702 Distributed Systems

ributed File Systems

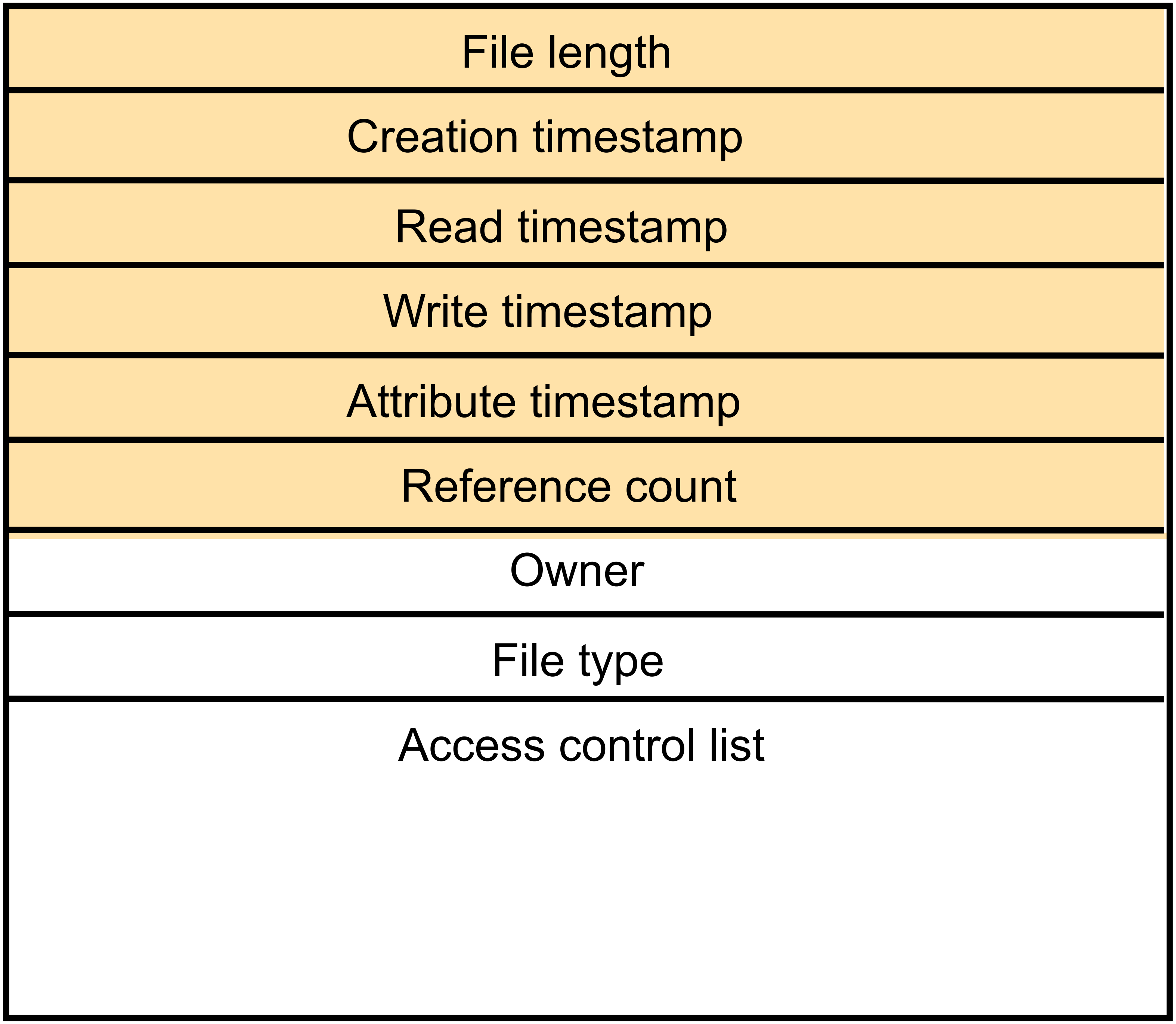
- eneric Distributed File System
- FS Network File System Developed at Sun (1984)
- FS Andrew File System Developed at CMU (1980's)
- oogle File System (GFS) (2004)
- DFS Open Source Hadoop Distributed File System(2008)

system modules

irectory module:	relates file names to file IDs
le module:	relates file IDs to particular files
ccess control module:	checks permission for operation requested
le access module:	reads or writes file data or attributes
lock module:	accesses and allocates disk blocks
evice module:	disk I/O and buffering

ypical non-distributed file system's layered organization. Each
ends only on the layer below it.

attribute record structure



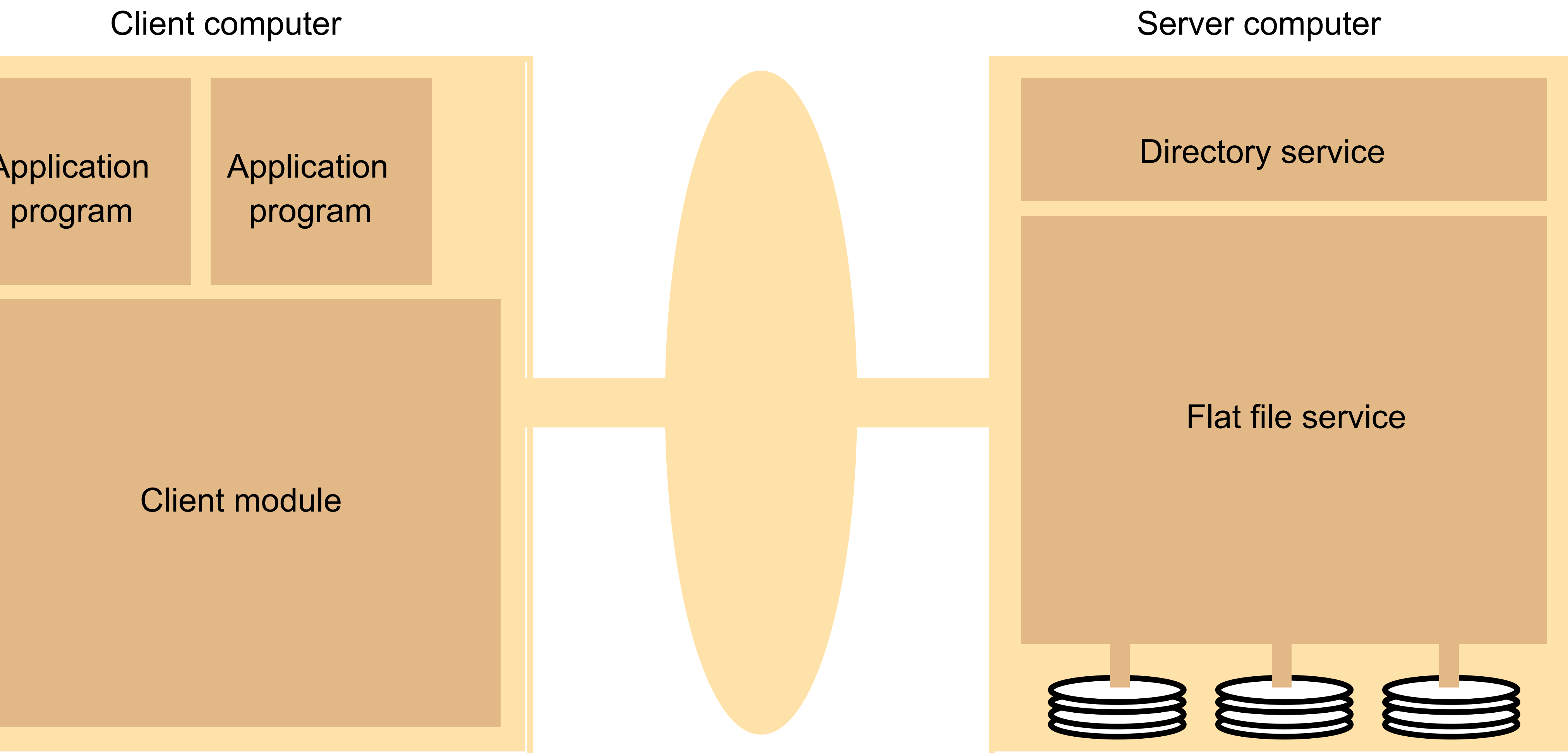
Files cont
both data
attributes.

