Distributed Systems

Chapter 3 Distributed Objects and Remote Invocation

5.1 Communication between Distributed Objects

Topics to be addressed

- Distributed objects
- The distributed object model
- Design issues
- Implementation
- Distributed garbage collection

5.2.2 Distributed Objects

- Physical distribution of objects into different processes or computers in a distributed system
  - Object state consists of the values of its instance variables
  - Object methods invoked by remote method invocation (RMI)
  - Object encapsulation: object state accessed only by the object methods
- Usually adopt the client-server architecture
  - Basic model
- Objects are managed by servers and
- Their clients invoke their methods using RMI

– Steps

1. The client sends the RMI request in a message to the server
2. The server executes the invoked method of the object
3. The server returns the result to the client in another message

– Other models

- Chains of related invocations: objects in servers may become clients of objects in other servers
- Object replication: objects can be replicated for fault tolerance and performance
- Object migration: objects can be migrated to enhancing performance and availability

5.2.3 The Distributed Object Model

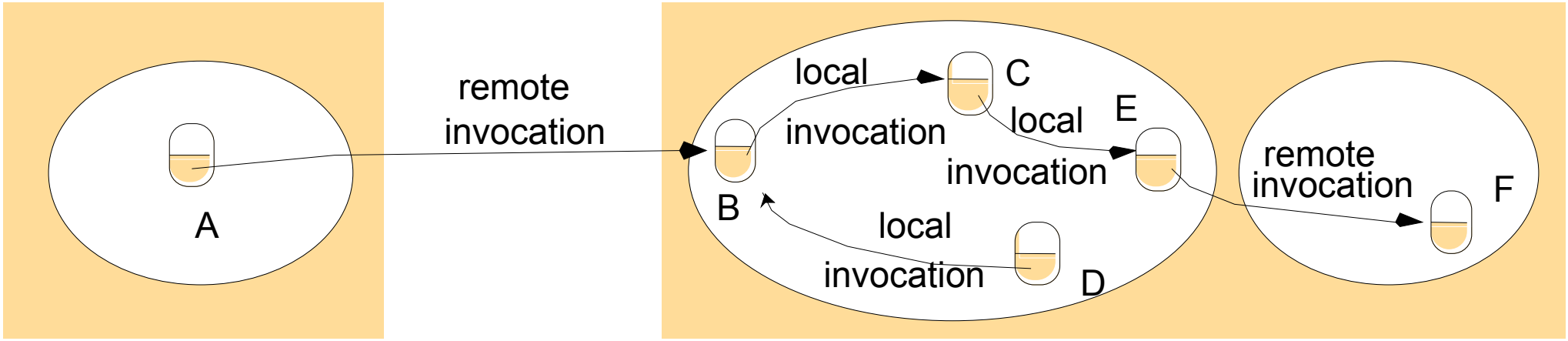

Figure 5.3 Remote and local method invocations

Two fundamental concepts: Remote Object Reference and Remote Interface

- Each process contains objects, some of which can receive remote invocations are called *remote objects* (B, F), others only local invocations
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods, called *remote method invocations*

– Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely

Five Parts of Distributed Object Model

- Remote Object References
  - accessing the remote object
  - identifier throughout a distributed system
  - can be passed as arguments
- Remote Interfaces
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - Interface Definition Language (IDL) used for defining remote interface

