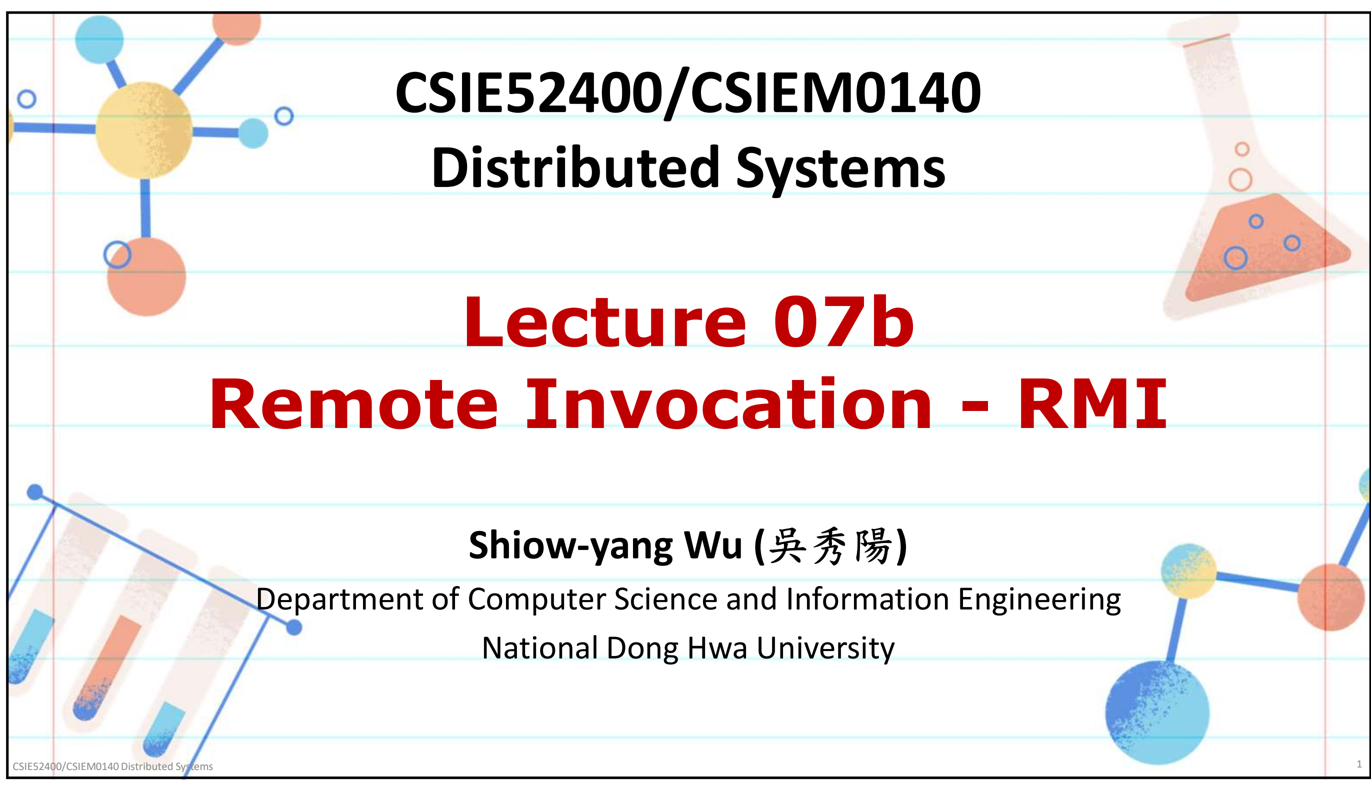
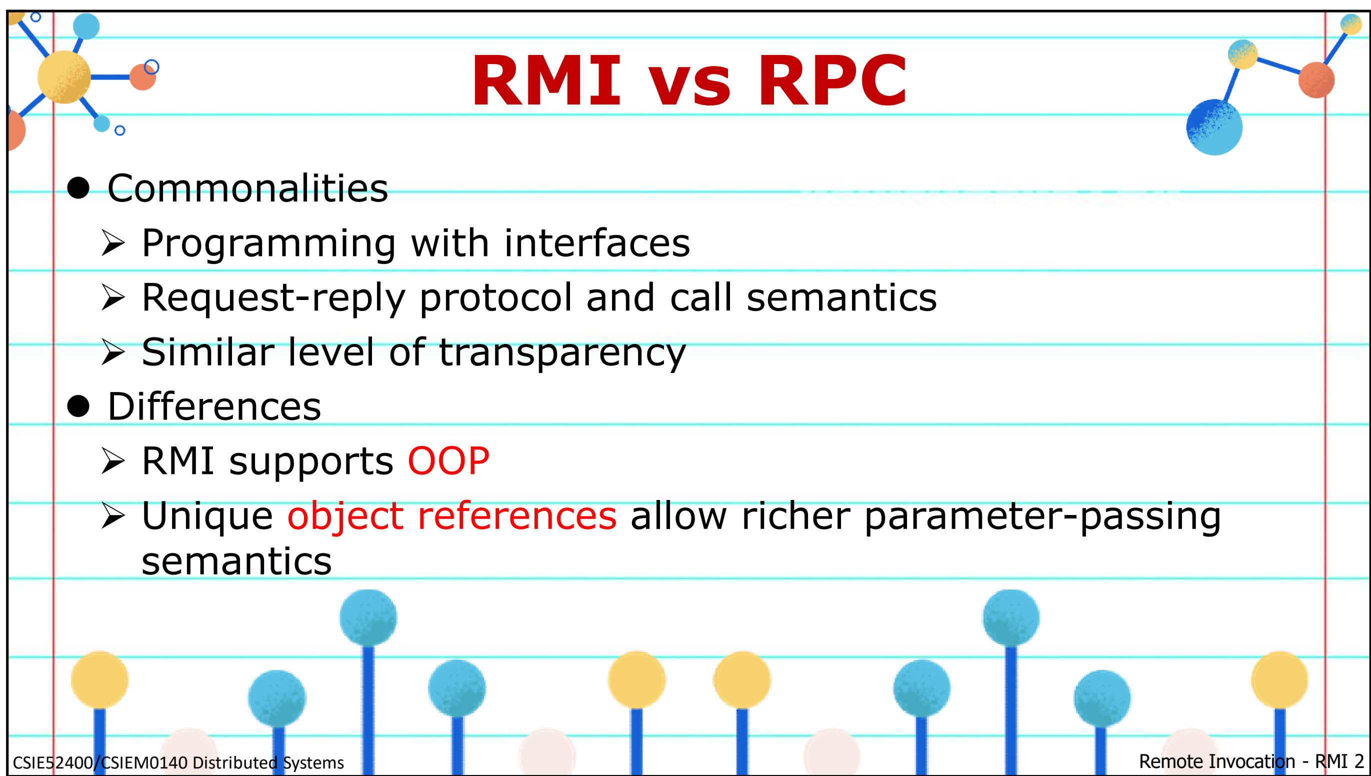
