Using IDLs

- To generate this thin layer between the caller and the RMI subsystem, and the RMI subsystem and the callee, you invoke an IDL compiler.
- It emits the code that both sides link against such that the method calls are transparently made through RMI.
- The benefit is transparency from the code perspective.
- The cost is some extra work at compile time to build and link this extra layer of code.
  - See page 189 for details on this layer of code.

Object models

- To understand the impact of being distributed on a distributed object system, we'll briefly revisit the Java/C++ object model.

Object model for Java/C++

- *Object references*: Instances of objects are referred to through references to their locations in memory. Object references are first class values, so they can be passed around and returned.
- *Interfaces* provide the signature an object provides without specifying the implementation.
  - E.g.: Java Interfaces, C++ class definitions
- *Actions* are performed by callers invoking methods on objects. A method invocation can:
  - Change the state of the receiver
  - Create a new object.
  - Result in other methods in other objects being called.

Object model for Java/C++

- *Exceptions*: Error conditions can result in exceptions being thrown. Exceptions are passed up the call graph from callee to caller until a method provides code to catch the exception.
- *Garbage collection*: In languages like Java, memory is not explicitly deallocated most of the time, so the runtime system is responsible for identifying memory no longer referred to and deallocating it automatically.
  - C++ does not have garbage collection, although “smart pointers” can help alleviate the need for explicit deallocation.

Distributed objects

- A distributed object program is composed of clients and servers.
  - Servers manage instances of objects accessible in the distributed system.
  - Clients invoke methods on these objects. Clients may contain objects themselves, but not necessarily ones that can be accessed by other processes.
- The request/reply protocol discussed earlier can be used to dispatch method calls from clients over the communication subsystem to servers, perform the computation, and reply with the result.

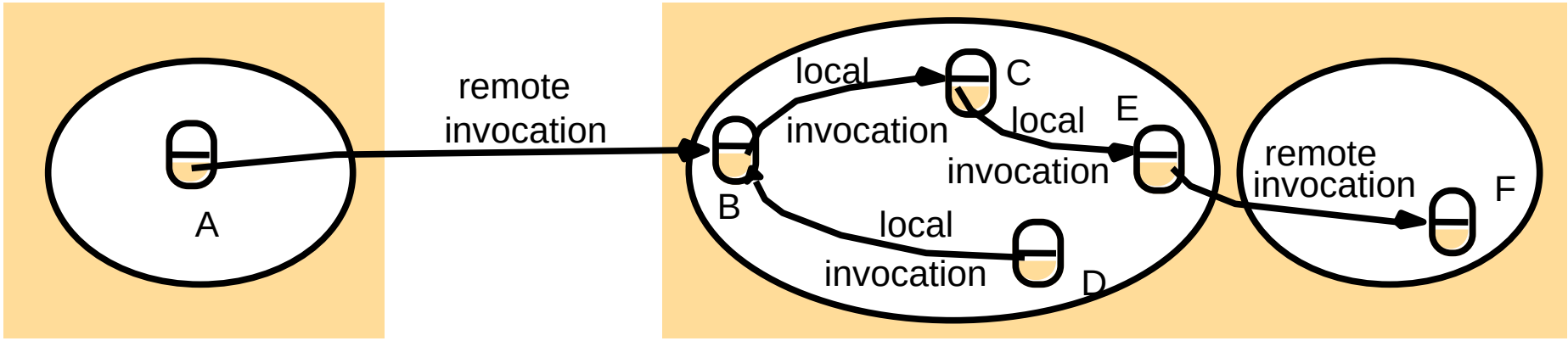
Distributed object model

- Distributed objects don’t inherit the object model of their parent languages completely. The distributed nature of the program adds constraints that may change the meaning of the base model.

Remote object references

- We saw earlier that a remote object reference is a structure that encapsulates the address/port information of the server and some unique identifier of the actual instance within that server that the remote object reference refers to.
- What changes? Method calls act the same as local object references.
- Access to public data may be restricted though. Some systems provide this for remote references if a proper setter/getter pattern is followed. Otherwise, they prohibit it.

Remote invocation illustration



Interfaces

- The interface that is callable remotely is either:
  - Described by an IDL
    - This is true with CORBA, SunRPC, Babel.
  - Provided by extending an interface
    - This is how Java works with the `Remote` interface.

Actions

- For local objects, actions occur as they would in a non-distributed case.
- For remote object references, the RMI system helps make the remote invocation occur.
  - Actions leading to instantiations of new objects will result in the new object being in the process that “owns” the method, not the caller.