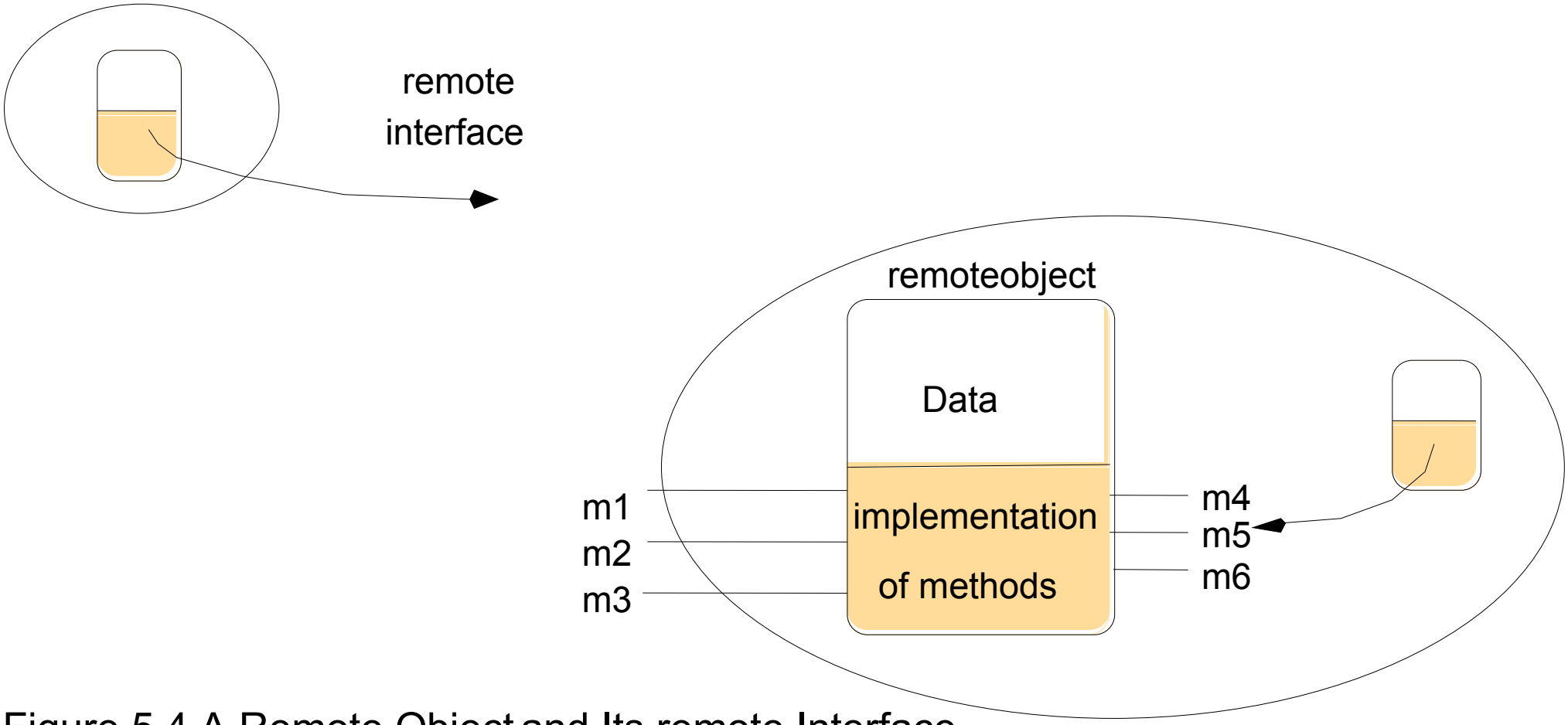

Figure 5.4 A Remote Object and Its remote Interface

{

Five Parts of Distributed Object Model (cont.)

- Actions
  - An action initiated by a method invocation may result in further invocations on methods in other objects located in difference processes or computers
  - Remote invocations could lead to the instantiation of new objects, ie. objects M and N of Figure 5.5
- Exceptions
  - More kinds of exceptions: i.e. timeout exception
- RMI should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked
- Garbage Collection
- Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

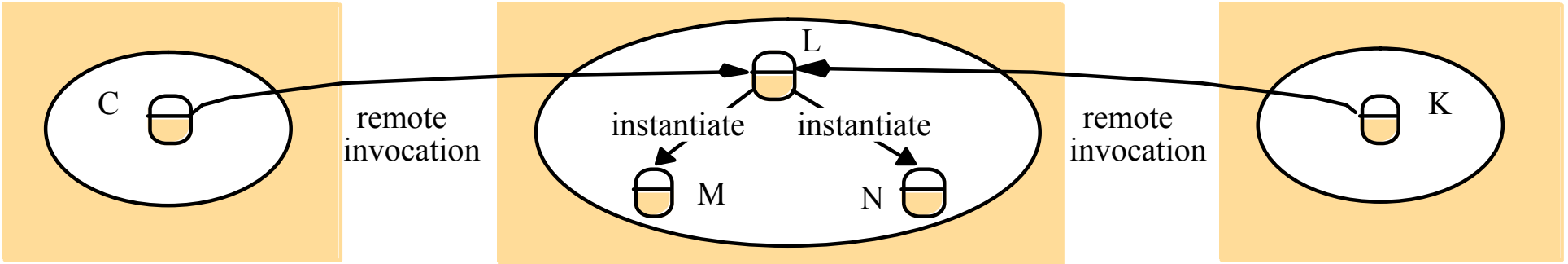

Figure 5.5 Instantiating Remote Objects

5.2.4 Design Issues for RMI

- Two design issues that arise in extension of local method invocation for RMI
  - The choice of invocation semantics
    - Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error
      - Either request or reply message may be lost
      - Either server or client may be crashed
  - The level of transparency
    - Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics

- Error handling for delivery guarantees

– Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed

– Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server

– Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations

- Choices of invocation semantics

– Maybe: the method executed once or not at all (no retry nor retransmit)

