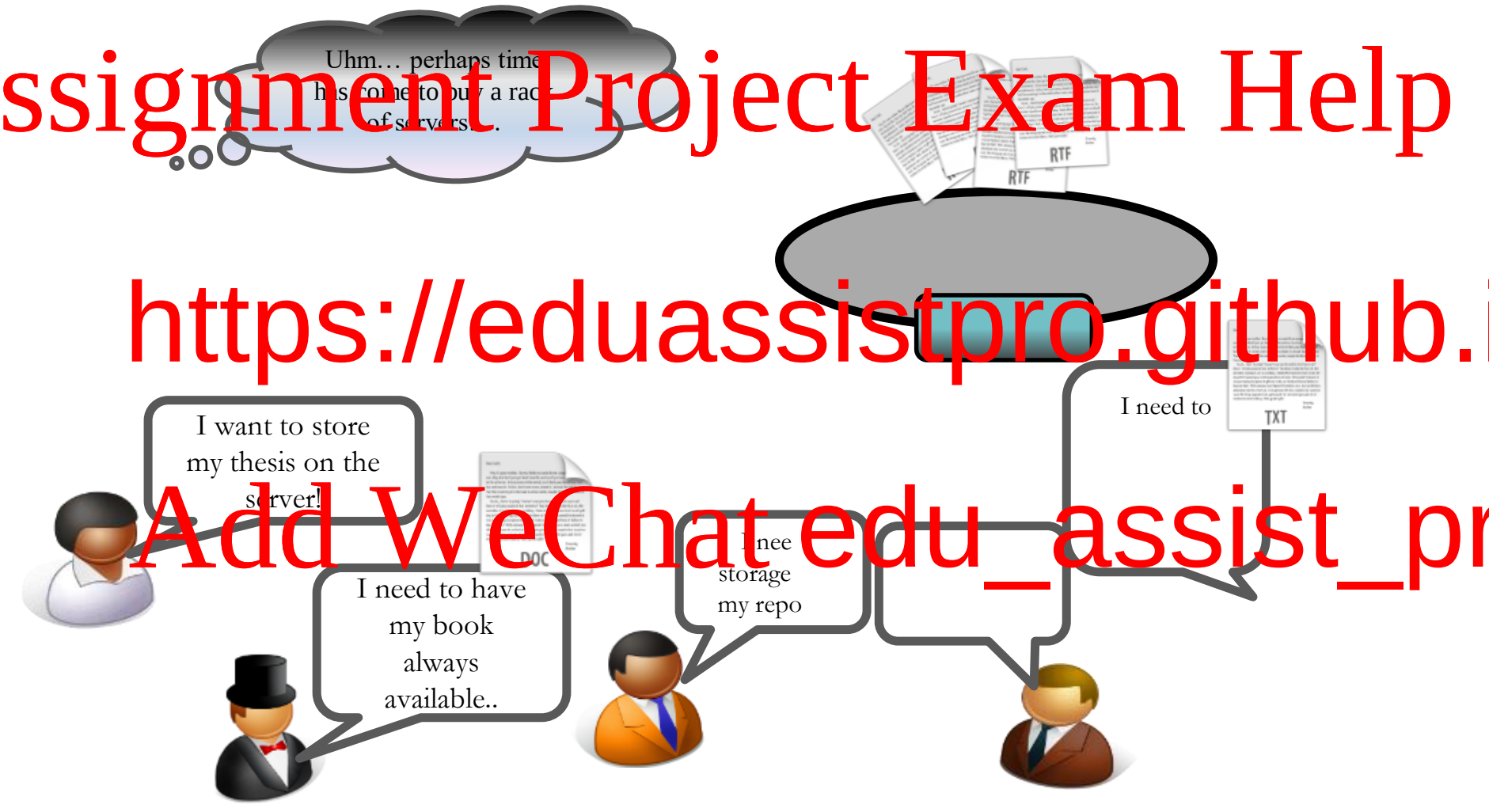
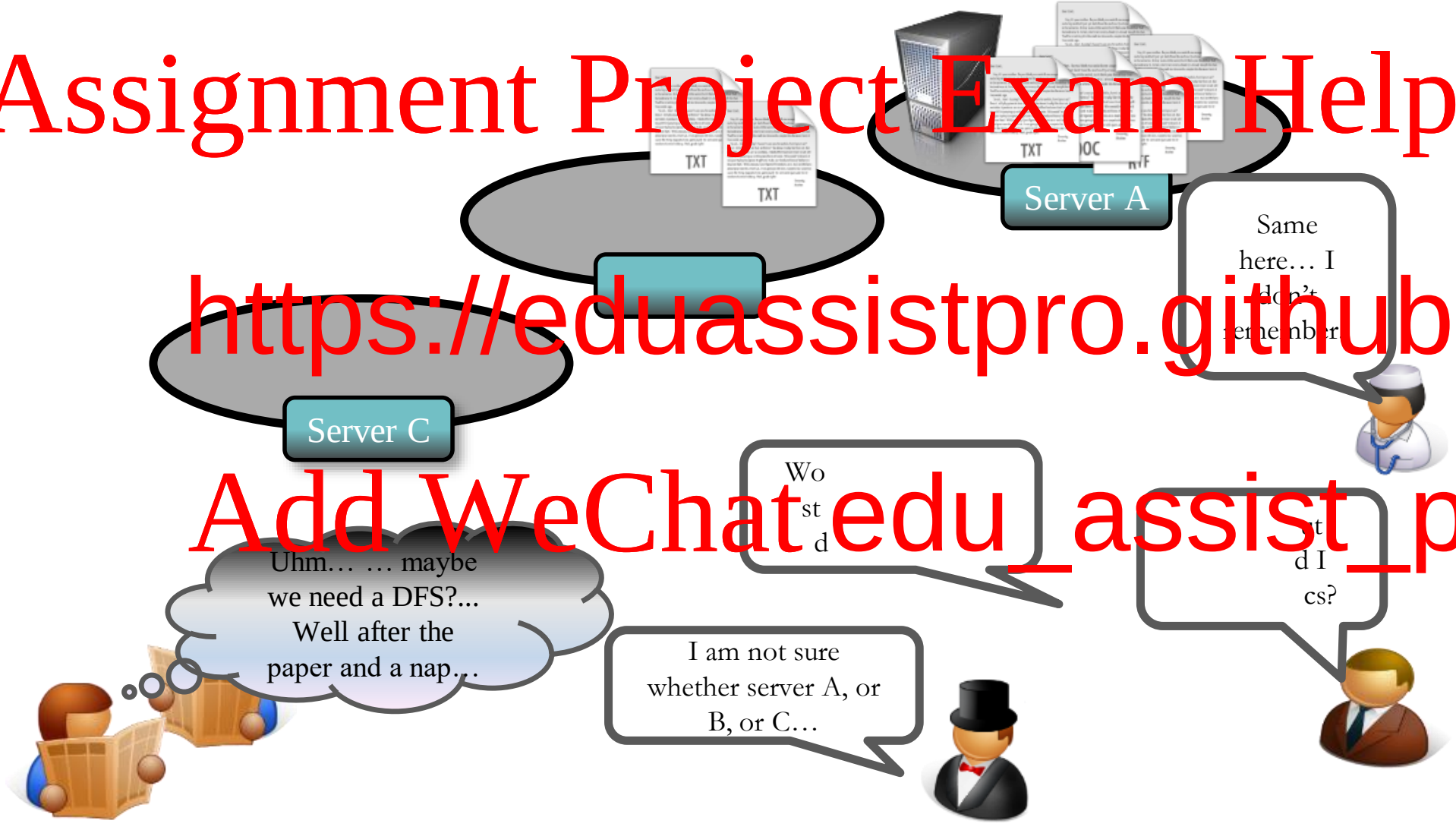
Learning Outcomes

