THE ANDREW FILE SYSTEM

Hayder Hamandi
Wayne State University
8 PUBLICATIONS **0** CITATIONS
SEE PROFILE

THE ANDREW FILE SYSTEM

BY: HAYDER HAMANDI

PRESENTATION OUTLINE

- Brief History
- AFSv1
  - How it works
  - Drawbacks of AFSv1
  - Suggested Enhancements
- AFSv2
  - Newly Introduced Notions
    - ✓ Callbacks
    - ✓ FID
  - Cache Consistency
    - ✓ Between Different Machines
    - ✓ On the Same Machine
    - ✓ The “Last Writer Wins” Approach
  - Crash Recovery
- AFSv2 vs NFS
- Summary

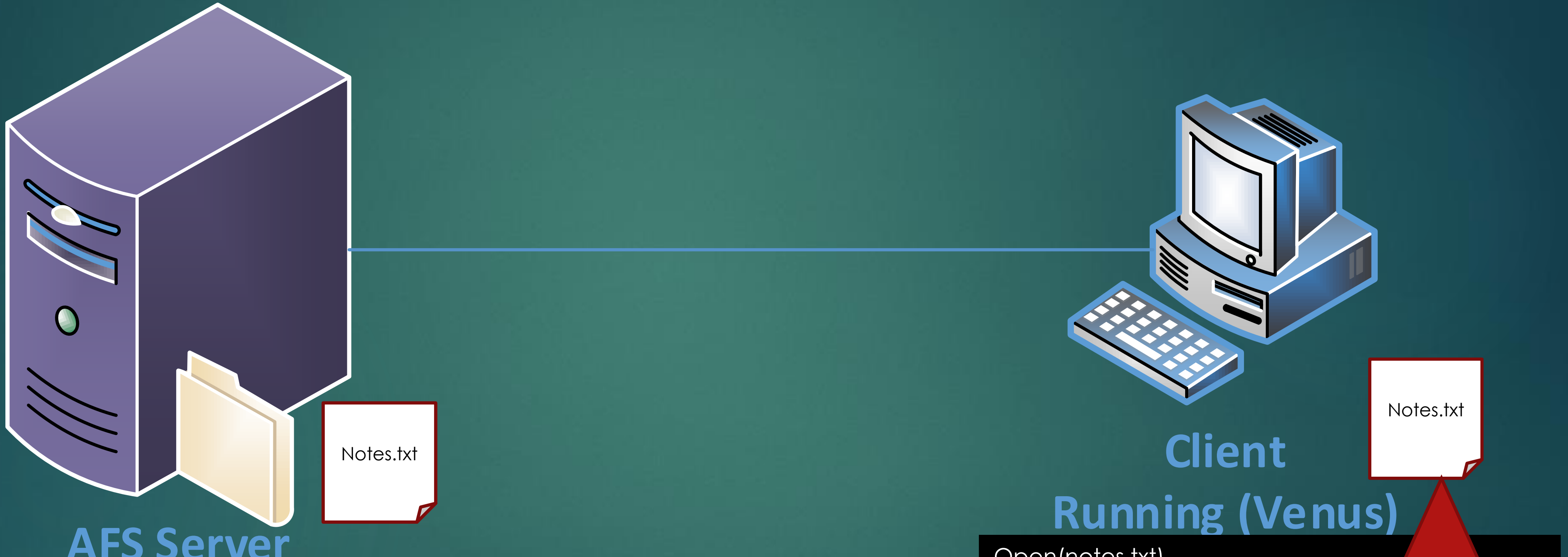
Brief History

- Introduced by researchers at CMU in the 1980's
- Project led by Prof. M. Satyanarayanan ("Satya" for short)
- Goals of project:
  - Create a scalable DFS to support as many clients as possible.
  - Design and implementation of protocol between clients and servers in an DFS environment.
  - Implement simple cache consistency.
  - Overcome the **limited scalability** problem facing **NFS**. NFS forces clients to check with server periodically to determine if cached contents have changed; each check uses resources (CPU, net. Bandwidth). Thus, frequent checks will limit the no. of clients a server can handle.