– At-least-once: the method executed **at least** once

– At-most-once: the method executed **exactly** once

Fault tolerance measures *Invocation*

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

Figure 5.6 Invocation semantics: choices of interest

RMI Design Issues: Transparency

- Transparent remote invocation: like a local call
  - marshalling/unmarshalling
  - locating remote objects
  - accessing/syntax
- Differences between local and remote invocations
  - latency: a remote invocation is usually several order of magnitude greater than that of a local one

  - availability: remote invocation is more likely to fail
  - errors/exceptions: failure of the network? server? hard to tell
- syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
  - consistency on the remote machine:
- Argus: incomplete transactions, abort, restore states [as if the call was never made]

5.2.5 Implementation of RMI

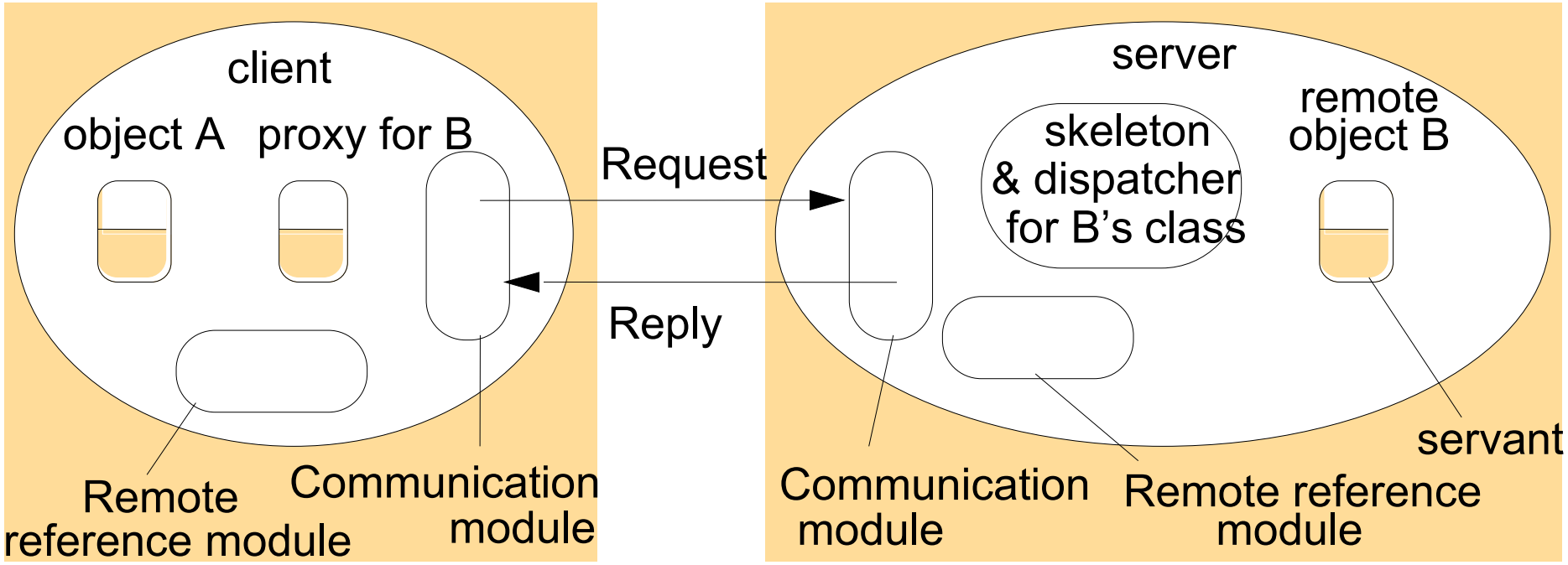

Figure 5.7 Role of proxy and skeleton in RMI

- Communication module
  - Two cooperating communication modules carry out the request-reply protocols: message type, requestID, remote object reference
    - Transmit request and reply messages between client and server
    - Implement specific invocation semantics

–The communication module in the server

- selects the dispatcher for the class of the object to be invoked,
- passes on local reference from remote reference module, • returns request

Implementation of RMI (cont.)

