Lecture

CMPU4021
Distributed Systems

Remote Invocation

Remote Invocation

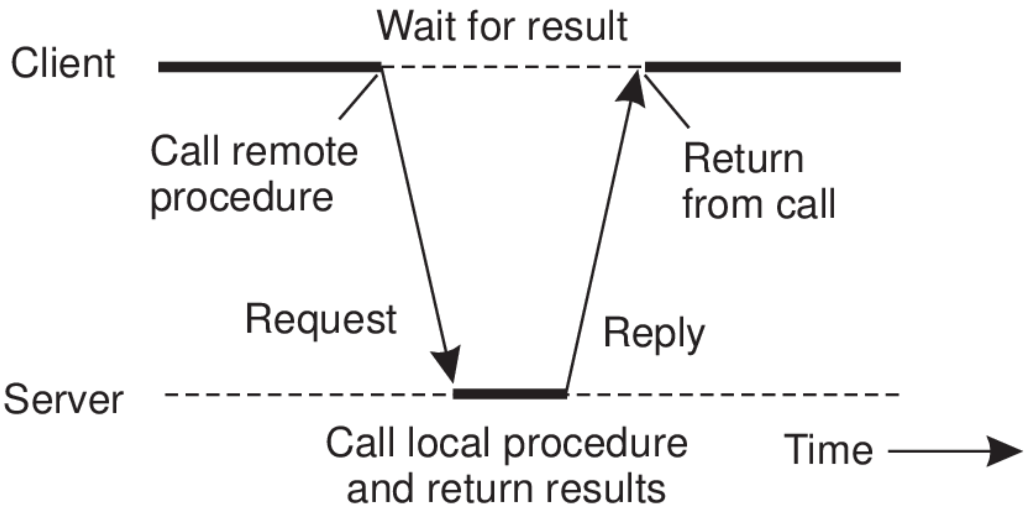
- Covers a range of techniques
- Based on a two-way exchange between communicating entities in a distributed system
  - resulting in the calling of a remote operation, procedure or method.

- Remote Procedure Calls (RPC)

- Remote method invocation (RMI)

Basic RPC operation

- Supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally.
- RPC systems offer (at a minimum)
  - access transparency
    - the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa.
  - location transparency



Remote method invocation (RMI)

- Strongly resembles remote procedure calls but in a world of distributed objects.

- A calling object can invoke a method in a remote object.

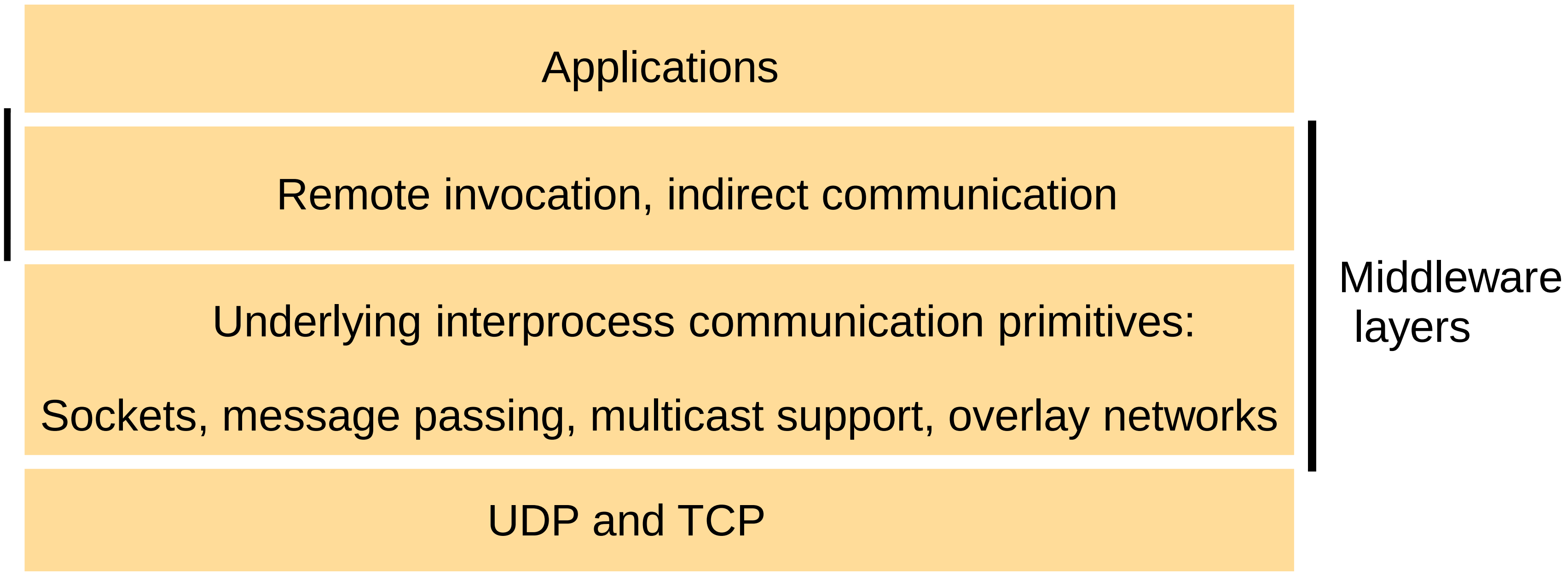
- Communication among distributed objects via RMI
  - Recipients of remote invocations are remote objects, which implement remote interfaces for communication

