Distributed Systems: Concepts and Design

Chapter 12 Exercise Solutions

12.1 Why is there no open or close operation in the interface to the flat file service or the directory service. What are the differences between our directory service Lookup operation and the UNIX open?

12.1 Ans.

Because both services are stateless. The interface to the flat file service is designed to make *open* unnecessary. The *Lookup* operation performs a single-level lookup, returning the UFID corresponding to a given simple name in a specified directory. To look up a pathname, a sequence of *Lookups* must be used. Unix *open* takes a pathname and returns a file descriptor for the named file or directory.

12.2 Outline methods by which a client module could emulate the UNIX file system interface using our model file service.

12.2 Ans.

Left to the reader.

12.3 Write a procedure PathLookup(Pathname, Dir) → UFID that implements Lookup for UNIX-like pathnames based on our model directory service.

12.3 Ans.

Left to the reader.

12.4 Why should UFIDs be unique across all possible file systems? How is uniqueness for UFIDs ensured?

12.4 Ans.

Uniqueness is important because servers that may be attached to different networks may eventually be connected, e.g. by an internetwork, or because a file group is moved from one server to another. UFIDs can be made unique by including the address of the host that creates them and a logical clock that is increased whenever a UFID is created. Note that the host address is included only for uniqueness, not for addressing purposes (although it might subsequently be used as a hint to the location of a file).

12.5 To what extent does Sun NFS deviate from one-copy file update semantics? Construct a scenario in which two user-level processes sharing a file would operate correctly in a single UNIX host but would observe inconsistencies when running in different hosts.

12.5 Ans.

After a *write* operation, the corresponding data cached in clients other than the one performing the *write* is invalid (since it does not reflect the current contents of the file on the NFS server), but the other clients will not discover the discrepancy and may use the stale data for a period of up to 3 seconds (the time for which cached blocks are assumed to be fresh). For directories, the corresponding period is 30 seconds, but the consequences are less serious because the only operations on directories are to insert and delete file names.

Scenario: any programs that depend on the use of file data for synchronization would have problems. For example, program A checks and sets two locks in a set of lock bits stored at the beginning of a file, protecting records within the file. Then program A updates the two locked records. One second later program B reads the same locks from its cache, finds them unset, sets them and updates the same records. The resulting values of the two records are undefined.

12.6 Sun NFS aims to support heterogeneous distributed systems by the provision of an operating system-independent file service. What are the key decisions that the implementer of an NFS server for an operating system other than UNIX would have to take? What constraints should an underlying filing system obey to be suitable for the implementation of NFS servers?

12.6 Ans.

The Virtual file system interface provides an operating-system independent interface to UNIX and other file systems. The implementor of an NFS server for a non-Unix operating system must decide on a representation for file handles. The last 64 bits of a file handle must uniquely define a file within a file system. The Unix representation is defined (as shown on page ??) but it is not defined for other operating systems. If the operating system does not provide a means to identify files in less than 64 bits, then the server would have to generate identifiers in response to *lookup*, *create* and *mkdir* operations and maintain a table of identifiers against file names.

Any filing system that is used to support an NFS server must provide:

- efficient block-level access to files;
- file attributes must include write timestamps to maintain consistency of client caches;
- other attributes are desirable, such owner identity and access permission bits.

12.7 What data must the NFS client module hold on behalf of each user-level process?

12.7 Ans.

A list of open files, with the corresponding v-node number. The client module also has a v-node table with one entry per open file. Each v-node holds the file handle for the remote file and the current read-write pointer.

12.8 Outline client module implementations for the UNIX open() and read() system calls, using the NFS RPC calls of Figure 8.3, (i) without, and (ii) with a client cache.

12.8 Ans.

Left to the reader.

12.9 Explain why the RPC interface to early implementations of NFS is potentially insecure. The security loophole has been closed in NFS 3 by the use of encryption. How is the encryption key kept secret? Is the security of the key adequate?

12.9 Ans.

The user id for the client process was passed in the RPCs to the server in unencrypted form. Any program could simulate the NFS client module and transmit RPC calls to an NFS server with the user id of any user, thus gaining unauthorized access to their files. DES encryption is used in NFS version 3. The encryption key is established at *mount* time. The mount protocol is therefore a potential target for a security attack. Any workstation could simulate the mount protocol, and once a target filesystem has been mounted, could impersonate any user using the encryption agreed at mount time..

