DES and storing the result into word 2 (words in NLTSS were 64 bits long) [16]. This scheme requires that capabilities use all possible 256 bits. While this scheme still leaves capabilities vulnerable to data theft, it prevents inadvertent modification of the capability and greatly reduces the chances that an attacker will guess a valid capability for some object [8].

Alternative methods of protecting the capability were considered, such as access lists, encrypted client address and public key client address protection, each had problems in practice though. Access list protection is where the server maintains a list of the addresses of clients that are allowed to access each object. Not only is this approach resource intensive for the server, but it is susceptible to a reflectivity attack against the directory server. Encrypted client address protection embeds the address of the client into the capability and encrypts it using a secret key known only to the server. This relieves the burden of storing access information on the server and, implemented properly, eliminates the susceptibility to reflexivity attacks. It does this at the cost of additional CPU time, something that is at a premium in the supercomputing environment. The final protection also encrypts the client address into the capability, but does so using public key encryption such that one client can encrypt the address of another client into the capability and pass the capability to that client without going to the server. This reduces the CPU load on the server, however the computational expense of public key cryptography, especially given processor technology in the 1980's when NLTSS was being written, made this approach infeasible [8].

4.2 CPU and Process Control

NLTSS supports breaking processes into lightweight tasks (threads) through the use of a user level library [20, 5]. This increases portability and allows the threading implementation to be modified independent of the kernel. The kernel itself was developed as a multitasked process so that multiple system calls could be handled concurrently on a multiprocessor system [20]. The CPU and process control is broken up into two pieces: the CPU driver and the Process Server. The CPU driver is the mechanism that actually controls the CPU. The Process Server is the process that enforces the policy by which processes get scheduled. There is one CPU driver per machine, however a process server may control the job scheduling on multiple machines [5, 1].

4.2.1 CPU Driver

The CPU driver has two tasks. One task handles the time slicing between multiple processes. It will perform the context switch between processes at a given interval (timeslicing). In the event of a significant state change, such as job completion or failure this task notifies the process server of the change. When a job sends a message, this task will forward the message to the message system and either switch to another process if the sending process has blocked or allow the process to continue [5]. The other task in the CPU driver simply receives new jobs from the Process Server and adds them to the queue. It also notifies the message system of the process' arrival so that any queued messages for it can be delivered [5].

4.2.2 Process Server

The Process Server handles all the functionality associated with processes, such as their creation, destruction, forking and interrupt handling. Each process has associated with it a header that contains information such as the conditions under which it should be halted or notified, protection level, time elapsed, account to charge, memory usage, message information, comment and name. In total, there are 39 fields associated with every process [1].

4.3 Message System

NLTSS provides one system call for input and output. It is the Message system call [20, 6]. This call was designed to achieve location independent and efficient communication, support for shared memory multiprocessing, process protection and domain separation along with functional simplicity.

4.3.1 Location Independent Communication

The NLTSS message system provides a system call that allows processes on the same computer as well as processes on different computers to communicate with each other. Simply specifying network addresses in the message call does this. The message system provides a transport level interface as defined by ISO model. When two processes from the same computer are communicating, bits are copied. When the processes communicating are on different computers on the network, packets are exchanged. These details are transparent to the user of the message system call. By changing the network address used, access can be changed from local to remote. Similarly, programs written to service resource requests locally can be changed to directly service equivalent requests from anywhere in the network. Clearly, NLTSS eliminates much of the programming work needed to access services and provide access to services in a communication network.

4.3.2 Efficient Input/Output

Efficient I/O is achieved by minimizing domain change overhead (Cray parlance for a context-switch) and providing direct access to the peripherals. NLTSS will only perform a domain change before a process' timeslice is up if computation can no longer proceed because it is completely blocked waiting for I/O to complete. The message system chains multiple I/O requests into a single call. Changing to the system domain is avoided by providing status of I/O requests directly in the memory of the requesting process. NLTSS only supports data transfer directly to and from peripherals, hence avoiding unnecessary copying and buffering of that data.

4.3.3 Support for Shared Memory Multiprocessing

With the support of the NLTSS process server, two or more parallel forked processes may share the same memory space. The NLTSS message system call allows such forked processes to initiate, control and synchronize both execution as well as I/O for any number of parallel tasks sharing memory.

