Statelessness and Statefulness in Distributed Services

Charles H. Sauer
Don W. Johnson
Larry Loucks
Amal A. Shaheen-Gouda
Todd A. Smith

IBM Advanced Engineering Systems
Austin, Texas 78758

INTRODUCTION

Distributed Services (DS) provides distributed operating system capabilities for the AIX[1] operating system. These include distributed files with local/remote transparency, a form of "single system image" and distributed interprocess communication. The distributed file design supports "traditional" AIX and UNIX[2] file system semantics. This allows applications, including data management/data base applications, to be used in the distributed environment without modification to existing *object* code. The design incorporates IBM architectures such as SNA and some of the architectures of Sun Microsystems'[3] NFS. This paper focuses on a number of implementation issues in DS, particularly use of *stateless* and *stateful* mechanisms. Before focusing on these mechanisms, we provide an overview of Distributed Services. For additional background information on DS, see Sauer *et al* [1,2] and Levitt [3].

DISTRIBUTED SERVICES DESIGN GOALS

The primary design goals in our design of Distributed Services have been

Local/Remote Transparency in the services distributed. From both the users' perspective and the application programmer's perspective, local and remote access appear the same.

Adherence to AIX Semantics and Unix Operating System Semantics. This is corollary to local/remote transparency: the distribution of services cannot change the semantics of the services. Existing object code should run without modification, including data base management and other code which is sensitive to file system semantics.

Remote Performance = Local Performance. This is also corollary to transparency: if remote access is noticably more expensive, then transparency is lost. Note that caching effects can make some distributed operations *faster* than a comparable single machine operation.

Network Media Transparency. The system should be able to run on different local and wide area networks.

1. AIX is a trademark of International Business Machines Corporation
2. Developed and licensed by AT&T. Unix is a registered trademark in the U.S.A. and other countries.
3. NFS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

Mixed Administrative Environments Supported. This includes support of single system image clusters, various servers (file, code, device, ...), and independently administered workstations. Additionally, services must be designed to make the administrator's job reasonable.

Security and Authorization Comparable to a Single Multiuser Machine.

We believe we have done well in meeting our design goals.

DISTRIBUTED SERVICES FILE SYSTEM OVERVIEW

Remote Mounts

Distributed Services uses "remote mounts" to achieve local/remote transparency. A remote mount is much like a conventional mount in the Unix operating system, but the mounted filesystem is on a different machine than the mounted on directory. Once the remote mount is established, local and remote files appear in the same directory hierarchy, and, with minor exceptions, file system calls have the same effect regardless of whether files(directories) are local or remote[4]. Mounts, both conventional and remote, are typically made as part of system startup, and thus are established before users login. Additional remote mounts can be established during normal system operation, if desired.

Conventional mounts require that an entire file system be mounted. Distributed Services remote mounts allow mounts of subdirectories and individual files of a remote filesystem over a local directory or file, respectively. File granularity mounts are useful in configuring a single system image. For example, a shared copy of /etc/passwd may be mounted over a local /etc/passwd without hiding other, machine specific, files in the /etc directory.

File System Implementation Issues

Virtual File Systems. The Distributed Services remote mount design uses the Virtual File System approach used with NFS [4,5]. This approach allows construction of essentially arbitrary mount hierarchies, including mounting a local object over a remote object, mounting a remote object over a remote object, mounting an object more than once within the same hierarchy, mount hierarchies spanning more than one machine, etc. The main constraint is that mounts are only effective in the machine performing the mount.

lookup. In conjunction with using the Virtual File System concept, we necessarily have replaced the traditional namei() kernel function, which translated a full path name to an i-number, with a component by component lookup() function. For file granularity mounts, the string form of the file name is used, along with the file handle of the (real) parent directory. This alternative to using the file handle for the mounted file allows replacement of the mounted file with a new version without loss of access to the file (with that name). (For example, when /etc/passwd is mounted and the passwd command is used, the old file is renamed opasswd and a new passwd file is produced. If we used a file handle for the file granulariity mount, then the client would continue to access the old version of the file. Our approach gives the, presumably intended, effect that the client sees the new version of the file.)

