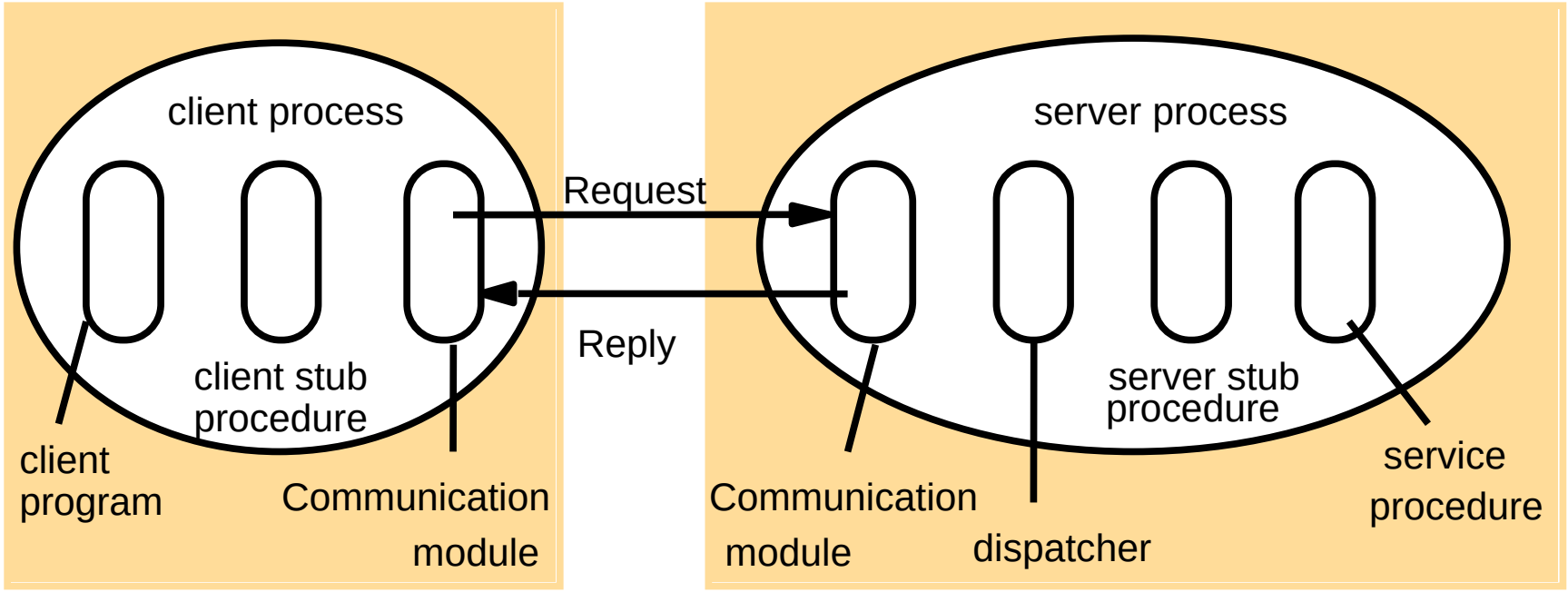
Please note:

- Please start working your research project (proposal will be due on Feb. 19 in class)
- Each group needs to turn in a printed version of their proposal and intermediate report. Also, before class each group needs to email me a DOC version.

Remote Procedure Call (1):

- at-least-once or at-most-once semantics
- client: "stub" instead of "proxy" (same function, different names)
  - behaves like a local procedure, marshal arguments, communicate the request
- server:
  - dispatcher
  - "stub": unmarshal arguments, communicate the results back

Remote Procedure Call (2)



Sun RPC (1):

- Designed for client-server communication in the SUN NFS (network file system)
- Supplied as a part of SUN and other UNIX operating systems
- Over either UDP or TCP
- Provides an interface definition language (IDL)
  - initially XDR is for data representation, extended to be IDL
  - less modern than CORBA IDL and Java
    - program numbers (obtained from a central authority) instead of interface names
    - procedure numbers (used as a procedure identifier) instead of procedure names
    - only a single input parameter is allowed (then we have to use a ?)
- Offers an interface compiler (rpcgen) for C language, which generates the following:
  - client stub
  - server main procedure, dispatcher, and server stub
  - XDR marshalling, unmarshaling

