Distributed Objects and Remote Method Invocation

Prof. Dave Bakken

Cpt. S 464/564 Lecture

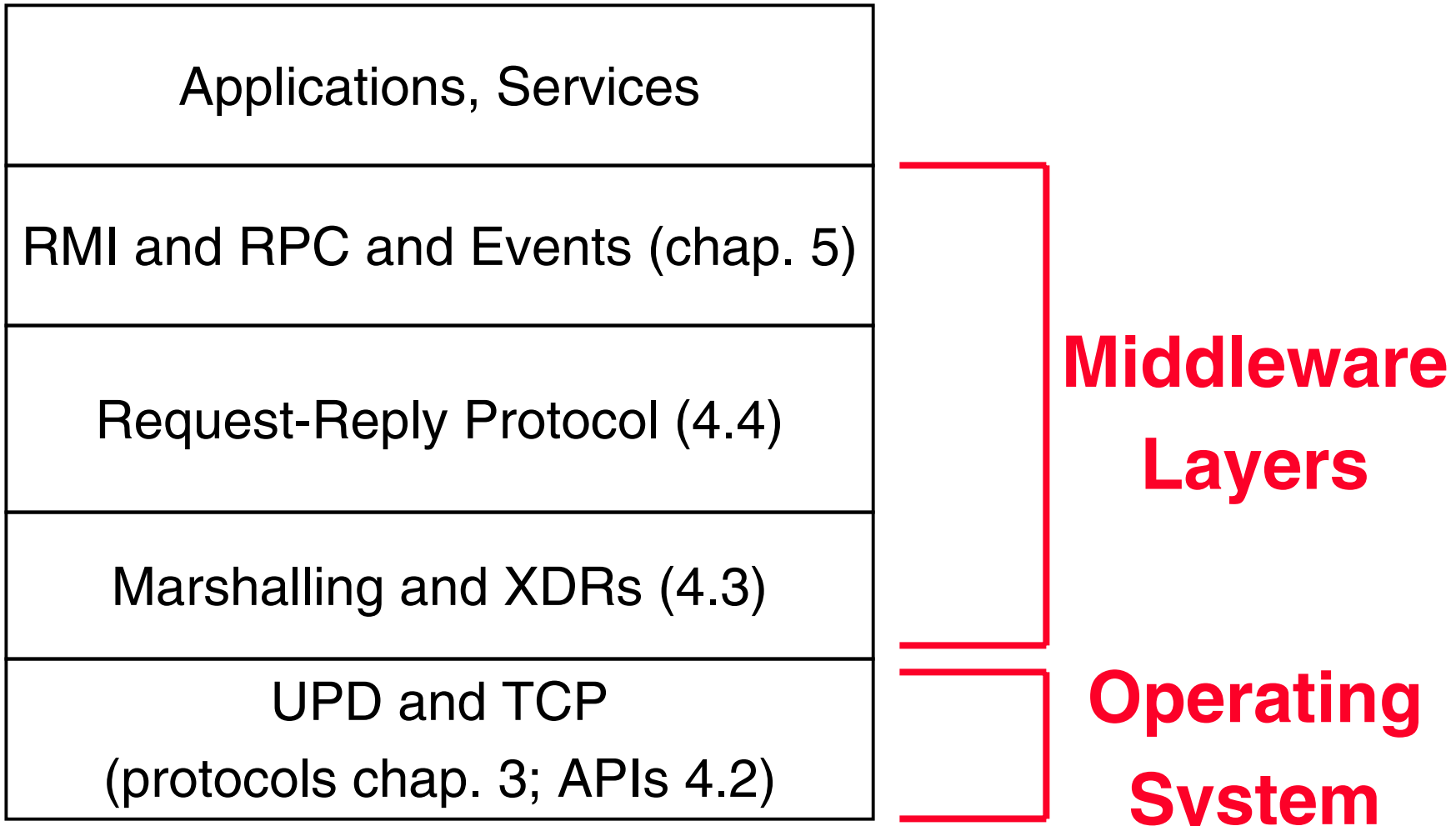
Textbook, Chapter 5

Oct 4+9, 2000

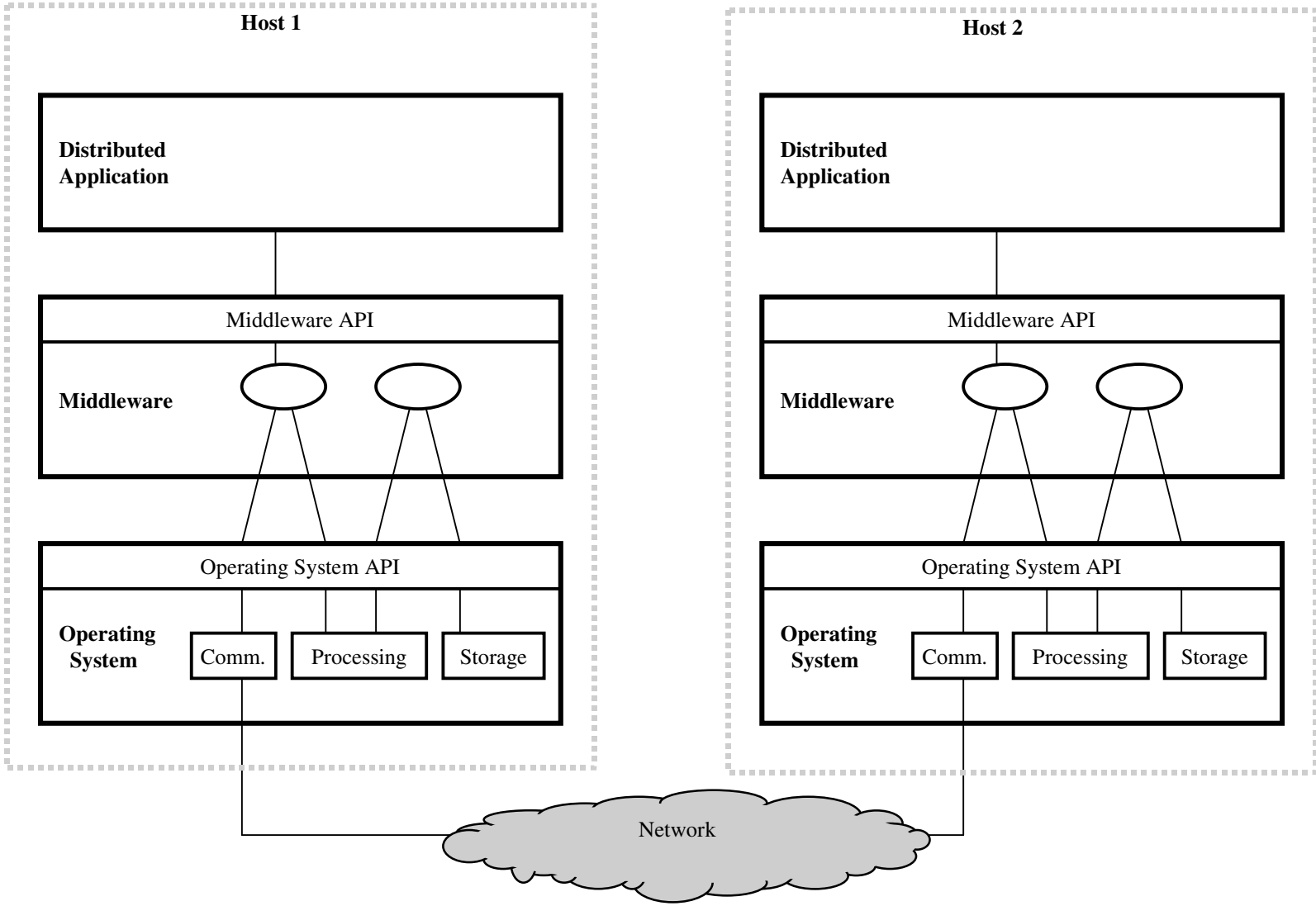
Introduction

- Distributed programs have pieces running in different processes
- So those processes need to be able to invoke remote operations
- Paradigms for remote invocations
  - Remote procedure call (5.3)
  - Remote method invocation (general RMI 5.2; Java RMI 5.5)
  - Distribute events and notifications (5.4)
- Difference between a collection of procedures and an object?

Middleware Layers and IPC



Middleware and Operating Systems



Goals of Higher-Level Middleware

- Location Transparency
- Heterogeneity across
  - Communication protocols
  - Computer hardware
  - Operating systems
  - Programming Languages
  - Vendor implementations
- How does middleware provide these?