AFSv1

- Actual name is ITC DFS
- How it works:
  - Basic idea is **whole-file caching** on the local disk of the client machine.
  - When you **open()** a file, the entire file is fetched from the server and stored on your local disk.
  - Subsequent **read()** and **write()** operations are done locally without any network communication. In other words, treated as LFS.
  - Locally the file is cached as blocks in client memory (somewhat similar to NFS)
  - Once a **close()** command is issued, the file (<u>if modified</u>) is flushed back to the server.

AFSv1: How it works (contd.)



AFS Server
Running (Vice)

Find path to: /home/remzi/notes.txt
Send notes.txt to client
Notes.txt has not been modified

Open(notes.txt)
Files is a remote file:
Fetch(/home/remzi/notes.txt)
Read()
Write()
Close(notes.txt)
Store(/home/remzi/notes.txt)
Open(notes.txt)
TestAuth(/home/remzi/notes.txt)

Drawbacks of AFSv1

- Path-traversal costs are too high:
  - In **Fetch()** and **Store()**, client sends the entire pathname to the server.
  - Server must perform full pathname traversal looking in all the directories in the path. Ex: find **root** directory, then **home**, then **remzi**, and so on.
  - With multiple clients, the server will spend all of its CPU time simply walking down directory paths.
- Client issues too many **TestAuth** protocol messages:
  - Large amount of traffic just to check whether a file is valid or not.
  - Server spend to much CPU time to check the file validity.
  - Most of the time, the reply was the file is valid.
- CPU became bottleneck and can only handle 20 clients.
- Load was not balanced across servers.
- Server used single distinct process per client. Context switching overhead.

Suggested Enhancements

- Solve the load imbalance problem by introducing **volumes**.
- Solve the context switching problem by building threads instead of processes. And we know that switching between threads (LWP) creates much less overhead.
- Minimize the number of server interactions to solve the CPU bottleneck problem.
- Many enhancements led to the introduction of AFSv2.

AFSv2

- **Callbacks**:
  - Reduce the number of client/server interactions.
  - More scalable: each server supports up to 50 clients.
  - A promise from the server to the client that the former will inform the latter when a file the latter is caching has been modified.
  - Client no longer needs to contact the server to find out if a cached file is valid.
  - Client assumes that the file is valid unless the server tells it otherwise.
  - Interrupts are used instead of polling.

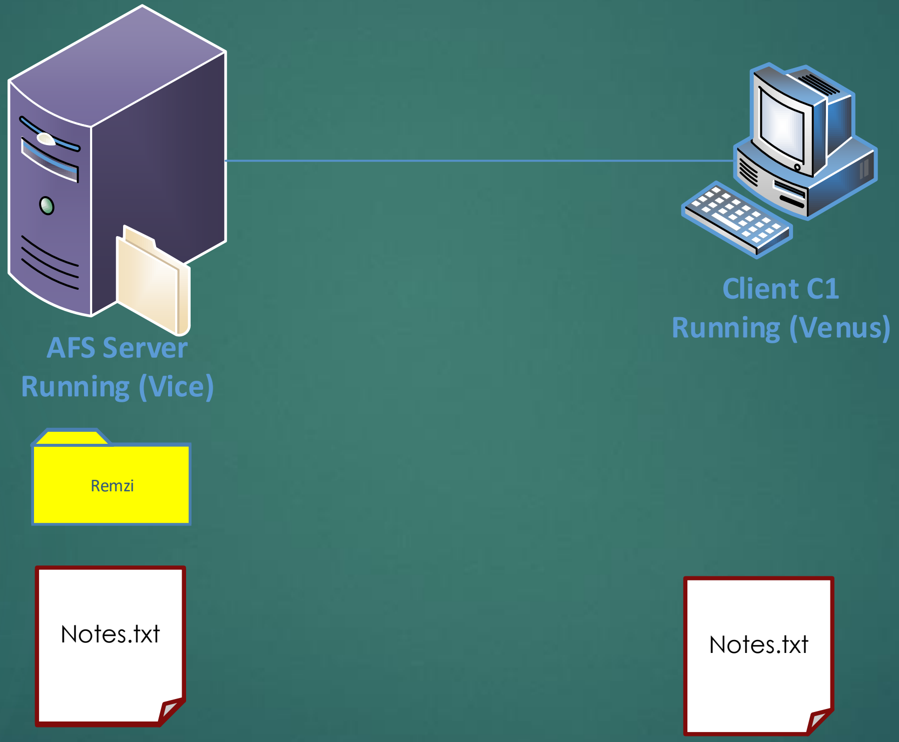
AFSv2 (contd.)

