04: Object-based Architectures

Distributed Software Architectures

Stefan Huber <shuber.lba@fh-salzburg.ac.at>

April 26, 2019

Section 1

Object-based Architectures

Middleware and distribution

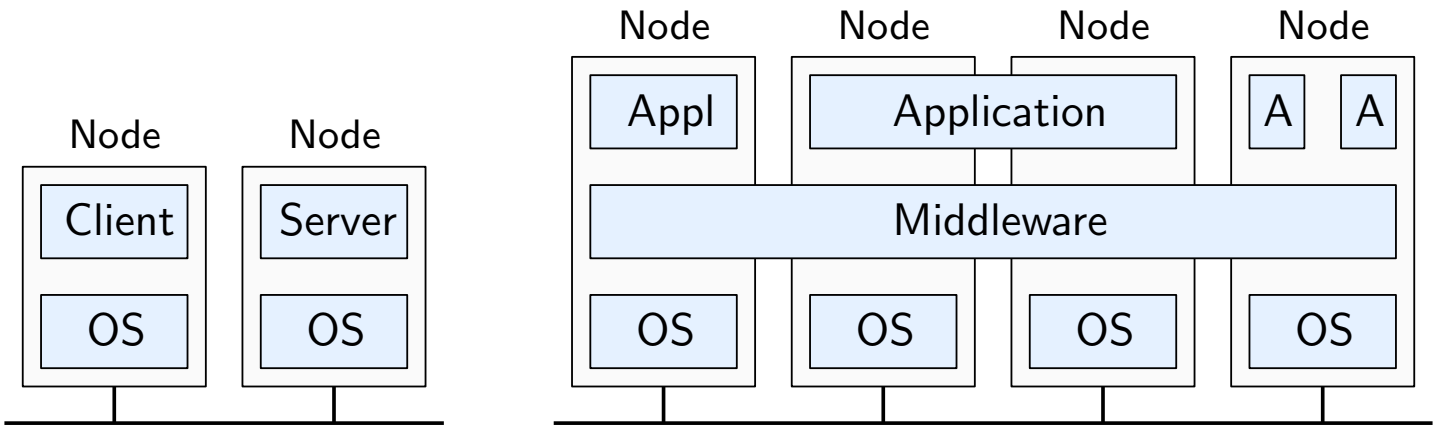
Distribution transparency

Distribution transparency allows for ignorance of the location of data or services.

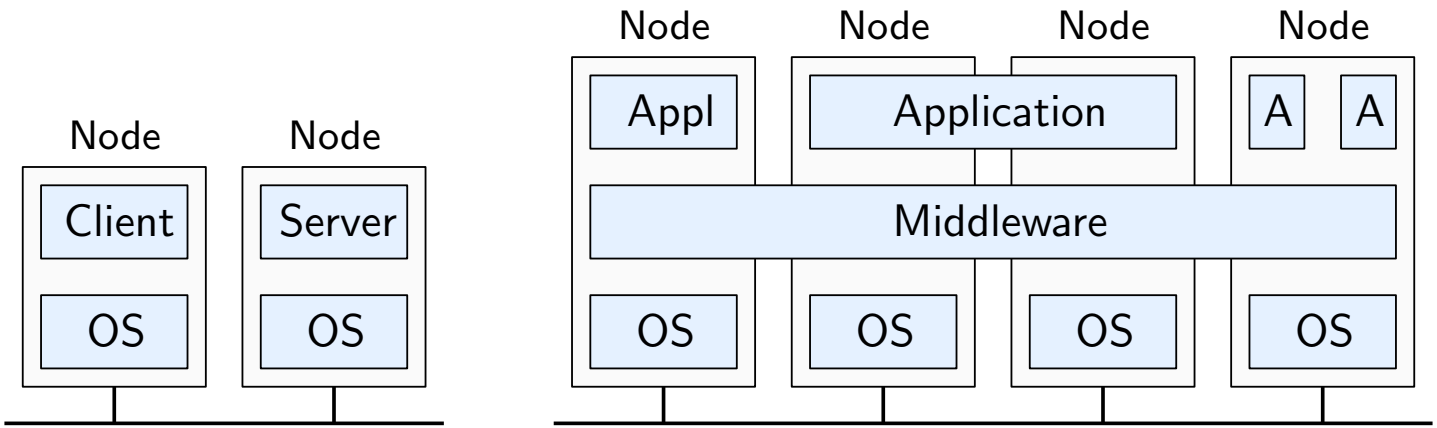
Middleware

Middleware is a cross-node layer on top of each OS to provide distribution transparency to distributed applications.