The shade
attributes
managed
file syster
not norma
directly m
by user pr

file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. 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 = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
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*.

se operations are implemented in the Unix kernel. These are
rations available in the non-distributed case. Programs cannot
erver any discrepancies between cached copies and stored data after
update. This is called strict one copy semantics.



he client module provides
single interface used by
pps – emulates traditional fs

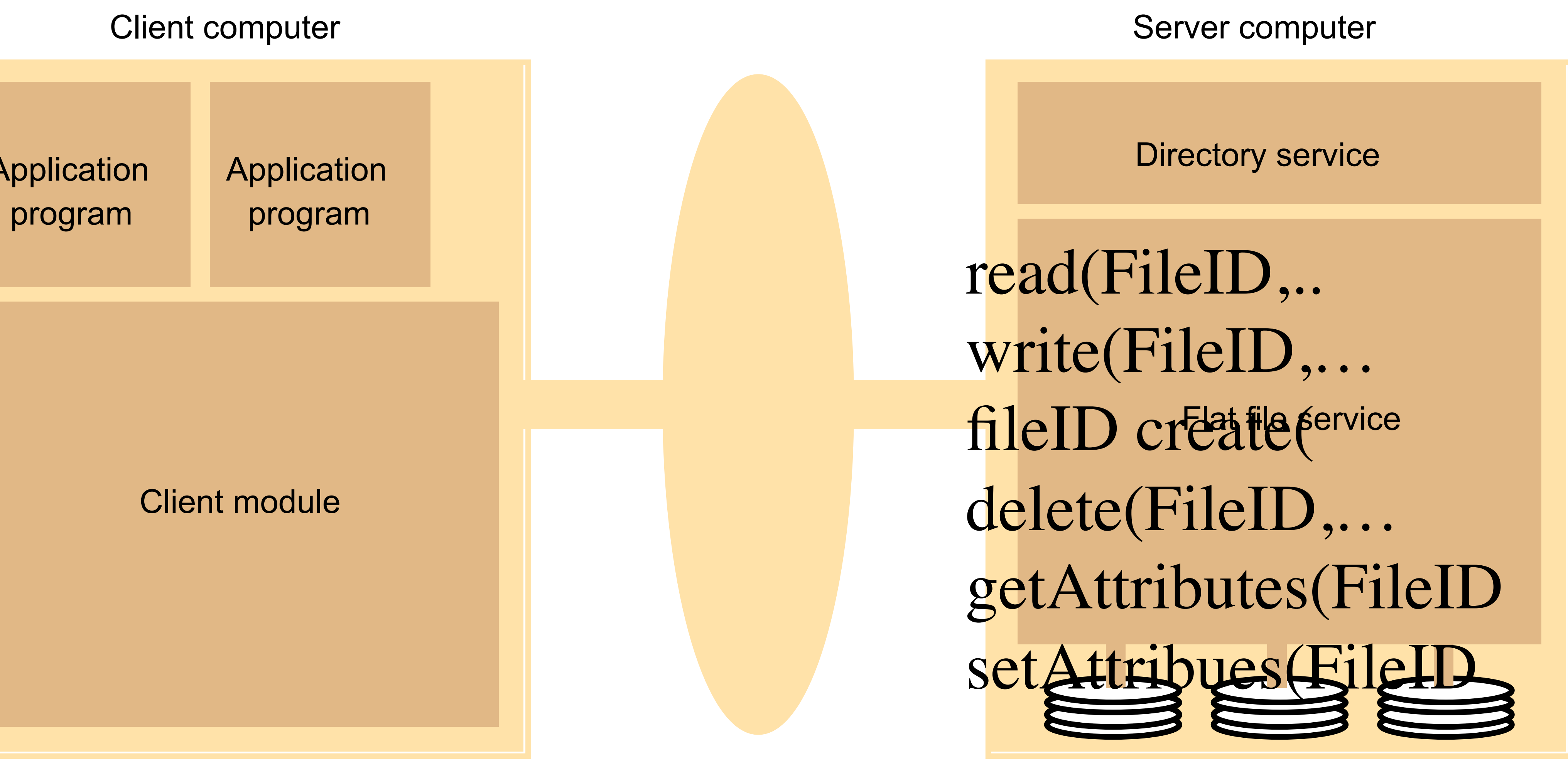
Flat file service and dir.
both provide an RPC in

ile service operations

l(FileId, i, n) -> Data hrows *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
e(FileId, i, Data) hrows *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
te() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
te(FileId)	Removes the file from the file store.
ttributes(FileId) -> Attr	Returns the file attributes for the file.
ttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

e client module will make calls on these operations and so will ectory service act as a client of the flat file service. Unique File ntifiers (UFID's) are passed in on all operations except create(

service architecture   Generic Distributed File System

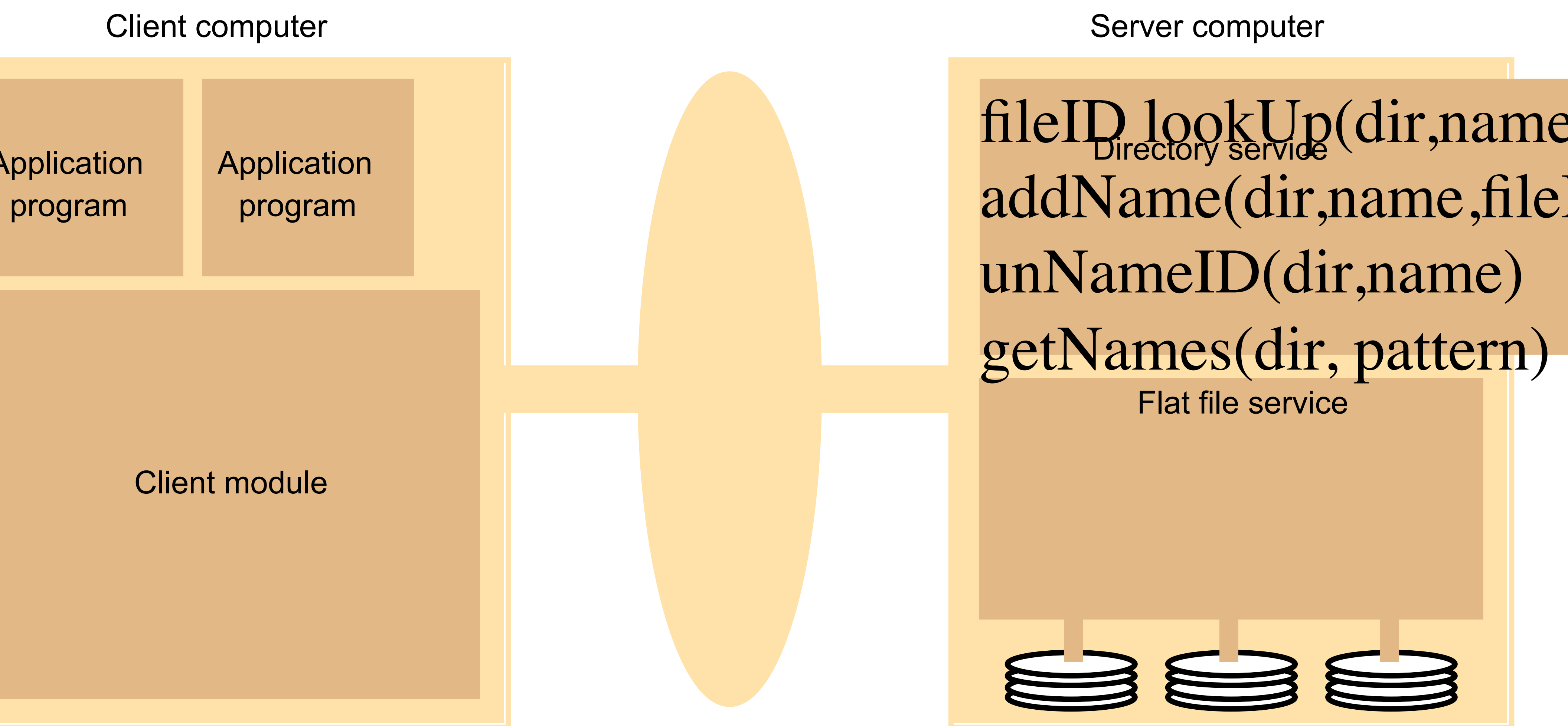


ctory service operations

ookup(Dir, Name) -> FileId - throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
ddName(Dir, Name, FileId) - throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
nName(Dir, Name) - throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
etNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