Garbage collection

- In garbage collected languages, distributed objects must “play well” with the native garbage collector.
  - I.e.: Remote object references should respect the reference counting scheme that the server uses.
- Distributed garbage collection requires tracking references on both sides.

Basic Distributed Garbage Collection

- Client has a proxy object for concrete object existing on server.
  - When client GC sees that proxy object isn't reachable anymore, it notifies server that the remote reference count should be decreased.
- Server maintains list of known clients that hold references to local objects.
  - GC collects an object after both remote and local references are all zero.
- Leases on objects allow server to deal with clients that vanish without proper `removeRef()` calls.
  - Otherwise, client crashes could cause server-side memory leaks.

Exceptions

- Nothing really special for exceptions. If an exception occurs on the remote side, the RPC system must respect the “throws” property of a method to transport the Exception to the client to handle.
  - Easily achieved in responses to client.
- Additional exceptions may be used to deal with RMI-related error conditions on the client side, such as a timeout on a method invocation.

Invocation semantics

- When describing the request/reply protocols, we noticed that the potential for request or reply loss could lead to multiple function invocations on the server side (or none at all).

- We have three different invocation semantics that are used to describe what an RMI system can provide with respect to how many times a remote method can be invoked.

Invocation semantics

- **Maybe invocation semantics**: The remote method may be executed one or zero times.
  - Omission failure may result in request never arriving.
  - Crash of server may cause result to never be delivered.
  - After the caller times out, it can't know whether or not the action actually occurred. If the result arrives after the timeout, this is considered to be a non-completion by the caller since it has moved on.
  - Useful in applications where occasional failures to execute remote methods can be tolerated.

Invocation semantics

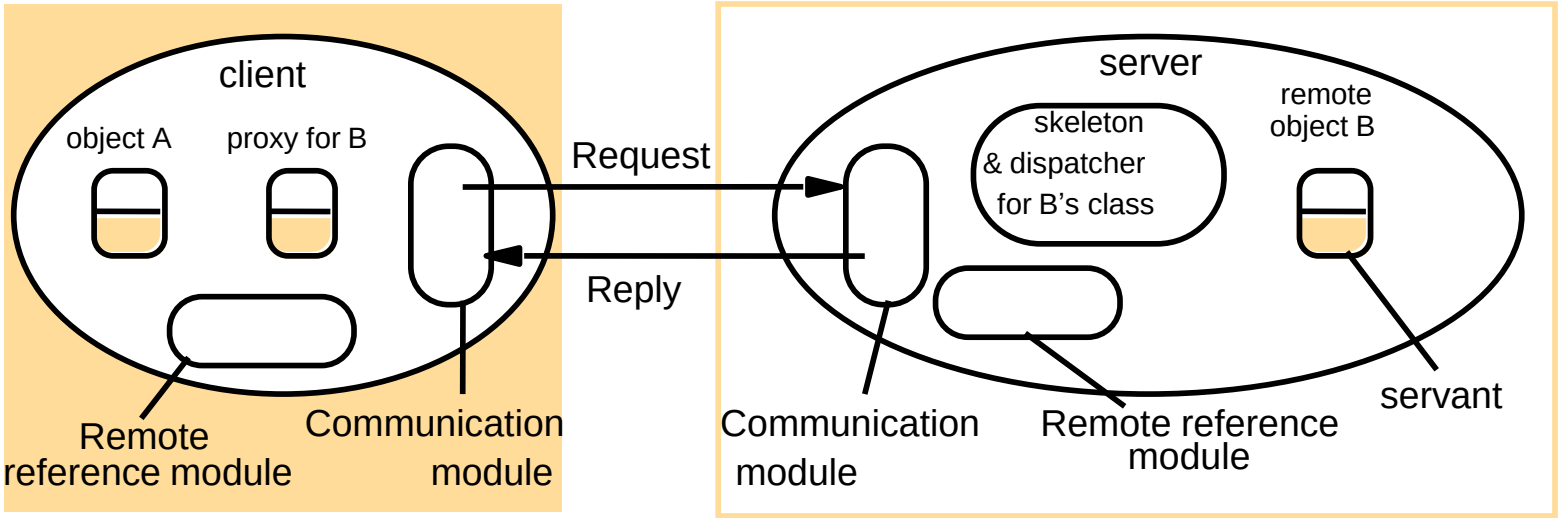
- **At-least-once semantics**: The caller receives a result if the method was executed at least once, and an exception otherwise.
  - If a method executes and the caller never receives the message, resulting in a retry, we may see the method executed multiple times.
  - If the method is idempotent, this is fine.
    - If not, this could lead to correctness problems.
  - These semantics clearly are OK if all operations on the server side are idempotent.