- It makes a distributed system appear as a single computer.
- Provides abstractions away from the details of communication (on a data level).



Middleware services



Middleware : distributed system $\widehat{=}$ operating system : computer.

- Manages resources
- Provides services
  Naming, inter-application communication, failure tolerance, security ...

Types of transparencies

Distribution transparency may refer to different transparency types:[1]

Transparency	**Description**
Access	Hide differences in data representation and method of access
Location	Hide where object is located
Relocation	Hide that an object may be moved while used
Migration	Hide that an object is moving/mobile object
Replication	Hide that an object is replicated
Concurrency	Hide that an object is shared by independent users
Failure	Hide failure and recovery of an object

[1] An object here may mean process or resource.

Remote Procedure Call

Within an application, how do components communicate?

- Procedure calls.[2]

Within a distributed system, how do components communicate (so far)?

- Message exchange.
- Even for IPC on the same host we have a message-based communication.

[2] Assuming an imperative programming paradigm.

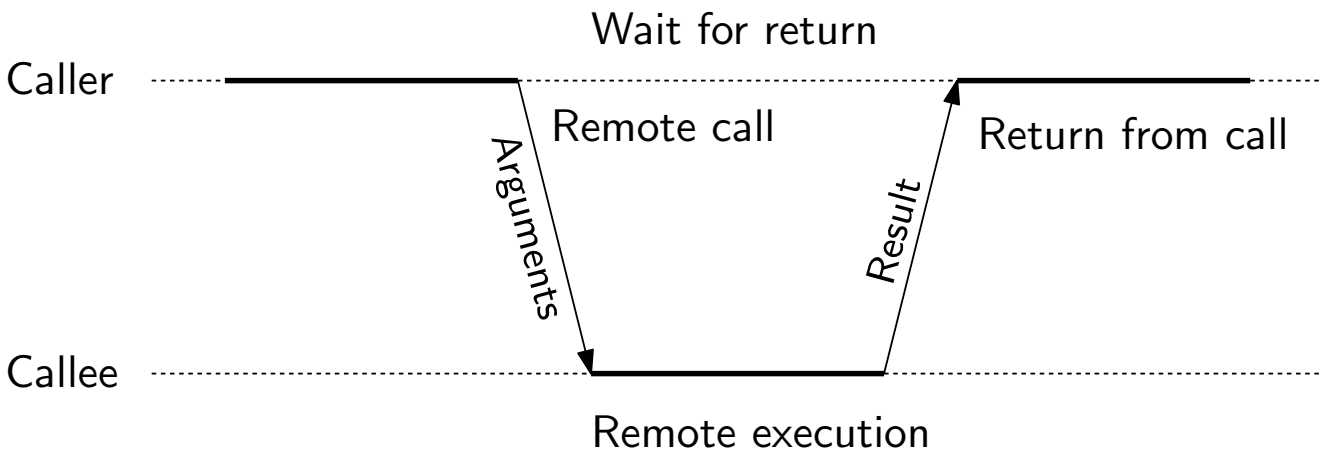
Remote Procedure Call

Within an application, how do components communicate?

- Procedure calls.[2]

Within a distributed system, how do components communicate (so far)?

- Message exchange.
- Even for IPC on the same host we have a message-based communication.



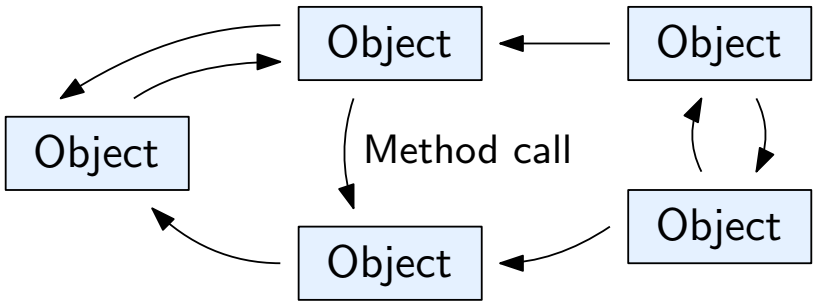
Simple yet powerful idea: Procedure calls within distributed applications.

- We can call a remote procedure as if it is local. No change of paradigm.
- Increased access transparency.