4. The traditional prohibition of links across devices applies to remote mounts. In addition, Distributed Services does not support direct access to remote special files (devices) and the remote mapping of data files using the AIX extensions to the shmat() system call. Note that program licenses may not allow execution of a remotely stored copy of a program.

Kernel Structured Using Sun "vnode" Definition. We have used the Sun vnode data structure (see Kleiman [6]) to support multiple file system types in the AIX kernel. This allows a clean division between the local AIX filesystem code and the remote filesystem code. Further, it allows a natural approach in our current development of NFS support in AIX. See Figure 1.

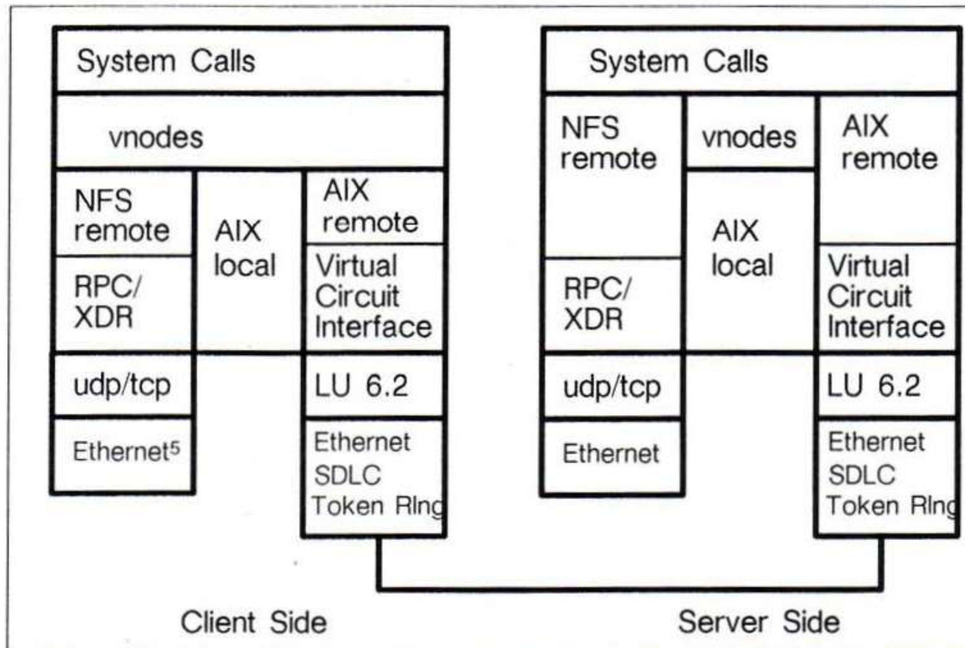

Figure 1. Architectural Structure of Distributed Services File System

5. Ethernet is a trademark of Xerox Corporation.

Virtual Circuit Interface. Distributed Services assumes virtual circuits are available for network traffic. One or more virtual circuits must remain in force between a client with a file open and the server for that file. (The mere existence of a remote mount does not require retention of a virtual circuit.) Execution of cleanup code, e.g., decrementing usage counts on open files, will be triggered by loss of a virtual circuit. The architecture of Distributed Services includes a *Virtual Circuit Interface* (VCI) layer to isolate the Distributed Services code from the supporting network code. Our current code uses the SNA LU 6.2 protocol to provide virtual circuit support, but, potentially, another connection oriented protocol, e.g., TCP, could be used. The basic primitives of the VCI are the `dsrpc()`, `dsrpc_got()` and `dsgetdata()` functions. `dsrpc()` acquires a connection with a specified machine and then issues `dsrpc_got()` to invoke a function on that machine. `dsrpc_got()` is called directly if the caller has a previously established connection available. Both of these calls return without waiting for the result of the remote function, allowing continued execution on the calling machine. `dsgetdata()` is used to request the result of a remote functions; it will wait until the result is available.

SNA LU 6.2 Usage. We chose to use LU 6.2 because of its popular position in IBM's networking products and because of its technical advantages. In particular, LU 6.2 allows for "conversations" within a session. Conversations have the capabilities of virtual circuits, yet with low overhead of the order typically associated with datagrams. Typically, one or two sessions are opened to support the flow between two machines, regardless of the number of virtual circuits required. We have methodically tuned the LU 6.2 implementation, utilizing the fully preemptive process model of the AIX Virtual Resource Manager (see Lang, Greenberg and Sauer [7]). By properly exploiting the basic architecture of LU 6.2 and careful tuning, we have been able to achieve high performance without using special private protocols or limiting ourselves to datagrams.