- Reliability
  - Either one or both the invoker and invoked can fail, and status of communication is supported by the interface. e.g.,
    - notification on failures,
    - reply generation,
    - parameter processing – marshalling/unmarshalling

- Local invocations target local objects, and remote invocations target remote objects

RMI in Middleware Layers

- A suite of API software that uses underlying processes and communication (message passing) protocols to provide its abstract protocol – simple RMI request-reply protocol



Applications

Remote invocation, indirect communication

Underlying interprocess communication primitives:

Sockets, message passing, multicast support, overlay networks

UDP and TCP

Middleware layers

Interfaces

- Interfaces hide the details of modules providing the service; and access to module variables is only indirectly via ‘getter’ and ‘setter’ methods / mechanisms associated with the interfaces
  - e.g., call by value/reference for local calls through pointers vs. input, output, and input paradigms in RMI through message-data and objects

- *Service interfaces*
  - client-server model, specification of the procedures offered by a server
    - defining the types of input and output arguments

- *Remote interfaces*
  - distributed object model, specifies the methods of an object that are available for invocation by objects in other processes
    - defining the types of the input and output arguments of each of them.

Interface definition languages (IDLs)

- IDL is a language that is used to define the interface between a client and server process in a distributed system.
- Each interface definition language also has a set of associated IDL compilers
  - one per supported target language.
- An IDL compiler compiles the interface specifications, listed in an IDL input file, into source code (e.g., C/C++, Java) that implements the low-level communication details required to support the defined interfaces.
- Provides a notation for defining interfaces in which each of the parameters of a method may be described as for *input* or output in addition to having its type specified.

The distributed object model

- RMI
  - invocations between objects in different processes (either on same or different computers) is *remote*. Invocations within the same process are *local*

- Each process contains objects, some of which can receive remote invocations, other only local invocations

- Those that can receive remote invocations are called remote objects

- Objects need to know the remote object reference of an object in another process in order to invoke its methods. How do they get it?
  - The remote interface specifies which methods can be invoked remotely

Distributed Garbage Collection

- Where any process includes remote objects, then it is equipped wlth both
  - Local garbage collector
  - Distributed garbage collector
- For any remote object O
  - `O.holders` is a list of all the processes that have a remote reference to that object i.e. got a stub for it
- When a client C receives a remote reference for O it makes an `addRef` call to O's garbage collector
  - resulting in its being added to `O.holders`
- When C's local garbage collector attempts to delete the stub object for O, it calls `removeRe`f on O's garbage collector, resulting
  - it its being removed from `O.holders`
- When `O.holders` is empty, O can be deleted