[2] Assuming an imperative programming paradigm.

Object-based architecture

An object-based architecture consists of a distributed set of objects that interact via method calls.



Encapsulation like in OOP:

- Objects encapsulate data, the object's state.
- Objects provide methods that operate on the state.

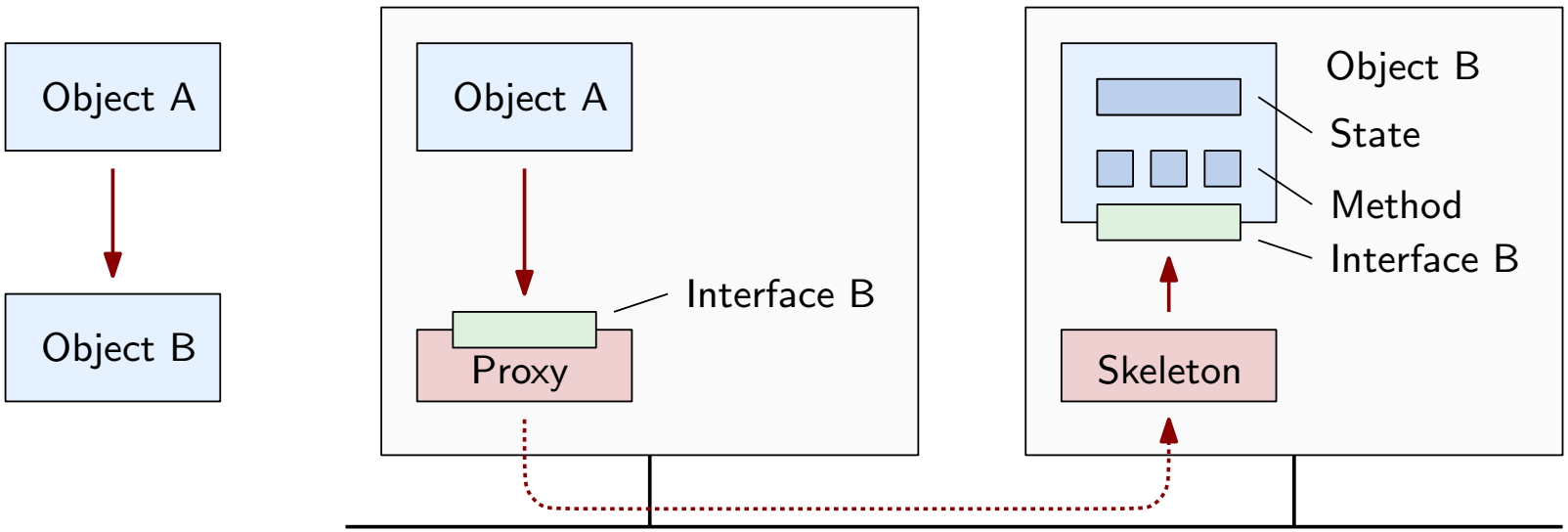
Method calls can take place over the network.

- But thanks to distribution transparency provided by a middleware, we (mostly) do not care.
- We speak of distributed objects.

Distributed objects

An interface defines the set of methods to access an object.

- The interface hides the actual implementation.