4.3.4 Process Protection and Domain Separation

The NLTSS message system call is the only interface between a process and everything else outside its memory space. The message system guarantees that only data in the buffer pointed to by Activated Send Buffer table is available to another process. It also guarantees that no data in the memory is externally modified unless it is in the buffer pointed to by an Activated Receive Buffer table. The only exception to this rule is when one process has a capability to another. The NLTSS message system allows protection levels to be assigned to all data so high protection data does not get exposed to processes that cannot provide that protection.

4.3.5 Functional Simplicity

The NLTSS message system call uses a single data structure to keep the call as simple as possible within the above design constraints. The data structure describes each data transfer, a Send or a Receive, using a single buffer. The buffer table contains To and From network addresses, status of an individual i/o request, three "action" bits that describe if the request is to Activate, Cancel or Wait. There are also quite a few other bits that can be set to give more information about the transfer. These include Done bit, BOM (beginning of message), EOM (end of message), Wake bit, etc.

4.3.6 Message System Implementation

The NLTSS Message System follows the Rendezvous and Transfer Mechanism [6]. Basic operation of this message system is as follows:

1. Message sending process activates Send Buffer Table and the message system looks for a corresponding active Receive Buffer Table in the receiving process.
2. If match is found, this rendezvous leads to data transfer from Send Buffer Table to the Receive Buffer Table until one or the other is exhausted. The exhausted buffer table is then marked Done which may lead to waking up of sender or receiver as required.
3. If no match was found in the receiving process then the Send Buffer Table is marked as Send Blocked waiting for matching receive. The receiving process is notified that a message is waiting to be sent to it. When a receive is activated in the receiving process, the Receive Buffer Table gets activated and it first checks if any send is blocked waiting to it. If so, the message system starts the data transfer from Send Buffer Table to the Receive Buffer Table. If not, the Receive Buffer Table is marked as Receive Blocked waiting for a matching send. Note that no notification to the sending process is done here.

If there is a receiving process waiting with its Receive Buffer Table for a send, then this rendezvous is more efficient since the additional step of notifying the receiver that sender is waiting to send is eliminated. This is more obvious where the rendezvous is between two processes not on the same machine. Planning to have the receives activated before corresponding sends helps with efficiency.

Transfer of buffered data is asynchronous. If a send buffer is larger than the matching receive buffer, send will do a partial data transfer and wait for subsequent receives to complete the send. Correspondingly, if the receive buffer is larger than the send, that receive only partially completes until more data can be received. The sender process could put a special message ending marker (EOM, Wake) to force the receive to complete.

NLTSS supports transfer to and from arbitrary bit boundaries. Messages sent to devices get transferred directly from the user's memory to the device if the device is capable of the transfer. Direct device transfers reduce the cost of transfer substantially.

4.3.7 Protocol Compatibility Issues

The NLTSS message system can provide local communication by itself, but to communicate with other computers, it requires communication support and protocol compatibility at various levels up to and including transport level (ISO) [6]. NLTSS supports the Delta-T transport protocol [19] and TCP/IP. There were problems with transport protocols other than Delta-T. For example, most transport protocols only support transfer of data in units of 8-bit bytes but the NLTSS message system interface supports the transfer of an arbitrary number of data bits. Additionally, most transport level protocols do not have the concept of protection levels, hence NLTSS users communicating with such protocols may get only single protection level.

Most transport level protocols are connection oriented in that an explicit connection open before data transfer and an explicit connection close after transfer is completed is expected. For performance reasons Delta-T chose not to explicitly open and close the connection for data transfer and the message system follows that trend. However, the message system also provides the BOM and EOM bits for functional and compatibility reasons. These bits mark the open and close of the connection, respectively.

The NLTSS message system and Delta-T use a fixed length 64 bit network address. Each network address uniquely identifies a communication port associated with a task or set of tasks in the same process. The network address is broken down into an 8 bit network identifier, an 8 bit machine identifier, an 8 bit sub-machine identifier, a 16 bit process identifier and a 24 bit port number [20]. This may create problems with other protocols that use extendible or other very large forms of network addresses.

The communication itself takes place over streams. Streams are unidirectional connections that are identified by the source address, destination address and a stream number. The stream number is similar to a capability in that knowledge of its value is necessary to access the stream and it is designed to be unguessable [20]. Interrupts and other out-of-band signals are sent on a separate parallel stream in NLTSS and Delta-T, as a matter of philosophy. This is done to keep the protocol and interface simpler.