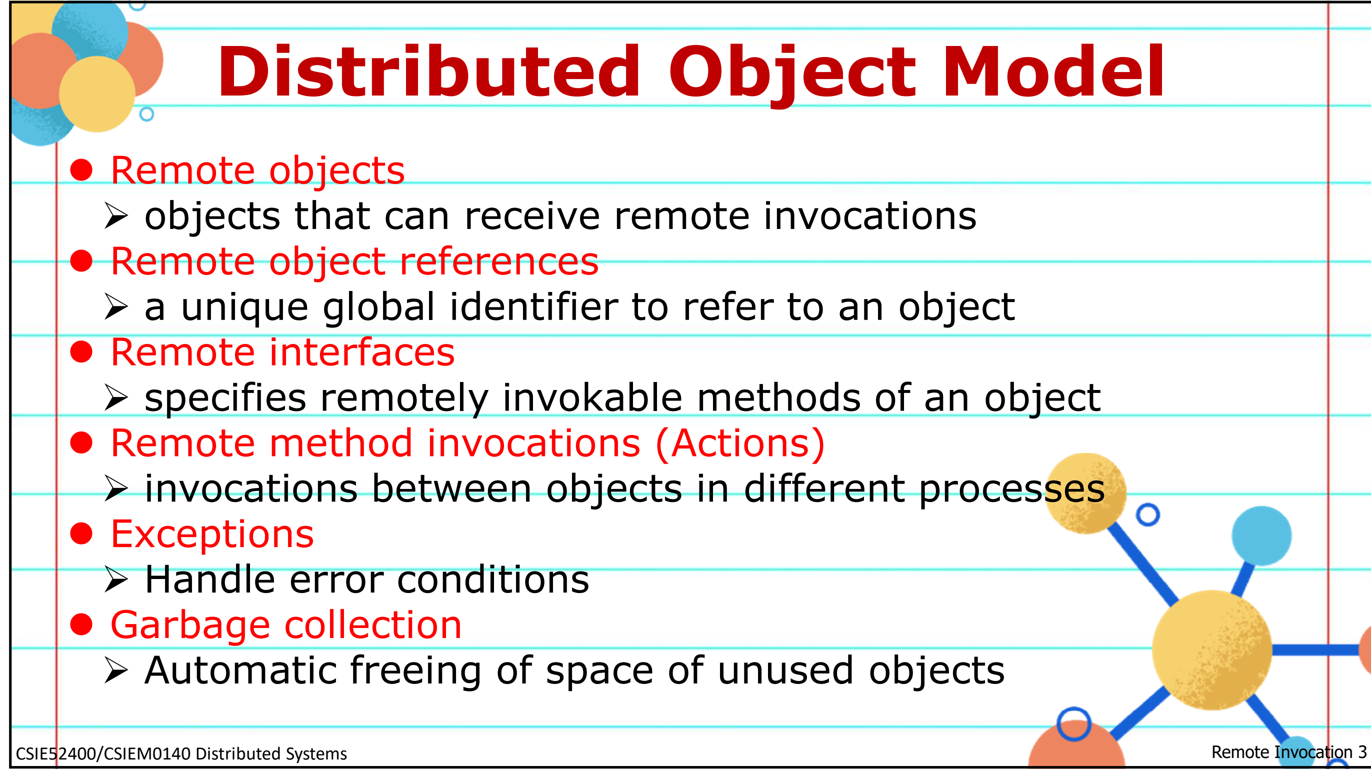
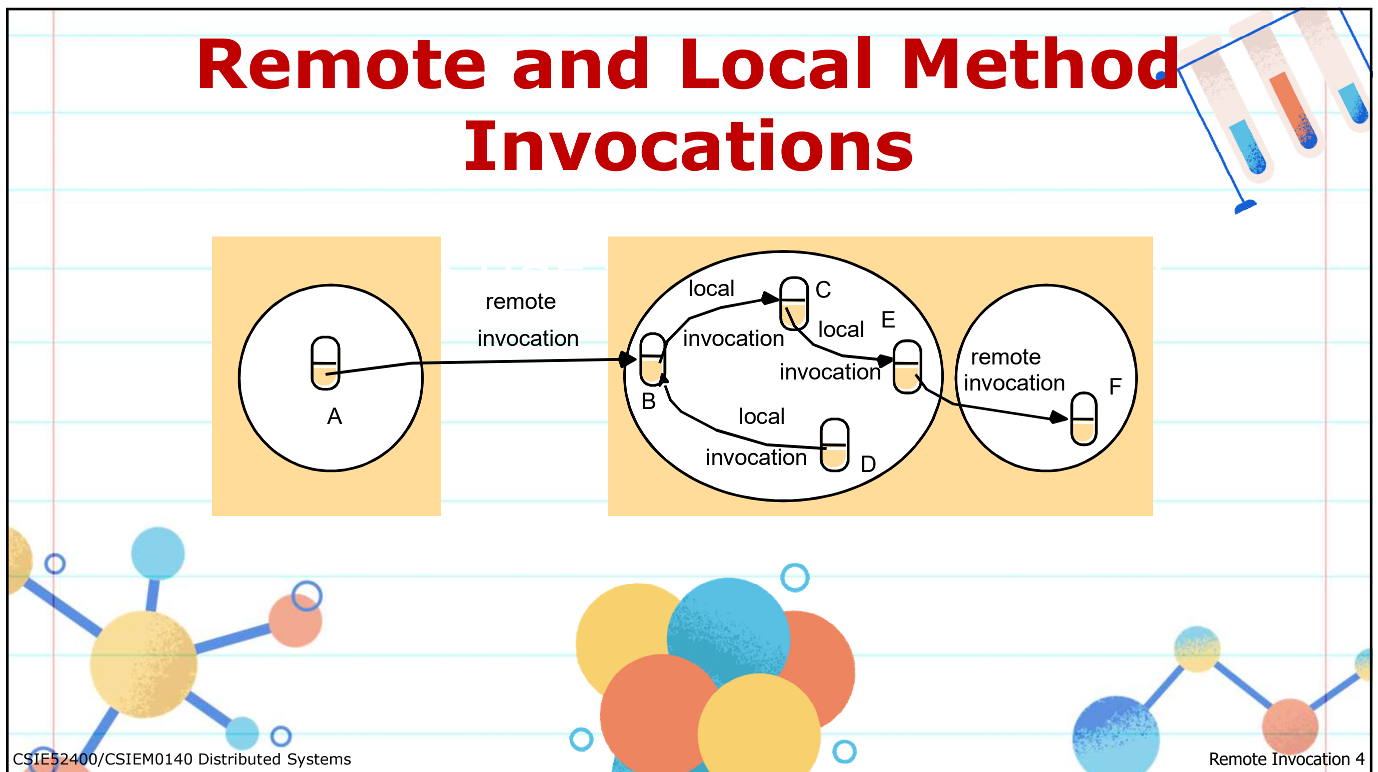