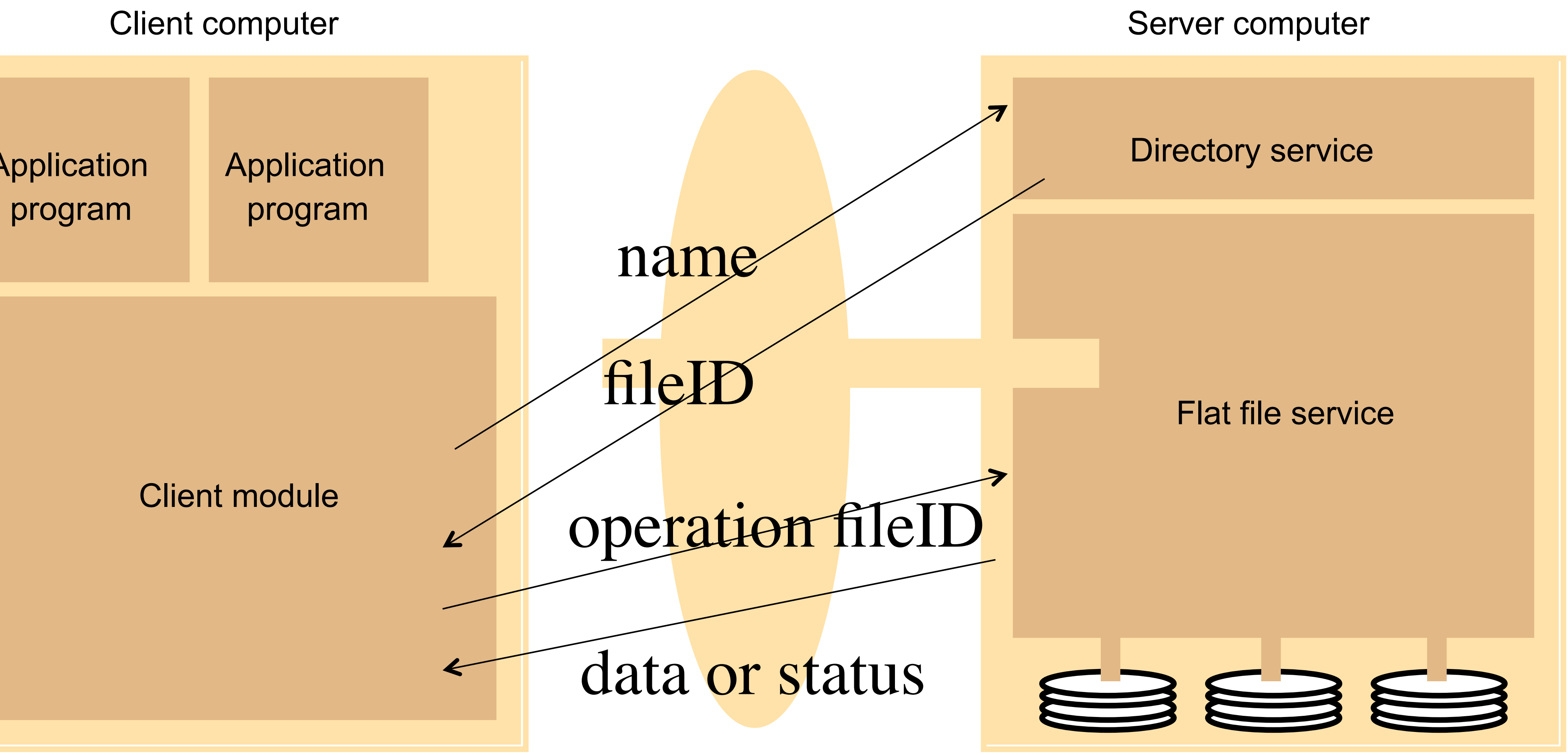
Primary purpose: translate text names to UFID's. Each directo
is stored as a conventional file and so this is a client of the flat
service.

Once a flat file service and directory service is in place, it



fileID lookUp(dir,name
addName(dir,name,file
unNameID(dir,name)
getNames(dir, pattern)

have seen this pattern before.


Client computer
Server computer
Application program
Application program
Directory service
name
fileID
Flat file service
Client module
operation fileID
data or status

l: Be unsurprising and look like a UNIX FS.

l: Implement full POSIX API. The **P**ortable **O**perating **S**ystem **I**nterface is an IEEE family of standards that describe how Unix like Operating Systems should behave.

l: Your files are available from any machine.

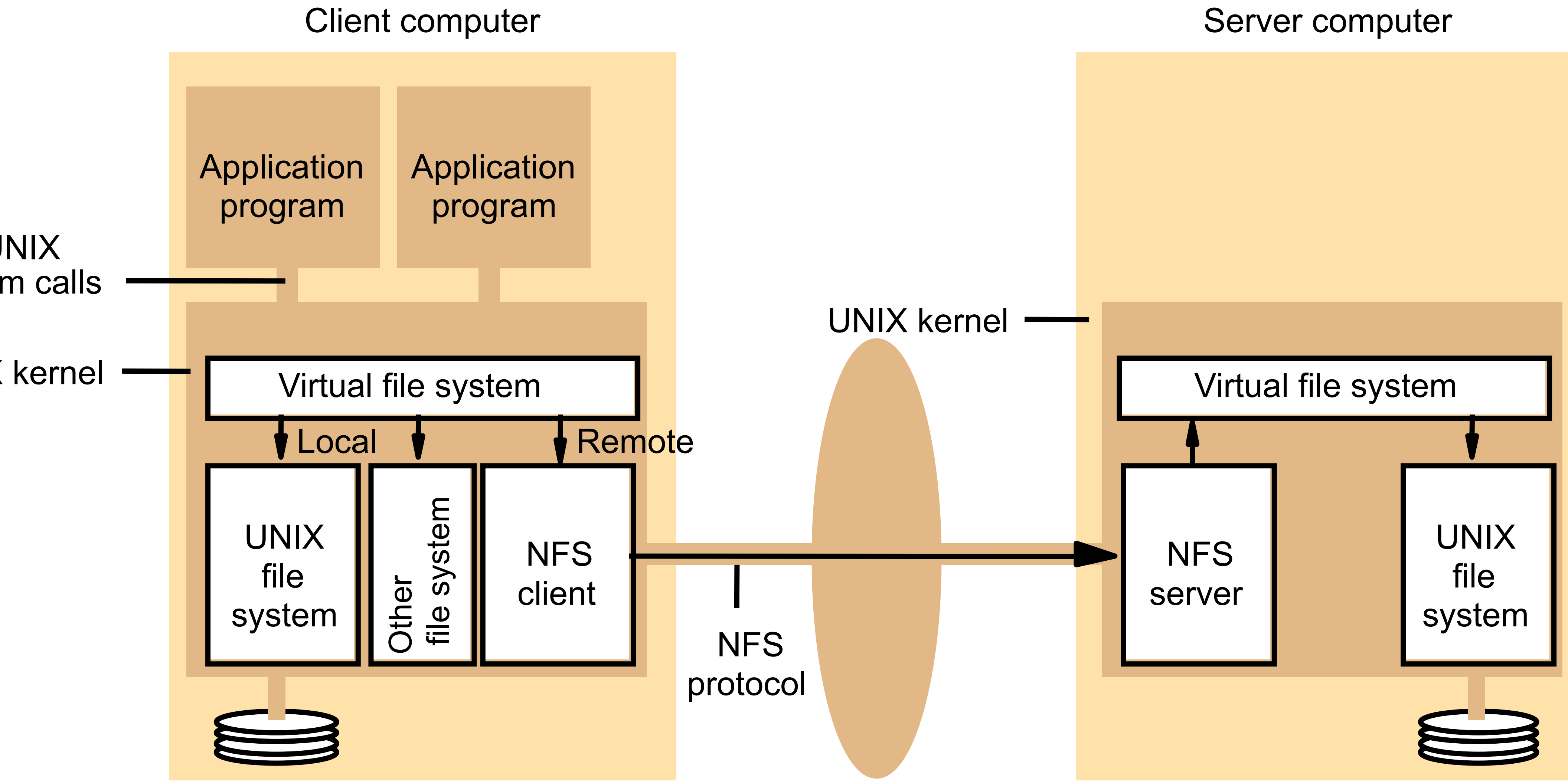
l: Distribute the files and we will not have to implement n protocols.