4.4 File System

The NLTSS file system provides distributed file access. The basic goal is to allow processes on any processor to access files that reside either locally or remotely. To some degree, the NLTSS file system was patterned after Sun's Network File System (NFS). However, NFS itself fell short of the requirements of a file system for a distributed supercomputer environment [25]. Specifically, NFS provides remote file access by keeping the file data in its remote location and by transferring it to the requesting process in fairly small pieces. While this strategy usually is sufficient for the requirements of a typical distributed environment, it is insufficient for the needs of a distributed supercomputer environment due to the high-bandwidth characteristics of supercomputers. To use these high-performance machines efficiently, it is necessary to migrate the data to the processor. Ideally, this migration is performed at a high speed, and the migrated data is then cached local to the processing in order to allow it to be accessed quickly during computation. The NLTSS file system provides this functionality.

4.4.1 Utility of a Distributed File System

The NLTSS file system allows a process on a given machine to read and write not only local files, but remote files as well [3]. This transparent distributed file system capability is of great utility in a distributed supercomputer environment, in which migrating certain functionality from number crunching supercomputers to workstations is a desirable goal. For example, a numerically intensive calculation can be run on a supercomputer, and the resulting data can be visualized by using graphics software on a local workstation. By doing this, the supercomputer is not burdened with graphics computations, which are more suited to be performed on the workstation. Moreover, the graphics algorithms on the workstation can use the services of the distributed file system to access the data on the supercomputer in a fairly transparent fashion, with no greater effort than would be required to analyze local data. Another example of the usefulness of the distributed file system was pointed out by the authors of NLTSS [3]. They observed that the file system allowed programmers to use a full-featured text editor on a VAX to directly edit a file on a Cray, for which the available text editing facilities were less sophisticated.

4.4.2 The NLTSS File Resource

A disk file under NLTSS is referred to as a file resource, and is identified by a capability that authorizes access to it. A file resource is essentially a record consisting of several fields [10]. The Body field is the field of the file resource that contains the actual file data; it contains unformatted data written there by a customer process. The other file resource fields describe the attributes and location of the Body, and are collectively referred to as the Header [2]. The Header consists of the following fields: Allocated Length, Altered Flag, Comment, Fragmented List, Length, Lifetime, Lock Type, Logical Unit, Notification Address, Physical Unit, Protection Level, Storage Account, Subtype, Time Last Read, Time Last Written, Time of Creation, Total Reads, Total Writes, Unique File ID, Usage Hint and User Checksum [10]. As can be seen from this list of file resource fields, a NLTSS file resource encompasses much more than just the file data.

4.4.3 File Capabilities

A file capability identifies a file resource and authorizes access to it. In order for a process to perform any operation on a file, the process must have a capability for that file. Each file capability includes the following access bits: Destroy, Modify, Add, and Read [10]. The process can perform a given type of operation on the file only if the corresponding access bit is set. The Read and Destroy access permissions have the expected meanings; Read gives the process read access to any field of the file resource, while Destroy gives the process the ability to destroy the file. Destroy also gives permission to revoke capabilities to the file. The Modify permission gives the process the ability to modify any field of the file resource except the Allocated Length field. The Add permission gives the process the ability to increase the size of the Body field, which necessarily includes the ability to alter the Allocated length field.

4.4.4 File System Components

The NLTSS file system performs disk I/O through the use of several basic file system components: file servers, hardware interface processes (HIPs), I/O interfaces, and disk drivers.

File Server The file server for a specific computer is responsible for managing all of the files on the disks of that computer. The file server handles the allocation and deallocation of disk space, maintains a catalog of files currently residing on the disk, manages file headers, controls access to files using the access bits of file capabilities, and handles I/O requests [2, 10]. NLTSS file servers support fragmented files, can operate in parallel with the process requesting disk access, and can reside on a machine other than the one whose disk is being managed [2]. Also, the file server was designed to ensure that the failure of a single disk would not result in the loss of files.

The file server provides four categories of functionality: file resource management (sometimes referred to as file header management), file access management, I/O request management, and operations pertaining to the handling of reply monologs [10]. Reply monologs and the NLTSS Control Monolog Record (CMR) are used to provide communication between the file server and the customer process during the process of file I/O, and will not be described in detail here. The other three categories will be discussed briefly.

File resource management encompasses

the following operations: Change, Create, Create and Write, Destroy, and Interrogate [10]. Change is used to change a writable field of a file resource. Create is used to create a new file. Create and Write is used to create a read-only file and to write the (static) data that this file is to contain. Destroy, as its name suggests, is used to destroy a file. The final file resource management operation, Interrogate, is used to query a readable field of a file resource.