The AIX implementation of LU 6.2 supports Ethernet, SDLC and Token Ring transport. The AIX LU 6.2 and TCP/IP implementations are designed to coexist on the same local area network (Ethernet or Token Ring). In our development environment, we use both protocols simultaneously on the same machines and media, e.g., TCP/IP for Telnet and/or X Windows, and LU 6.2 for Distributed Services.

STATELESSNESS AND STATEFULNESS

Definition and Implementation Approach

One of the key implementation issues in remote file system implementation is the approach to "statelessness" and "statefulness." We loosely define "stateless" as meaning that a client for a file (system) does not retain state about the server for that file (system) and vice versa.[6] The NFS designers have heavily exploited statelessness [4] and demonstrated the advantages gained by using stateless mechanisms. Where it is practical to use a stateless approach, without compromising our design goals, we have done so. For example, our remote mounts are stateless -- a server can be recycled without a client of that server losing mounts. However, in some areas where we believe a stateful approach is necessary, we maintain state between server and client and are prepared to clean up this state information when a client or server fails. For example, we maintain state with regard to caching mechanisms, so that cache consistency can be assured. We also add mechanisms "on top of" stateless mechanisms to simplify administration and operation.

Virtual File Systems and Inherited Mounts

The Distributed Services remote mount design uses the Virtual File System approach used with NFS [4,5]. This allows for stateless mounts, in the sense described above. When a client successfully requests a mount from a server, it receives an handle for the object it is mounting and stores it in its mount table. The handle is effectively a pointer to the on disk inode for the mounted object and a generation number for that inode. The generation number is used for subsequent validity tests. When the client is parsing a file pathname, e.g., for `open()`, and encounters the mounted object, the handle is given to the server as an argument in the `lookup()` remote procedure call. Typically, the mounted object is a directory, and the server will lookup an object within that directory.

For example, let us suppose that a client mounts server's `/B` over `/a/b`. The client then opens `/a/b/c`. When the client gets to `b/c`, it passes the handle for `b` and the component `c` to the server, requesting the server to lookup and return a handle for `c` that can be used in the actual `open()` call. The server will return a handle for `/B/c`.

6. We say "loosely" because some state information is retained even in designs normally thought of as "stateless." For example, file handles given to clients by servers contain generation information that would be invalid if the corresponding file were lost in a disk failure and recovered from backup media.

There are several points to notice here. First, this approach is stateless in that the server can be recycled (e.g., powered off and on) and the handle(s) given to the client(s) performing a mount(s) is still valid, so the mount need not be repeated. This is true because the handle refers to an on disk structure, not an in memory structure. Second, the path resolution process must necessarily ignore mounts on the server, since these are not reflected in the on disk structures and are not necessarily repeated when the server is recycled. Third, as an immediate consequence, the client must explicitly perform all mounts "for itself," since it does not "see" mounts performed by the server.

Inherited Mounts

However, it is desirable for one machine to be able to "inherit" mounts performed by other machines. As an example, consider the series of mounts depicted by Figure 2. Though this example is slightly exaggerated for emphasis, it is highly relevant to actual configurations. The thicker lines indicate mount points. Thus, from the client, /usr/src refers to .../src on "srcsrvr," /usr/src/cmd refers to .../cmd on "cmdsrvr," and so on. Without the inheritance mechanism described below, the client must be configured to explicitly perform each mount, e.g.,

```
mount -n srcsrvr .../src /usr/src
mount -n cmdsrvr .../cmd /usr/src/cmd
mount -n ccsrvr .../cc /usr/src/cmd/cc
```

if the mounts are entered from the command line, or, in the more likely situation, the client's mount profile (/etc/filesystems in AIX) would contain an entry for each of these mounts. If the srcsrvr machine were reconfigured to obtain /usr/src/cmd from a different mounted object (say, a new local disk), then all of the client machines would have to be explicitly reconfigured to reflect the new object.