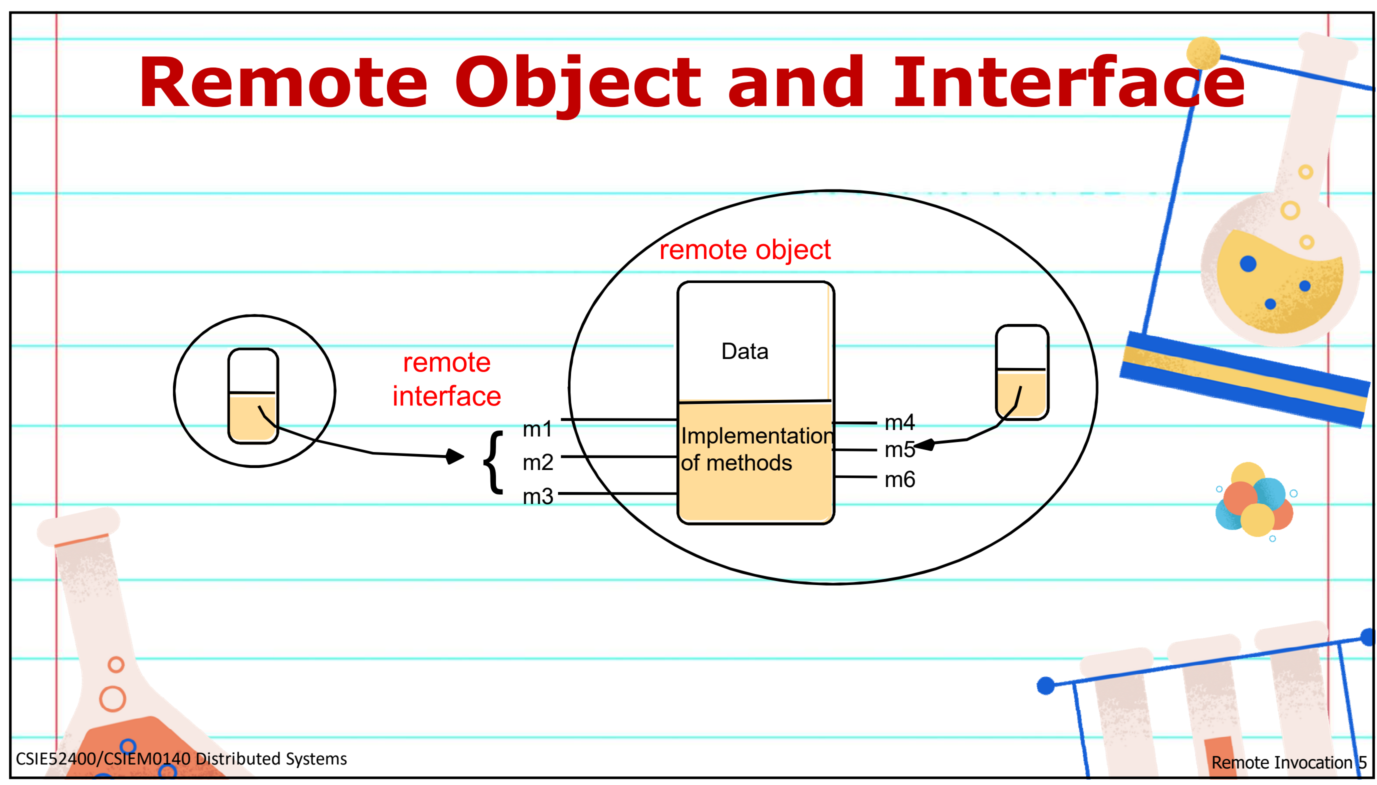
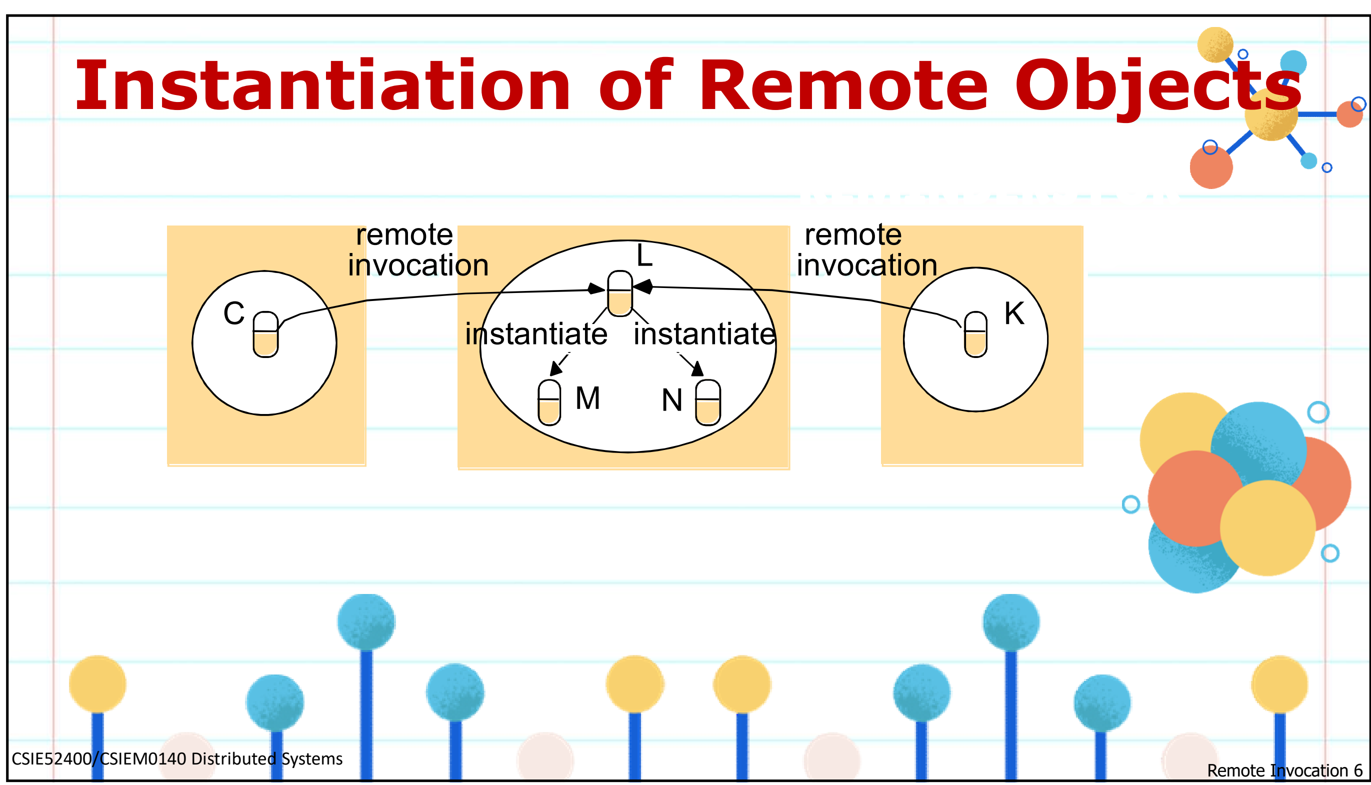

