Recap

NFS

- Stateless Server
- Server's Commands:
  - Lookup, Read, Write, GetAttr
  - Idempotency
- How to implement FS API based on the above commands

Client-Side Caching

Sending every read and write request over network has big performance penalty, orders of magnitude slower than a local file system

Locality observed in typical file accesses, therefore obvious solution is to add a **cache** on the client

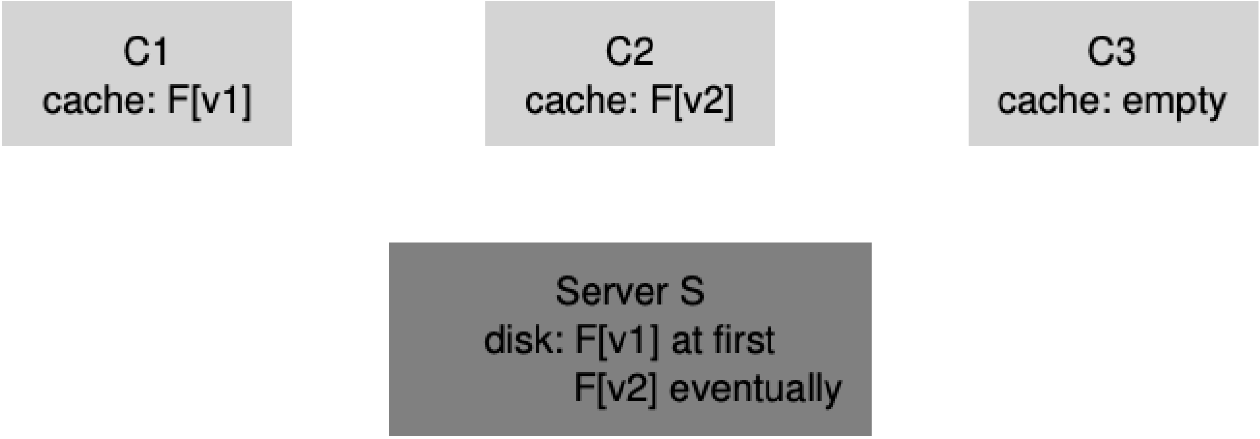
Recently accessed file data is kept in client cache so it can be quickly read again

Write buffering means write goes to cache first and then later the changes are pushed to the server

Advantage: client responds quickly to a write system call, doesn’t need to block application for network operation

Cache Consistency Problem

Big problem: **cache consistency**



Example 1:

C1 reads file F

C2 overwrites file F

C3 reads file F

What version of F does C3 get?

When client can't get most recent version of file from server it is an **update visibility** cache consistency problem

Example 2:

C1 reads file F

C2 overwrite file F

C2 flushes cache to the server

C1 reads again from file F

What version of F does C1 read the second time?

When client reads from out-of-date cache it is a **stale cache** consistency problem

Addressing Update Visibility

Flush-on-close semantics means cache is always flushed when the application closes a file

Ensures that subsequent opens from another node will see the latest file version

Not perfect solution, update visibility problem still exists, but is mitigated for common file usage patterns

Addressing Stale Cache

Check if file has changed before using cached contents of file

The GETATTR command will indicate time of last modification to file

Results in a flood of GETATTR commands, solution is to add a local **attribute cache** that updates contents only after a timeout