Interfaces

- Modern languages let you
  - Organize a program into a set of modules that can communicate with each other
  - Export the operations that can be invoked on each module
- Interface: procedures and variables that can be accessed from other modules
  - Everything else is hidden from other modules: information hiding
  - Allows implementation to change much easier
- Distribute interfaces
  - Cannot access a variable directly
  - Pointers are invalid
  - Don't want to send all parameters in both direction: input and output parameters declared
  - Service interface: specification of procedures of a server available for use in clients
  - Remote interface: specification of methods of an object instance which may be invoked by clients

Interface Description Languages

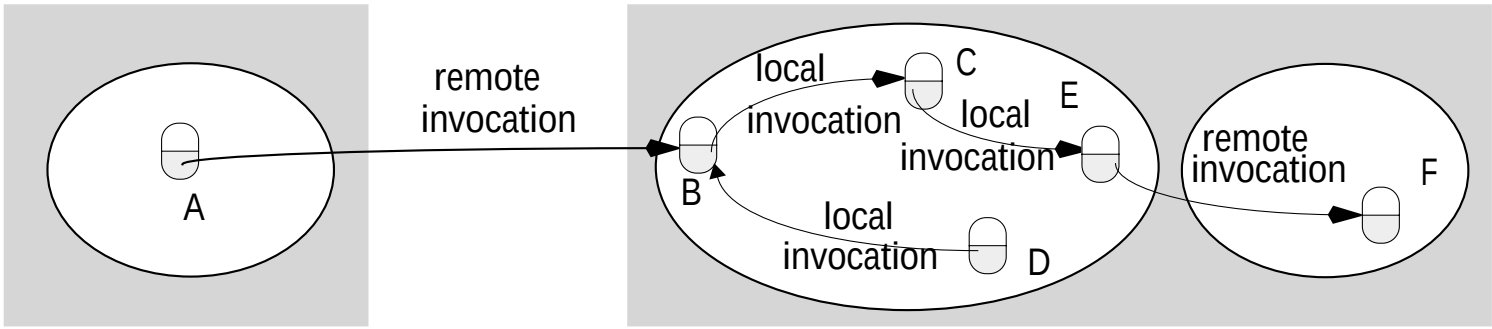
- Specification of interfaces in a separate language
- Allows for heterogeneity across programming languages
- Used to generate proxies, skeletons, ….
- OSF DCE (RPC), CORBA (distributed object), DCOM IDL (based on DCE), …
- Project #1 IDL:

```
module Grade {
        interface Grader {
                boolean add_grade(in string tid, in string pwd, in float grade);
                float show_grade(in string sid, in string pwd);
        };
        interface Security {
                boolean check_teacher_pwd(in string tid, in string pwd);
                boolean check_student_pwd(in string sid, in string pwd);
        };
};
```

Object Model

- So what exactly is an object???
- Collection of data and code
- Can be invoked via its methods
- Can have its public data members directly accessed
- Object references
  - How a caller invokes an object
  - First-class values: have a type, can be assigned to variables, passed as params, returned as return value
- Interface: signature of the methods that can be invoked
- Action in an OO program
  - Initiated by an object invoking another object's method
  - Can have two effects
    - Modify state of object
    - Call another object (nested)
- Exceptions: handling error or boundary conditions
- Garbage collection: freeing memory not in use (Java vs. C++ …)

Figure 5.3 Remote and local method invocations



Distributed Object Model

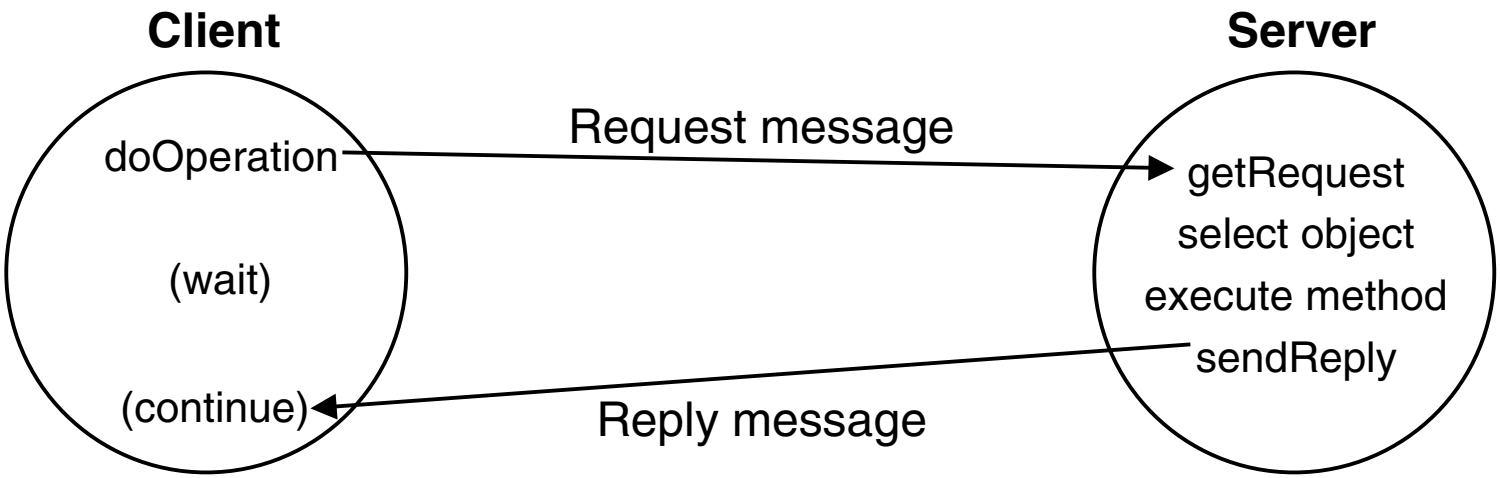
- Program consists of a collection of interacting objects
- Objects interact via remote method invocations
- Remote object: any object which can receive a remote invocation
- Remote object references
  - An identifier that is used to invoke a remote object
  - Can be used throughout a distributed system
  - Analogous to local object references
    - Identifies target object
    - May be passed as parameters and return values
  - Represented very differently from a local object reference
- Remote interface: specifies methods available for invocation
- Garbage collection: reclaim memory when no remote clients need object
- Exceptions: many more things can go wrong….