12.10 After the timeout of an RPC call to access a file on a hard-mounted file system the NFS client module does not return control to the user-level process that originated the call. Why?

12.10 Ans.

Many Unix programs (tools and applications) are not designed to detect and recover form error conditions returned by file operations. It was considered preferable to avoid error conditions wherever possible, even at the cost of suspending programs indefinitely.

12.11 How does the NFS Automounter help to improve the performance and scalability of NFS?

12.11 Ans.

The NFS *mount service* operates at system boot time or user login time at each workstation, mounting filesystems wholesale in case they will be used during the login session. This was found too cumbersome for some applications and produces large numbers of unused entries in mount tables. With the Automounter filesystems need not be mounted until they are accessed. This reduces the size of mount tables (and hence the time to search them). A simple form of filesystem replication for read-only filesystems can also be achieved with the Automounter, enabling the load of accesses to frequently-used system files to be shared between several NFS servers.

12.12 How many lookup calls are needed to resolve a 5-part pathname (for example, /usr/users/jim/code/xyz.c) for a file that is stored on an NFS server? What is the reason for performing the translation step-by-step?

12.12 Ans.

Five *lookups*, one for each part of the name.

Here are several reasons why pathnames are looked up one part at a time, only the first of those listed is mentioned in the book (on p. 229):

i) pathnames may cross mount points;

ii) pathnames may contain symbolic links;

iii) pathnames may contain '..';

iv) the syntax of pathnames could be client-specific.

Case (i) requires reference to the client mount tables, and a change in the server to which subsequent *lookups* are dispatched. Case (ii) cannot be determined by the client. Case (iii) requires a check in the client against the 'pseudo-root' of the client process making the request to make sure that the path doesn't go above the pseudo-root. Case (iv) requires parsing of the name at the client (not in itself a bar to multi-part lookups, since the parsed name could be passed to the server, but the NFSD protocol doesn't provide for that).

12.13 What condition must be fulfilled by the configuration of the mount tables at the client computers for access transparency to be achieved in an NFS-based filing system.

12.13 Ans.

All clients must mount the same filesystems, and the names used for them must be the same at all clients. This will ensure that the file system looks the same from all client machines.

12.14 How does AFS gain control when an open or close system call referring to a file in the shared file space is issued by a client?

12.14 Ans.

The UNIX kernel is modified to intercept local *open* and *close* calls that refer to non-local files and pass them to the local Venus process.

12.15 Compare the update semantics of UNIX when accessing local files with those of NFS and AFS. Under what circumstances might clients become aware of the differences?

12.15 Ans.

UNIX: strict one-copy update semantics;

NFS: approximation to one-copy update semantics with a delay (~3 seconds) in achieving consistency;

AFS: consistency is achieved only on *close*. Thus concurrent updates at different clients will result in lost updates – the last client to close the file wins.

See the solution to Exercise 8.1 for a scenario in which the difference between NFS and UNIX update semantics would matter. The difference between AFS and UNIX is much more visible. Lost updates will occur whenever two processes at different clients have a file open for writing concurrently.

12.16 How does AFS deal with the risk that callback messages may be lost?

12.16 Ans.

Callbacks are renewed by Venus before an *open* if a time T has elapsed since the file was cached, without communication from the server. T is of the order of a few minutes.

12.17 Which features of the AFS design make it more scalable than NFS? What are the limits on its scalability, assuming that servers can be added as required? Which recent developments offer greater scalbility?

12.17 Ans.

The load of RPC calls on AFS servers is much less than NFS servers for the same client workload. This is achieved by the elimination of all remote calls except those associated with *open* and *close* operations, and the use of the *callback* mechanism to maintain the consistency of client caches (compared to the use of *getattributes* calls by the clients in NFS). The scalability of AFS is limited by the performance of the single server that holds the most-frequently accessed file volume (e.g. the volume containing */etc/passwd*, */etc/hosts*, or some similar system file). Since read-write files cannot be replicated in AFS, there is no way to distribute the load of access to frequently-used files.

Designs such as xFS and Frangipani offer greater scalability by separating the management and metadata operations from the data handling, and they reduce network traffic by locating files based on usage patterns.