CSIE52400/CSIEM0140
Distributed Systems
Lecture 07b
Remote Invocation - RMI
Shiow-yang Wu (吳秀陽)
Department of Computer Science and Information Engineering
National Dong Hwa University
CSIE52400/CSIEM0140 Distributed Systems
1

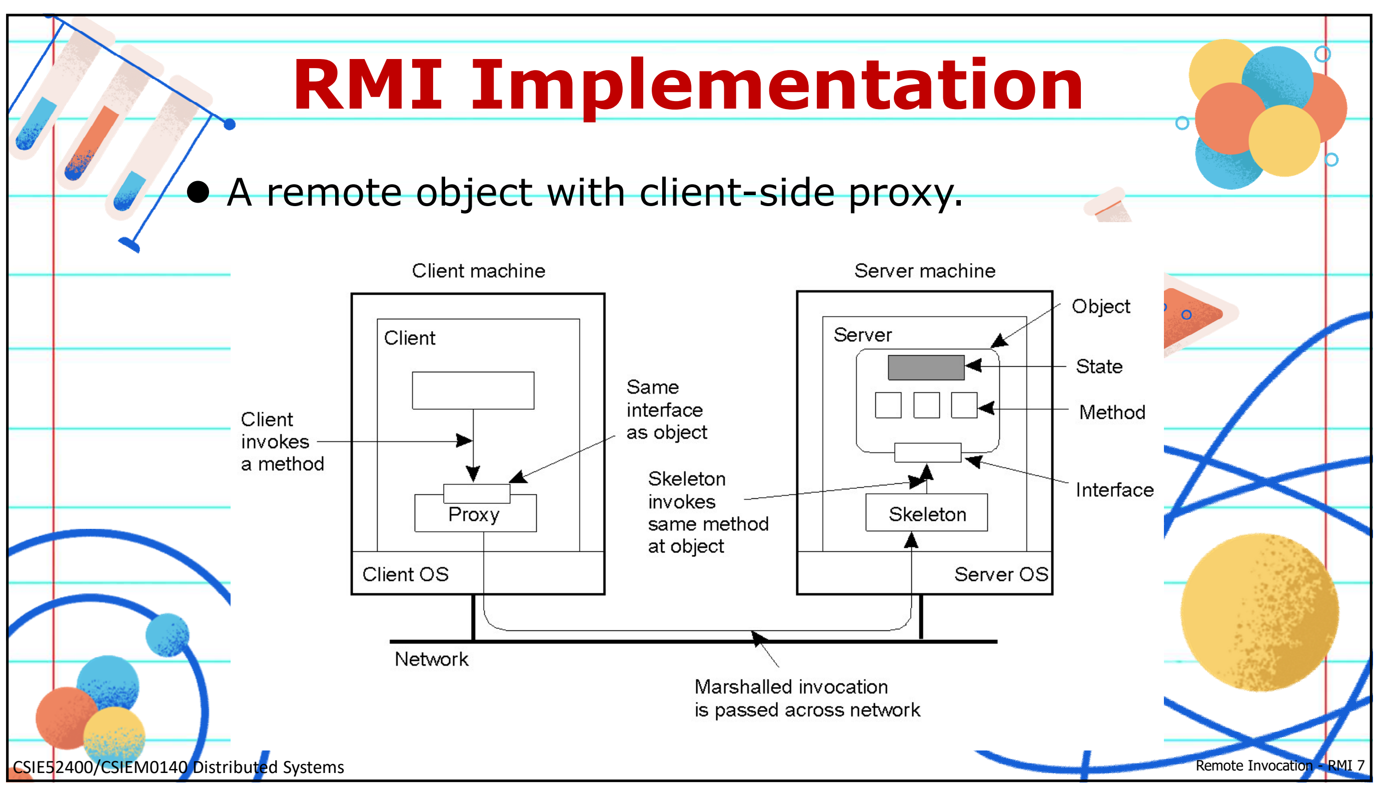

RMI vs RPC
Commonalities
Programming with interfaces
Request-reply protocol and call semantics
Similar level of transparency
Differences
RMI supports OOP
Unique object references allow richer parameter-passing semantics
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 2