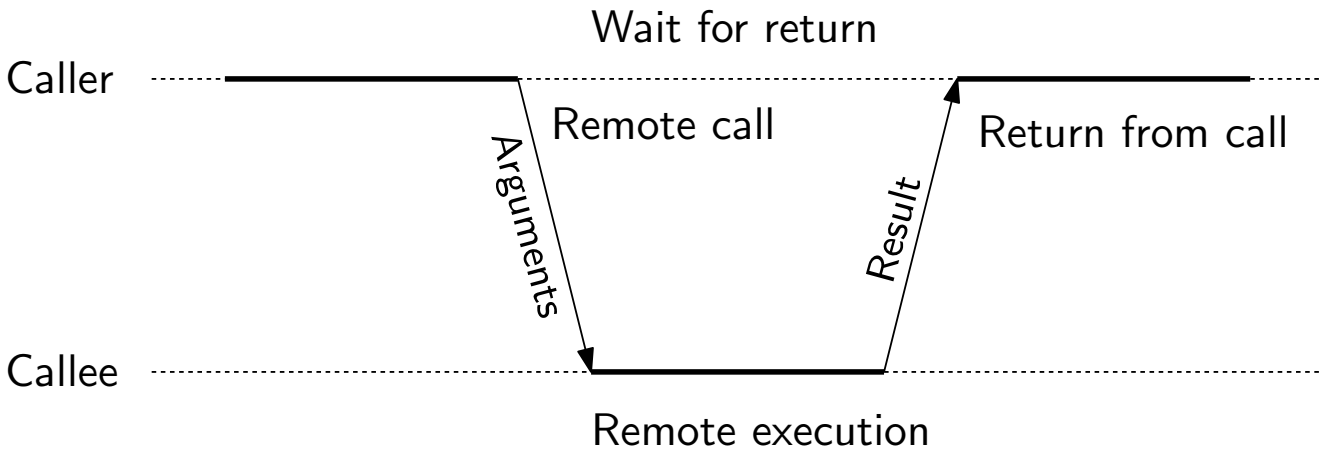


Assume Object A calls a method of an Object B on a different node.

- For Object A all that matters is the interface of B.
- A proxy of Object B, which implements the same interface, is loaded at the location of Object A. The proxy then performs a remote procedure call.
- Proxy and skeleton provide the illusion of co-located objects.

Basic operation of RPC

Main goal: Remote procedure calls shall look like local procedure calls.



Local (remote) procedure call:

1. Parameters are passed by stack or registers (via network).
2. Procedures code is executed (by remote machine).
3. Result is passed back by register (via network).
4. Execution resumes for caller.

Key element: Parameter (and result) passing.

Barefooted Python RPC

We would like to call `is_prime()` and `add()` like local procedures.

```
1     for i in range(20):
2         print("is_prime({}): {}".format(i, is_prime(i)))
3
4     a, b = 10, 23
5     print("add({}, {}) = {}".format(a, b, add(a, b)))
```

Steps:

1. Marshalling arguments
2. Sending over to remote side
3. Remote execution to produce result
4. Receive result
5. Unmarshalling return value

Barefooted Python RPC

The client stubs (proxy) implementations:

```
1  def remote_procedure_call(endpoint, fun, *args):
2      # Marshalling arguments and function name
3      fun_args = pickle.dumps([fun, args])
4
5      # Pass function call to remote side and receive result
6      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7      sock.connect(endpoint)
8      sock.send(fun_args)
9      res = sock.recv(4096)
10
11     # Unmarshalling return value
12     return pickle.loads(res)
13
14
15 def is_prime(num):
16     return remote_procedure_call(("localhost", 1700), "is_prime", num)
17
18
19 def add(a, b):
20     return remote_procedure_call(("localhost", 1700), "add", a, b)
```

Barefooted Python RPC

The server stubs and remote procedure implementations:

```
class RPCHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # Unmarshalling arguments and function name
        data = self.request.recv(4096)
        fun, args = pickle.loads(data)

        proc = getattr(self, fun, None)
        res = None
        if proc is not None:
            print("Call {}{}".format(fun, args))
            res = proc(*args)

        # Marshalling return value
        self.request.sendall(pickle.dumps(res))

    def is_prime(self, num):
        num = abs(num)
        if num <= 1:
            return False

        return all([num % i != 0 for i in range(2, int(sqrt(num)+1))])

    def add(self, a, b):
        return a + b
```

Marshalling

Openess

A distributed system shall be open; it shall be easy to use and integrate its components in other systems.

An open RPC mechanism has to deal with heterogeneous systems:

- Little endian versus big endian
- Programming languages
- Operating system
- Processor instruction sets (x86, amd64, arm, powerpc, alpha, . . .)

Marshalling

Marshalling and unmarshalling is the transformation of parameters into a neutral data format forth and back.

The goal of marshalling is openess and access transparency.

XML-RPC

XML-RPC:

- An RPC protocol based on XML over HTTP.
- Developed by UserLand Software and Microsoft (1998).
- Later evolved to the SOAP protocol.