has been a major success.

was originally based on UDP and was stateless.

added later.

defines a virtual file system. The NFS client pretends to architecture



NFS uses RPC over TCP or UDP.
External requests are translated into
RPC calls on the server. The virtual

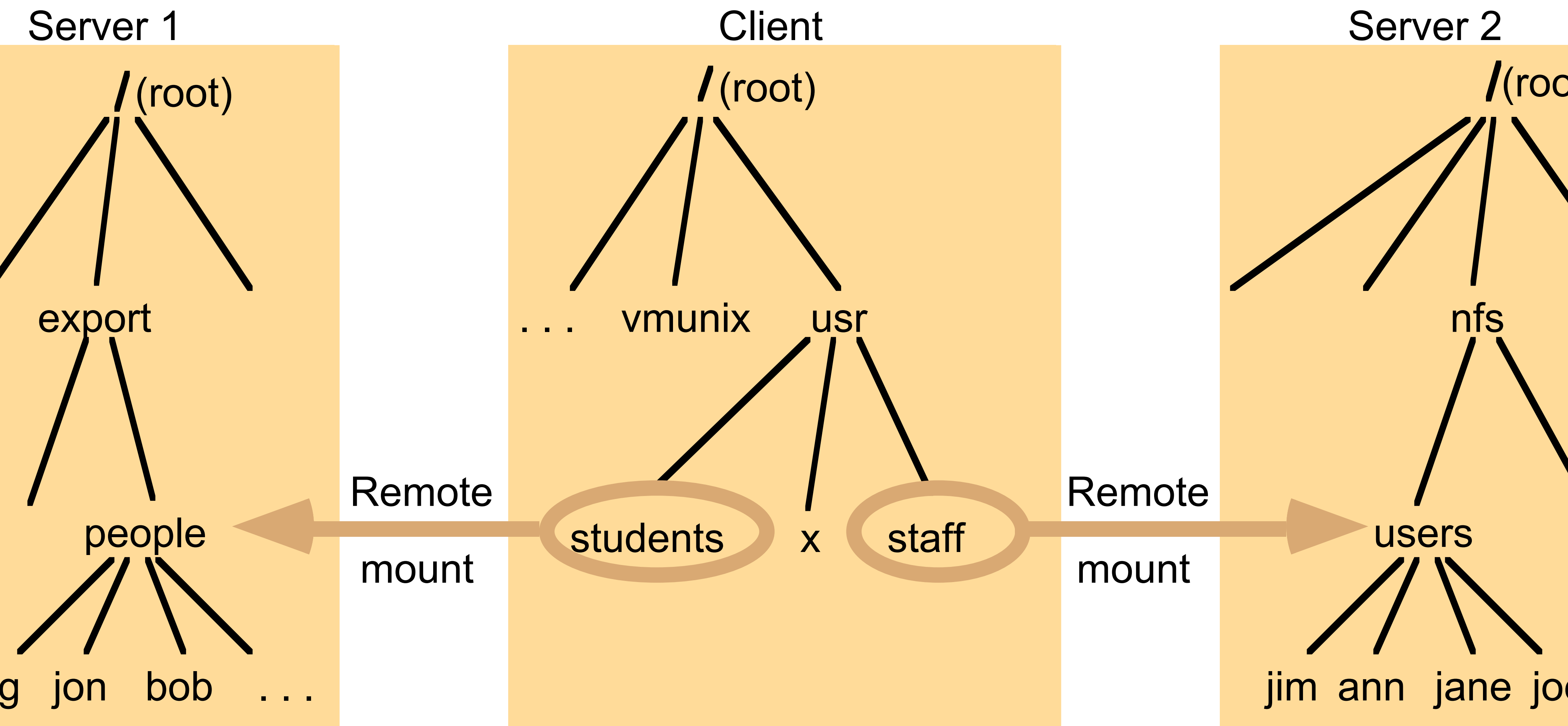
server operations (simplified) – 1

up(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
te(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
ve(dirfh, name) status	Removes file name from directory *dirfh*.
r(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
r(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
ne(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
ewdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

server operations (simplified) – 2

ink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpre the *string* but makes a symbolic link file to hold it.
link(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
ir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
r(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *di* Fails if the directory is not empty.
dir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaq pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the follow entry. If the value of *cookie* is 0, reads from the first entry in the directory.
s(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

l and remote file systems accessible on an NFS client



te: The file system mounted at */usr/students* in the client is actually the sub-tree loc
xport/people in Server 1; the file system mounted at */usr/staff* in the client is actuall

rew File System

ike NFS, the most important design goal is
calability.

achieve scalability, whole files are cached in clie
odes. Why does this help with scalability?

reduce client server interactions.

ient cache would typically hold several hundred
files most recently used on that computer.

e cache is permanent, surviving reboots.

en the client opens a file, the cache is examined
nd used if the file is available there.

ew File System - Typical Scenario – Modified from Coulouris

e client code tries to open a file the client cache is tried fi
not there, a server is located and the server is called for t
.

copy is stored on the client side and is opened.

sequent reads and writes hit the copy on the client.

n the client closes the file - if the files has changed it is
nt back to the server. The client side copy is retained for
ssible more use.