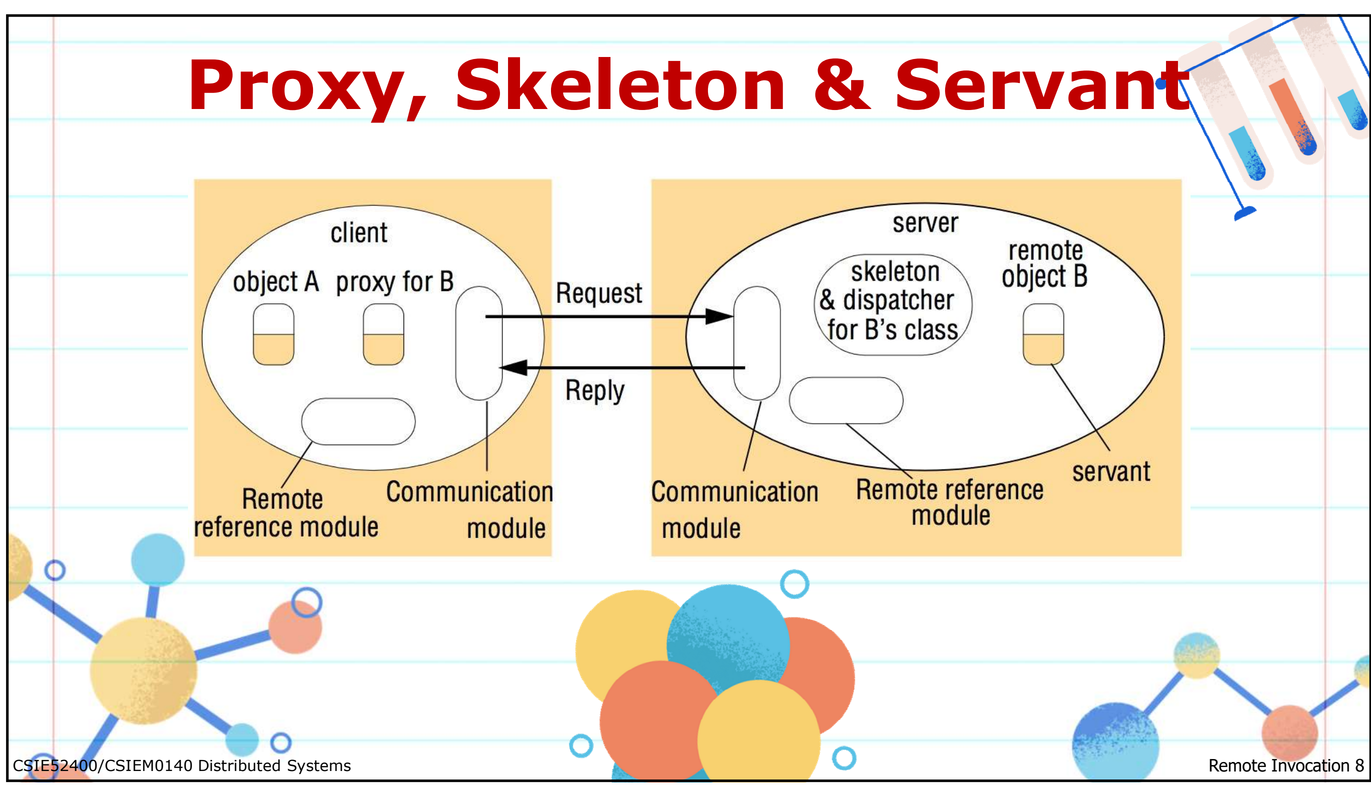

Distributed Object Model
Remote objects
objects that can receive remote invocations
Remote object references
a unique global identifier to refer to an object
Remote interfaces
specifies remotely invokable methods of an object
Remote method invocations (Actions)
invocations between objects in different processes
Exceptions
Handle error conditions
Garbage collection
Automatic freeing of space of unused objects
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 3


Remote and Local Method Invocations
A
remote
invocation
B
local
invocation
C
local
E
invocation
local
invocation
D
remote
invocation
F
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 4


Remote Object and Interface
remote object
Data
Implementation of methods
remote interface
m1
m2
m3
m4
m5
m6
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 5


Instantiation of Remote Objects
remote invocation
remote invocation
C
L
K
instantiate
instantiate
M
N
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 6


RMI Implementation
A remote object with client-side proxy.
Client machine
Server machine
Client
Server
Object
State
Method
Same interface as object
Client invokes a method
Proxy
Skeleton invokes same method at object
Skeleton
Interface
Client OS
Server OS
Network
Marshalled invocation is passed across network
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 7


Proxy, Skeleton & Servant
client
object A
proxy for B
Request
Reply
server
skeleton & dispatcher for B's class
remote object B
Remote reference module
Communication module
Communication module
Remote reference module
servant
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 8

The RMI Software

- Proxy: one proxy for each remote object to make remote method invocation transparent to clients
- Skeleton
  - each remote object class has a skeleton that implements the methods in the remote interface
  - unmarshals the request message, invokes the corresponding remote object, waits for its completion, and marshals the result
- Dispatcher
  - one dispatcher and skeleton for each class representing a remote object
  - receives request message, selects appropriate method, passes on the request to the skeleton

Binding a Client to an Object

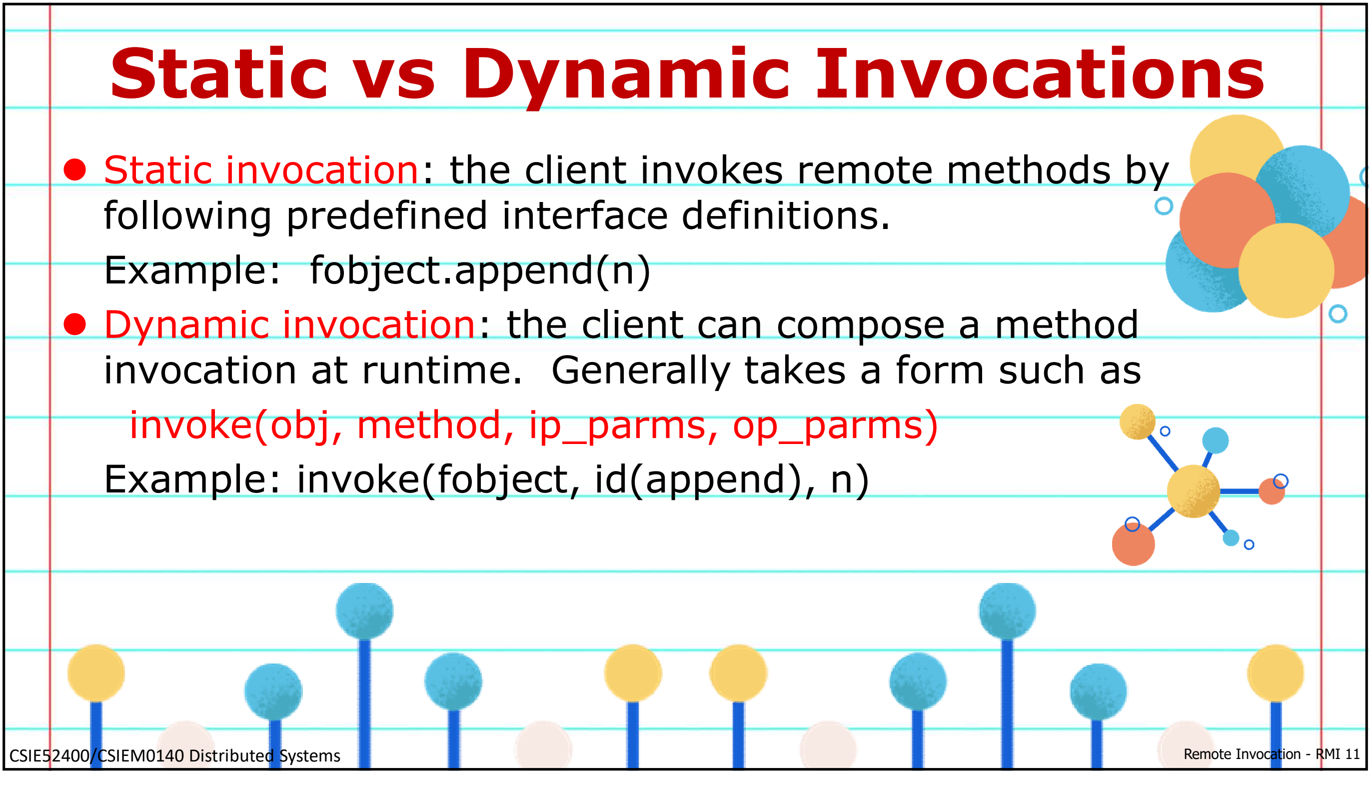
```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref = …;                    //Initialize the reference to a distributed object
obj_ref-> do_something();       //Implicitly bind and invoke a method
```