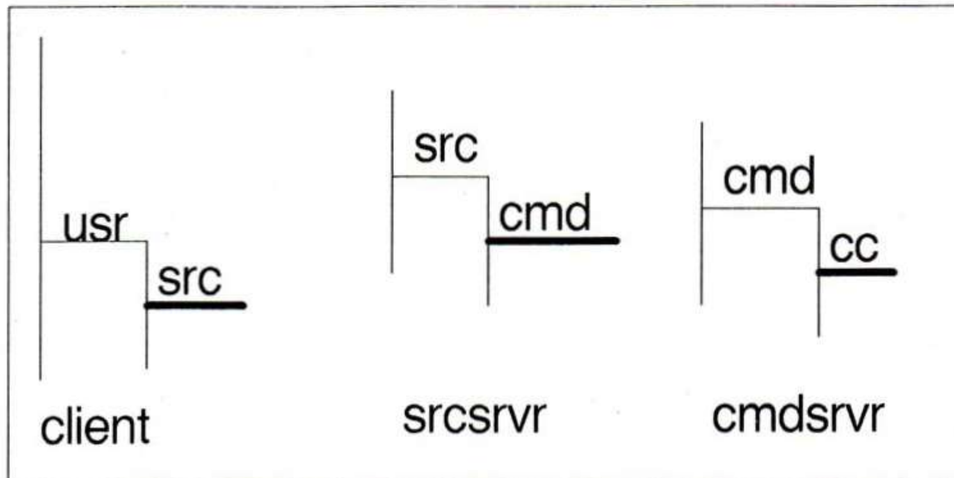

Figure 2. Inherited Mount Example

Distributed Services implements inherited mounts on top of virtual file systems. There is a mntctl() system call and corresponding remote procedure call. One of the options of mntctl() is to query and return a list of all mounts currently in effect on a given server. The mount command in

AIX supports a -i (inherited) flag which causes the query to be performed and the additional mounts to be made. For the above example,

```
mount -i -n srcsrvr .../src /usr/src
```

would have the same net effect as the three separate commands illustrated previously. Similarly, an entry of the form

```
/usr/src:
     nodename = srcsrvr
     dev = .../src
     mount = true,inherit
```

in /etc/filesystems would cause this set of mounts to be performed at system startup time. By use of inherited mounts, clients of a file server need not know of restructuring of the server's mounts underneath the initial mount. For example, if a client always uses an inherited mount of /usr/src, it does not need to change it's configuration files when the server uses additional/different mounts to provide the subdirectories of /usr/src.

The mount extensions associated with Distributed Services are also usable and useful for local mounts. If a stanza of the form

```
/LOCAL:
     dev = /
     mount = true,inherit
```

is placed in /etc/filesystems after all local device mounts and before any remote mounts, then even after the remote mounts are performed, and normal paths to local files are hidden, alternate paths to local files are available through /LOCAL. For example, if /usr/bin is mounted over to obtain a version from a code server, the local version will still be accessible as /LOCAL/usr/bin.

Separate Machine Operation

Clearly, it is desirable that a client machine be able to operate if one or more of the servers it uses is not available. For example, as part of the DS single system image support, all of the machines in a cluster share a copy of /etc/passwd stored on the "/etc server." A critical aspect of this is having recent copies of the shared files from the "/etc server." As part of the mounting of these files, before the mount is actually performed, the file is copied from the server to the client. For example, before mounting the shared /etc/passwd over the client /etc/passwd, the shared version is mounted temporarily over another file and copied to /etc/passwd. In addition to sample configuration files for this process, sample recovery daemons are provided with DS. The recovery daemon can be used to repeatedly try to open critical files such as /etc/passwd. When the daemon is unable to open /etc/passwd, it unmounts /etc/passwd, revealing the local copy. (Since mounts are stateless, unmounting can be performed without server notification or involvement.)

For each user that is to be able to use a machine when the "home directory server" is not available, a home directory must be created and stocked with essential data files. Similarly, for a machine to be able to use an application when the "application server" is not available, that application must be installed in the client's /usr/lpp, when the server's /usr/lpp is not mounted. The

resulting machine is certainly not as useful as when the servers are available, but it is usable, and much better than no machine at all.