AFS

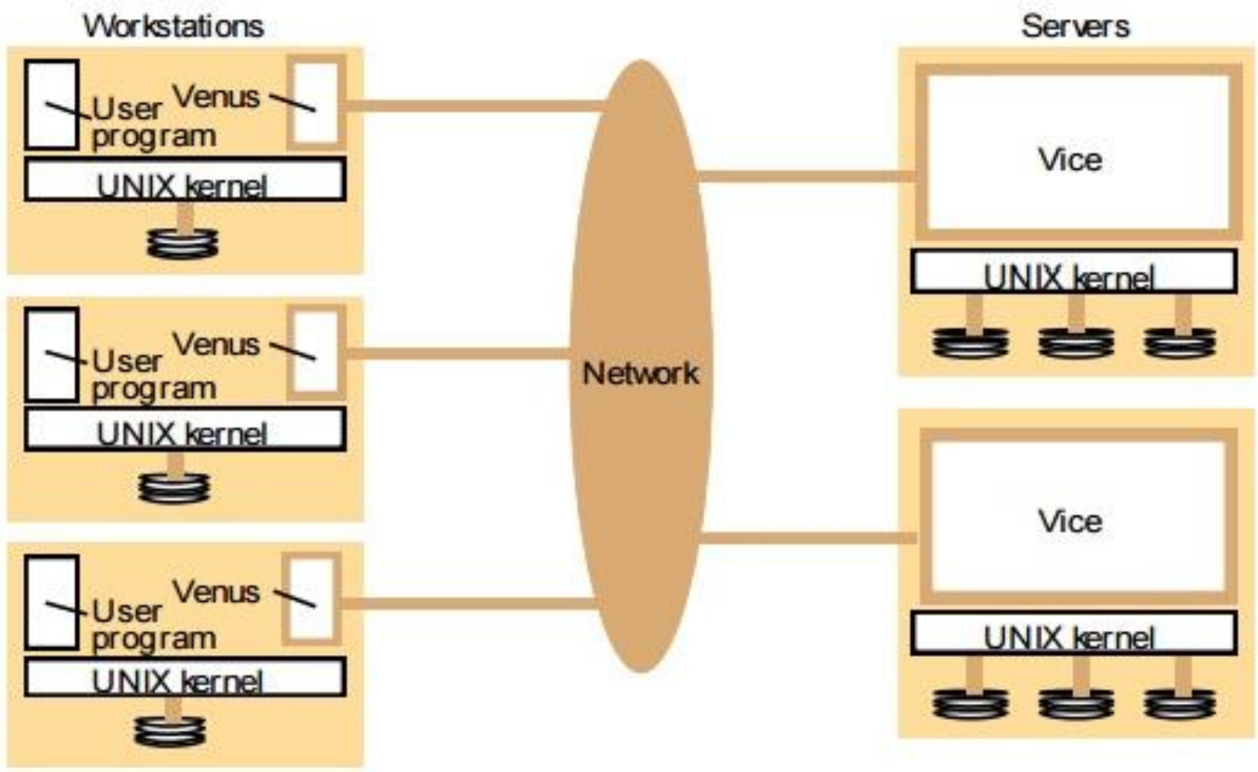
Based on Ch. 49

Slides revised from
Matthew Tancreti
Iowa State University

Architecture

Client-side code is Venus and server-side is Vice

Both sides take advantage of existing Unix file system to store files



Client/Server Protocol

TestAuth	Test whether a file has changed (used to validate cached entries)
GetFileStat	Get the stat info for a file
Fetch	Fetch the contents of file
Store	Store this file on the server
SetFileStat	Set the stat info for a file
ListDir	List the contents of a directory

Fetch/Store entire file

Whole-File Caching

Performs **whole-file caching** on local disk

1. When application calls open() the entire contents are copied to the local disk
2. All read() and write() operations are performed only on the local copy of the file
3. On close() file is flushed back to server

File is kept in cache even after flush to server, if it is opened again client sends TestAuth to server to check if local copy is out-of-date

Application System Calls	Client Action	Message to Server
open()	check if already in cache	TestAuth
		Fetch
read()	send read() to local file	
write()	send write() to local file	
close()		Store

Problems with Version 1

Path-traversal cost too high

- Fetch command contains the entire absolute path
- Requires server to traverse path every time

Too many TestAuth messages

- Same issue that NFS had with GETATTR messages
- Servers spending to much time responding to TestAuth messages
- Most of the time the response is that the file has not changed

These problems resulted in server CPU becoming a bottleneck (a server could handle only about 20 clients)

Callback

To solve problem of too many AuthTest messages, use **callback** to reduce number of interactions with server

Sever promises to inform client when a file is modified

Callback is similar to idea of interrupts (wait until an event happens)

Different from NFS approach which is more like polling

File Identifier (FID)

To solve problem of too many path traversals on server, use **file identifier** (FID)

Similar in concept to NFS file handle

Client no longer requires server to traverse absolute path every time

Example of Reading a File

Client (C_1)	**Server**
fd = open("/home/remzi/notes.txt", ...); Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX); perform local `read()` on cached copy	
close(fd); do local `close()` on cached copy if file has changed, flush to server	

Example Opening a File the Second Time

```
fd = open("/home/remzi/notes.txt", ...);
 Foreach dir (home, remzi)
  if (callback(dir) == VALID)
   use local copy for lookup(dir)
  else
   Fetch (as above)
 if (callback(notes.txt) == VALID)
  open local cached copy
  return file descriptor to it
 else
  Fetch (as above) then open and return fd
```

Cache Consistency

	Client$_1$		Client$_2$		Server	*Comments*
P$_1$	P$_2$	Cache	P$_3$	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	

Open after close is always consistent

Update Visibility

Different machines

Client$_1$ P$_1$	P$_2$	Cache	Client$_2$ P$_3$	Cache	Server Disk	*Comments*
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() $\rightarrow$ B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() $\rightarrow$ A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() $\rightarrow$ B	B	B	
		B	close()	B	B	

Processes share cache,
so consistent

Not consistent with Client1 until after close

Last to Close Wins

	Client$_1$		Client$_2$		Server	*Comments*
P_1	**P_2**	**Cache**	**P_3**	**Cache**	**Disk**	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P_3 the last writer wins
		D	read() $\rightarrow$ D	D	D	
		D	close()	D	D	

Client 1 overwrites Client 2 changes

Recovery After Crash

What if server sends callback while client is rebooting?

- Client must consider all cache suspect after reboot

What if server crashes?

- Callbacks are kept in memory and are lost
- Clients must have some way of realizing that server crashed

Heartbeat protocol, client sends periodic message and expects response

Performance of AFS vs NFS

Workload	NFS	AFS	$\frac{\text{AFS}}{\text{NFS}}$
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$