Andrew File System

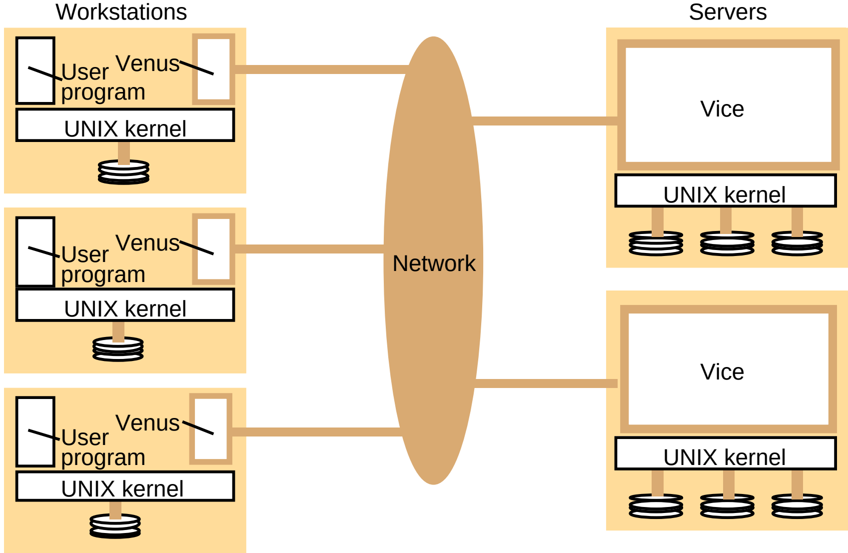
- **Andrew File System (AFS)**
  - started as a joint effort of Carnegie Mellon University and IBM
  - today basis for DCE/DFS: the distributed file system included in the Open Software Foundations's Distributed Computing Environment
  - some UNIX file system usage observations, as pertaining to caching
    - infrequently updated shared files and local user files will remain valid for long periods of time (the latter because they are being updated on owners workstations)
    - allocate large local disk cache, e.g., 100 MByte, that can provide a large enough working set for all files of one user such that the file is still in this cache when used next time
    - assumptions about typical file accesses (based on empirical evidence)
      - usually small files, less than 10 Kbytes
      - reads much more common than writes (appr. 6:1)
      - usually sequential access, random access not frequently found
      - user-locality: most files are used by only one user
      - burstiness of file references: once file has been used, it will be used in the nearer future with high probability

Andrew File System

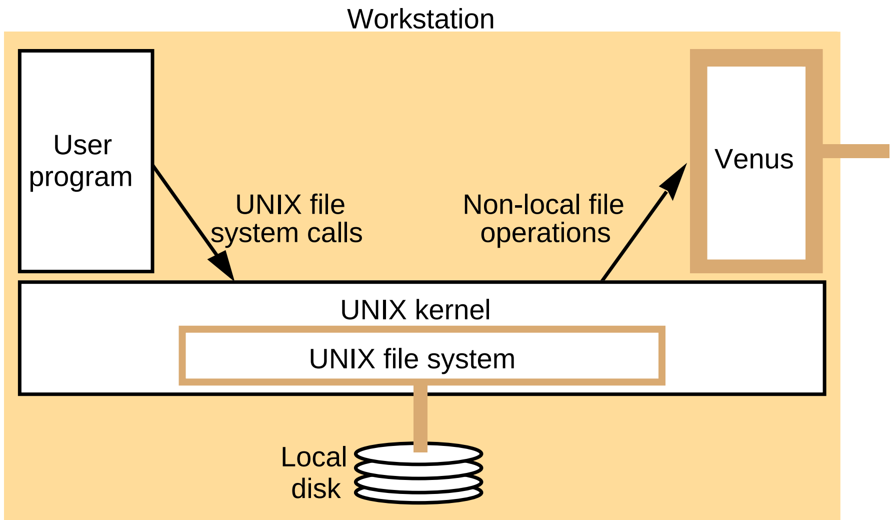
- **Andrew File System (AFS)**
  - design decisions for AFS
    - whole-file serving: entire contents of directories and files transfered from server to client (AFS-3: in chunks of 64 Kbytes)
    - whole file caching: when file transfered to client it will be stored on that client's local disk

Andrew File System

- **AFS architecture: Venus, network and Vice**



- **AFS system call intercept, handling by Venus**



Andrew File System

- Implementation of file system calls - callbacks and callback promises

User process	UNIX kernel	Venus	Net	Vice
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid ***callback promise***, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a ***callback promise*** to the workstation. Log the callback promise.
		Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
	Open the local file and return the file descriptor to the application.			
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a ***callback*** to all other clients holding *callback promises* on the file.