- Understanding the need for Distributed File Systems (DFS)
- Revisiting the basics of Unix File Systems
- Understanding the key requirements for DFS
- Exp
- Cas
- Reading: Distributed Systems: Concepts and Design by George Coulouris (5th edition). Chapter 12. Se

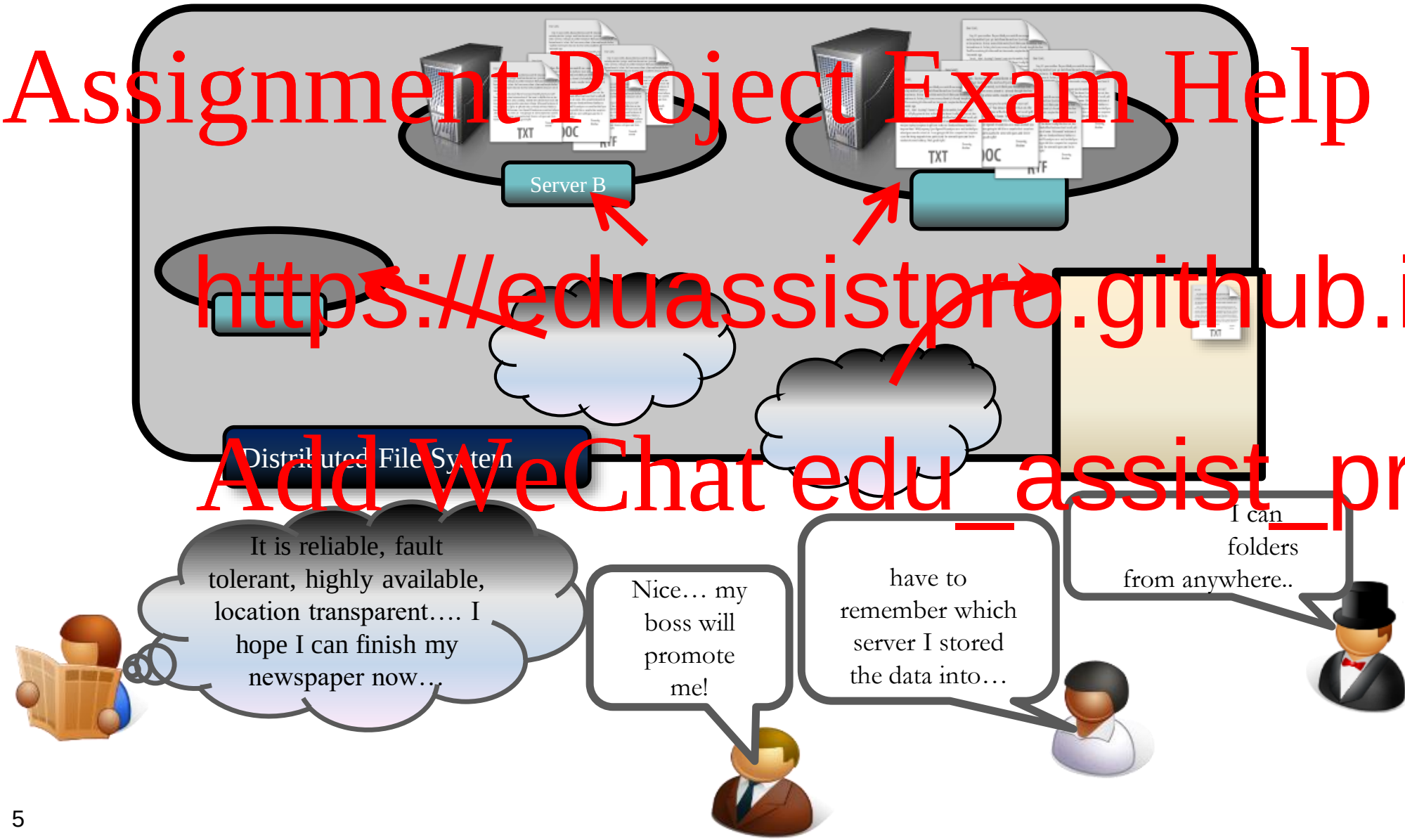
A Case for DFS



A Case for DFS



A Case for DFS



Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File sys	×	✓	×		file system