Distributed Objects vs. Other Middleware

Middleware Category	Communications Resources?	Processing Resources?	Storage Resources?
Distributed Relational Databases	Yes	Limited	Yes
Remote Procedure Call	Yes	Yes	No
Message-Oriented Middleware	Yes	No	Limited
Distributed Objects	Yes	Yes	Yes

Delivery Guarantees for RMIs

- Techniques for providing reliable delivery
  - Retry request message
  - Duplicate request filtering
  - Retransmission of replies



Invocation Semantics

- Maybe invocation semantics
  - Invoker cannot tell if a remote method has been executed
  - Can suffer from omission failures: request is dropped
  - Can suffer from crash failures: server fails
  - Useful only when these failures are acceptable
- At-least once invocation semantics
  - Invoker receives reply and knows method executed, or receives exception
  - Can suffer from crash failure: server fails
  - Can suffer from arbitrary failures: multiple executions cause errors
  - Useful only with idempotent operations
  - Provided by Sun RPC
- At-most-once invocation semantics
  - Invoker receives reply and knows method executed, or receives exception
  - Reply implies executed exactly once
  - Exception implies 0 or 1 executions
  - Provided by Java RMI and CORBA
- What are semantics of a local method call?

Figure 5.5 Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

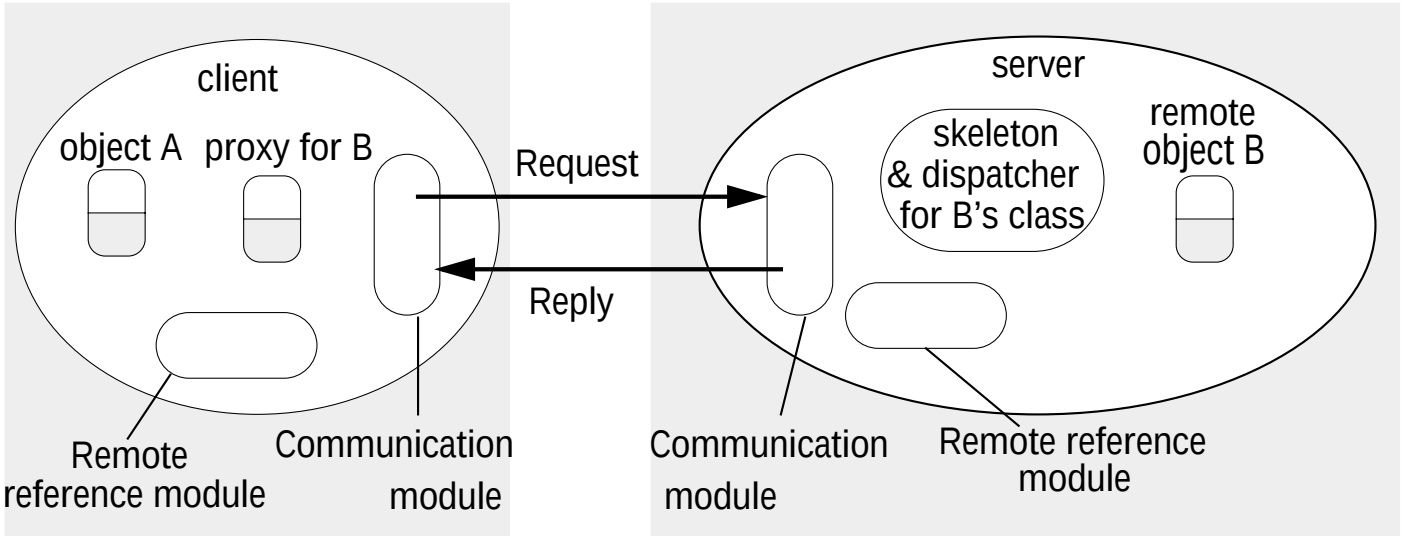
Transparency

- Original RPC [Birrell and Nelson, 1984] tried to make remote call like local one
  - No distinction in syntax
  - Hiding marshalling and message passing
  - Retransmissions hidden
- But local calls are not local ones!
- What to do?
  - Try to provide complete transparency
  - Allow hooks to abort a remote invocation that is taking too long (Argus)
  - Have separate syntax for remote interfaces (Sun Labs 1994)

Implementing RMIs

- Different (logical) modules on client and server sides…
- Communication module
  - On both client and server
  - (re)transmits request and reply messages
  - Uses fields in message (how?)
    - Message type (request or reply)
    - RequestID
    - MethodID
- Remote reference module
  - Translates between local and remote object references
  - Creates object references
  - Maintains a remote object table on each host
    - Server: remote object instances
    - Client: proxies for a given remote object

Figure 5.6 The role of proxy and skeleton in remote method invocation



Implementing RMIs (cont.)

- Proxy: provide access transparency to client
  - Implement exact same methods
  - Hide details of
    - Remote object reference details
    - Marshalling
    - Message sending
- Dispatcher
  - Demux point for communication software
  - Upcalls to method in skeleton
  - Uses methodID
- Skeleton
  - Unmarshalls arguments from request data structure
  - Upcalls to object instance
  - Marshalls reply data structure
  - Returns to dispatcher

Implementing RMIs (cont.)

- Binder
  - Maintains mappings from text names to remote object references
  - Lets clients find the remote object instances they need
  - E.g., VisiBroker osagent program
- Activation of remote objects
  - Activator: processes that start server processes or object instances
  - Active object: one that can accept invocations
  - Passive object: one that cannot, but can be made active