- **File Identifier (FID)**:
  - Used instead of pathnames to specify files.
  - Consist of:
    - ✓ Volume Identifier
    - ✓ File identifier
    - ✓ Uniquifier (to enable reuse of the volume and file IDs when a file is deleted).
  - Client walks pathname, one piece at a time, caching the results and reducing the load on the server.
  - Ex: Client want to access the file (/home/remzi/notes.txt)
    - ✓ Fetch directory contents of **home** and load them into local disk cache
    - ✓ Setup callback on **home**
    - ✓ Fetch directory **remzi**, put on local disk and setup callback
    - ✓ Fetch **notes.txt**, cache to local disk and setup callback
    - ✓ Return file descriptor to the calling application

AFSv2: How it works

```
Receive Fetch request
    look for remzi in home dir
    establish callback(C1) on remzi
    return remzi's content and FID

Receive Fetch request
      look for notes.txt in remzi dir
      establish callback(C1) on notes.txt
      return notes.txt's content and FID
```



```
fd = open("/home/remzi/notes.txt", ...);
      Send Fetch (home FID, "remzi")

Receive Fetch reply
      write remzi to local disk cache
      record callback status of remzi
Send Fetch (remzi FID, "notes.txt")

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

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
```

AFSv2 Cache Consistency

- **Update Visibility**: When will the server be updated with a new version of the file
- **Cache Staleness**: Once the server has a new version, how long before clients see the new version instead of an older cached copy?
- Case 1: Consistency between processes on **different machines**
- Case 2: Consistency between processes on the **same machine**
- Case 3 (Cross-machine): processes on different machines modifying the same file at the same time.

AFSv2 Cache Consistency (contd.)

Case 1: Different Machines

- Client updates (writes) to a file are not visible to the server as well as other machines until the files is closed.
- When an updated file is closed by a client, AFS makes updates visible on the server and invalidates cached copies simultaneously.
- Subsequent opens of the file will require a re-fetch of the new version of the file from the server.