Distribu	✓	✓	✓	✓	FS
Web	✓	✓	✓	✓	erver
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×		
Persistent object store	✓	✓	×		stent e
Peer-to-peer storage store	✓	✓	✓	2	OcceanStore

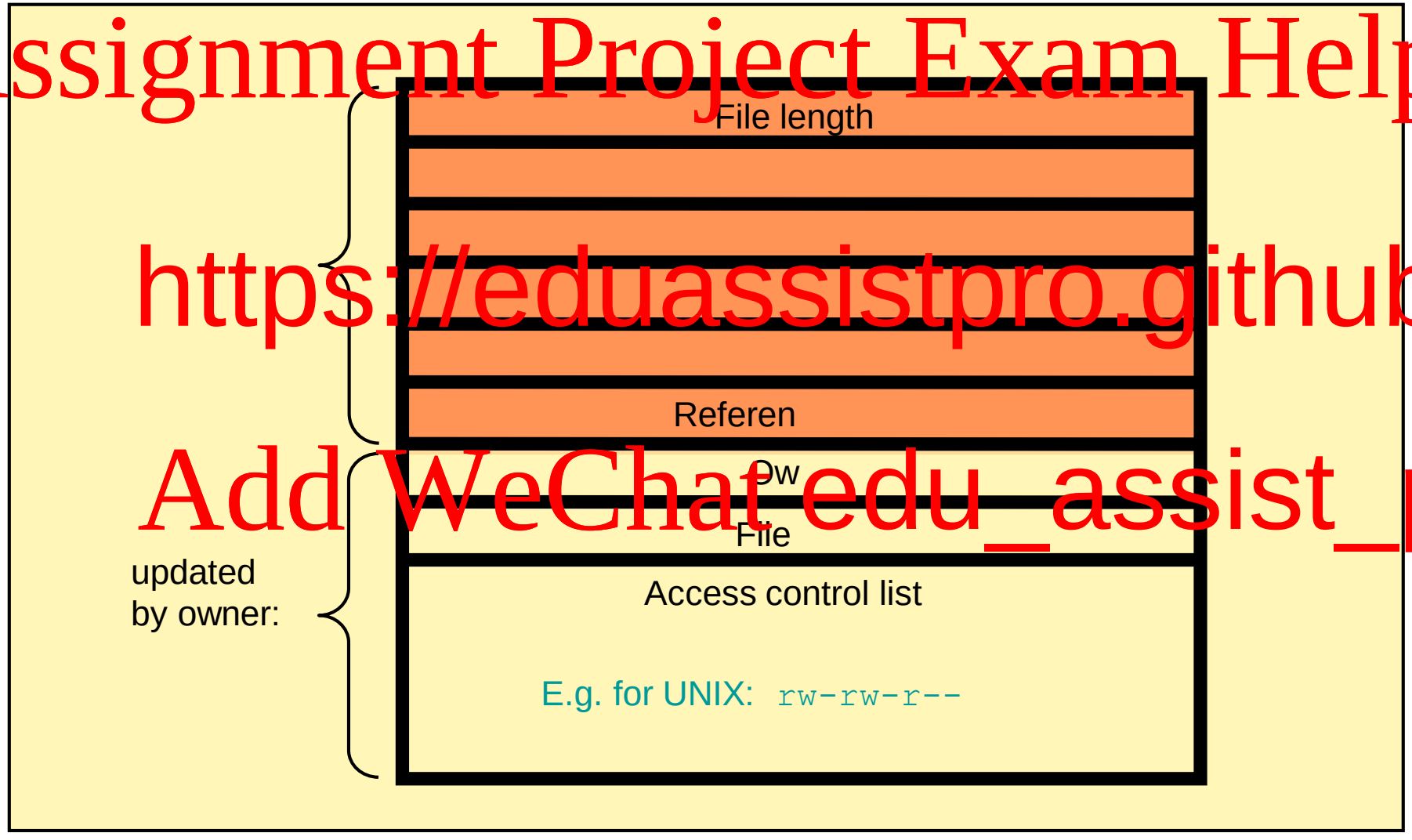
Types of consistency between copies: 1 - strict one-copy consistency √ - approximate/slightly weaker guarantees
X - no automatic consistency 2 – considerably weaker guarantees