STATEFUL CACHING

Directory Caching

Use of component by component lookup associated with the virtual file systems approach means, in the worst case, that there will be a `lookup()` remote procedure call for each component of the path. To avoid this overhead in typical path searches, the results of `lookup()` calls are cached in kernel memory. Consistent with the above objectives, the intent of the DS design is that cached results will always be valid (and will be invalidated when they are not valid). Since DS performs userid and groupid translations to manage the global set of user id's and group id's, the cache must not allow authorization checks to be bypassed or performed in an inconsistent manner. With these points in mind, we have chosen to limit our cache to directory entries only, since ordinary file entries are likely to be more volatile, and have chosen to only cache directory entries where we are assured that all users have search permission, i.e., all three execute bits are set. Cached results may become invalid because of directory changes in the server. The policy for cache invalidation is as follows. Whenever any directory in a given server is changed, client directory caches are purged, for all clients of that server. Only clients performing a `lookup()` at the given server since the previous directory change are notified, since only those clients could have cached results for the given server. Of course, the clients only purge the entries for the server that had the directory change. The purpose of this strategy is to keep the directory cache entries correct, with little network traffic. (A client will also purge cached results if it loses connection with the server, even temporarily.)

Data Caching

Distributed Services uses data caching in both client and server, to avoid unnecessary network traffic and associated delays. The caching achieves the traditional read ahead, write behind and reuse benefits associated with the kernel buffer cache, but with both client and server caches. As a result, read ahead(write behind) can be occuring in the client cache with regard to the network and in the server cache with regard to the disk. *As a result, disk to disk transfer rates to/from remote machines can be substantially greater than local rates.* In AIX we have methodially tuned the local disk subsystem, yet use of cp for remote files (copying remote file to local file, or vice versa) yields measurably higher disk to disk throughput than for local only files. Note that stateless data caching designs may not support write behind, in order to guarantee that all data will be actually on the server's disk before the write rpc returns to the client.

In general, it is difficult to keep multiple cached data blocks consistent. We chose to implement a state machine based on current opens of a given file. We emphasize that this mechanism is applied at a file granularity, and that it is strictly a performance optimization — the mechanism is designed to preserve the traditional multireader/multiwriter semantics of the Unix file system. Caching is always allowed at the server for a file (the machine where the disk copy is stored), but caching at clients may not be allowed, if such caching conflicts with consistency requirements. See Figure 3.