Sun RPC (2): Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
      int length;
      char buffer[MAX];
};
struct writeargs {
      FileIdentifier f;
      FilePointer position;
      Data data;
};
```

```
struct readargs {
      FileIdentifier f;
      FilePointer position;
      Length length;
};

program FILEREADWRITE {
   version VERSION {
      void WRITE(writeargs)=1;      1
      Data READ(readargs)=2;        2
   }=2;
} = 9999;
```

Sun RPC (3):

- A local binding service running at a well-known port
  - local binder—port mapper
  - server registers its program/version/port numbers with port mapper
  - client contacts the port mapper at a fixed port with program/version numbers to get the server port
  - different instances of the same service can be run on different computers--different ports
- authentication
  - request and reply have additional fields to allow authentication
  - unix style (uid, gid), shared key for signing, Kerberos

Case Study: Java RMI (1): Remote interface

Both ordinary objects and remote objects can appear as arguments and results in a remote interface

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject  getAllState() throws RemoteException;    1
}
public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws RemoteException; 2
   Vector allShapes() throws RemoteException;
   int getVersion() throws RemoteException;
}
```

All serializable non-remote objects are copied and passed by value. (see line 1 for return value of getAllState)

Java RMI (3): Server main method

Create a security manager to protect an RMI server

A default RMISecurityManager protects local resources

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
             ShapeList aShapeList = new ShapeListServant();          1
            Naming.rebind("Shape List", aShapeList );                2
             System.out.println("ShapeList server ready");
         }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Why protection? Because Java allows classes to be downloaded from one virtual machine to another. If the recipient does not already possess the class of an object passed by value, its code is download automatically.

Java RMI (4): client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    ShapeList aShapeList = null;
    try{
        aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;        1
        Vector sList = aShapeList.allShapes();                               2
    } catch(RemoteException e) {System.out.println(e.getMessage());
    }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

The client looks up a remote object reference using the *lookup* operation of the RMIregistry (line 1). After having a remote object reference, the client invokes *allShape* method (line 2) and receives a vector of remote object references

Java RMI (5): Java RMIregistry

RMIregistry must run on every server computer that hosts remote objects

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the

A table mapping textual names to references to remote objects hosted on that computer

Java RMI (6):