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>strlen</methodName>
  <params>
    <param> <value><string>hello</string></value> </param>
  </params>
</methodCall>
```

```xml
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><int>5</int></value>
    </param>
  </params>
</methodResponse>
```

XML-RPC in Python

```python
from xmlrpc.server import SimpleXMLRPCServer

class MyFuncs:
    def strlen(self, s):
        return len(s)

def adder_function(x, y):
    """A function that returns the sum of the two arguments."""
    return x + y

if __name__ == '__main__':
    server = SimpleXMLRPCServer(("localhost", 8000))

    # Provide system.{listMethods,methodHelp,methodSignature}
    server.register_introspection_functions()
    # Register adder_function under the name 'add'
    server.register_function(adder_function, 'add')
    # Use pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)
    # Register all the methods of the instance.
    server.register_instance(MyFuncs())

    try:
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()
```

XML-RPC in Python

```python
import xmlrpc.client

if __name__ == '__main__':
    with xmlrpc.client.ServerProxy('http://localhost:8000') as proxy:
        print(proxy.pow(2, 3))          # Returns 2**3 = 8
        print(proxy.add(2, 3))          # Returns 5
        print(proxy.strlen("hi"))       # Returns 2
        print(proxy.system.listMethods())
```

Or on the console using curl:

```
curl -X POST http://localhost:8000/RPC2 -d '<?xml version="1.0"?>
<methodCall>
  <methodName>strlen</methodName>
  <params>
    <param> <value><string>hello</string></value> </param>
  </params>
</methodCall>'
```

References and pointers

Particular difficulty: How to handle references (pointers)?

- An address is only meaningful within the address space of a process.
- Especially relevant in object-based architectures.

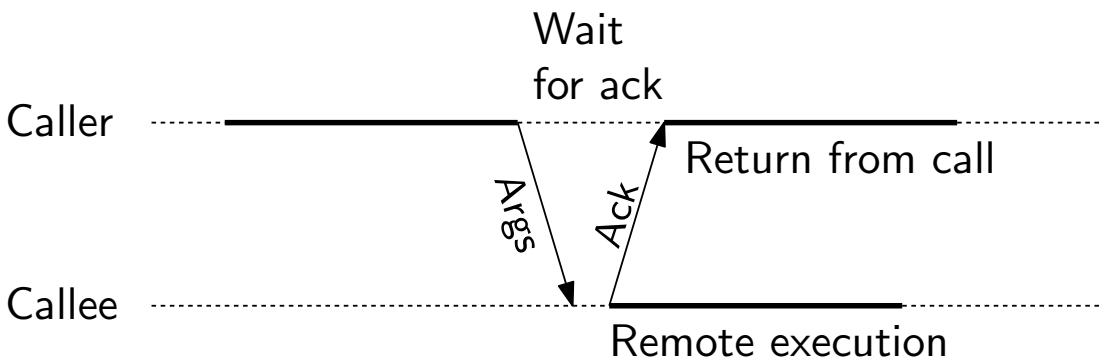
Solutions:

- Forbidding it.