Distributed Garbage Collection

- Possible difficulties
  - `removeRef` and `addRef` sent at same time from different processes

- Possibility that O would be deleted, even though a client *thinks* it has a reference

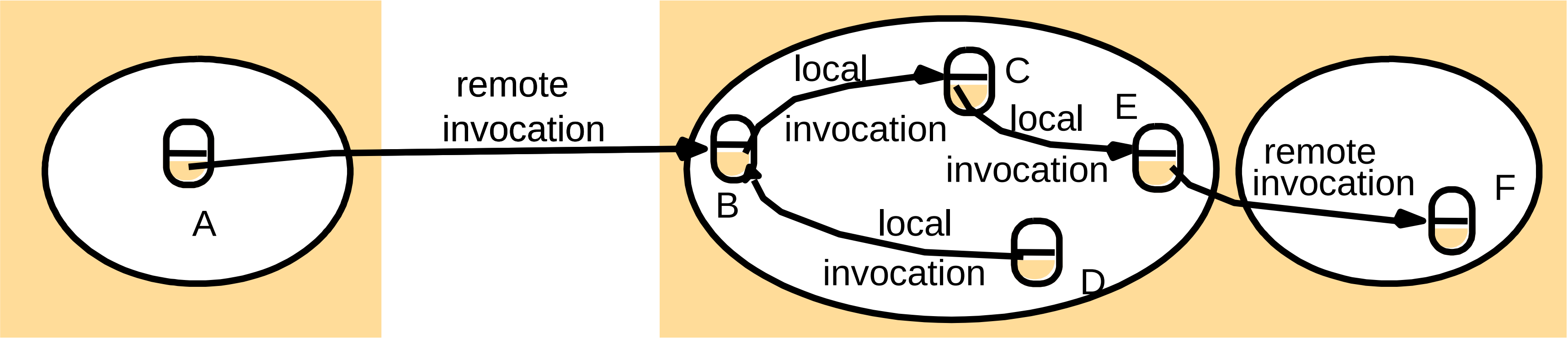
- Incorporate a delay / temporary reference to solve
  - `addRef` goes missing

- Client must detect, exception returned
  - `removeRef` message goes missing / not sent

- Time based leases are allocated for objects

The distributed object model: Remote and local method invocations

- Objects receiving remote invocations (service objects) are remote objects, e.g., B and F
- Object references are required for invocation, e.g., C must have E’s reference or B must have A’s reference
- B and F must have remote interfaces (of their accessible methods)