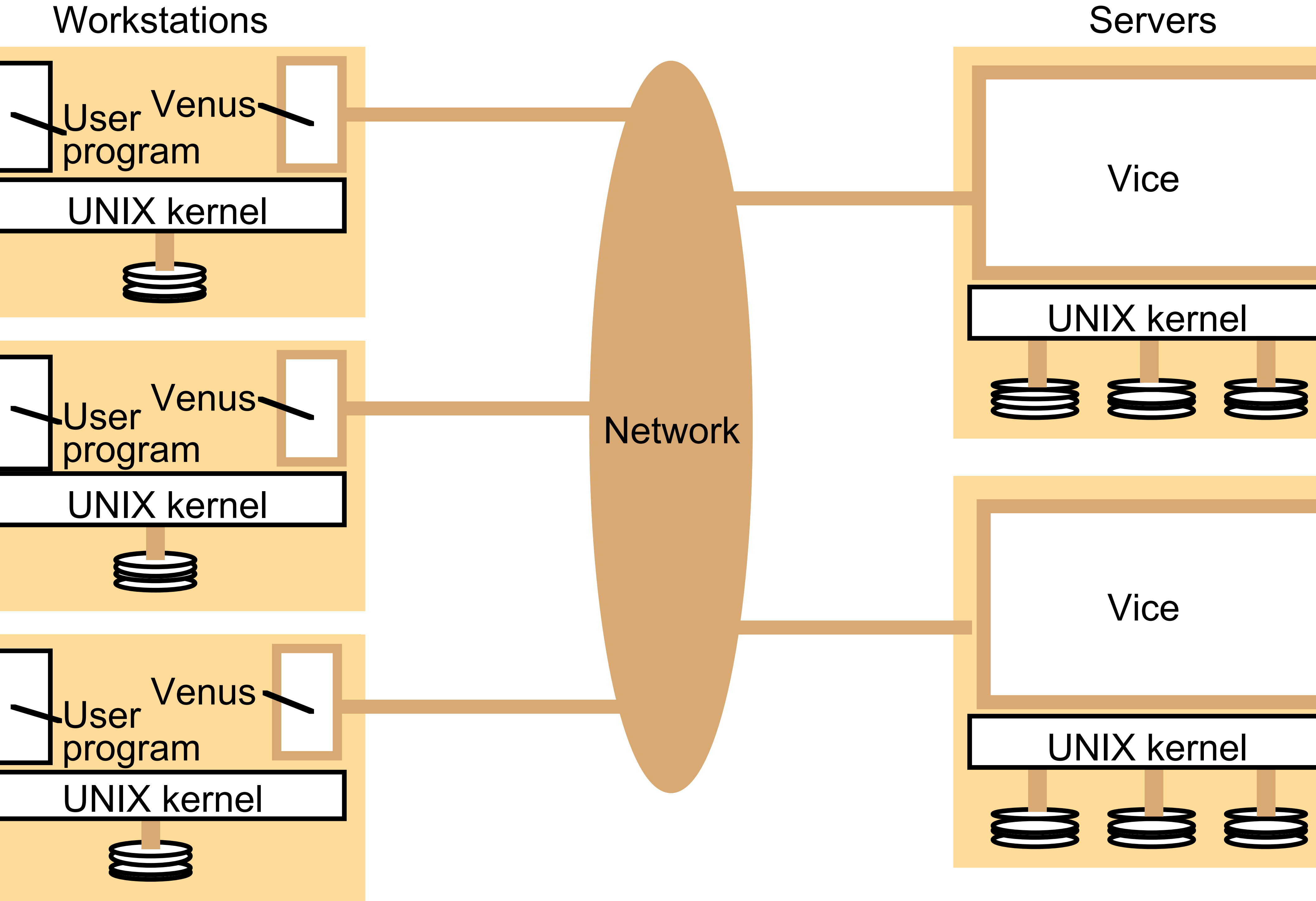
sider UNIX commands and libraries copied to the client.

sider files only used by a single user.

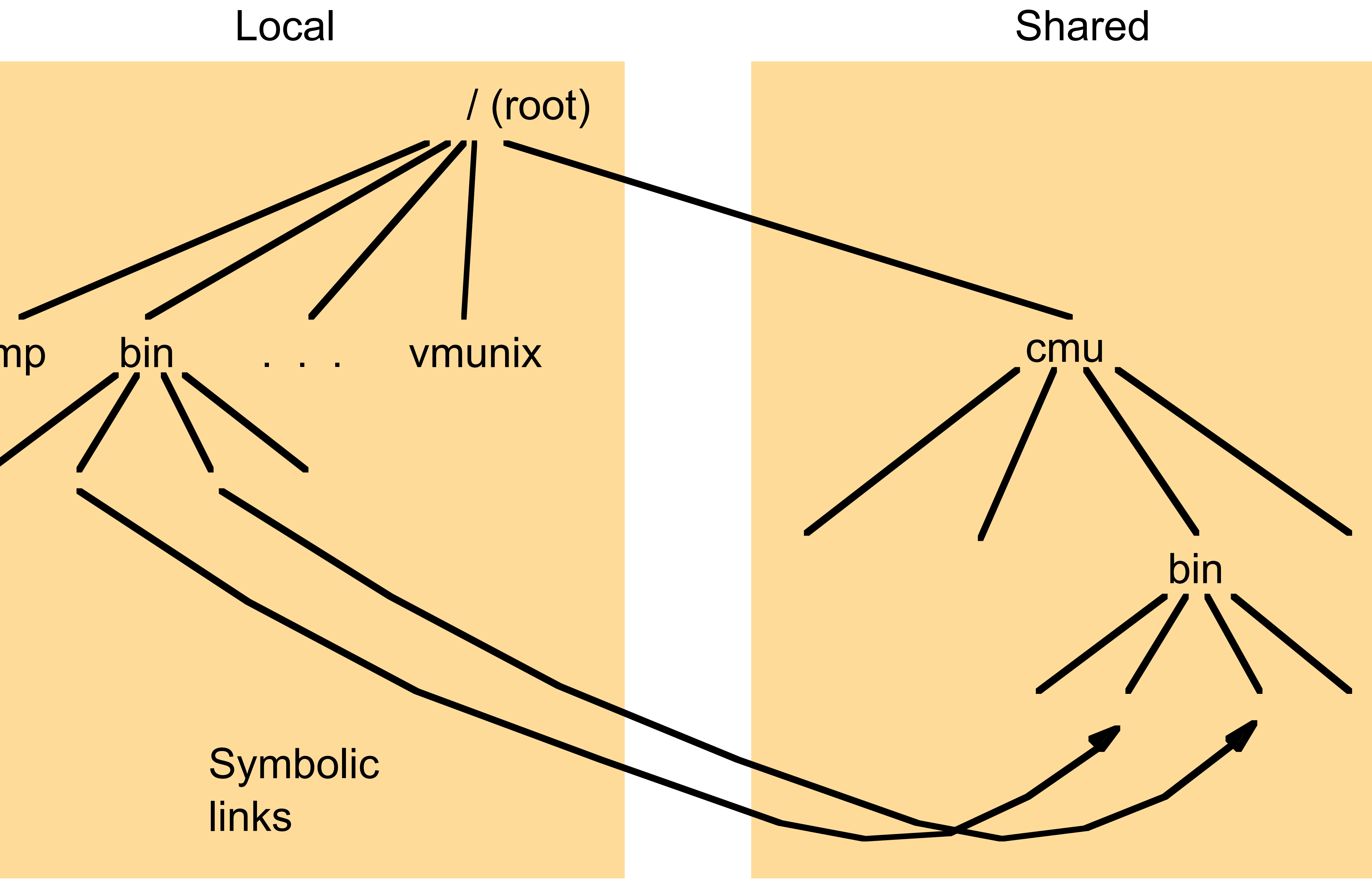
se last two cases represent the vast majority of cases.

: Your files are available from any workstation.