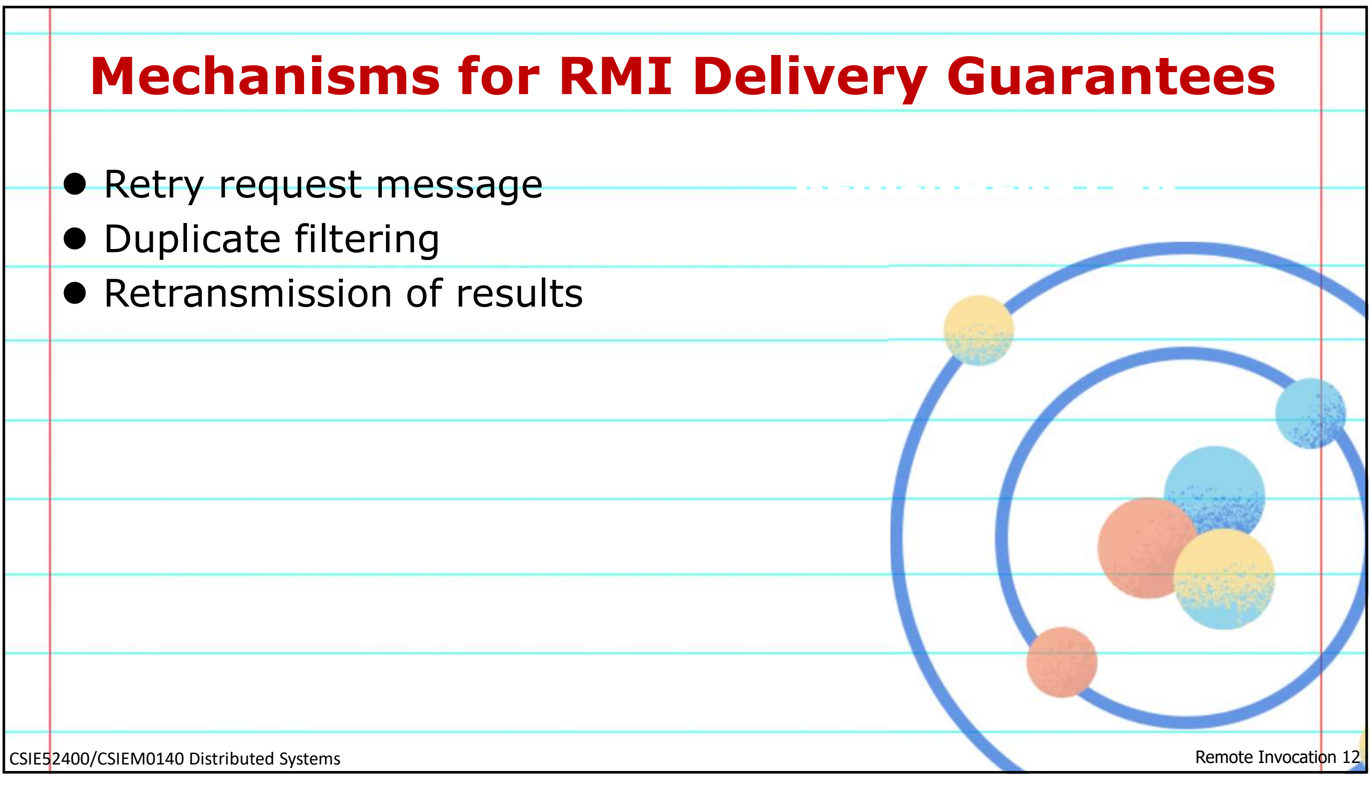
(a)