- Remote reference module
  - Responsible for translating between local and remote object references and for creating remote object references
  - remote object table: records the correspondence between local and remote object references
- remote objects held by the process (B on server)
- local proxy (B on client)

– When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table

- Servant
  - An instance of a class which provides the body of a remote object – handles the remote requests

- RMI software
  - Proxy: behaves like a local object, but represents the remote object
  - Dispatcher: look at the methodID and call the corresponding method in the skeleton
  - Skeleton: implements the method

  Generated automatically by an interface compiler

Implementation Alternatives of RMI

- Dynamic invocation
  - Proxies are static—interface complied into client code
  - Dynamic—interface available during run time
- Generic invocation; more info in “Interface Repository” (COBRA)
- Dynamic loading of classes (Java RMI)
- Binder

– A separate service to locate service/object by name through table mapping for names and remote object references

- Activation of remote objects

– Motivation: many server objects not necessarily in use all of the time

• Servers can be started whenever they are needed by clients, similar to inetd–

Object status: active or passive

- active: available for invocation in a running process
- passive: not running, state is stored and methods are pending – Activation of objects:
- creating an active object from the corresponding passive object by creating a new instance of its class
- initializing its instance variables from the stored state

– Responsibilities of ***activator***

- Register passive objects that are available for activation
- Start named server processes and activate remote objects in them
- Keep track of the locations of the servers for remote objects that it has already activated

Implementation Alternatives of RMI (cont.)

- Persistent object stores
  - An object that is guaranteed to live between activations of processes is called a ***persistent object***
  - Persistent object store: managing the persistent objects
- stored in marshaled from on disk for retrieval

- saved those that were modified

– Deciding whether an object is persistent or not:
  - persistent root: any descendent objects are persistent (persistent Java, PerDiS)
  - some classes are declared persistent (Arjuna system)
  - Object location
– specifying a location: ip address, port #, …
– location service for migratable objects
  - Map remote object references to their probable current locations
  - Cache/broadcast scheme (similar to ARP)
– Cache locations
– If not in cache, broadcast to find it
  - Improvement: forwarding (similar to mobile IP)

5.2.6 Distributed Garbage Collection

- Aim: ensure that an object
  – continues to exist if a local or remote reference to it is still held anywhere
  – be collected as soon as no object any longer holds a reference to it

- General approach: reference count
- Java's approach
  - the server of an object (B) keeps track of proxies
  - when a proxy is created for a remote object
- addRef(B) tells the server to add an entry
  - when the local host's garbage collector removes the proxy
    - removeRef(B) tells the server to remove the entry
  - when no entries for object B, the object on server is deallocated

5.3 Remote Procedure Call

- client:
  "stub"
  instead of
  "proxy"

(same function, different names) – local call, marshal arguments, communicate the request