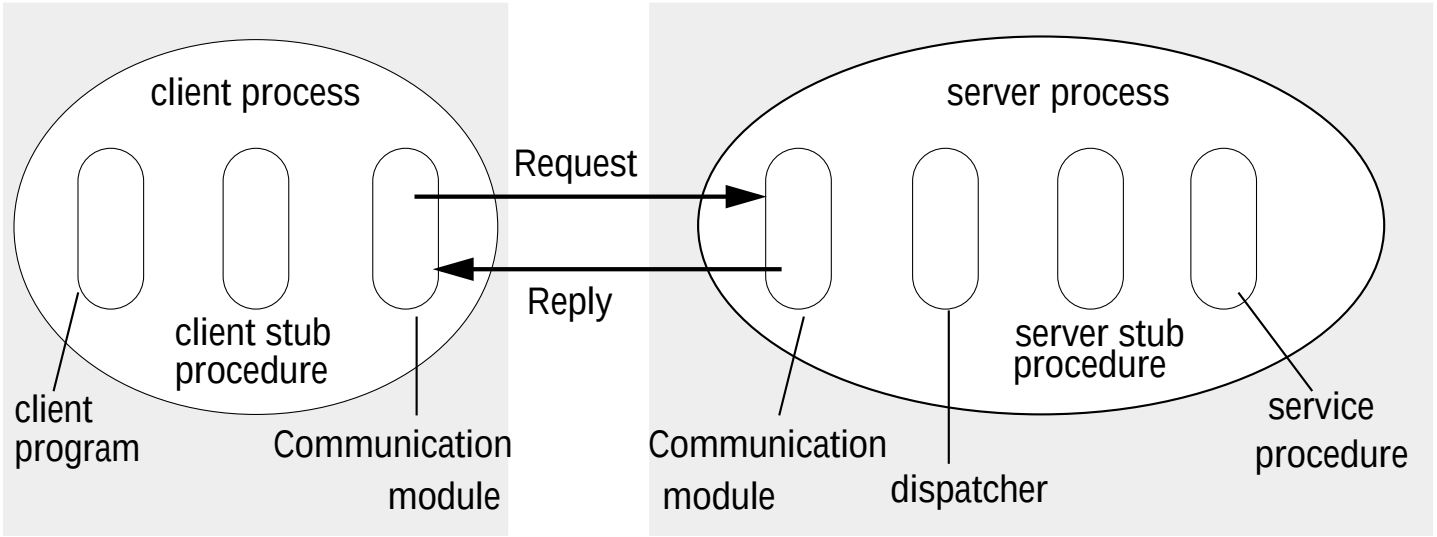
Distributed Garbage Collection

- Goals
  - If an object reference is still held anywhere, keep it existing (alt: active)
  - If nobody holds a reference, shut down object (alt: make it passive)
- Java distributed garbage collection algorithm
  - Based on reference counting
  - Works in conjunction with local garbage collector (GC)
  - Server tracks which processes that hold ref for each object it manages
  - Client gets reference: addRef(obj) invocation to server, then proxy created
  - Client's local GC notices proxy unreachable: removeRef(B) invocation to server, then delete proxy
  - Server: nobody else refers to object, then reclaim space
  - Tolerates comm. failures: addRef(obj) and removeRef(obj) idempotent
  - Tolerates client failures: leases
  - Why care about client failures here?

Remote Procedure Call (RPC)

- An RPC is similar to a remote method invocation
- Service interface: description of procedures that can be invoked in an RPC service
- Normal semantics provided are at-least-once or at-most-once

Figure 5.7 Role of client and server stub procedures in RPC



Sun RPC

- Developed in mid-1980s for Sun's Network File System (NFS)
- Sometimes called Open Network Computing (ONC)
- Came with Sun's Unix products and others'
- High-level implementation details
  - Can be implemented over either UDP or TCP
  - At-least-once call semantics
  - Broadcast RPC is an option
- Sun RPC interface language: "XDR"
- Interface compiler for XDR called rpcgen

Sun XDR

- Originally designed just to describe external data representations
- Later extended to be an interface language
- Meant for use with C
- May be used to define a service interface for Sun RPC by
  - Specifying set of procedure definitions
  - Specifying supporting type definitions
- Very primitive compared to CORBA IDL or Java
  - Not objects, but procedures (OK, its RPC…)
  - No service/program names, just program number and version number
  - No procedure names, just procedure number and procedure sugnature
  - Only one input parameter allowed (can be a struct)
  - Only output parameter is the return value (can be a struct)