File access management involves the operations New Capability, Reduce Access, and Set Lock [10]. New Capability provides the functionality for generating a new capability to a given file, or to invalidate existing capabilities. Reduce Access is used to reduce the access permissions associated with the capability to a file. Set Lock is used to set a lock (pertaining to either read, write, or read/write operations) on all fields of a file resource.

I/O request management operations include Pattern, Supply Server Data Address, Read, and Write [10]. Read and Write are used to read data from the Body field of a file resource or to write data to this field, respectively. Pattern is used to write a specified pattern over the Body field of a file. Supply Server Data Address is used to request the file server to provide a capability containing the address of the data mover by which data will be transferred to or from the customer process.

Disk Driver and I/O Interface The disk driver is low-level software that controls hardware-specific aspects of the process of actually reading data from and writing data to the physical disk hardware. The I/O interface provides a somewhat higher level interface to the low-level disk driver software. At the time NLTSS was designed, it was decided that the disk driver and interface that were already resident on the Crays at LLNL would be retained and incorporated into the NLTSS file system [11].

Hardware Interface Process (HIP) Because of the decision to reuse pre-existing disk driver and interface software, the NLTSS file system was obligated to replicate the I/O request format expected by this software. Therefore, an additional level of software was needed to translate between NLTSS disk I/O requests and disk driver I/O requests. This software interface was named the Hardware Interface Process (HIP) [11]. Another benefit of this arrangement is that it provides an abstraction that encapsulates the device specific code of the device driver and provides a uniform I/O interface to the HIP. Therefore, the transition to different disk hardware, for example, would be simplified. The HIP consists of several different tasks, which are coordinated with each other using standard semaphore synchronization mechanisms [11].

4.4.5 Input Output Descriptors (IODs) and Component Interaction

IODs Input Output Descriptors (IODs) are 192-bit data structures that are passed as messages between NLTSS file servers, HIPs, and disk driver I/O interfaces, and which facilitate the communication between these components. There are four varieties of IOD: a File-Server-to-Disk-HIP IOD, a Disk-HIP-to-I/O-Interface IOD, an I/O-Interface-to-Disk-HIP IOD, and a Disk-HIP-to-File-Server IOD. Although all four types of IOD are 192 bits in size, the exact structure of the fields that compose each IOD differs somewhat from type to type [11].

File System Component Interaction To illustrate the roles played by the various components of the NLTSS file system, and to describe the interactions (via IODs) between these components, we must consider what happens during a generic file I/O operation. First, the file server receives a logical I/O request from a customer process. This request for a data transfer contains a file capability, the address of the first bit of data to be transferred, and the total number of bits to be transferred, the type of data transfer, and the network addresses to be used for the transfer [11]. The file server then examines the file capability and determines if the requesting process has the requisite permissions to perform the requested file operation. If so, the file server proceeds to construct one or more IODs from the data in the logical I/O request. These are File-Server-to-Disk-HIP IODs, and there is one for each physical disk fragment specified in the logical I/O request (up to a maximum of eight fragments) [11]. These IODs are then transferred to the Hardware Interface Process (HIP).

The HIP is responsible for translating the request(s) from the file server into a format that the disk driver understands. In addition, it has the crucial role of obtaining the memory addresses within the customer process at which the data transfer is to take place. It obtains these data directly from the customer process itself, since the file server has no knowledge of this information [11]. Note that this is required only if the data transfer is a disk-to-memory or memory-to-disk transfer, and is not necessary for a disk-to-disk transfer (which is just a file copy, and does not involve any memory address within the customer process). Likewise, for a disk-to-memory or memory-to-disk transfer, the HIP needs to "lock down" the requesting process to prevent it from being swapped out of memory on a context switch during the file transfer. It does this by issuing a privileged message system request called an I/O Block [11]. Next, the HIP constructs one or more Disk-HIP-to-I/O-Interface IODs based on the information in the File-Server-to-Disk-HIP IODs received from the file server (again, there is one IOD for each fragment). These IODs (or, as many of them as possible) are then sent to the I/O interface, which essentially relays them to the disk driver by placing them on the disk driver queue.

The completion of disk operations is monitored by the I/O interface. When all of the physical fragments of a single logical I/O request are completed, the I/O interface constructs a single I/O-Interface-to-Disk-HIP IOD (regardless of how many fragments there were), and passes it up to the HIP [11]. Alternatively, if an error occurs, the IOD passed up to the HIP indicates this fact.