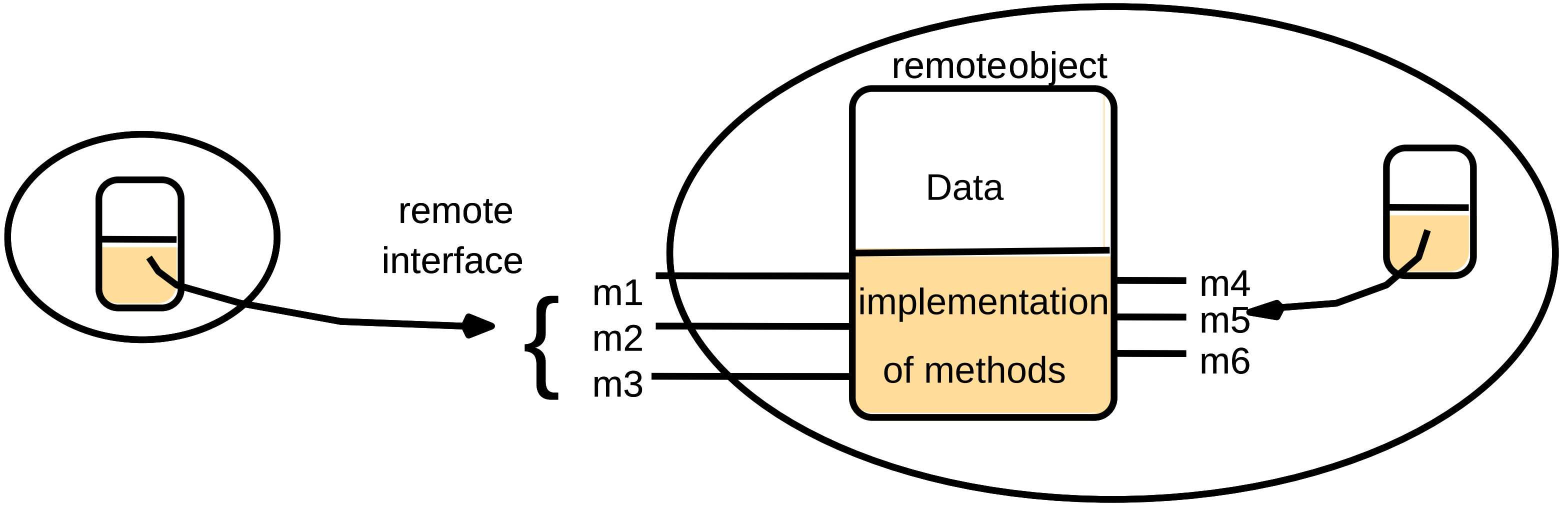


The distributed object model

- Remote object references
  - An unique identifier of a remote object, used throughout a distributed system
  - The remote object reference (including the ‘interface’ list of methods) can be passed as arguments or results in rmi.

The distributed object model:Remote interfaces

- Remote objects have a class that implement remote methods (as public).
- Local objects can access methods in an interface plus methods implemented by remote objects
  - Remote interfaces can't be constructed – no constructors



Invocation Semantics

- Unreliable network
  - – For all request –reply protocols, messages may get lost

- Solutions for lost / retransmitted messages
  - – Retry request
  - – Filter Duplicates
  - – Retransmit results

Invocation Semantics

- In local OO system
  - all methods are invoked exactly once per request –guaranteed – unless whole process fails
- In distributed object system, we need to know what has happened if we do not hear result from remote object i.e. did the request go missing, did the response go missing
- 3 different types of guarantee (invocation semantics) may be provided
  - could be implemented in a middleware platform intended to support remote method invocations:
    - *Maybe*
    - *At-Least-Once*
    - *At-Most-Once*

Maybe Invocation Semantics

- If the invoker cannot tell whether a remote method has been invoked or not

- Very inexpensive, but only useful if the system can tolerate occasional failed invocations

At-Least-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked at least once

- Achieved by resending requests to mask omission failure

- Only useful if the operations are idempotent (x = 10, rather than x = x + 10)

- Inexpensive on server

At-Most-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked only once
- If no result is received, then the method was executed either never or once
- Achieved by resending requests, and storing and resending responses
- More expensive on a server / remote object, which must maintain results and recognise duplicate messages