- Notation for defining the expected things: constants, typedefs, …
- Rpcgen uses XDR code to generate
  - Client stub procedures
  - Server main procedure, dispatcher, and server stub procedures
  - XDR marshalling and unmarshalling procedures fro dispatcher and client and server stub procedures

Sun XDR Example (Figure 5.8)

```
const MAX = 1000;
typedef int FilePointer;
typedef int FileIdentifier;
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;
        DATA READ(readargs)=2;
    }=2;
}=9999;
```

Sample Sun RPC Client

```
/* File C.c – Simple client for the
 * FileReadWrite service */
#include …
main(int argc, char **argv) {
    CLIENT *clientHandle;
    char *serverName="foobar";
    readargs a;
    Data *data;

    /* create socket and client handle */
    clientHandle = clnt_create(
                    serverName,
                    FILEREADWRITE,
                    VERSION,
                    "udp");
    assert(clientHandle != NULL);
    a.f = 10;
    a.position = 1000;
    /* call stub */
    data = read_2(&a, clientHandle);
    …
    clnt_destroy(clientHandle);
}
```

Sample Sun RPC Server Procedures

```
/* File S.c – server procedures for
 * the FileReadWrite service */
#include …

void *write_2(writeargs *a) {
    /* do the actual writing */
}

data *read_2(readargs *a) {
    static Data result;  /* must be static */
    result.buffer = … /* file reading */
    result.length = … /* amt read */
    return &result;
}
```

Sun RPC Binding

- No network-wide naming (binding) service in Sun RPC
  - Clients must specify hostname of server when importing a service interface
- Sun RPC runs a port mapper:
  - Local binding service on each host
  - Runs on a well-known port on each computer
- Server startup: register with portmapper: program #, version #, port #
- Client startup
  - Finds out server's port by asking portmapper on that host
  - Gives program # and version #
- Problem: multiple service instances can run on multiple computers
  - May be on different ports
  - Q: how to let client multicast using direct broadcast?
  - A: broadcast to portmappers and the forward