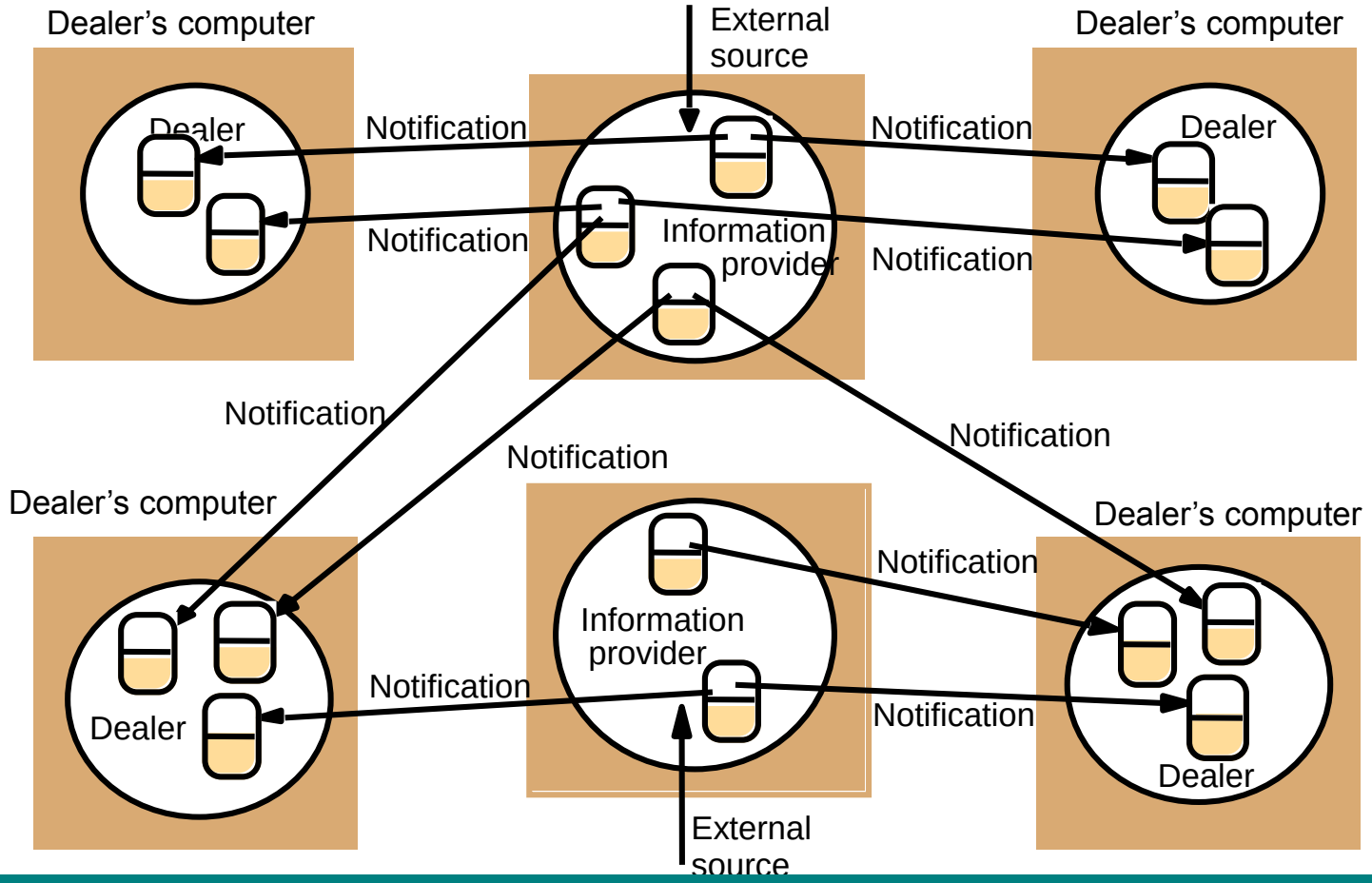
- Callbacks
  - server notifying the clients of events
  - why?
    - polling from clients increases overhead on server
    - client cannot notify users of updates in a timely manner
  - how
    - remote object (callback object) on client for server to call
    - client tells the server about the callback object, server put the client on a list
    - server call methods on the callback object when events occur
  - client might forget to remove itself from the list
    - lease--client expire

Events and notifications (1):

- events of changes/updates...
- notifications of events to parties interested in the events
- publish events to send
- subscribe events to receive
- main characteristics in distributed event-based systems:
  - a way to standardize communication in heterogeneous systems (not designed to communicate directly)
  - asynchronous communication (no need for a publisher to wait for each subscriber--subscribers come and go)
- event types
  - each type has attributes (name of the object that generated the event, the operation, and the time)
  - subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" type)

Events and Notifications (2):