Characteristics of File Systems

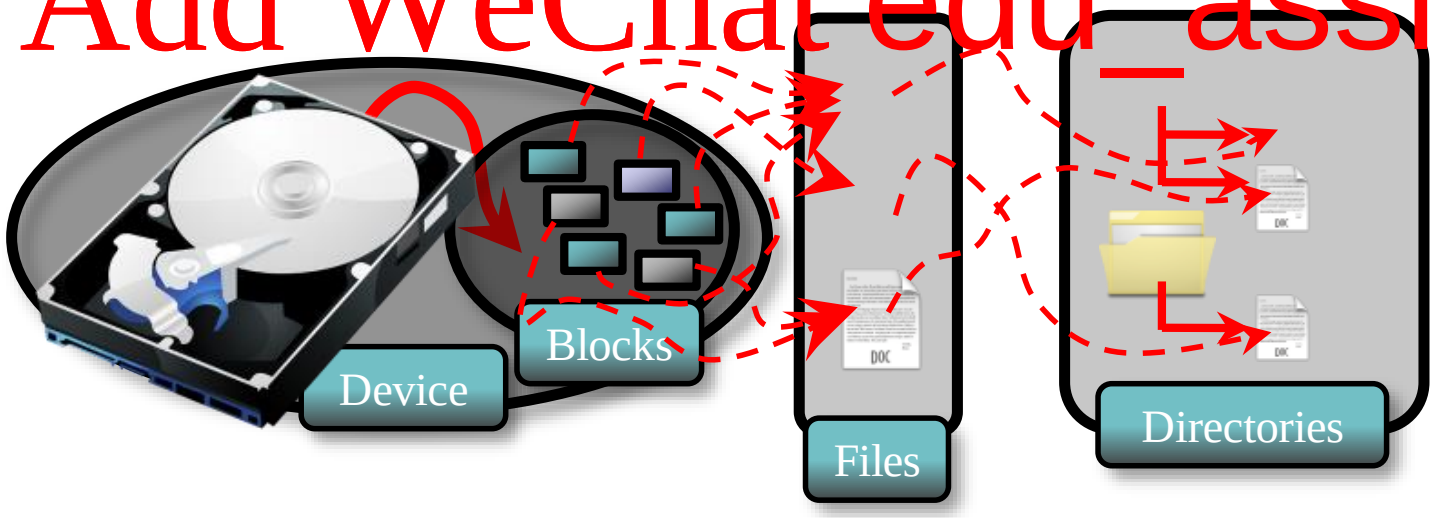
- Files contain both *data* and *attributes*. Data is usually in the form of a sequence of bytes and can be accessed and modified. Attributes include things s identity and ac

- Files h of other files; and they may themselves be (sub-) hierarchical naming scheme for the file. T concatenation of the directory names an

File Attribute Record Structure



File System Modules (non-DFS)



UNIX File System Operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status	
count =	*s* to *buffer*.
count =	rom buffer. y transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write e, depending on *whence*
status = unlink(name)	Removes the file *nam* e file has no other names, it
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Code in C: Copy File Program

Write a simple C program to copy a file using the UNIX file system operations.

```
#define BUFSIZE 1014
void copyfile(char* oldfile, char* newfile)
{                       char buf[BUFSIZE]; int i,n=1, fdold, fdnew;

                                                    PTEND);

                                        if(write(
                                    }
                                close(fdold);
                        }
                        else printf("Copyfile: couldn't open f
}
main(int argc, char **argv)  {
                        copyfile(argv[1], argv[2]);
}
```

Distributed File Systems (DFS)

- A *file system* provides a convenient programming interface for disk storage along with features such as access control and file-locking that allows file sharing.
- A basic *distributed file system* emulates the same functionality as a (non-distri e remote comp
- A *file* exactly as they do local ones, allowing users to access their files from any computer in an intranet.
- Hosts that provide a file service can be o rage devices, e.g., for multiple disk drives, and r a wide range of other services in an organization, e.g., for the web services and email services. This further facilitates management of the persistent storage, including backups and archiving.