Invocation semantics: failure model

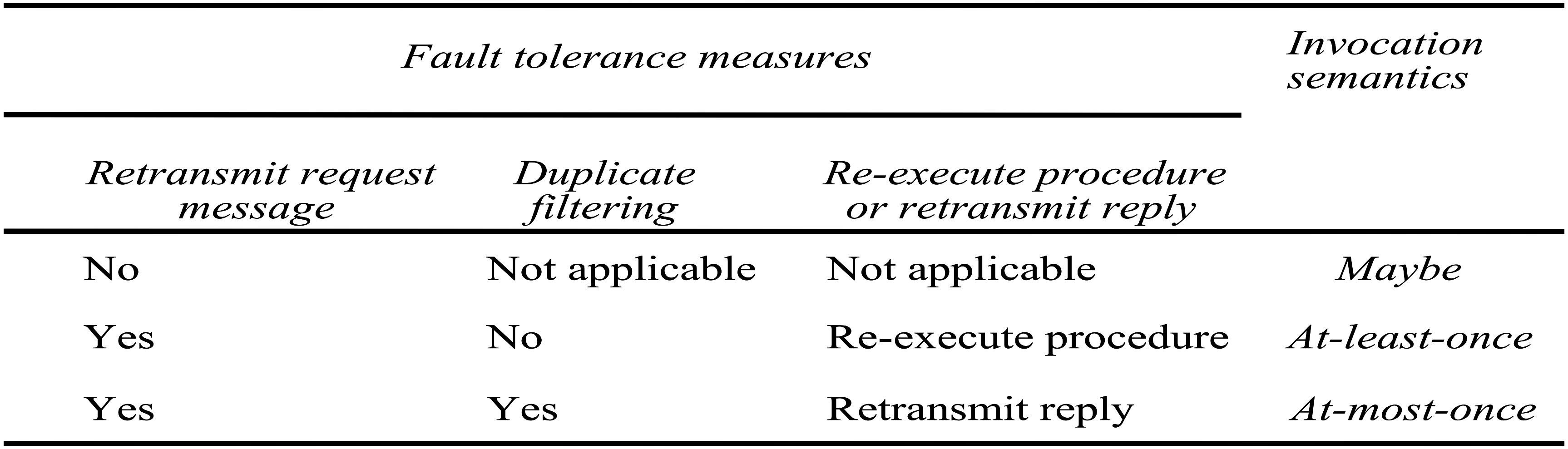
- Maybe, At-least-once and At-most-once
  - can suffer from crash failures when the server containing the remote object fails.

- *Maybe*
  - if no reply, the client does not know if method was executed or not
    - omission failures - if the invocation or result message is lost

- *At-least-once*
  - the client gets a result (and the method was executed at least once) or an exception (no result)
    - arbitrary failures. If the invocation message is retransmitted, the remote object may execute the method more than once, possibly causing wrong values to be stored or returned.
    - if *idempotent* operations are used, arbitrary failures will not occur

- At-most-once
  - the client gets a result (and the method was executed exactly once) or an exception (instead of a result, in which case, the method was executed once or not at all)

Invocation semantics: failure model



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*

Design Issues of RMI

- Local invocations have
  - at-most-once, or
  - exactly-once semantics

- Distributed RMI, the alternative invocation semantics are:
  - *Retry request message* – retransmit until reply is received or on server failure – at-least-once semantics;

  - *Duplicate message filtering* – discard duplicates at server (using seq #s or ReqID);

  - Buffer result messages at server for *retransmission* – avoids redo of requests (even for idempotent ops) – at-most-once semantics.
    - Idempotent operation – the one which can be performed repeatedly with the same effect as if it had been performed exactly once.

Transparency

- Remote invocations should be made transparent
  - the syntax of a remote invocation is the same as that of a local invocation, but
  - the difference between local and remote objects should be expressed in their interfaces.

- E.g. Java RMI:
  - Remote objects implement `Remote` interface and throw *RemoteExceptions*

- Remote object should be able to keep its state consistent in the presence of concurrent accesses from multiple clients.

Representation of a remote object reference

- a remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted.

Why not?

- the first two fields locate the object unless migration or re-activation in a new process can happen
- the fourth field identifies the object within the process
- its interface tells the receiver what methods it has (e.g. class *Method*)
- a remote object reference is created by a remote reference module when a reference is passed as argument or result to another process
  - it will be stored in the corresponding proxy
  - it will be passed in request messages to identify the remote object whose method is to be invoked

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Distributed garbage collection

Ensures

- if a local or remote reference to an object is still held anywhere in a set of distributed objects
  - then the object itself will continue to exist,
- As soon as no object holds a reference to it
  - the object is collected and the memory it uses recovered.

Summary

RMI

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.

- Local method invocations provide exactly-once semantics
  - the best RMI can guarantee is at-most-once

- Middleware components proxies, skeletons and dispatchers hide details of marshalling, message passing and object location from programmers.

References

- Chapter 5: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5/E

- Chapter 4: Maarten van Steen, Andrew S. TanenbaumDistributed Systems, 3rd edition (2017)