Information provider continuously receives new trading information

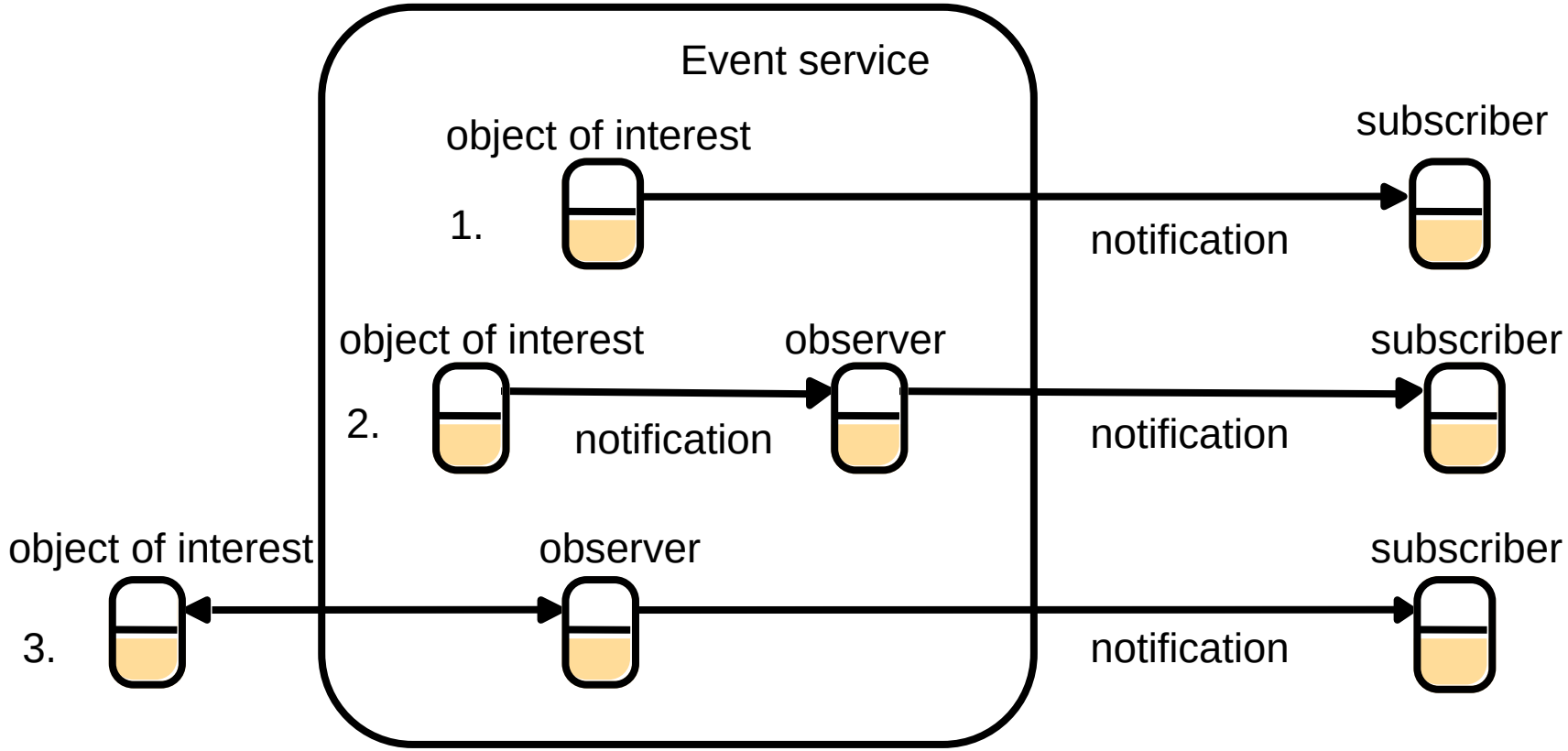


A dealer process creates an object to represent each named stock that the user asks to have displayed

Events and notifications (3):

- Distributed event notification
  - decouple publishers from subscribers via an event service (manager)
- Architecture:
  - object of interest (the object that experiences changes of state, which might be of interest to other objects)
  - event (occurs at an object of interest as the result of the completion of method execution)
  - Notification (an object that contains information about an event)
  - Subscriber (an object that has subscribed to some type of events in another object)
  - observer object (proxy) [reduce work on the object of interest]
    - forwarding
    - filtering of events types and content/attributes
    - patterns of events (occurrence of multiple events, not just one)
    - mailboxes (notifications in batches, subscriber might not be ready)
  - publisher (object of interest or observer object)
    - generates event notifications