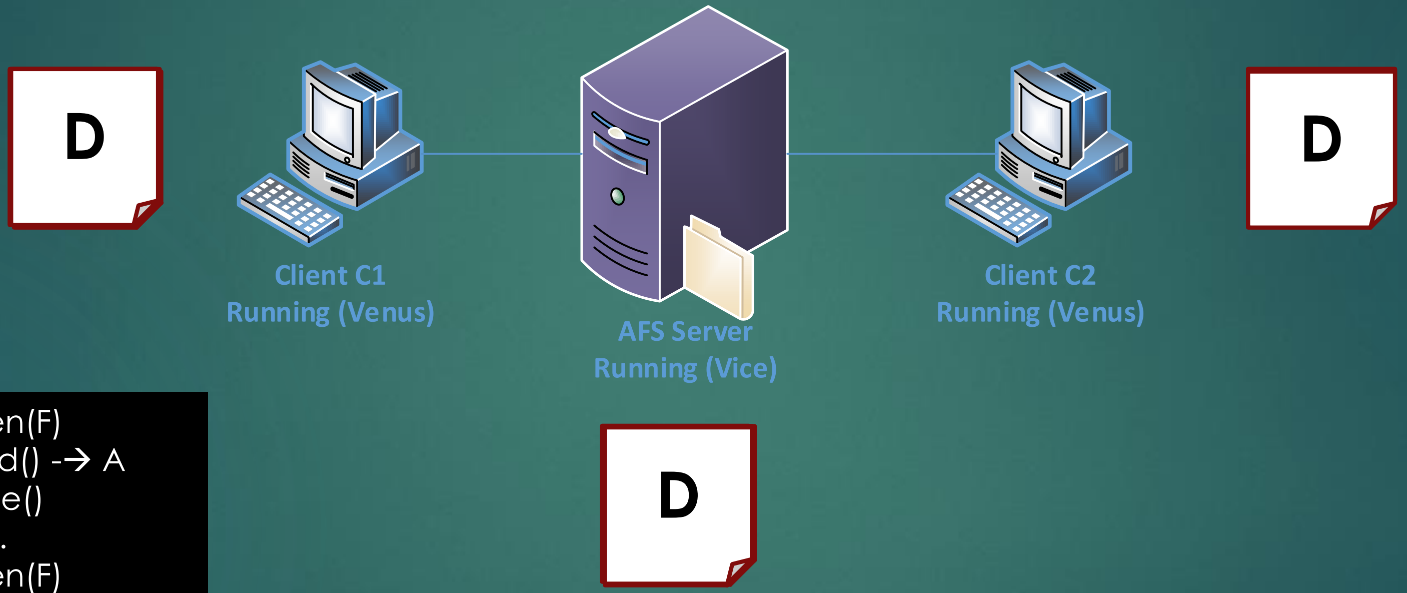
Case 2: Same Machine

- Client updates (writes) are immediately visible to other processes on the same machine (local processes).
- Processes do not have to wait until the file is closed to see the latest updates.

Case 3: The Last Writer Wins Approach (Last Closer Wins)

Open(F)
Write(A)
Close()
…….
Open(F)
Write(B)
…….
Close()
…….
Open(F)
Write(D)
Close()

P1



Open(F)
Read() -→ A
Close()
…….
Open(F)
Read() -→ B
Close()