- server: – dispatcher

– "stub": unmarshal arguments, communicate the results back

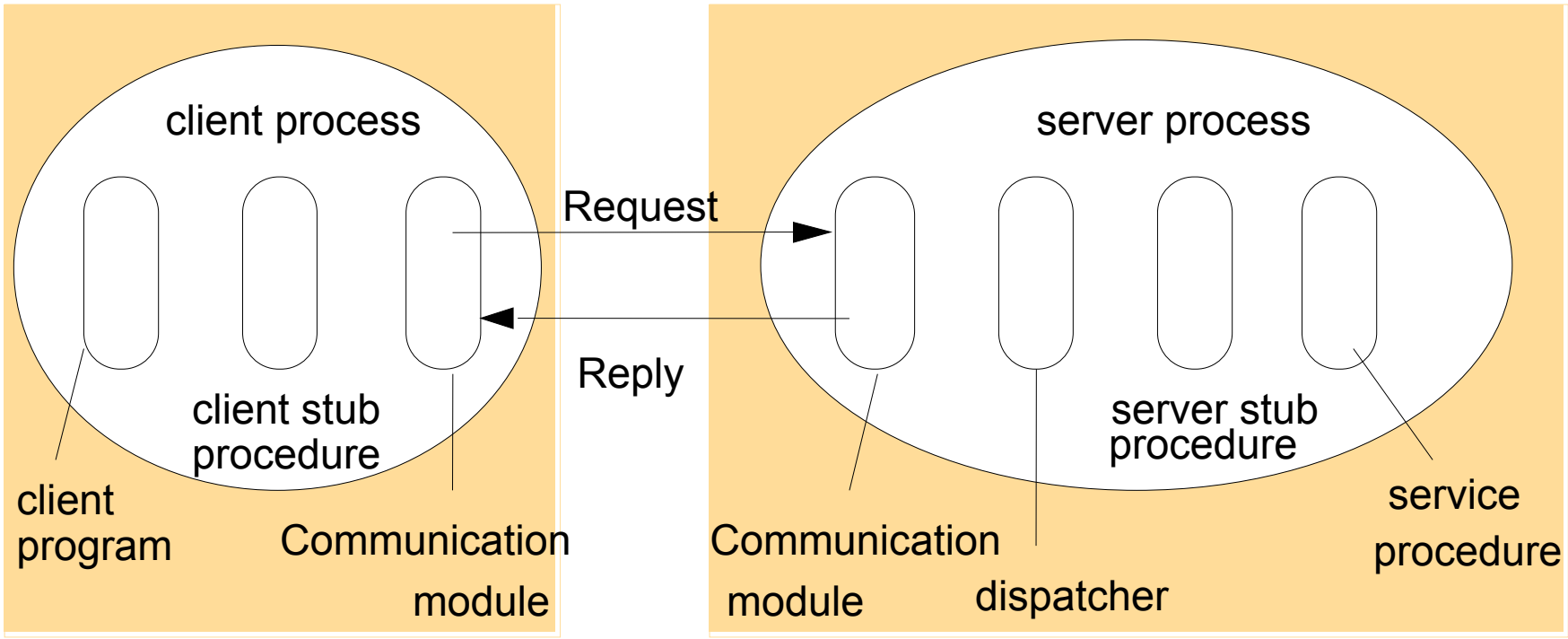

Figure 5.8 Role of client and server stub procedures in RPC in the context of a procedural language

Distributed Event Notification

- Distributed event notification – decouple publishers from subscribers via an event service (manager)
- Architecture: roles of participating objects
  - object of interest (usually changes in states are interesting)
  - event
  - notification
  - subscriber
  - 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

Example: Distributed Event Notification

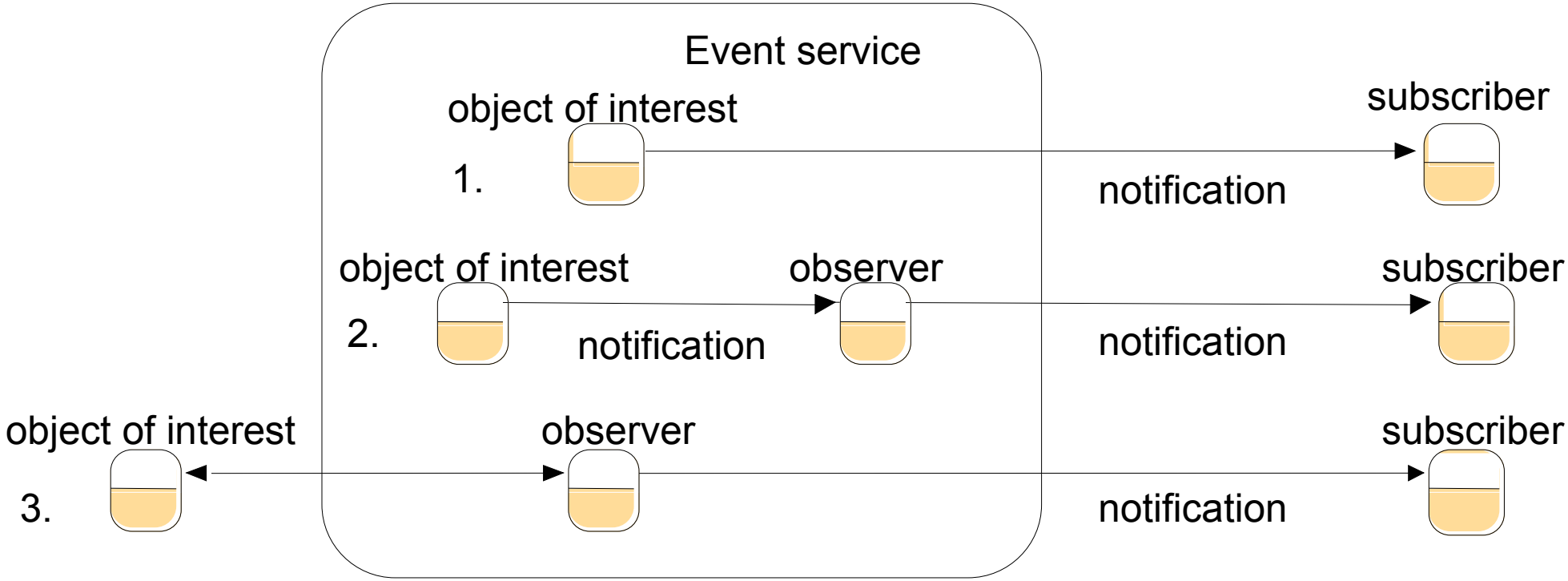

Figure 5.11 Architecture for distributed event notification

- Three cases
  - Inside object without an observer: send notifications directly to the subscribers

– Inside object with an observer: send notification via the observer to the subscribers
  – Outside object (with an observer)

1. an observer queries the object of interest in order to discover when events occur 2.
The observer sends notifications to the subscribers