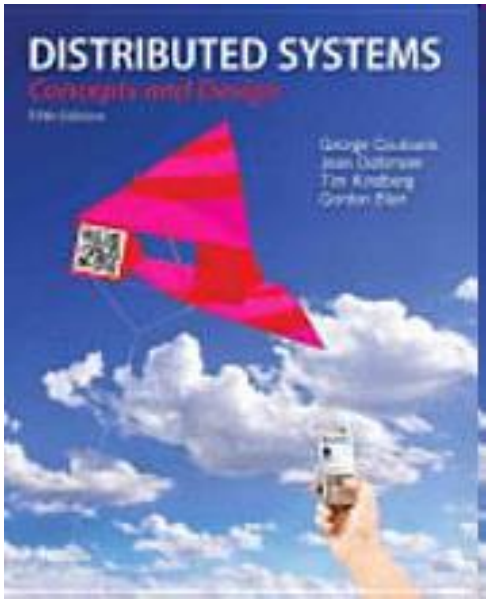
Events and Notifications (4):



Events and Notifications (5)

- Jini
  - event generators (publishers)
  - remote event listeners (subscribers)
  - remote events (events)
  - third-party agents (observers)

Slides for Chapter 12: Distributed File Systems



From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

Learning Objectives

- Understand the architecture for file systems
- Understand two basic distributed file service implementations (Sun NFS and Andrew File System)

Basic topics in this chapter

- file service architecture
- network file system (focus)
- enhancements and further developments

Refresh your memory

- A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them.
- File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files.
- They might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS).

Common Terms

- Meta Data is data about data; meta data is used to facilitate the understanding, use and management of data
- Disk file system: a file system designed for the storage of files on a disk drive. Examples of disk file systems include FAT, FAT32, NTFS, etc.
- Flash file system: a file system designed for storing files on flash memory devices
- Network file system: a network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.

Basic concepts

- A basic *distributed file system* emulates the same functionality as a (non-distributed) file system for client programs running on multiple remote computers.
- Advanced distributed file systems go much further by e.g. maintaining replica files and provide bandwidth and timing guarantees for multimedia data streaming.
- A *file service* allows programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet.

Distributed storage systems

- Earlier storage systems are file systems (e.g. NFS); units are files.
- More recently, distributed object systems (e.g. CORBA, Java); units are objects.

Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM, Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