Invocation semantics

- **At-most-once semantics**: If a result is received by the caller, it knows the method executed exactly once. If the method does not execute, and exception occurs.
  - Requires all fault-tolerance measures to be in places to guarantee safe delivery of the request and result.
  - Works for operations that are not idempotent.
  - Java RMI uses at-most-once semantics. CORBA allows at-most-once, and maybe semantics only in the case where no return value results from the operation.
  - SunRPC provides at-least-once semantics.

Implementation details

- To make RMI happen, the software is broken up into many interacting pieces.



Implementation details

- Communication module
  - The communication module is concerned with the request/reply protocol, such as maintaining the requestID counter and talking to the underlying UDP or TCP layer.
  - Marshalling and the addressing of specific methods is handled in the RMI software.
- Remote reference module
  - The remote reference module manages the remote object references, and handling their local proxies.
  - Primarily acts as a translator between remote object references and local objects.

Implementation details

- Servants
  - This is the object instance on the server that actually provides the concrete methods that are executed.
- RMI software
  - Layer between the application and the communication and remote reference modules.
  - Proxies live here. Since we want to make remote objects transparent to the caller, proxy objects sit in the client that provide the same interface as the remote objects, but their methods pass invocations to the comm. and remote reference modules to be handed to the server for execution.

Implementation details

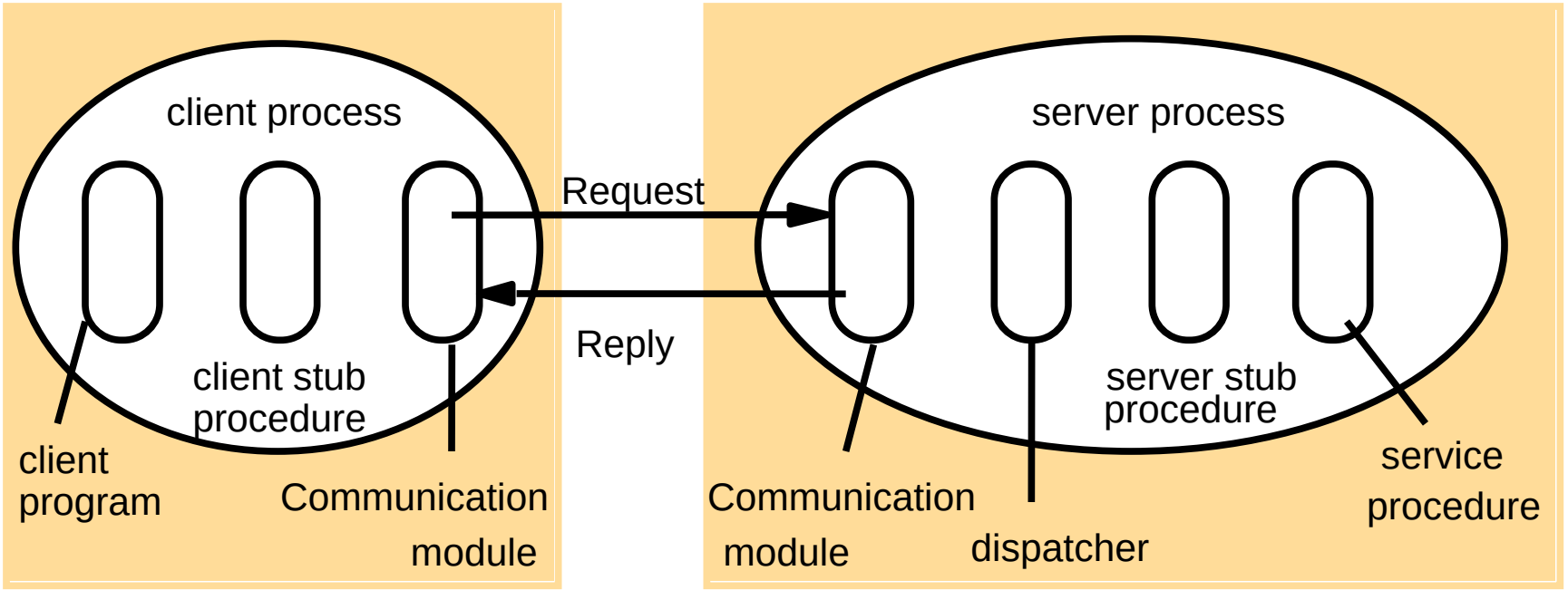
- Dispatcher
  - The dispatcher on the server side translates the methodID from the request method into a method invocation on the remote object skeleton.
- Skeleton
  - The remote object is not called directly by the dispatcher. Instead, an object with the same methods is placed between the dispatcher and actual remote object. It has the responsibility of unmarshalling the data contained within the request message, invoking the function, and marshalling the return values to return in a reply.