P2

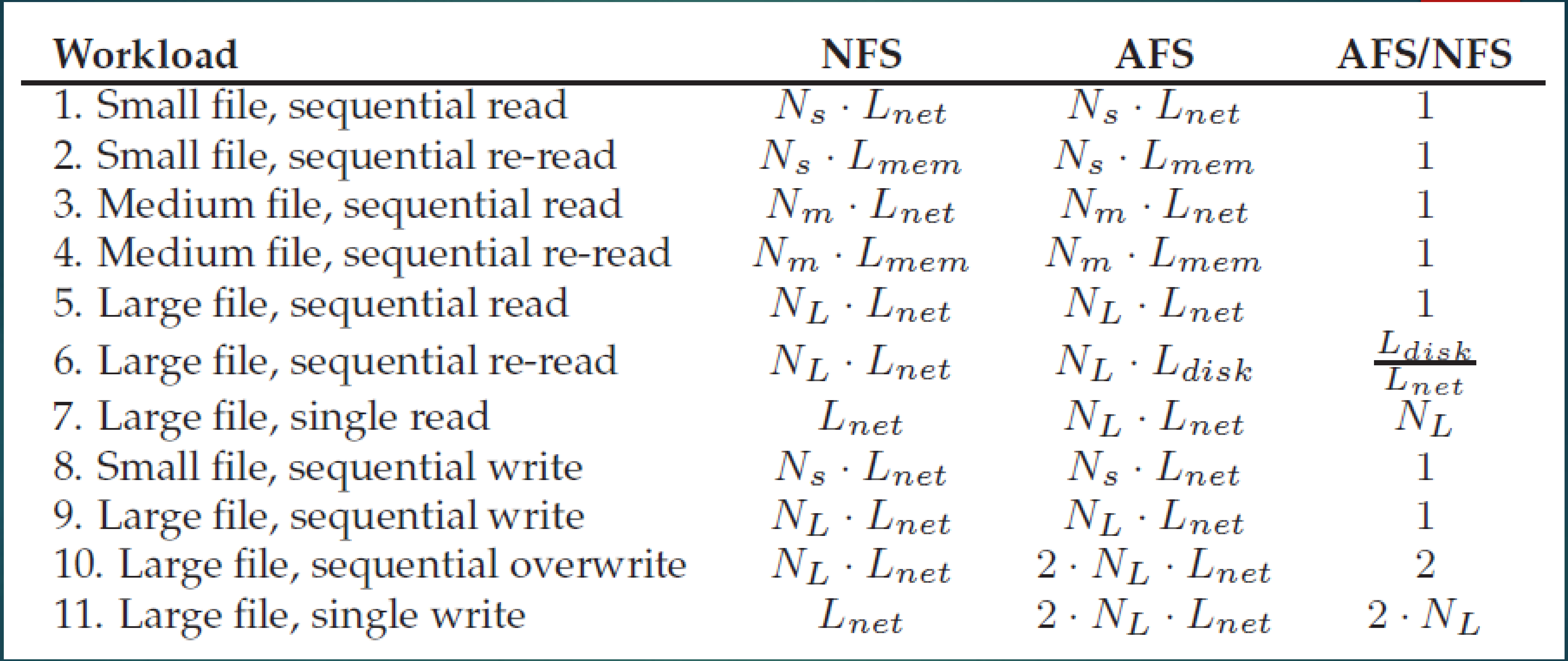
Open(F)
Read() -→ A
Close()
……..
Open(F)
Read() --→ B
Close()
Open(F)
Write(C)
Close()
Open(F)
Read() --→ D
Close()

P3

Crash Recovery

- Client Recovery: Imagine same scenario above (Server S with two clients C1 and C2)
  - C1, C2 and S had the same cached copy of file F.
  - C1 has rebooted the machine.
  - Meanwhile, C2 has updated the file F and flushed the new version of the file back to S.
  - S tried to contact C1 via callback in order to inform C1 to invalidate its copy of F. However, since C1 is rebooting it is unable to receive the invalidation messages from S.
  - Now that C1 is back online, C1 should treat all its cached contests as suspect. Thus, C1 will send TestAuth protocol message to S to check whether its copy of F is still valid for use; if not, C1 will fetch the newer version of F from S.
- Server Recovery:
  - When a crashed server reboot, it has no idea which client has which files.
  - Each client must realize that the server has crashed, and thus treat all their cached contents as suspect.
  - Can be implemented in two ways:
    - ✓ Server send messages to all clients "**Don't trust your cache contents**" after it recovers from the crash
    - ✓ Clients check that the server is alive periodically using **Heartbeat** messages

AFS vs NFS



Workload	NFS	AFS	AFS/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}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

N_s: #Blocks for **small** file, N_m: #Blocks for **medium** file, N_L: #Blocks for **large** file.

L_{net}: Access time for 1 **remote** block, L_{disk}: Access time for 1 block on **local disk**, L_{mem}: Access time for 1 block in **memory**

$L_{net} > L_{disk} > L_{mem}$

Summary

- AFS minimizes client/server interactions through whole-file caching and callbacks.
- Supports many clients; thus, reducing the number of servers needed.
- Consistency model is simple to understand.
- AFS provides global namespace to clients, thus ensuring all files were named the same way on all client machines. However, a program that uses a particular pathname to access a file on AFS must be changed in order to run on a different client, due to the fact that mount points differ from one machine to another.
- Incorporates security mechanisms to authenticate users.
- Provides flexible user managed access control to files.
- Adds tools to enable simpler management of servers by system administrators.

THANK YOU