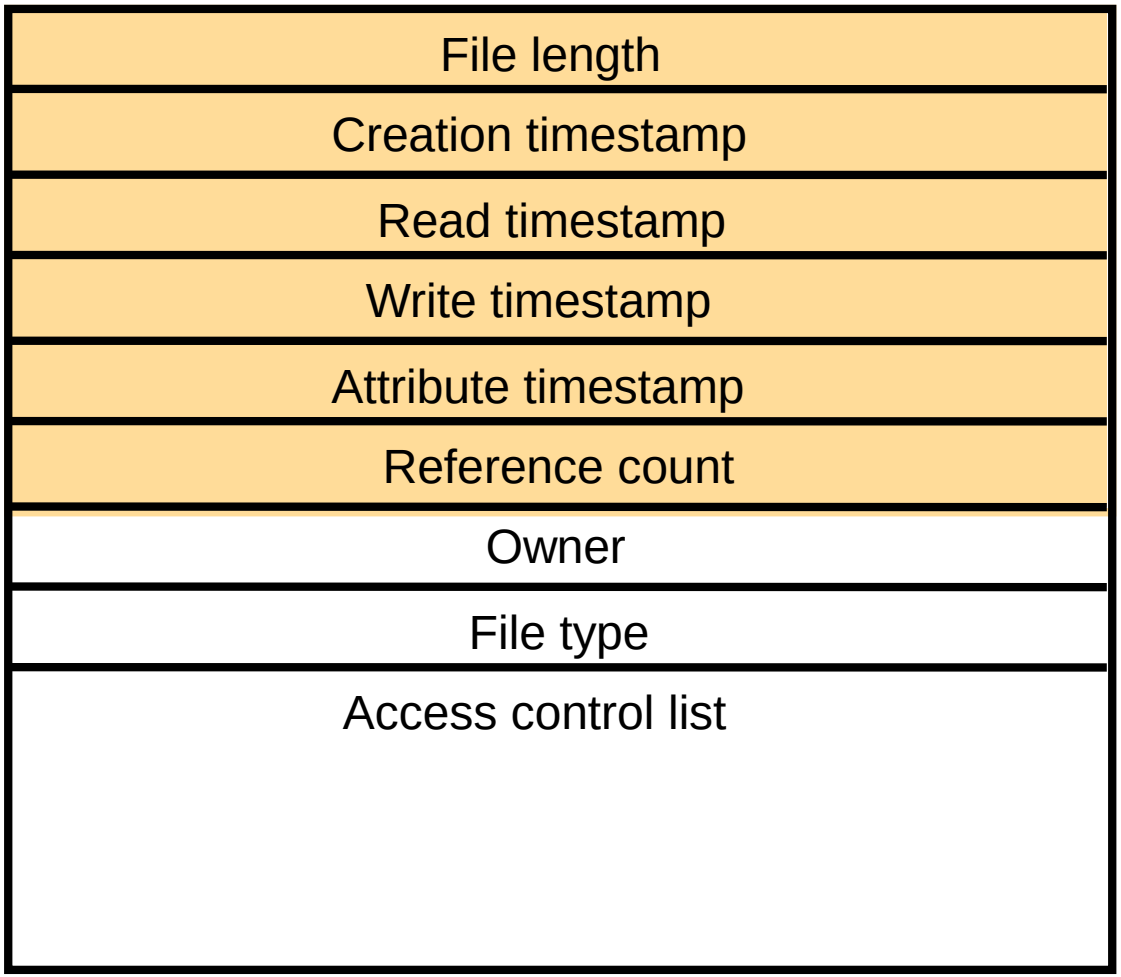
Types of consistency:
1: strict one-copy. : ✓ slightly weaker guarantees. 2: considerably weaker guarantees.

Characteristics of (non-distributed) file systems

- data and attributes (Fig 12.3)
- directory: mapping from text names to internal file identifiers
- layers of modules in file systems (Fig 12.2)
- file operation system calls in UNIX (Fig. 12.4)

File attributes

The shaded attributes are managed by the file system



File system modules

What else are needed for a distributed file service?

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Client-server communication and with the distributed naming and location of files.

UNIX 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*.

Distributed file system requirements(1)

