AFS

How to design a scalable distributed file system?

COM S 352
Iowa State University
Matthew Tancreti

AFS

AFS introduced by CMU researchers in 1980s

Designed with goal of **scalability**, server should support as many clients as possible

Main idea is whole-file caching, different than NFS which caches blocks of data

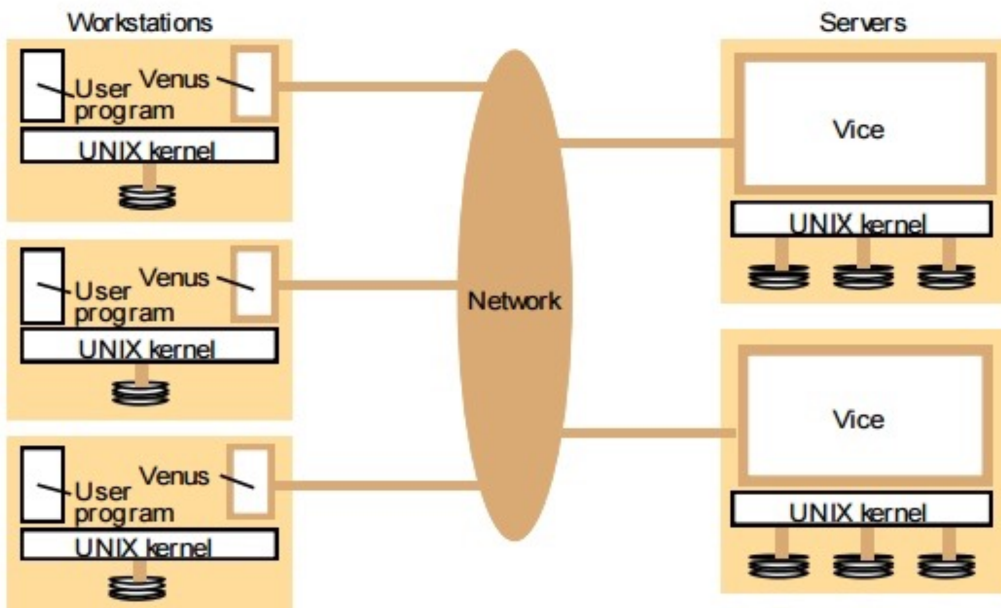
How to design a scalable distributed file system?

OpenAFS logo. [source]

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in cache even after flush to server, if it is opened again client sends TestAuth to server to check if local copy is out-of-date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

Load not balanced across servers

- Some server get much more traffic because they have more popular files

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

Volumes

Attempt to solve load balancing problem - directories are mounted to **volumes**

Administrator can move volumes across servers to balance load

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() $\rightarrow$ A	A		-	A	
	close()	A		-	A	

Open after close is always consistent

Update Visibility

Different machines

Client$_1$			Client$_2$		Server	*Comments*
P$_1$	**P$_2$**	**Cache**	**P$_3$**	**Cache**	**Disk**	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → 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() → B	B	B	
		B	close()	B	B	

Processes share cache,
so consistent

Not consistent with Client1 until after close

Last Writer 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		$\cancel{C}$	D	
		D	open(F)	D	D	Unfortunately for P$_3$
		D	read() $\rightarrow$ D	D	D	the last writer wins
		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 is 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}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$