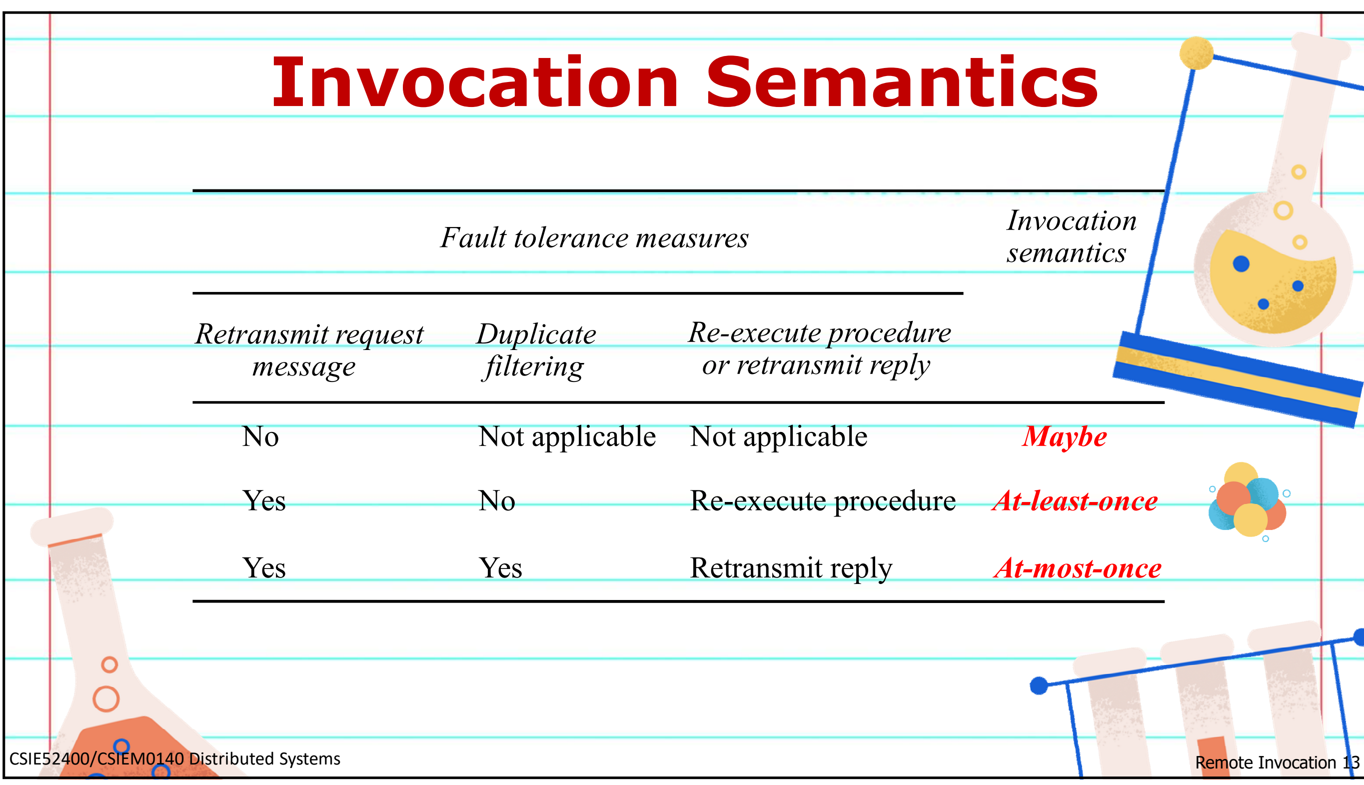
```
Distr_object obj_ref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref = …;                    //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);        //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();      //Invoke a method on the local proxy
```

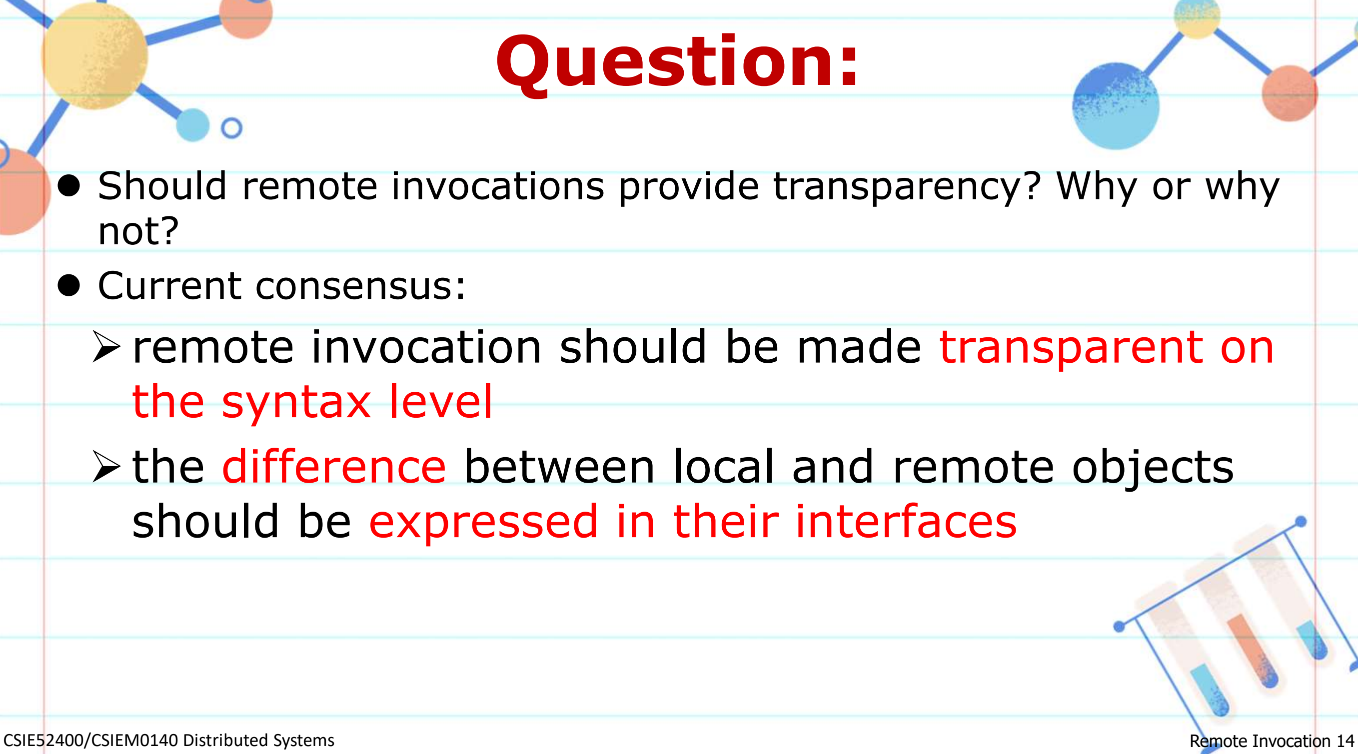
(b)