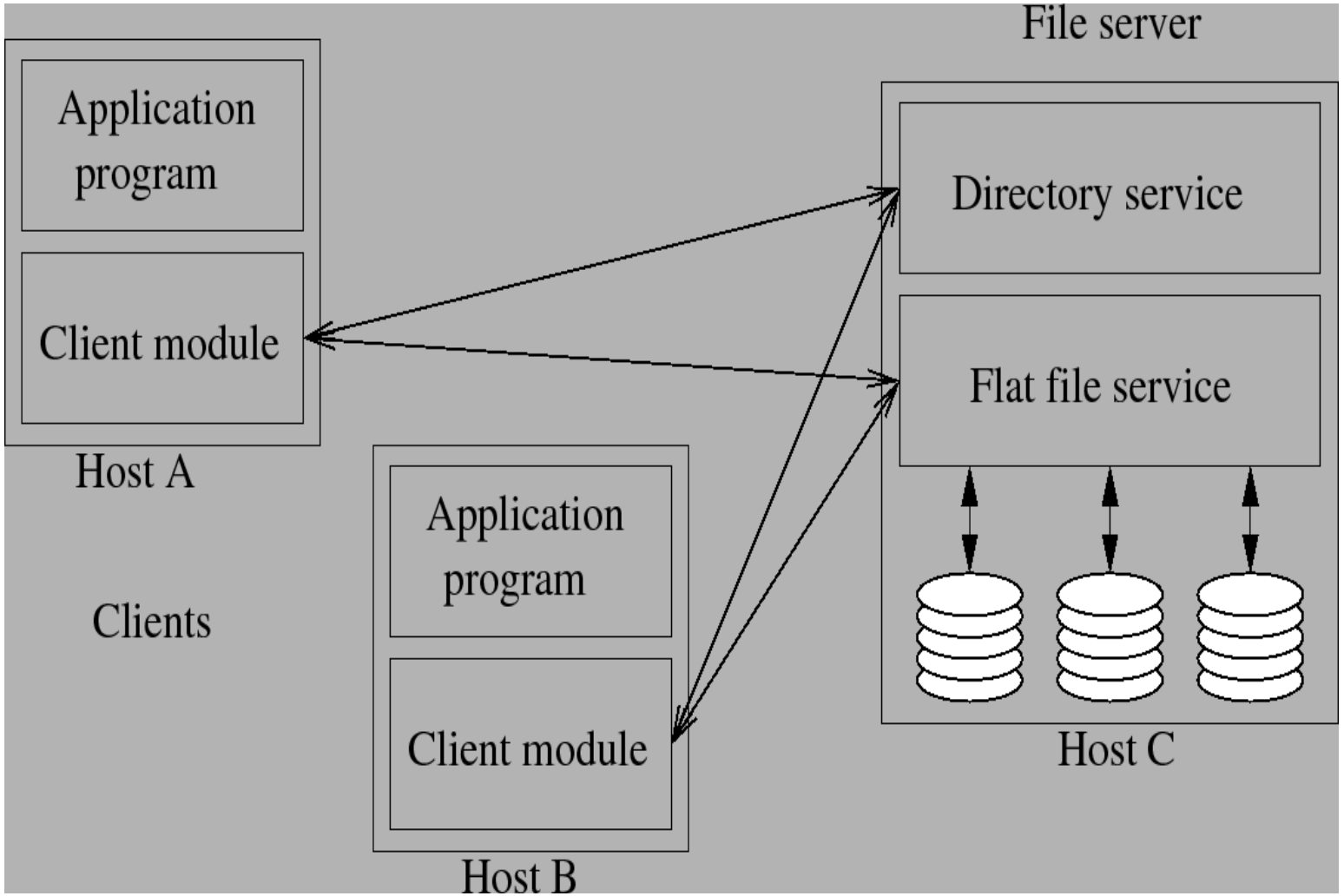
Transparency

- *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 files can, unchanged, operate on remote files.
- *Location transparency* - Client programs should see a uniform file name space; the names of files should be consistent regardless of where the files are actually stored and where the clients are accessing them from.
- *Mobility transparency* - Client programs and client administration services do not need to change when the files are moved from one place to another.
- *Performance transparency* - Client programs should continue to perform satisfactorily while the load on the service varies within a 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 system 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 several servers, that provides better capacity for accessing the file and better fault tolerance.
- **Hardware and operating system heterogeneity**
  - The service should not require the client or server to have specific hardware or operating system dependencies.
- **Fault tolerance**
  - Transient communication problems should not lead to file corruption. Servers can use at-most-once invocation semantics or the simpler at-least-once semantics with *idempotent* operations. Servers can also be *stateless*.
- **Consistency**
  - Multiple, possibly concurrent, access to a file should see a consistent representation of that file, i.e. differences in the files location or update latencies should not lead to the file looking different at different times. File meta data should be consistently represented on all clients.
- **Security**
  - Client requests should be authenticated and data transfer should be encrypted.
- **Efficiency**
  - Should be of a comparable level of performance to conventional file systems.

File service architecture



File service architecture

- 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 creates.

- Directory service

  The directory service provides a mapping between text names and their UFIDs. The directory service creates directories and can add and delete files from the directories. The directory service is itself a client of the flat file service since the directory files are stored there.

- Client module
  - integrate/extend flat file and directory services
  - provide a common application programming interface (can emulate different file interfaces)
  - stores location of flat file and directory services
  - Though the client module is shown as directly supporting application programs, in practice it integrates into a virtual file system.