Andrew File System

- **Callback mechanism**
  - ensures that cached copies of files are updated when another client performs a close operation on that file
  - callback promise
    - token stored with cached file
    - status: *valid* or *cancelled*
  - when server performs request to update file (e.g., following a `close`), then it sends callback to all Venus processes to which it has sent callback promise
    - RPC from server to Venus process
    - Venus process sets callback promise for local copy to *cancelled*
  - Venus handling an `open`
    - check whether local copy of file has *valid* callback promise
    - if *canceled*, fresh copy must be fetched from Vice server

Andrew File System

- **Callback mechanism**
  - Restart of workstation after failure
    - retain as many locally cached files as possible, but callbacks may have been missed
    - Venus sends cache validation request to the Vice server
      - contains file modification timestamp
      - if timestamp is current, server sends *valid* and callback promise is reinstantiated with *valid*
      - if timestamp not current, server sends *cancelled*
  - Problem: communication link failures
    - callback must be renewed with above protocol before new open if a time T has lapsed since file was cached or callback promise was last validated
  - Scalability
    - AFS callback mechanism scales well with increasing number of users
      - communication only when file has been updated
      - in NFS timestamp approach: for each open
    - since majority of files not accessed concurrently, and reads more frequent than writes, callback mechanism performs better

Andrew File System

- **File Update Semantics**
  - to ensure strict one-copy update semantics: modification of cached file must be propagated to any other client caching this file before any client can access this file
    - rather inefficient
  - callback mechanism is an approximation of one-copy semantics
  - guarantees of currency for files in AFS (version 1)
    - after successful open: latest(F, S)
      - current value of file F at client C is the same as the value at server S
    - after a failed open/close: failure(S)
      - open close not performed at server
    - after successful close: updated(F, S)
      - client's value of F has been successfully propagated to S

Andrew File System

- **File Update Semantics in AFS version 2**
  - Vice keeps callback state information about Venus clients: which clients have received callback promises for which files
  - lists retained over server failures
  - when callback message is lost due to communication link failure, an old version of a file may be opened after it has been updated by another client
  - limited by time T after which client validates callback promise (typically, T=10 minutes)
  - currency guarantees
    - after successful open:
      - latest(F, S, 0)
        - copy of F as seen by client is no more than 0 seconds out of date
      - or (lostCallback(S, T) and inCache(F) and latest(F, S, T))
        - callback message has been lost in the last T time units,
        - the file F was in the cache before open was attempted,
        - and copy is no more than T time units out of date

Andrew File System

- **Cache Consistency and Concurrency Control**
  - AFS does not control concurrent updates of files, this is left up to the application
    - deliberate decision, not to support distributed database system techniques, due to overhead this causes
  - cache consistency only on open and close operations
    - once file is opened, modifications of file are possible without knowledge of other processes' operations on the file
    - any close replaces current version on server
      - all but the update resulting from last close operation processed at server will be lost, without warning
    - application programs on same server share same cached copy of file, hence using standard UNIX block-by-block update semantics
  - although update semantics not identical to local UNIX file system, sufficiently close so that it works well in practice

 tele

Enhancements

- **Spritely NFS**
  - goal: achieve precise one-copy update semantics
  - abolishes stateless nature of NFS -> vulnerability in case of server crashes
  - introduces open and close operations
    - open must be invoked when application wishes to access file on server, parameters:
      - modes: read, write, read/write
      - number of local processes that currently have the file open for read and write
    - close
      - updated counts of processes
  - server records counts in open files table, together with IP address and port number of client
  - when server receives open: checks file table for other clients that have the same file open
    - if open specifies write,
      - request fails if any other client has file open for writing,
      - otherwise other read clients are instructed to invalidate local cache copy
    - if open specifies read,
      - sends callback to other write clients forcing them to modify their caching strategy to *write-through*
      - causes all other read clients to read from server and stop caching

Enhancements

- **WebNFS**
  - access to files in WANs by direct interaction with remote NFS servers
  - permits partial file accesses
    - http or ftp would require entire files to be transmitted, or special software at the server end to provide only the data needed
  - access to "published" files through public file handle
  - for access via path name on server, usage of lookup requests
  - reading a (portion of) a file requires
    - TCP connection to server
    - lookup RPC
    - read RPC
- **NFS version 4**
  - similarly for WANs
  - usage of callbacks and leases
  - recovery from server faults through transparent moving of file systems from one server to another
  - usage of proxy servers to increase scalability