Distributed File Systems Requirements (1)

Transparency :

- *Access transparency* – Client programs should be unaware of the distribution of files. Same API is used for accessing local and remote files and so programs written to operate on local

a uniform
ent
here the

- *Mobility transparency* – Client services do not need to chan one place to another.

istration
from
tinue to
within a

- *Performance transparency* – perform satisfactorily while th specified range.
- *Scaling transparency* – The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Distributed File Systems Requirements (2)

Concurrent file updates : Multiple clients' updates to files should not interfere with each other. Policies should be manageable.

File replication : Each file can have multiple copies distributed over

cessing

Hardwar ld not require the client or server to operating system dependenci

Fault tolerance : Transient communication o file corruption. Servers can use at semantics or the simpler at-least-once semantics with *idempotent* operations. Servers can also be *stateless*.

Distributed File Systems Requirements (3)

Consistency : Multiple, possibly concurrent, access to a file should see a consistent representation of that file, i.e. differences in the the file should be

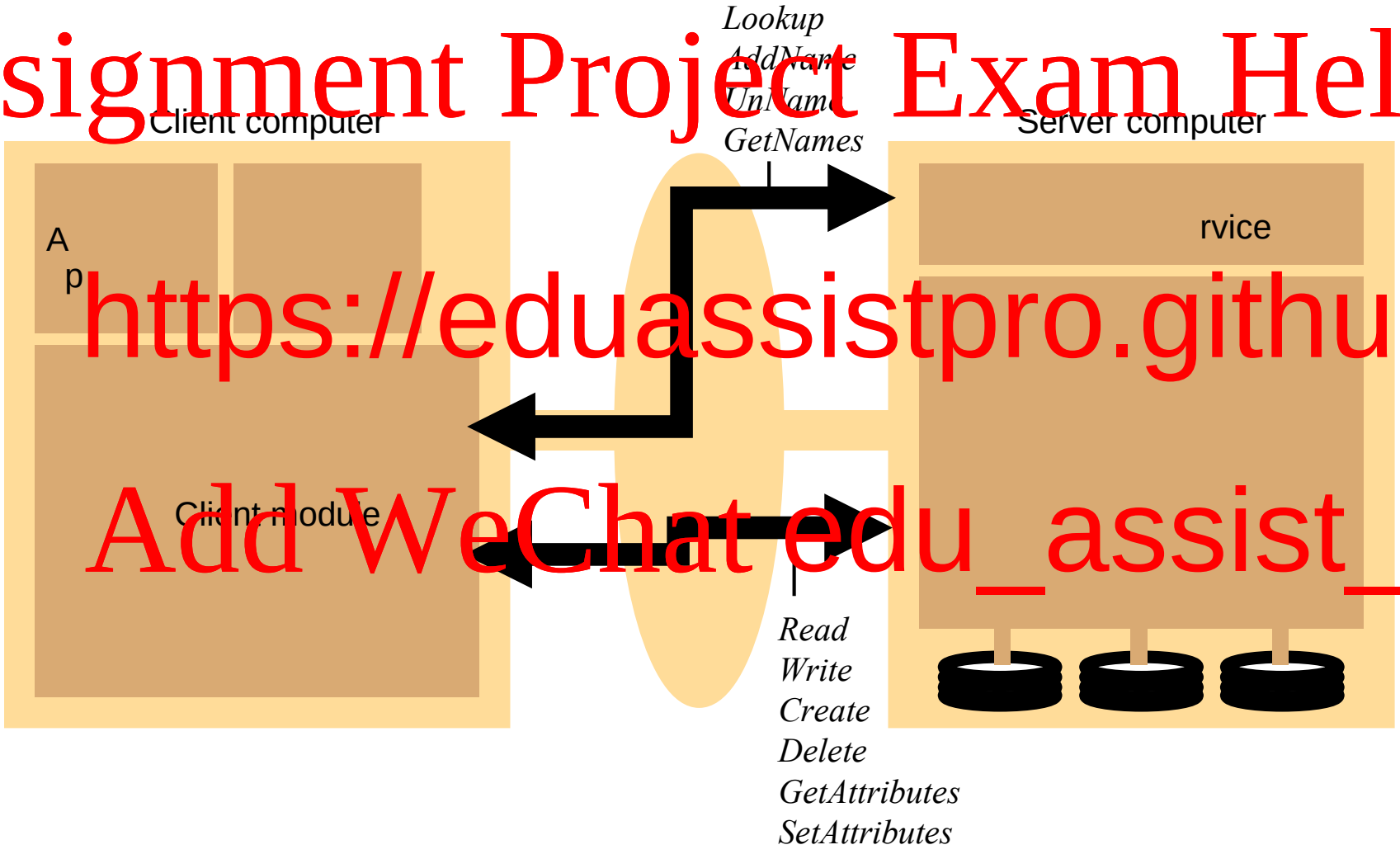
Security : Client requests should be aut fer should be encrypted.

Efficiency : Should be of a comparable le conventional file systems.