- Passing a deep copy, i.e., turning a call-by-reference into a call-by-value-and-restore. For read-only procedures a call-by-value suffices.
- Using global references or "pseudo-pointers" (e.g., handles or IDs).

Variations of RPC

Asynchronous call:

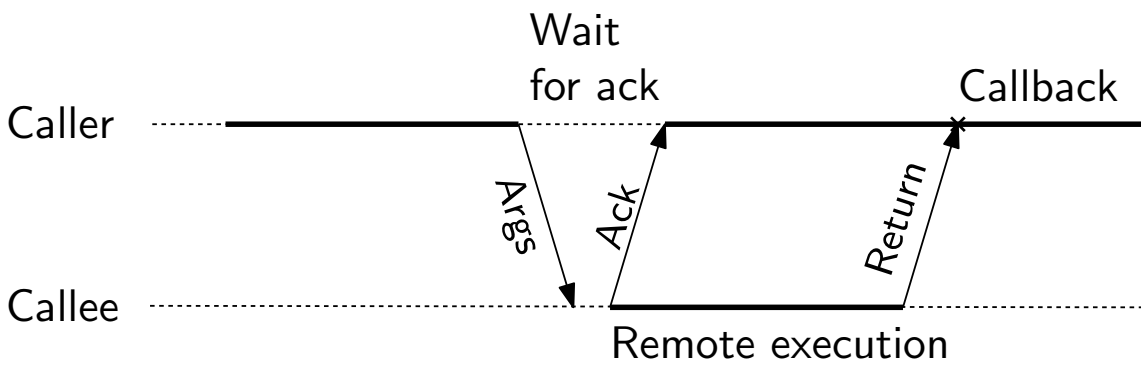
- If the remote procedure has no result or we do not care.
- Server immediately sends acknowledgment rather than after execution.
- **One-way RPC**: Do not even wait for ack at expense of reliability.



Variations of RPC

Deferred synchronous RPC:

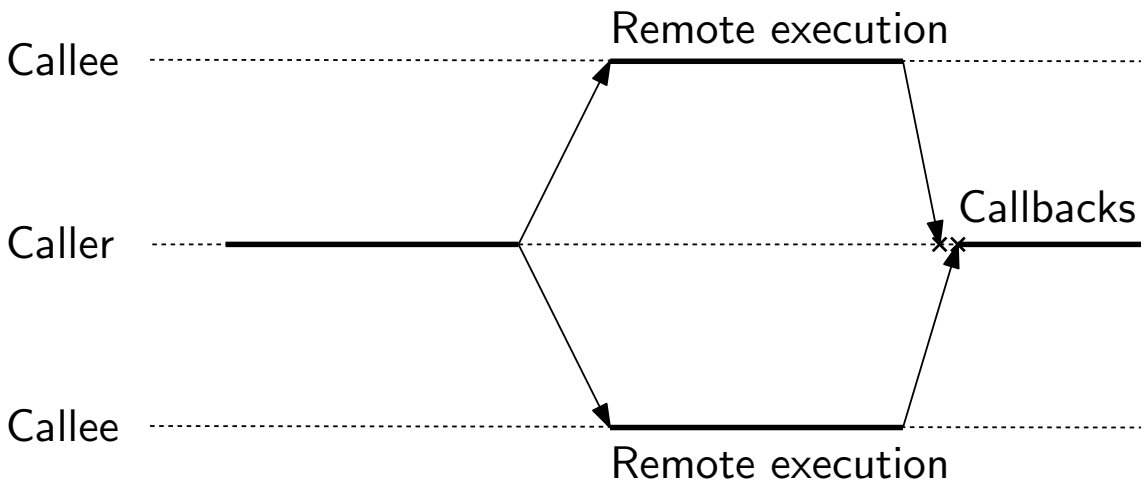
- Immediate ack, but returned values are sent later.
- Often returned values are signaled via a callback function. Alternative: Polling.
- For instance for parallel communication with server(s).



Variations of RPC

Multicast RPC:

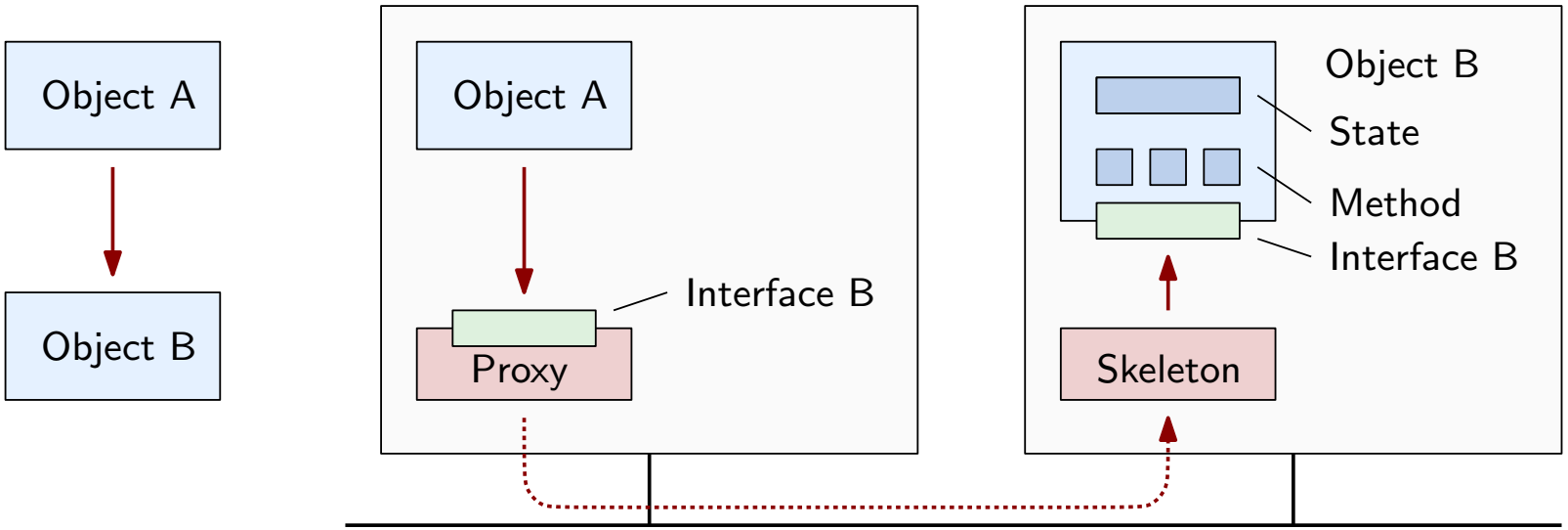
- Sending RPC request to a group of servers.
- Return is signaled by callback by each server.
- Waiting for first callback versus waiting for all.
  - Multicast for fault tolerance: Wait for first callback.
  - Multicast for parallelization: Wait for all (and merge results?).



Interface definition language

Being open towards various programming languages requires an IDL:

- An interface is defined using the IDL.
- From the interface definition file code is generated for the target language, e.g., Python.
- The generated code basically is the glue between the actual client code and the server implementation.



Ice

Internet Communications Engine:

- Full-fledged, open-source distributed computing framework.
- Developed by ZeroC, influenced by CORBA.
- Provides an object server, but also mechanisms for object registration and discovery, load balancing, or failover handling.
- Supports C++, Objective C, C#, Java, JS, MATLAB, PHP, Python, Ruby

Documentation (2954 pages) available at
`https://download.zeroc.com/ice/3.7/Ice-3.7.1.pdf`.

Installing Ice for a local user:

```
pip3 install --user zeroc-ice
```

Steps to write an Ice application:

1. Write a Slice definition (IDL) and compile it.
2. Write a server and compile it.
3. Write a client and compile it.

Ice – Demo (Step 1)

Example for a Slice file:

```
// Printer.ice
module Demo
{
    interface Printer
    {
        void printString(string s);
    }
}
```

Step 1 is to compile the Slice file:

```
slice2py Pinter.ice
```

The results are:

- A file `Printer_ice.py` with interface definitions and proxy code.
- A subdirectory `Demo` which forms the Python module `Demo`.

Ice – Demo (Step 2)

Step 2 is the implementation of the server:

- Implement the interface `Demo.Printer`. By convention we call it `PrinterI`.
- The main code does the following:
  1. Initialize the Ice runtime system.
  2. Create an object adapter named SimplePrinterAdapter and make it listen on TCP (default protocol) port 10000.
  3. Register and activate an instance of PrinterI under the name SimplePrinter.

```
1  import sys, Ice, Demo
2
3  class PrinterI(Demo.Printer):
4      def printString(self, s, current=None):
5          print("printString():", s)
6
7  if __name__ == '__main__':
8      with Ice.initialize(sys.argv) as communicator:
9          adapter = communicator.createObjectAdapterWithEndpoints(
10                 "SimplePrinterAdapter", "default -p 10000")
11         obj = PrinterI()
12         adapter.add(obj, communicator.stringToIdentity("SimplePrinter"))
13         adapter.activate()
14         communicator.waitForShutdown()
```

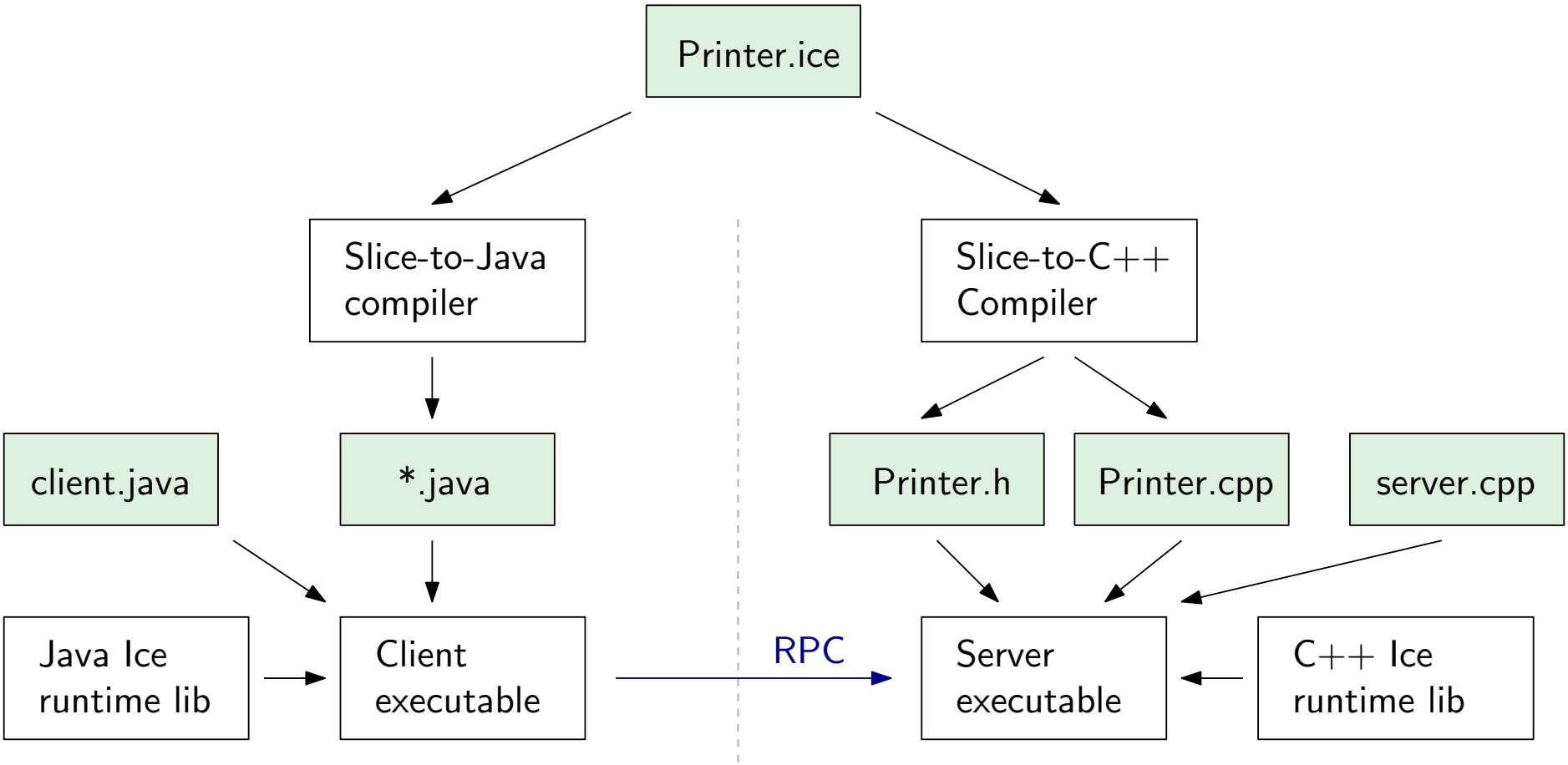
Ice – Demo (Step 3)

Step 3 is the implementation of a client:

1. Initialize the Ice runtime system.
2. Create a proxy of the SimplePrinter object provider by the adapter on TCP port 10000. The proxy object is of a generic type `Ice.ObjectPrx`.