IDLs, Proxies, and Skeletons

- Remember: the IDL describes the abstract interface that a remote object provides and that a client invokes.
- We need a thin layer on both sides to marshal data from the native representation for the caller or callee, pass it to the underlying communication system, and unmarshal and invoke methods on the other side.
- These are the proxies and skeletons.
- The IDL compiler typically creates these when the IDL interfaces are compiled.

RPC

- Very similar to RMI, but simpler because no remote object references are required.



Events and notifications

- Event-based systems are ones in which an object reacts to a change that occurs in another object.
- GUIs typically are based on events.
  - An example event is pushing the mouse button.
- There is asynchronous interaction with events.
  - The receiver typically has no idea when an event might come in.
- Objects that receive events call the receipt of an event *notification*.

Publish/subscribe

- Providers of events *publish* them.
  - In the GUI world, this might be an object saying that it will provide the “rightClick”, “leftClick”, and “mouseMove” events.
- Objects that wish to receive events when they occur in a specific provider will *subscribe* to them.
- When an event occurs, a publisher will send the event to all subscribers.
- Event objects are called *notifications*.

Events and distributed systems

- Examples of events in a distributed context.
  - Chat room participant leaves the chat room.
  - A device joins a network.
  - A data element is modified on a server that a set of clients are watching.
  - E-mail arrives.

Events and heterogeneity

- Events make systems not designed in advance to work together actually interoperable.

Interesting event designs

- Some interesting design patterns arise in event-based systems.
  - **Forwarding**:
    - These are subscribers who act as providers to other subscribers. You can imagine a hierarchy of these in some cases.
  - **Filters**:
    - Subscribers who filter out events and provide those that pass the filter to subsequent subscribers.
  - **Pattern subscribers**:
    - Some subscribers are not interested as much in individual events as they are in patterns amongst multiple events.
  - **Mailboxes**:
    - Allows events to be cached safely in the event that the subscriber is not available when the event occurs.

Multicast and group communications

- So far we've focused on point-to-point communications.
- In some cases we want group communications.
  - E.g.: Sending a message to a group of processes in a single operation, not a set of point-to-point operations.
- The simplest method (and least flexible for arbitrary sets of machines) is the special IP host ID composed all of ones.
  - E.g.: 128.223.255.255, 192.168.1.255
  - This simply broadcasts to all hosts connected to the network specified in the NetworkID part of the address.
- One of the goals is transparency of the members of the group to the sender. The sender just specifies the group and message, and the multicast layer gets it to the members.

Multicast and group communications

- Multicast is useful in distributed systems with characteristics like:
  - **Fault tolerance based on replicated services**
    - Client messages sent to all servers providing the services, each of which performs the same operation. If a server fails, it's OK, since others will have done the work.
  - **Discovery servers for spontaneous networking**
    - Use multicast IPs to dynamically find resources on a network (e.g.: a printer).
  - **Data replication for better performance**
    - Many machines contain replicas of data, and updates are broadcast to all data holders.
  - **Propagation of event notifications**
    - Consider a set of processes subscribed to an event to be a multicast group.

IP multicast

- IP multicast is built on top of IP.
- A multicast group is identified by a class D IP address (first 4 IPv4 bits are 1110).
- A client receives UDP messages on an ordinary port, and joins a multicast group to participate.
  - So, basic multicast suffers UDP-style failures.
- These multicast groups can be on the local network or Internet.
- Those on the Internet make use of multicast routers to propagate.
  - A time to live (TTL) specifies how many multicast routers a message can traverse through to limit how far a multicast message goes on the Internet.

IP multicast

- Allocation of multicast addresses is a little tricky on the Internet.
  - 224.0.0.1 through 224.0.0.255 are reserved.
  - The rest are available for temporary purposes.
  - For small local groups, a small TTL is often enough to avoid conflicts by limiting how far the packets can go.
  - Otherwise, the session directory (sd) program can be used to start and join multicast sessions. This tool keeps track of and broadcasts known multicast sessions to help both find ones that a program is looking for, and determine what multicast IPs are available.

Moving on

- So far we've covered:
  - Basic concepts and models in distributed systems
  - Networking topics: How you make processes on different systems talk to each other.
  - Abstraction layers and separation of concerns in distributed systems.
  - Remote procedure call/ remote method invocation.
  - Multicast and group communication concepts.
    - We'll talk more about this soon in more detail.