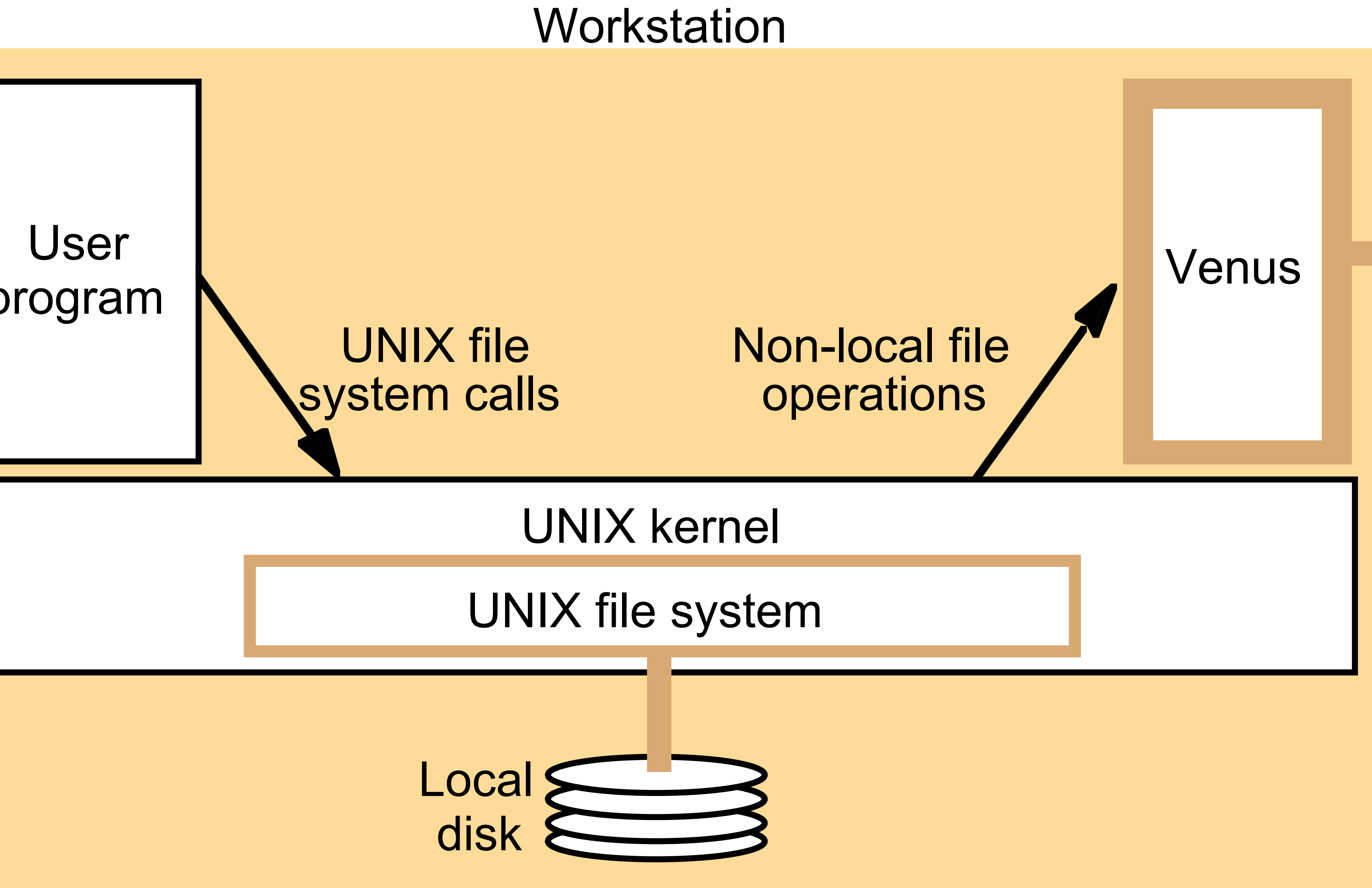
bution of processes in the Andrew File System



name space seen by clients of AFS



em call interception in AFS



ementation of file system calls in AFS

nt
nd the
anged
e makes
ls on all
ents to
ne
.

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
		Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
	Open the local file and return the file descriptor to the application.			
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

A call
promi
token
with t
cache
either
or can

In the
that tv
clients
write
then c
The la
writer

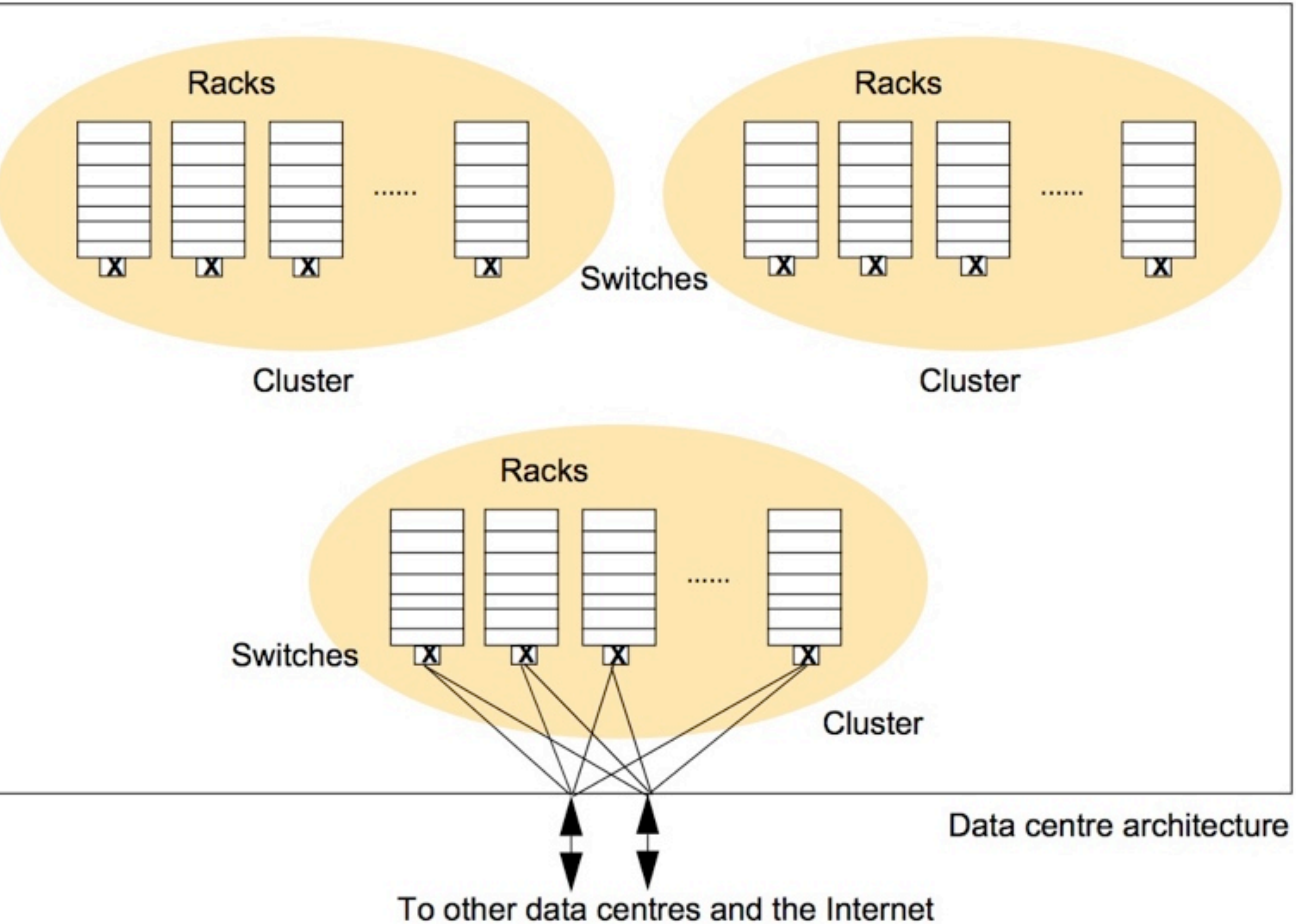
main components of the Vice service interface

h(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
e(fid, attr, data)	Updates the attributes and (optionally) the contents of a specifie file.
te() -> fid	Creates a new file and records a callback promise on it.
ove(fid)	Deletes the specified file.
ock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
aseLock(fid)	Unlocks the specified file or directory.
oveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
kCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

op (HDFS)

- What is Hadoop?
- Sort of the opposite of virtual machines where or machine may act like many. Instead, with Hadoo many machines act as one.
- Hadoop is an open source implementation of GF
- Microsoft has Dryad with similar goals.
- At its core, an operating system (like Hadoop) is all about:
  - a) storing files
  - b) running applications on top of files

nization of the Google physical infrastructure



Data centre architecture

C
P
a
to
4
p
R
o
in
E
>
E
>
3
a
p
A
C
1

quirements of Google File System (GFS)

n reliably with component failures.

ve problems that Google needs solved – not a
assive number of files but massively large files
e common.

cess is dominated by long sequential streaming
ads and sequential appends. No need for cachi
n the client.

oughput more important than latency.

nk of very large files each holding a very large
umber of HTML documents scanned from the w
hese need read and analyzed.

ch file is mapped to a set of fixed size chunks.

ch chunk is 64Mb in size.

ch cluster has a single master and multiple
sually hundreds) of chunk servers.