3. Ask server whether the proxy object returned implements the `Demo.Printer` interface, and if yes, cast to `Demo.PrinterPrx`.
4. Invoke the remote method `printString`.

```
1 import sys, Ice, Demo
2
3 if __name__ == '__main__':
4     with Ice.initialize(sys.argv) as communicator:
5         base = communicator.stringToProxy("SimplePrinter:default -p 10000")
6         printer = Demo.PrinterPrx.checkedCast(base)
7         if not printer:
8             raise RuntimeError("Invalid proxy")
9         printer.printString("Hello World!")
```

Interface definition language (revisited)



Python addendum: MRO

Method resolution order:

- Says how the inheritance graph is traversed to find a member. It is a linearization of the inheritance graph.
  - Also concerns constructor invocation of base classes.

- In Python 3 the MRO is determined by the *C3 linearization*[3].[4]
  - For single inheritance, it is simply the consecutive list of base classes.
  - For an arbitrary inheritance graph, however, it is quite subtle.
  - C3 is monotone:
    If B precedes C in the MRO of A then it does so for every sub-class of A.[5]
- Some practice:
  - `A.__mro__` tells the MRO for class `A`.
  - `super(X, self).f()` calls the method `f()` of the class after `X` in the MRO.
  - `super().f()` calls `f()` of the class after the current one in the MRO.

[3] `https://www.python.org/download/releases/2.3/mro/`

[4] Prior Python 2.3 it used to be depth-first left-right traversal. Also, from Python 2.2 there are new-style classes inheriting from the class `object`.

[5] Actually, there are hierarchies that do not admit a monotone hierarchy. Then an exception is raised.

Upcoming test

- Test on May, 9.
- 45 minutes.
- Focus on lecture slides, including the lecture today.