When the HIP receives an I/O-Interface-to-Disk-HIP IOD, it either sends more Disk-HIP-to-I/O-Interface IODs to the I/O interface or, if the entire logical I/O job is complete, it constructs a Disk-HIP-to-File-Server IOD to pass back up to the file server [11]. Of course, if the HIP receives an error IOD from the I/O interface or detects an error itself, it will send an error IOD of its own to the file server. In either case, it then issues a privileged message system request called an I/O Unblock , which causes the locked down process to be released; since the I/O operation is complete, the customer process is once again eligible to be swapped out of memory on a context switch. Upon receipt of the Disk-HIP-to-File-Server IOD, the file server can communicate to the customer process that the I/O operation is finished, or that an error has occurred, depending on the content of the IOD. At this point, the entire logical I/O operation is finished.

4.4.6 Performance Considerations

In the initial design of the NLTSS file system, the file server, hardware interface process, and driver/interface components of the file system were all separate processes. Communication between the processes was achieved by passing IOD messages through the NLTSS Message System. Although the encapsulation achieved in this model is conceptually attractive, these components were eventually merged due to performance reasons [11]. At first, the file server (FS) and the hardware interface process (HIP) were merged into a single process called the FSHIP. Later in the evolution of the system, the driver functionality was incorporated into the FSHIP as well [11]. In the FSHIP process, there are two queues, one to hold the File-Server-to-Disk-HIP IODs and the other to hold the Disk-HIP-to-File-Server IODs. The FS and the HIP are part of the same FSHIP process, and these two queues are shared between both components; access control is provided through semaphore-based synchronization [11].

Although the individual NLTSS file system components were merged in the sense that they were brought into the same process space, the codes for the components remained modularized, thereby preserving much of the original encapsulation, at least conceptually. Meanwhile, a performance enhancement was achieved by converting to a system with a single file system process (per physical disk, of course). Such tradeoffs are instructive, and this modification in design was one of the lessons learned in the NLTSS experience.

5 Lessons Learned

NLTSS was intended to be a revolutionary advance in operating system technology. For the most part, it succeeded. Of its long list of goals, most were successfully implemented. The goals that weren't met were primarily those for which there was insufficient user demand, such as encryption of communication channels or automatic distributed deadlock detection. The primary change in the design that came about after putting the system into action was the movement of critical services such as the file server and CPU server into the kernel, which was a sacrifice made purely to avoid the context switch overhead to these heavily used services. The NLTSS team made a conscious effort to keep a clean separation between these components and the low-level kernel though in the hopes that given advances in hardware technology those services could one day be moved back into user space [7]. It is suspected that some of these boundaries were crossed though while the team was under pressure to provide higher performance [14].

In retrospect, NLTSS's features do not seem very revolutionary by today's standards. The methods that NLTSS's architects used can be found in most any computer science text that covers distributed operating systems such as [18]. What is notable about NLTSS is that addressed issues such as scalability, reliability and integrity before they were identified as being issues common to all distributed systems. Furthermore, NLTSS was more than an academic exercise: it actually supported a demanding user community for over three years of production use. Finally, it implemented all these features on the real memory Cray architecture which was not amenable to being controlled by an advanced operating system [15].

During the shutdown ceremony for NLTSS, Dick Watson attributed its demise to the widespread acceptance of UNIX. While NLTSS was backwards compatible with its predecessor, LTSS, it was not compatible with UNIX, which many users demanded[22]. In retrospect, it probably wasn't that the users actually wanted UNIX, just that UNIX was good enough to perform the tasks that the users required. While NLTSS was designed to address what the designers saw as numerous shortcomings in UNIX, the users didn't actually use these advanced features as they were locked into the LTSS way of doing things. Instead of taking advantage of the new features, users complained that operations took longer to complete and they couldn't understand the rational for the change [7]. In the end, the cost of developing and maintaining an operating system was just too high when a version of UNIX that was good enough was readily available. Eight years after his speech at the last NLTSS shutdown, Dick Watson attributed the reason for NLTSS's demise was that "it was too revolutionary," it tried to fix all the problems with existing systems which made it so different from anything else that it never really got accepted [23].