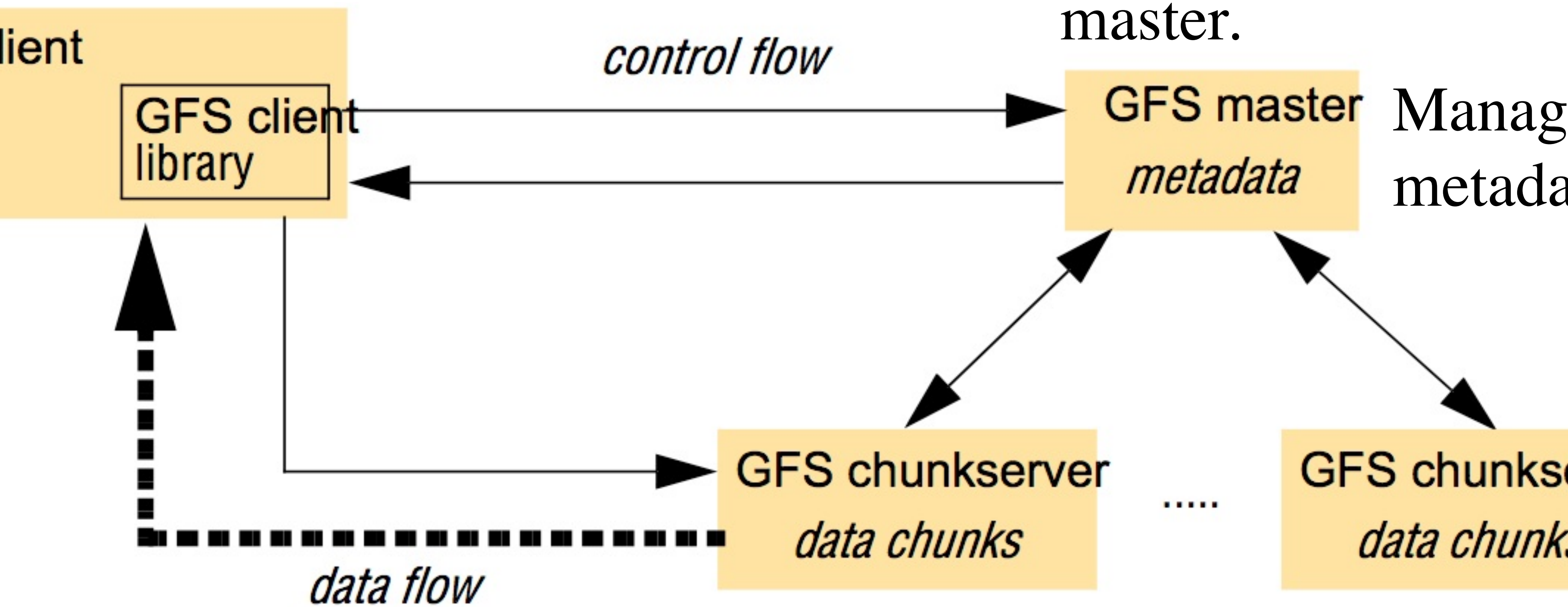
ch chunk is replicated on three different chunk
ervers.

e master knows the locations of chunk replicas.

e chunk servers know what replicas they have a
e polled by the master on startup.

all architecture of GFS

Each GFS cluster ha
master.



Manag
metada

Hundreds of chunkservers

Data is replicated on three independent chun
Locations known by master.
With log files, the master is restorable after f

S Reading a file sequentially

pose a client wants to perform a sequential rea
ocessing a very large file from a particular byte
fset.

- The client can compute the chunk index from th
byte offset.
- Client calls master with file name and chunk
index.
- Master returns chunk identifier and the locations
of replicas.
- Client makes call on a chunk server for the chun
and it is processed sequentially with no caching
may ask for and receive several chunks

Mutation operations

ppose a client wants to perform sequential writes the end of a file.

- The client can compute the chunk index from th byte offset. This is the chunk holding End Of Fil
- Client calls master with file name and chunk index.
- Master returns chunk identifier and the locations of replicas. One is designated as the primary.
- The client sends all data to all replicas. The primary coordinates with replicas to update files consistently across replicas.

pReduce Runs on Hadoop

- Provide a clean abstraction on top of parallelization and fa
olerance.
- Easy to program. The parallelization and fault tolerance is
utomatic.
- Programmer implements two interfaces: one for mappers
nd one for reducers.
- Map takes records from source in the form of key value
pairs.
- Map produces one or more intermediate values along wit
n output key from the input.
- When Map is complete, all of the intermediate values for a
iven output key are combined into a list. The combiners
n the mapper machines.

pReduce

- duce combines the intermediate values into one
ore final values for the same output key (usually
ne final value per key)
- e master tries to place the mapper on the same
achines as the data or nearby.

pReduce – From the Google Paper

- p, written by the user, takes an input pair and oduces a set of output key/value pairs.
- e MapReduce library groups together all termediate values associated with the key I and asses them to the reduce function.
- e Reduce function, also written by the user, ccepts an intermediate key I and a set of values r that key. It merges together these values to fo possibly smaller set of values. Typically, just ze one output value is produced per Reduce vocation.

pReduce From the Google Paper

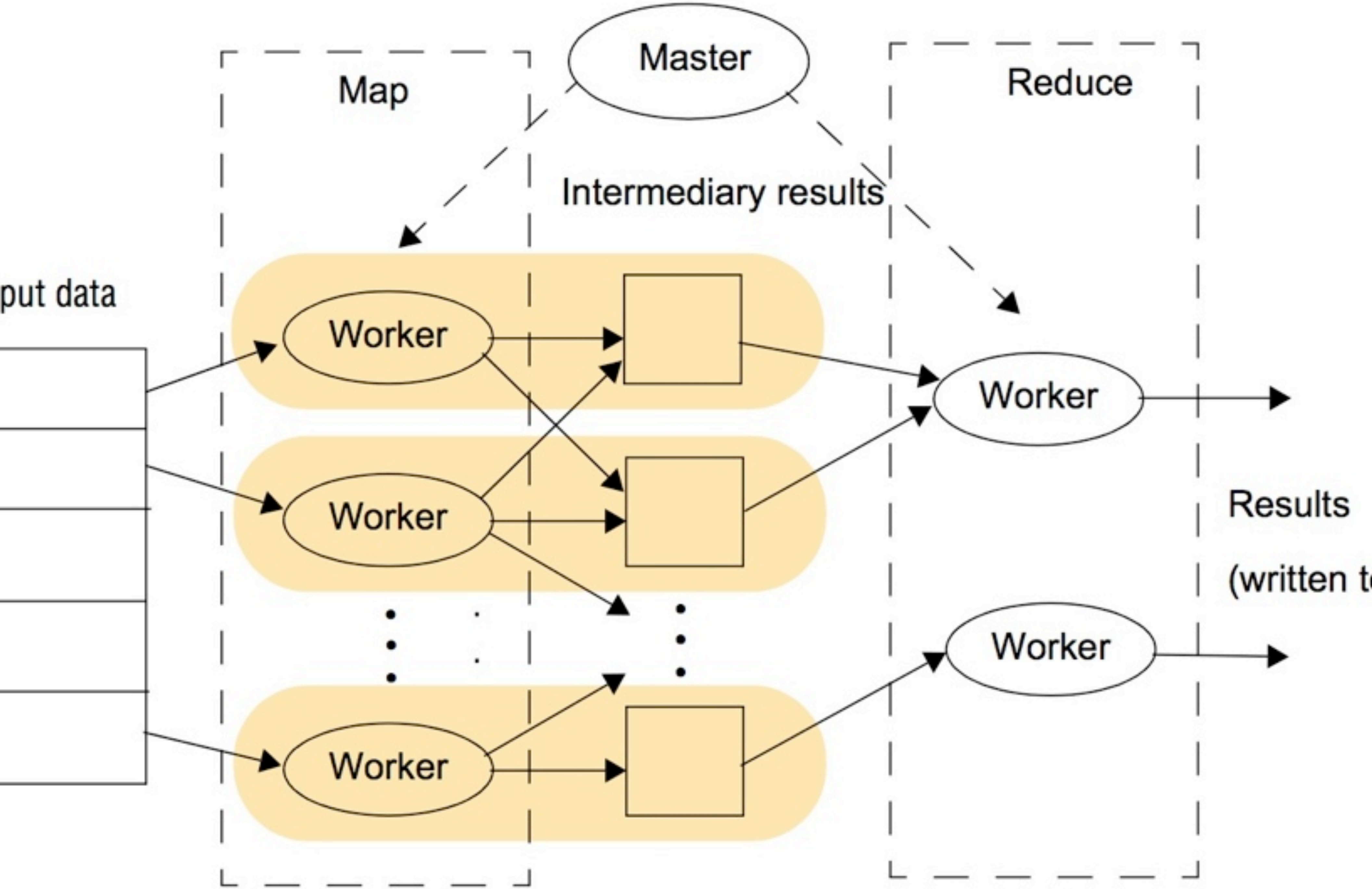
p (k1,v1) --> list(k2,v2)