File Service Architecture (FSA)

- *Flat file service*: The flat file service is concerned with implementing operations on the contents of files. A *unique file identifier* (UFID) is given to the flat file service to refer to the file to be operated on. The UFID is unique over all the files in the distributed system. The flat file service creates a new UFID for each new file that it
- *Directo* text names and th d and delete files fr e flat file service since the directory files are stored there.
- *Client module*: The client module integrates t file service to provide whatever application progr d by the application programs. The client module le servers. It can also cache data in order to imp

File Service Architecture (FSA)



FSA: Flat File Service Interface

Read(UFID, i, n) → Data	Reads up to n items from position i in the file.
Write(U	he file is
Create()	o it.
Delete(	
GetAttributes(UFID) → Attr	Returns the file attributes for the file.
SetAttributes(UFID,Attr)	Sets the file attribute

FSA: Flat File Service Interface

Difference with UNIX interface:

- Recall that the UNIX interface shown earlier requires that the UNIX file system maintains state, i.e., a file pointer, that is manipulated during reads

- The fl ue to fault
- *repeatable operations* – with the exception o idempotent, allowing the use of at-least-once
- *stateless server* – the flat file service does not r can be restarted after a failure and resume o clients or the server to restore any state.

- Also note that UNIX files require an explicit open command before they can be accessed, while files in the flat file service can be accessed immediately.

Flat File Service Access Control

- The service needs to authenticate the RPC caller and needs to ensure that illegal operations are not performed, e.g., that UFIDs are legal and that files a
- The s
idem

break the

Two ways to do this:

1. An access check can be made whenever a f
the results can be encoded in the form of a
client for submission to the flat file server.

ID, and
s the

2. A user identity can be submitted with every client request, and access checks can be performed by the flat file server for every file operation.

Directory Service Interface

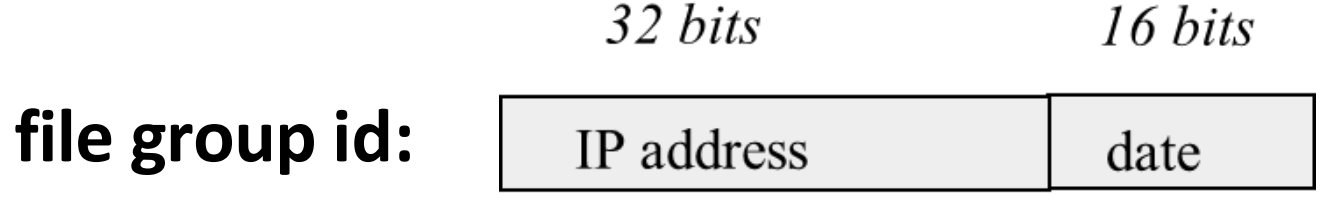
- The primary purpose of the directory service is to provide a translation from file names to UFIDs. An abstract directory service interface is shown on the next slide.
- The d ppings betw edin the flat fil e flat file server.
- A hierarchical file system can be built up .g.,the root directory has name “/” and the con ames “usr”, “home”, “etc”, which themselves ies or files. A client function can make requests for the UFIDs in turn, to proceed through the path to the file or directory at the end.

Directory Service Interface

Lookup(Dir,	given directory.
AddName(D	tory.
UnName(Dir, Name)	Remove the file name from the directory.
GetNames(Dir, Pattern) → Names	Returns all t h the pat tern.