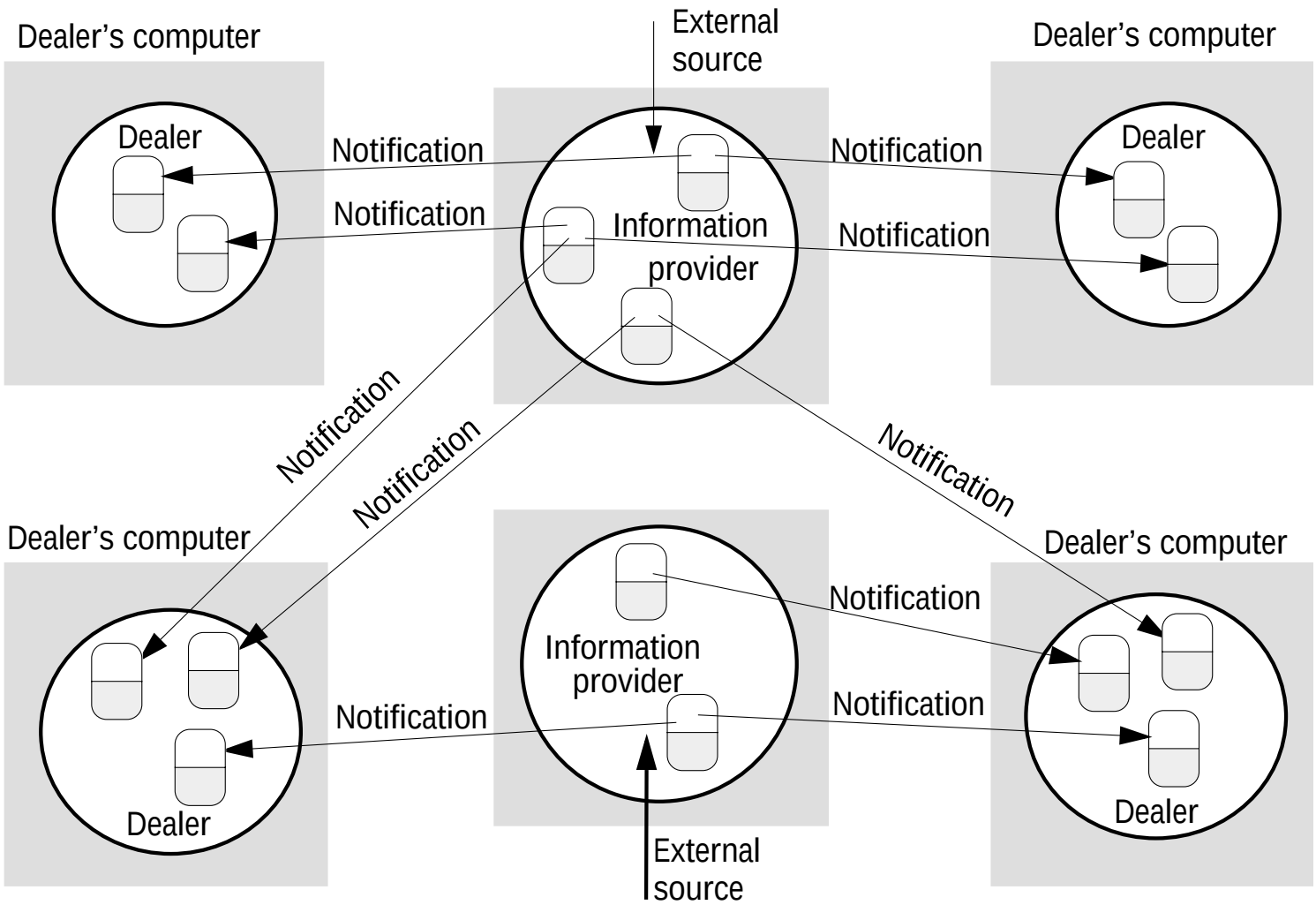
Events and Notifications

- Not all actions in all programs are initiated by a client invocation!
- Events allow one object to react to a change happening in another
  - Mouse click on computer
  - Price of a stock changing
  - Modification to a document
  - A person with a smart badge entered a room
- Objects that care about an event are notified when the state changes
- AKA publish and subscribe paradigm
  - One object publishes type of events it makes available, then sends a stream of events
  - Other objects that want to get events from the object subscribe or register interest
- Objects representing events are notifications
- Main characteristics of event-based systems
  - Heterogenous: can glue together components not designed to work together
  - Asynchronous: publishers and subscribers are decoupled

Events (cont.)