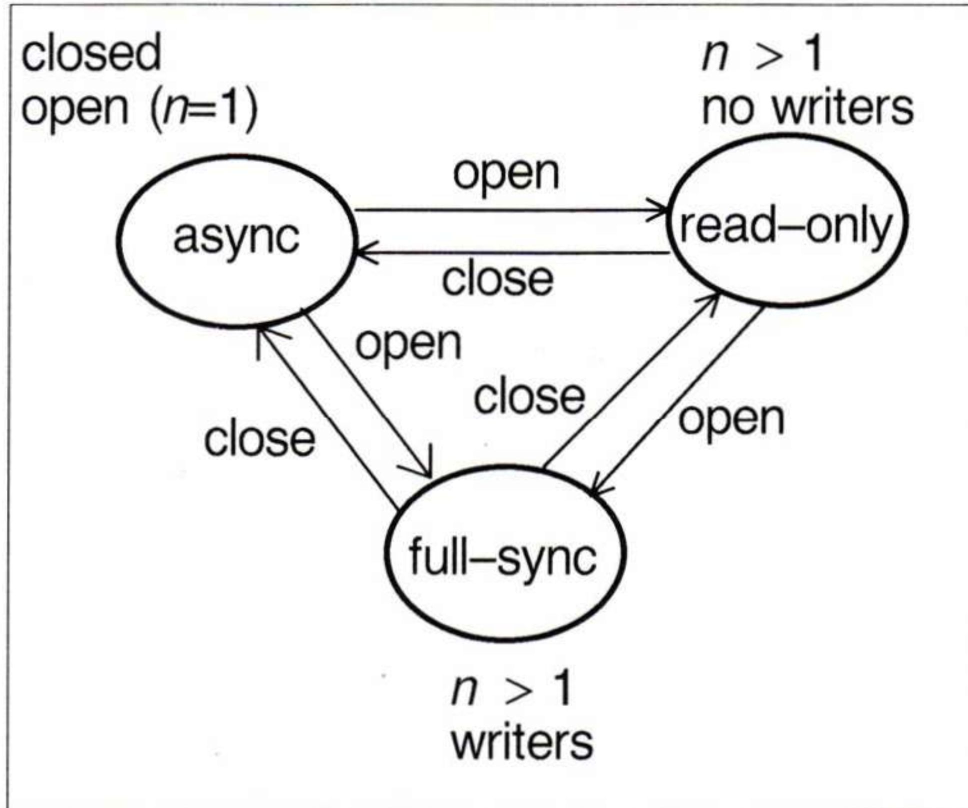

Figure 3. Data Cache Synchronization Modes

For each file, the caching of data for the file is controlled by the current number of *machines* with the file open (n in Figure 3) and the authorities granted at the open. Note that within a given machine, the number of processes with the file open is not relevant, because traditional Unix buffer cache mechanisms are employed. When a file is opened (n goes from 0 to 1), then the machine that has opened the file can cache data for the file, for both readers and writers. This is referred to as "async" mode because the client cache is managed asynchronously with respect to the server cache. When a file is opened, at a given time, by processes on only one machine, this mode allows full read and write caching with little server involvement. We believe that this is typical of actual file usage. When a second machine opens the same file, then the client machines are only allowed to cache data for the file if there are no writers for the file. If there are no writers, then the file is placed in "read–only" mode and clients are allowed to cache. This, like async mode, is very common, if not dominant, in typical file usage. When multiple machines have the file open, with at least one writer, then the file is placed in full–sync mode. In this mode, reads and writes are fully synchronous with respect to the server cache, and data blocks for this file are not kept in client caches. When all processes of a machine have closed a file, then transitions back to states with higher degrees of cach-

ing take place. (In the determination of caching mode, the server must also be counted as a client if processes on the server open the file.)

Close/Reopen Optimization. A frequent scenario is that a file is closed, say by an editor, and then immediately reopened, say by a compiler. Our data cache consistency mechanisms are extended to allow reuse of cached data blocks in the client data cache, if and only if the file is not modified elsewhere between the close and subsequent reopen.

Lock Table Caching

One of our objectives has been that data base applications, e.g., SQL/RT, be able to run effectively on the distributed file system. For this to be true, it is necessary that locking performance be optimized by caching lock tables. For the locking associated with the `lockf()` and `fcntl()` system calls, lock tables are used to keep track of current locks. We use caching mechanisms for lock tables similar to the ones used for data caching. However, there is no state corresponding to the read-only state of Figure 3. When all processes holding or requesting locks for a file are on a single machine, then the lock tables for that file are managed at the machine where the locks are held/requested. This corresponds to the async caching mode for data. Again, we believe this to be the typical case. When processes on more than one machine hold or request locks for a given file, then the lock tables are managed at the server for the file, corresponding to the full-sync data caching mode.

SUMMARY

Where appropriate, e.g., in managing remote mounts, stateless mechanisms function as well as or better than stateful mechanisms, and are simpler to implement. Thus they are clearly preferable. However, where stateless mechanisms are at a disadvantage with regard to stateful mechanisms, e.g., in cache consistency and caching performance, one must make a conscious tradeoff between simplicity of implementation and the advantages of stateful mechanisms.

REFERENCES

1. Charles H. Sauer, Don W. Johnson, Larry Loucks, Amal A. Shaheen-Gouda, Todd A. Smith, "RT PC Distributed Services: Overview," *Operating Systems Review 21,* 3 (July 1987) pp. 18-29.
2. Charles H. Sauer, Don W. Johnson, Larry Loucks, Amal A. Shaheen-Gouda, Todd A. Smith, "RT PC Distributed Services: File System," *;login: 12,* 5 (September/October 1987) pp. 12-22.
3. Jason Levitt, "The IBM RT Gets Connected," *BYTE 12,* 12 (1987) pp. 133-138
4. R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh and B. Lyon, "Design and Implementation of the Sun Network File System," *USENIX Conference Proceedings,* Portland, June 1985.
5. Sun Microsystems, Inc., *Networking on the Sun Workstation,* Feburary 1986.
6. S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Conference Proceedings,* Atlanta, June 1986.
7. T.G. Lang, M.S. Greenberg and C.H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology,* SA23-1057, January 1986.