File Group

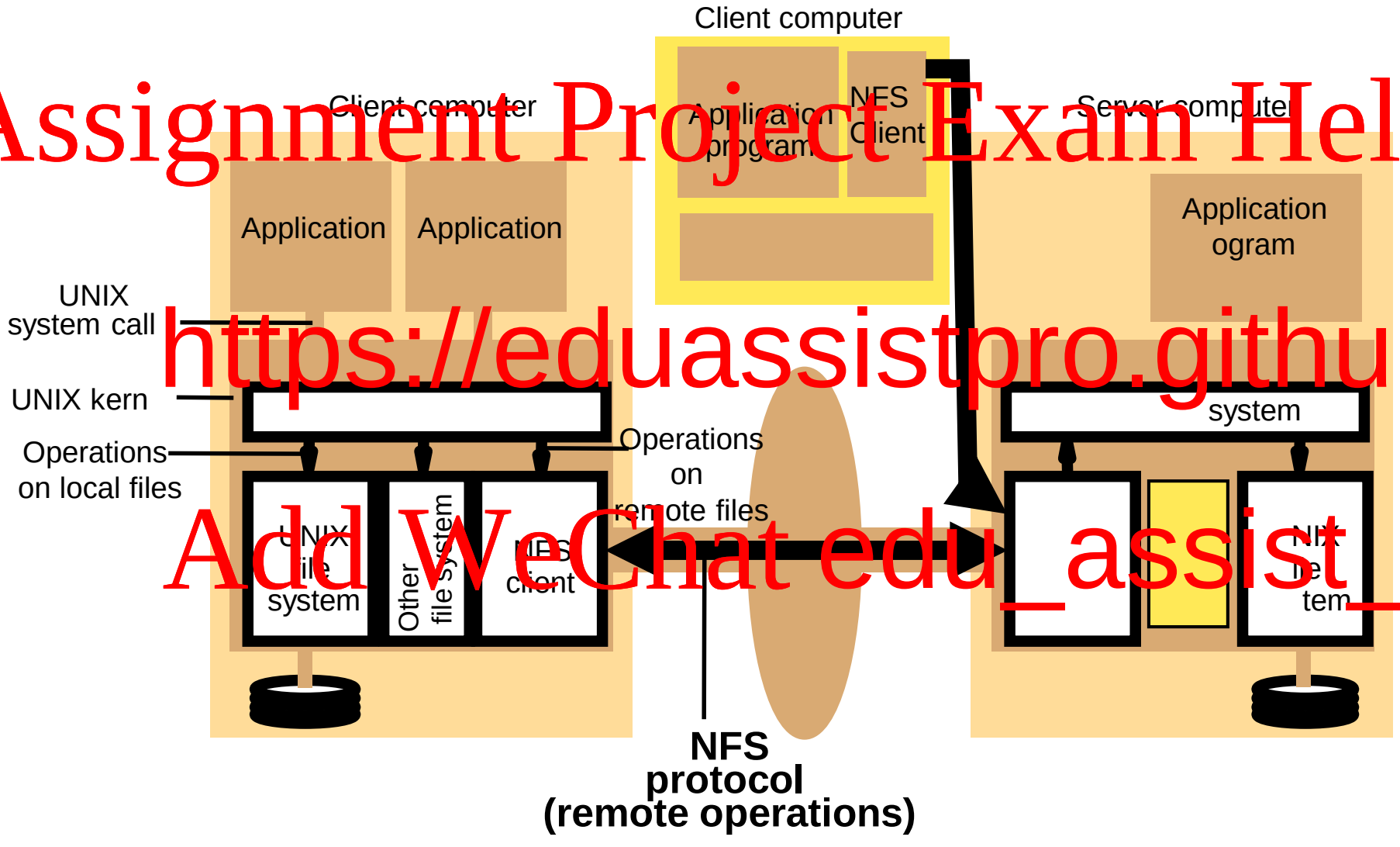
- A *file group* is a collection of files located on a given server. A server may hold several file groups and file groups can be moved between servers, but a
- File gr ral servers. Files erver, e.g., by co at the file was created (16 bits). This allows the file a common part to their UFID called the *file* cated to a different server without conflicting erver.
- The file service needs to maintain a mapping of UFIDs to servers. This can be cached at the client module.



Sun Network File System (NFS)

- The Sun Network File System (NFS) follows the abstract system shown earlie
- There he NFS proto lient to perform operations on the remote file store.
- We consider a UNIX implementation.
- The NFS client makes requests to the NF

NFS: System Architecture



NFS: Virtual File System

- UNIX uses a *virtual file system* (VFS) to provide transparent access to any number of different file systems. The NFS is integrated in the same way.
- The VFS maintains a VFS structure for each filesystem in use. The VFS struct .e., it comb tem.
- The V rds an indicator as to whether the file is local or
- If the file is local, then the v-node contai *i-node* on the UNIX file system.
- If the file is remote then the v-node con e's NFS *file handle* which is a combination of *filesystem identifier*, i-node number and whatever else the NFS server needs to identify the file.

NFS: Client Integration

The NFS client is integrated within the kernel so that:

- user programs can access files via UNIX system calls without recompilation or reloadi
- a singl ared cache;
- the en er can be retaine s.

The client transfers blocks of files from the st and caches them, sharing the same buffer c input-output system. Since several hosts ma remote file, caching presents a problem of c

NFS: Server Interface

- The NFS server interface integrates both the directory and file operations in a single service. The creation and insertion of file names in directories is performed by a single create operation, which takes the textname of the new file and file handle for the target directory as arguments.
- The p ystem primi