- An event source can generate events from one or more types
- Subscribers subscribe using
  - Type
  - Attributes (e.g., name or identifier, values, …)
- Publishers send an event when one matches type and attributes
- Q: examples of matching?
- Simple event-based dealing room system (Figure 5.9)
  - Allow “dealers” of stocks (“traders” in the US) to get latest information on prices of stocks they care about
  - Information providers provide a constant stream of new information
  - Each update to a stock object is an event

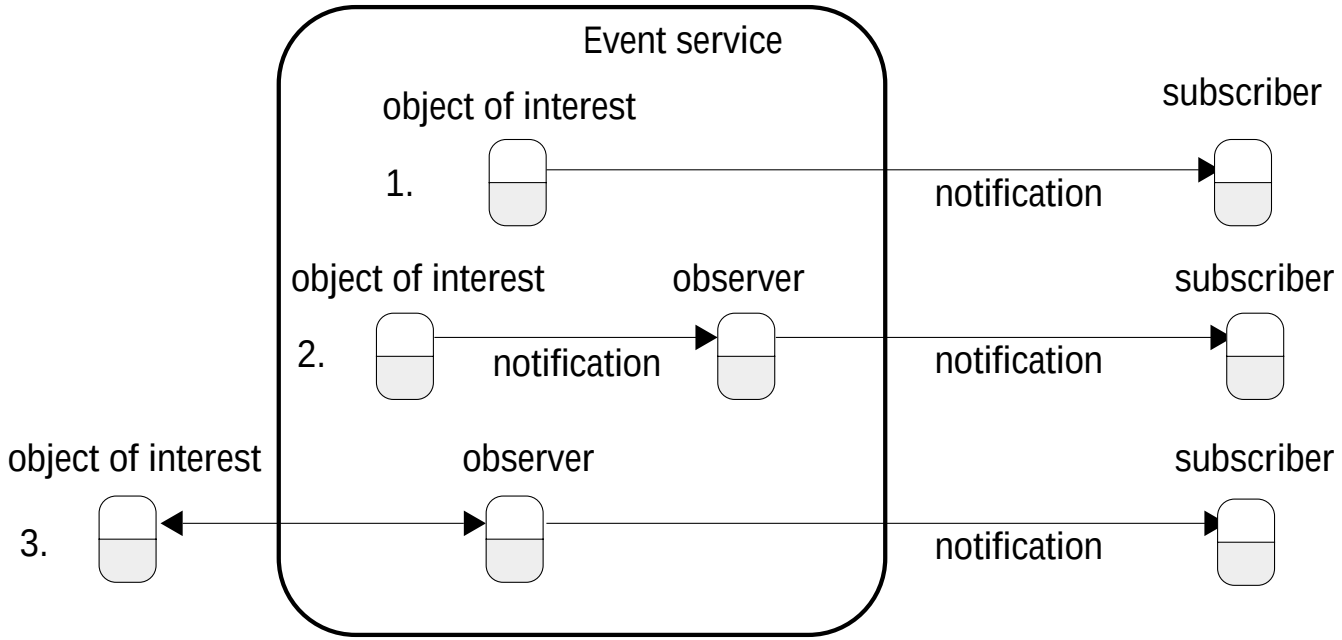
Figure 5.9 Dealing room system



Distributed Event Notification Roles

- Basic idea: support decoupling between publishers and subscribers
- Roles involved in a sample event system supporting this:
  - Object of interest: the object that experiences changes of state that others care about
  - Event: a method execution completing execution
  - Notification: an object containing event info
  - Subscriber: an object that has registered to receive notifications about some object of interest
  - Observer objects: an intermediate object to decouple publishers and subscribers
  - Publisher: object that declares it will generate notifications

Figure 5.10 Architecture for distributed event notification



Sample Event Example (cont.)

- Delivery Semantics
  - Suffers from failure model of whatever underlying transport is used…
  - Some systems need reliable multicast
  - Some systems need realtime guarantees
  - E.g., TIBCO (www.tibco.com) and stock traders
- What observers are used for
  - Forwarding
  - Filtering of notifications: by values on one object of interest
  - Patterns of events: multiple events in one or more object of interest
  - Notification mailboxes: store and forward

Jini Distributed Event Specification

- Jini allows a subscriber in one JVM to receive notifications from an object of interest in another JVM
- Chains of observers may be inserted between object of interest and subscriber
- Main objects involved
  - Event generators: object that allows other objects to subscribe to its events, and generates notifications
  - Remote event listeners: objects that can receive notifications
  - Remote events: objects passed by value to remove event listeners
    - I.e., this is what is called “notifications” above
  - Third-party agents: objects that may be interposed between an object of interest and a subscriber
    - Can be set up by event generator or subscriber to provide different QoS or implement different policies
- Constrast with CORBA’s Event Service later
  - CORBA-IV lecture
  - Project 5

Java RMI

- Read text chapter 5.5: not covering in class
- Don't worry about
  - Low-level details
  - Memorizing class names
  - …
- Do
  - Understand high-level features
  - Similar features, tools, hooks, etc. compared to CORBA