duce (k2, list(v2)) --> list(v2)

e examples of the use of MapReduce

ction	*Initial step*	*Map phase*	*Intermediate step*	*Reduce ph*
l count		For each occurrence of word in data partition, emit <*word*, 1>		For each wo the intermed set, count th number of 1
		Output a line if it matches a given pattern		Null
This heavily nediate	*Partition data into fixed-size chunks for processing*	For each entry in the input data, output the key-value pairs to be sorted	*Merge/sort all key-value keys according to their intermediary key*	Null
ted		Parse the associated documents and output a <*word*, document ID> pair wherever that word exists		For each wo produce a lis (sorted) document ID

overall execution of a MapReduce program



ail Execution of MapReduce

rs run on the input data scattered over n machines:

n Disk 1 =>(key,value) => map_1

n Disk 2 => (key,value) => map_2

n Disk n => (key,value) => map_n

ap tasks produce (key, value) pairs:

=> (key 1, value)

(key 2, value)

=> (key 1, value)

(key 2, value)

(key 3, value)

(key 1, value)

tput of each map task is collected and sorted on the key. These key, value pairs

ssed to the reducers:

, value list) => reducer1 => list(value)

, value list) => reducer2 => list(value)

Maps run in parallel.
Reducers run in parallel.
Map phase must be comp
finished before the reduce
phase can begin.
The combiner phase is ru
mapper nodes after map p
This is a mini-reduce
on local map output.

For complex activities, b

Reduce to Count Word Occurrences in Many Documents

=> (Document name,Document) => map$_1$ On machine near disk 1

=> (Document name,Document) => map$_2$ On machine near disk 2

=> (Document name, Document) => map$_n$

=> (ball, 1)

(game, 1)

=> (ball, 1)

(team, 1)

(ball, 1)

map output and sort by key. Send these pairs to reducers.

1,1,1) => reducer => (ball, 3)

, 1) => reducer => (game, 1)

1) => reducer => (team, 1)

ne MapReduce Examples

Count the number of occurrences of each word
a large collection of documents.

Distributed GREP: Count the number of lines w
a particular pattern.

From a web server log, determine URL access
frequency.

Reverse a web link graph. For a given URL, fin
URL's of pages pointing to it.

For each word, create list of documents
containing it. (Same as 4.)

Distributed sort of a lot of records with keys

pReduce Example (1)

unt the number of occurrences of each word in a
e collection of documents.

Doc1
car
bell
Doc2
car

// (K1,V1) → List(K2,V2)

nap(String key, String value)

/ key: document name

/ value: document contents

or each word w in value

emitIntermediate(w,"1")

(car,1),(bell,1),(car,1

==

/ (K2, List(V2)) → List(V2)

(bell,[1]), (car,[1,1])

reduce(String key, Iterator values)

/ key: a word

/ values: a list of counts

result = 0

pReduce Example (2)

tributed GREP: Count the number of lines with a ticular pattern. Suppose searchString is "th".

,V1) → List(K2,V2)

(fileOffset, lineFromFile)

if searchString in lineFromFile

emitIntermediate(lineFromFile,1)

2, List(V2)) → List(V2)

ce (K2, iterator values)

s = sum up values

emit (sum,k2)

(0, the line) (8, a line) (14, the

(22, the line)

(the line, 1), (the store, 1), (the

(the line, [1,1]), (the store,[1]

(2 the line),(1 the store)

pReduce Example (3)

m a web server log, determine URL access
quency.

age request log:

was visited

was visted

was visted

was visted

(0,URL1),(45,URL1),(90,URL2),(135,U

(URL1,1),(URL1,1),(URL2,1),(URL1,1)

/1) → List(K2,V2)

ffset, url)

itIntermediate(url,1)

(URL1, [1,1,1]), (URL2, [1])

List(V2)) → List(V2)

(url, values)

n values into total

(URL1, 3),(URL2,1)

pReduce Example (4)

Reverse a web link graph. For a given URL, find
URL's of pages pointing to it.

(URL1, {P1,P2,P3}) (URL2, {

/1) → List(K2,V2)

String SourceDocURL, sourceDoc)

for each target in the document

emitIntermediate(target, SourceDocURL)

(P1, URL1), (P2,URL1), (P3, U

(P1, URL2), (P3, URL2

List(V2)) → List(V2)

e(target, listOfSourceURL's)

mit(target, listOfSourceURL's)

(P1, (URL1, URL2)), (P2, (URL1

(P3,(URL1,URL2))

pReduce Example (6)

Distributed sort of a lot of records with keys.

,V1) → List(K2,V2) (0, k2, data), (20, k1, data), (30, k3, data)

(offset, record)

sk = find sort key in record

emitIntermediate(sk, record) (k2,data),(k1,data),(k3,data)

, List(V2)) → List(V2)

(k1,data),(k2,data),(k3,data)

e emits records unchanged

cal Example : Word Count

unt the number of occurrences of each word in a
e collection of documents.

Doc1
car
bell
Doc2
car

// (K1,V1) → List(K2,V2)
nap(String key, String value)
/ key: document name
/ value: document contents
or each word w in value
emitIntermediate(w,"1")

(car,1),(bell,1),(car,1)

==

/ (K2, List(V2)) → List(V2)

(bell,[1]), (car,[1,1])

reduce(String key, Iterator values)
/ key: a word
/ values: a list of counts
result = 0

e

```
lic static class MapClass extends MapReduceBase
nplements Mapper<LongWritable, Text, Text, IntWritable> {

rivate final static IntWritable one = new IntWritable(1);
rivate Text word = new Text();

ublic void map(LongWritable key, Text value,
              OutputCollector<Text, IntWritable> output,
              Reporter reporter) throws IOException {
String line = value.toString();
StringTokenizer itr = new StringTokenizer(line);
while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  output.collect(word, one);
```

d Counting in Java - Reducer

```
blic static class Reduce extends MapReduceBase
mplements Reducer<Text, IntWritable, Text, IntWritable> {

ublic void reduce(Text key, Iterator<IntWritable> values,
                  OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
int sum = 0;
while (values.hasNext()) {
  sum += values.next().get();
}
output.collect(key, new IntWritable(sum));
```

outing π

you think of an embarrassingly parallel approach
pproximating the value of π ?

00 monkeys to each throw one thousand darts
00 square 1 X 1 boards, all with inscribed
es.

t the number of darts landing inside the circles
those landing outside. Compute the area A =
ding inside)/(landing inside + landing outside).
know that A = π r ^{2} = π (1/2) ^{2} = ¼ π.

= 4A.