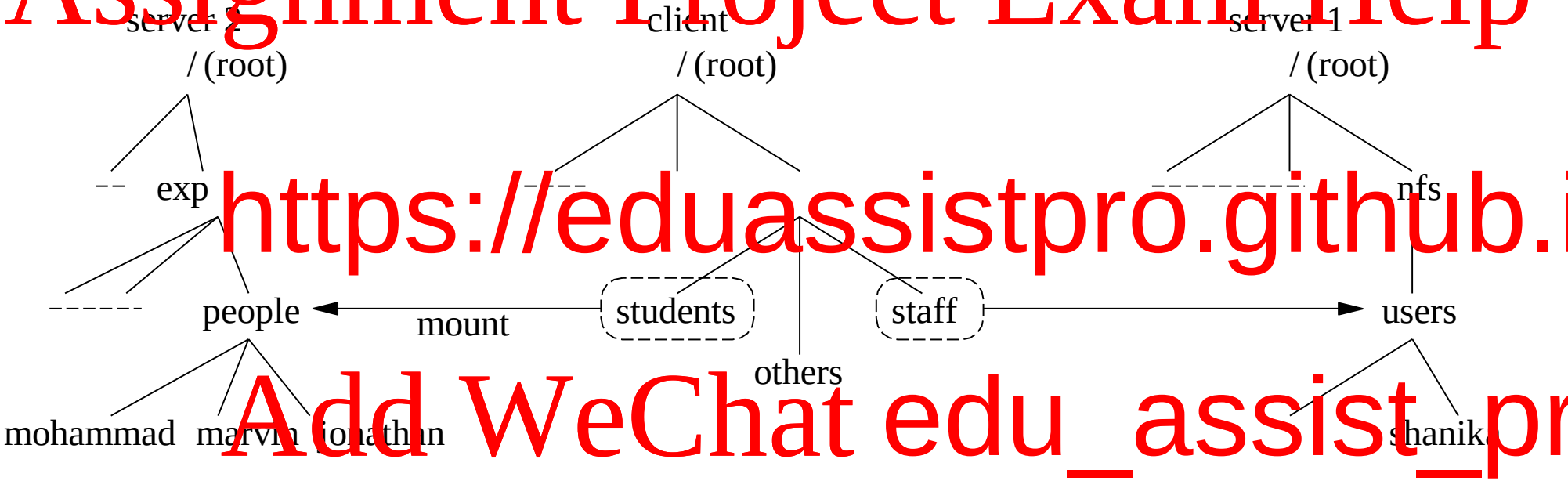
NFS: Mount Service

- Each server maintains a file that describes which parts of the local filesystems that are available for remote mounting.

  tawfiq@htpc:~$ cat /etc/exports
  # /et ay be
  # exp
  / stor

- In the above example all hosts on the su
  filesystem directory /store with read an
- A *hard-mounted* filesystem will block on ess is complete. A *soft-mounted* filesystem will retry a few times and then return an error to the calling process.

NFS: Example NFS Mount



Example NFS mounting in two different file systems

NFS: Server Caching

In conventional UNIX systems:

- data read from the disk or pages are retained in a main memory buffer cache and are evicted when the buffer space is required for other pages. Accesses to cached data does not require a disk access.
- *Read-a* ng those that h
- *Delaye* ing pages when they have been both modified and evicted. A UNIX sync operation flushes modified pages to disk every 30 seconds. Thi esystem, on a single host, because there is only one ca t bypass the cache.

NFS: Server Caching

- Use of the cache at the server for client reads does not introduce any problems. However, use of the cache for writes requires special care to ensure that client can be confident that the writes are persistent, especially in the event of a server crash.

There are

- *Write-t* is increases disk I/ n compl
- *Commit* – data is written to cache and is written to disk when a commit operation is received for the data. A reply to the commi een written to disk
- The first option is poor when the server rec requests for the same data. It however save
- The second option uses more network bandwidth and may lead to uncommitted data being lost. However, it receives the full benefit of the cache.

NFS: Client Caching

- The NFS client also caches data reads, writes, attributes and directory operations in order to reduce network I/O.
- Caching at the client introduces the cache consistency problem since now there is a cache at the client and the server, and there may be more than one c
- Note rite on anot e second read operation being incorrect.
- In NFS, clients poll the server to check fo

NFS: Client Caching

- Let T_c be the time when a cache block was last validated by the client. Let T_m be the time when a block was last modified.
- A cache block is said to be valid at time T if (i) $T - T_c < t$ where t is a given *freshness interval* , or (ii) the value of T_m at the client matches the value at the serve

- A small value for t leads to a close approximation of one-copy consistency, at the cost of greater network I/O.
- In Sun Solaris clients t is set adaptively in the ce on file update frequency. The range is 30 to 6 nding there is a lower risk of concurrent update.

NFS: Client Caching

- The validity check is made on each access to cache block. If the first half of the check is true, then the second half of the check need not be made. The first half of the check does not require network I/O.
- A sep first half of the check trieves $T_{m,\,server}$ for th valid, and the client
- If they do not match, then the cache block is invalid, and the client must request a new copy from the server.
- Traffic can be reduced by applying new value he blocks and by piggy-backing attribute values
- Write-back is used for writes, where modified files are flushed when a file is closed or when a sync operation takes place in the VFS. Special purpose daemons are used to do this asynchronously.

NFS: Summary

- *Access transparency* : Yes. Applications programs are usually not aware that files are remote and no changes are need to applications in order to access remote files.
- *Location transparency* : Not enforced. NFS does not enforce a global namespace since client filesystems may mount shared filesystems at different points. Thus an application that works on one client may not work on another.
- *Mobili* ust be update
- *Scalabi* odate more file ser cess a single f
- *File replication*: Not supported for updates. A d to facilitate this.
- *Hardware and operating system heterogeneit* don almost every known operating system and ha
- *Fault tolerance*: Acceptable. NFS is stateless ist for how to handle failures.
- *Consistency* : Tunable. NFS is not recommended for close synchronization between processes.
- *Security* : Kerberos is integrated with NFS. Secure RPC is also an option being developed.
- *Efficiency* : Acceptable. Many options exist for tuning NFS.