6 Conclusion

NLTSS was a distributed, object-oriented operating system based on a pure message passing kernel and capabilities developed at Lawrence Livermore National Lab from the late 1970's through 1993. As it was designed to be a full-featured production level operating system supporting a multicomputing environment, it had numerous goals concerning its use of multiprocessing, message passing, interprocess communication and distributed processing. Other goals focused on backwards compatibility, operational considerations and issues common to distributed and object-based operating systems. An overview of the issues in distributed operating systems and distributed object-based programming systems was presented along with how NLTSS addressed each issue. The design and implementation of NLTSS with particular regard for the capabilities & directory server, CPU & process control, message system, and file system were covered in detail. Finally, we looked at what can be learned from NLTSS in retrospect.

7 Auspices & Acknowledgements

This paper is a personal work of the authors and was not developed under the auspices of the United States Department of Energy, the University of California or Lawrence Livermore National Lab. Any opinions expressed herein are solely those of the authors and may or may not reflect the views of their respective employers or the University of California, Davis. NLTSS was developed under the auspices of the United States Department of Energy by Lawrence Livermore National Lab under contract number W-7405-ENG-48.

The authors gratefully acknowledge the assistance of Jed Donnelley, David Fisher, Donna Mecozzi, Jim Minton and Dick Watson, without whose assistance this paper would not have been possible.

8 References

References

[1] Ralph Allen. NLTSS process server. Technical report, Lawrence Livermore National Laboratory, December 1984.

[2] Charles Athey III, Jeffrey Clark, James E. Donnelley, Pierre Du Bois, Donna T. Mecozzi, and James A. Minton. NLTSS. Lawrence Livermore National Laboratory, Internal presentation, date unknown.

[3] Charles Athey III, Kent Crispin, Pierre Du Bois, Donna T. Mecozzi, and James A. Minton. NLTSS/LINCS for a presentation to Nuclear Software Systems Division. Lawrence Livermore National Laboratory, Internal presentation, November 1982.

[4] Roger S. Chin and Samual T. Chanson. Distributed object-based programming

systems. *ACM Computing Surveys*, 23(1), March 1991.

[5] James E. Donnelley. The internal structure of NLTSS. Technical report, Lawrence Livermore National Laboratory, June 1983.

[6] James E. Donnelley. The NLTSS message system: Definition and implementation for the Cray-1 and Cray X-MP computers. Technical report, Lawrence Livermore National Laboratory, May 1984.

[7] James E. Donnelley. Personal communication, February 16, 2001.

[8] James E. Donnelley and John G. Fletcher. Resource access control in a network operating system. In *ACM Pacific '80 Conference, San Francisco, CA*, November 1980.

[9] Invitation to NLTSS shutdown ceremony. Lawrence Livermore National Laboratory, Memorandum, March 1993.

[10] Donna T. Mecozzi. NLTSS file server. Technical report, Lawrence Livermore National Laboratory, April 1986.

[11] Donna T. Mecozzi, James A. Minton, Donald L. von Buskirk, and Joe Requa. NLTSS disk I/O. Technical report, Lawrence Livermore National Laboratory, July 1988.

[12] James A. Minton. NLTSS directory server. Technical report, Lawrence Livermore National Laboratory, July 1981.

[13] James A. Minton. Personal communication, February 23, 2001.

[14] James A. Minton. Personal communication, March 6, 2001.

[15] James A. Minton and David Fisher. Personal communication, February 16, 2001.

[16] James A. Minton and Donna T. Mecozzi. What are capabilities? Technical report, Lawrence Livermore National Laboratory, February 1986.

[17] Norm Samuelson. NLTSS account server. Technical report, Lawrence Livermore National Laboratory, September 1985.

[18] Mukesh Singhal and Nirnajan G. Shivratri. *Advanced Concepts in Operating Systems.* McGraw-Hill, 1994.

[19] Richard W. Watson. DELTA-T protocol specification. Technical report, Lawrence Livermore National Laboratory, December 1982.

[20] Richard W. Watson. Working notes: Motivation, goals and development strategy of NLTSS. Lawrence Livermore National Laboratory, Working notes, July 1987.

[21] Richard W. Watson. The architecture of future operating systems. Lawrence

Livermore National Laboratory, Internal presentation, January 1989.

[22] Richard W. Watson. Eulogy to NLTSS. Lawrence Livermore National Laboratory, Internal presentation, March 22, 1993.

[23] Richard W. Watson. Personal communication, January 29, 2001.

[24] Richard W. Watson. Requirements and architectural features of future operating systems. Technical report, Lawrence Livermore National Laboratory, circa 1989.

[25] Richard W. Watson. Distributed computing and operating systems for the supercomputer environment. Technical report, Lawrence Livermore National Laboratory, date unknown.