AFS Early Prototype

- Venus: client, user-level process, intercept file open and close calls
- Vice: server, one server process per client
  - shared data structure put in files, since processes can't share memory
  - .admin directory hierarchy, have stub directories
  - Vice-Venus interface name file by full path-name
  - validation of cached copies upon every use

Performance Problems of the Prototype

- Problems:
  - stat calls expensive
  - server context switches, network resource exhausted, CPU bound
- Quantify the problem: AFS benchmark
  - makedir, copy, scandir, ReadAll, make
  - server saturated at about at 5 instances of the benchmark

Identifying Performance Problems

- Top calls to the AFS server: TestAuth, GetFileStat
- Cache hit ratio: > 80%
- server load spread 5 to 1
- Why CPU bound?
  - context switch
  - namei resolution

Changes for Better Performance

- Cache Management
- Name resolution
- Communication and server process structure
- Low-level storage representation

Changes for Better Performance

- cache files on disk; cache attributes in memory
- use invalidation (callbacks)
  - race conditions in breaking callbacks
  - network failures
- client pathname lookup
  - clients cache entire directory at a time
- reducing context switches

Characteristics of AFS

- Properties of AFS:
  - ***scalability***
  - network transparency
  - location independence
  - user mobility and file mobility
  - fault tolerance

Scalability in Distributed FS

- network traffic
- server load

AFS Architecture

- reducing network traffic
  - clients cache recently accessed files on local disks
- delegating work to clients:
  - clients cache entire directories and perform name lookup
- dedicated servers for each cluster in the network

Volumes

- a logical unit of file storage; forms a subtree in the shared file hierarchy
- separating volumes from disk partitions:
  - advantages
  - volume location database
- uniform name space
  - file identifier
  - unique in time and space?

Consistency Semantics

- session semantics
  - do you really have to check on every read and write access to implement UNIX sharing semantics?
- strict consistency for metadata

Results of the Changes

- system scale much better
- CPU load significantly reduced
- gettime becomes the most frequently used call
- met the goal of 50 users per server

Changes for Better Admin

- security:
  - Kerberos
  - access control lists
    - read, write, lock, lookup, insert, delete, administer
    - retain UNIX permission bits as well
- clone of volumes for migration and backup
- organized update of read-only replicas

Compare with Other DFSs

- caching of whole files
  - advantages and disadvantages
- comparison with NFS
  - later versions of NFS caches attributes and do not check data validity on every open
  - NFS is slow, has a higher CPU load, disk load, and network load
  - but, it doesn't require workstations to have local disks

A Few More Years Later...

- Even more changes
  - what we advocated wasn't quite right…
  - whole file caching -> cache 64K chunks
  - discard stale data at next open -> discard stale data immediately

DCE DFS

- strict UNIX sharing semantics
  - data tokens, status tokens, open tokens, lock tokens
  - token manager operates at V_OP macro level
- both client and server uses vnode/vfs to increase interoperability with other fs
- client caches result of individual lookups
- complex and difficult to implement