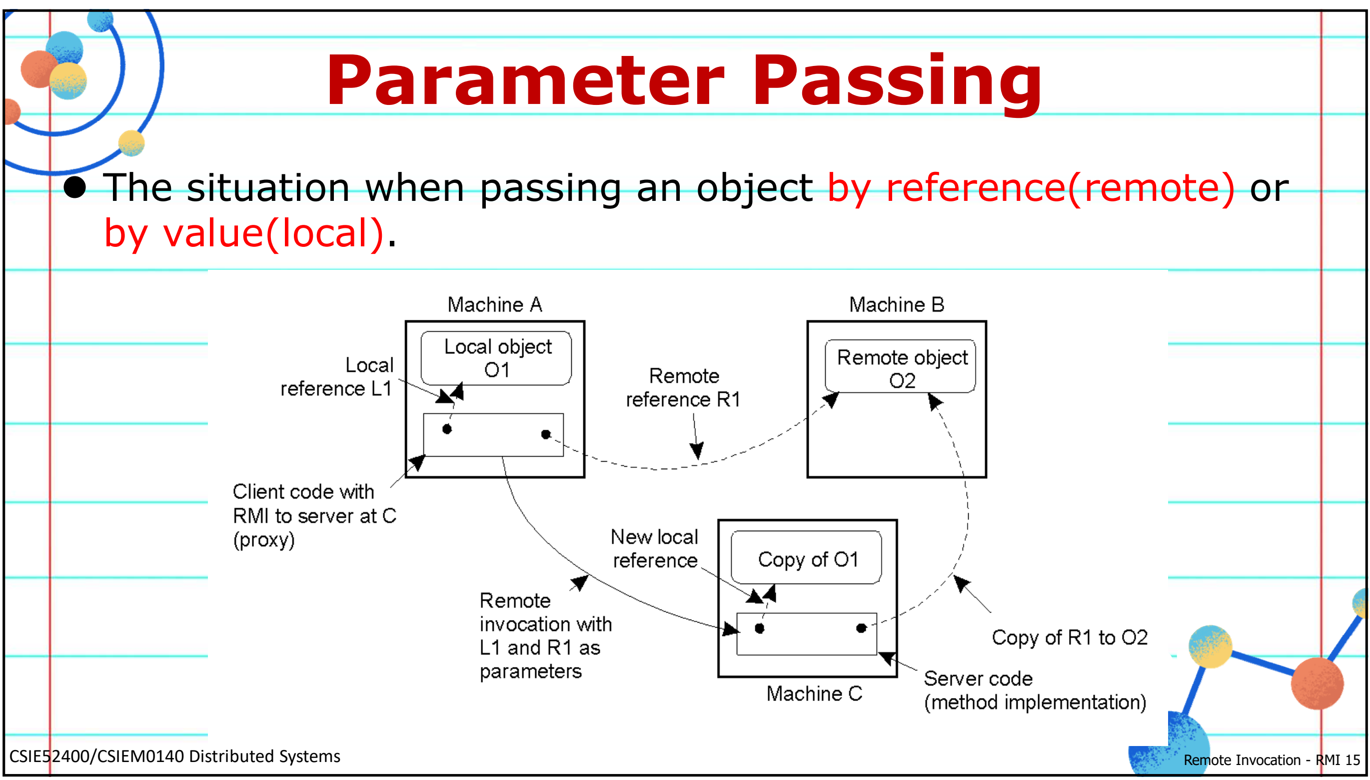
a) Implicit binding using global references
b) Explicit binding using global and local references
*: A binder is a service to keep the mappings between names and object references

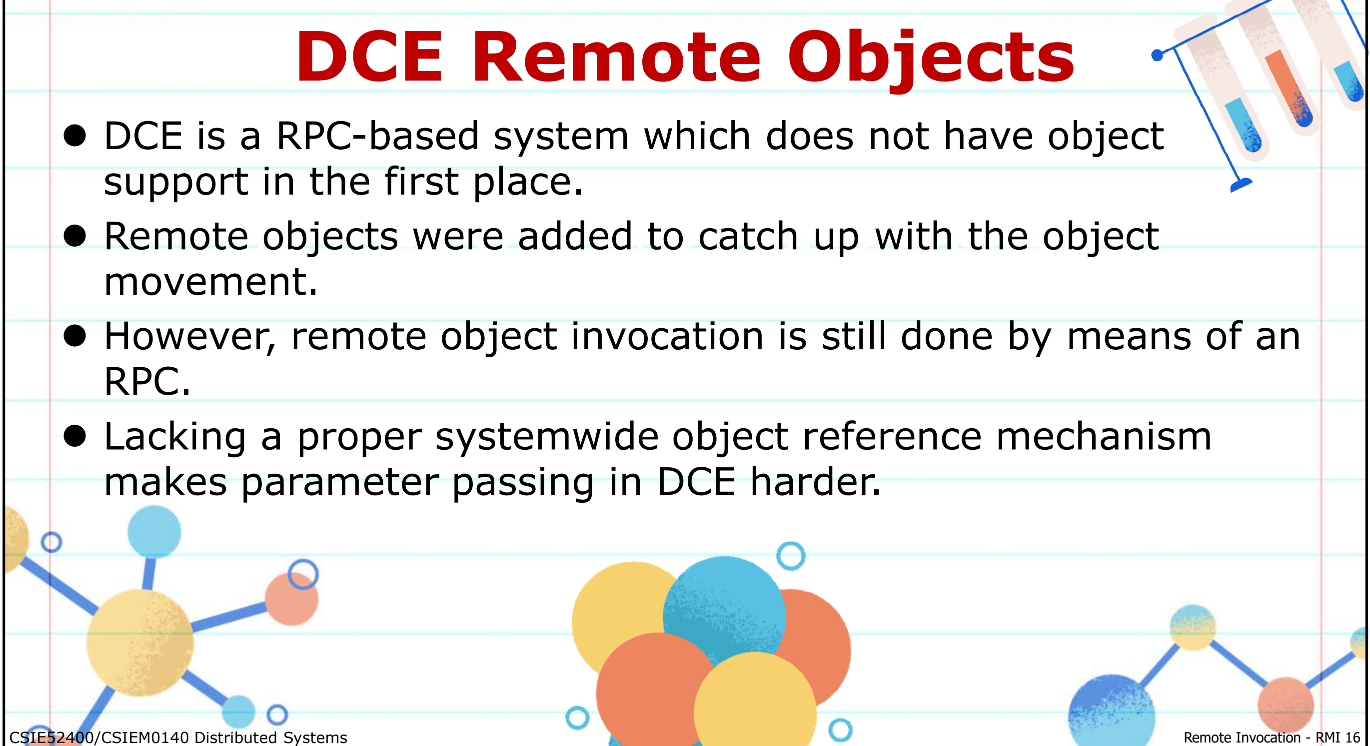

Static vs Dynamic Invocations
Static invocation: the client invokes remote methods by following predefined interface definitions.
Example: fobject.append(n)
Dynamic invocation: the client can compose a method invocation at runtime. Generally takes a form such as
invoke(obj, method, ip_parms, op_parms)
Example: invoke(fobject, id(append), n)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 11

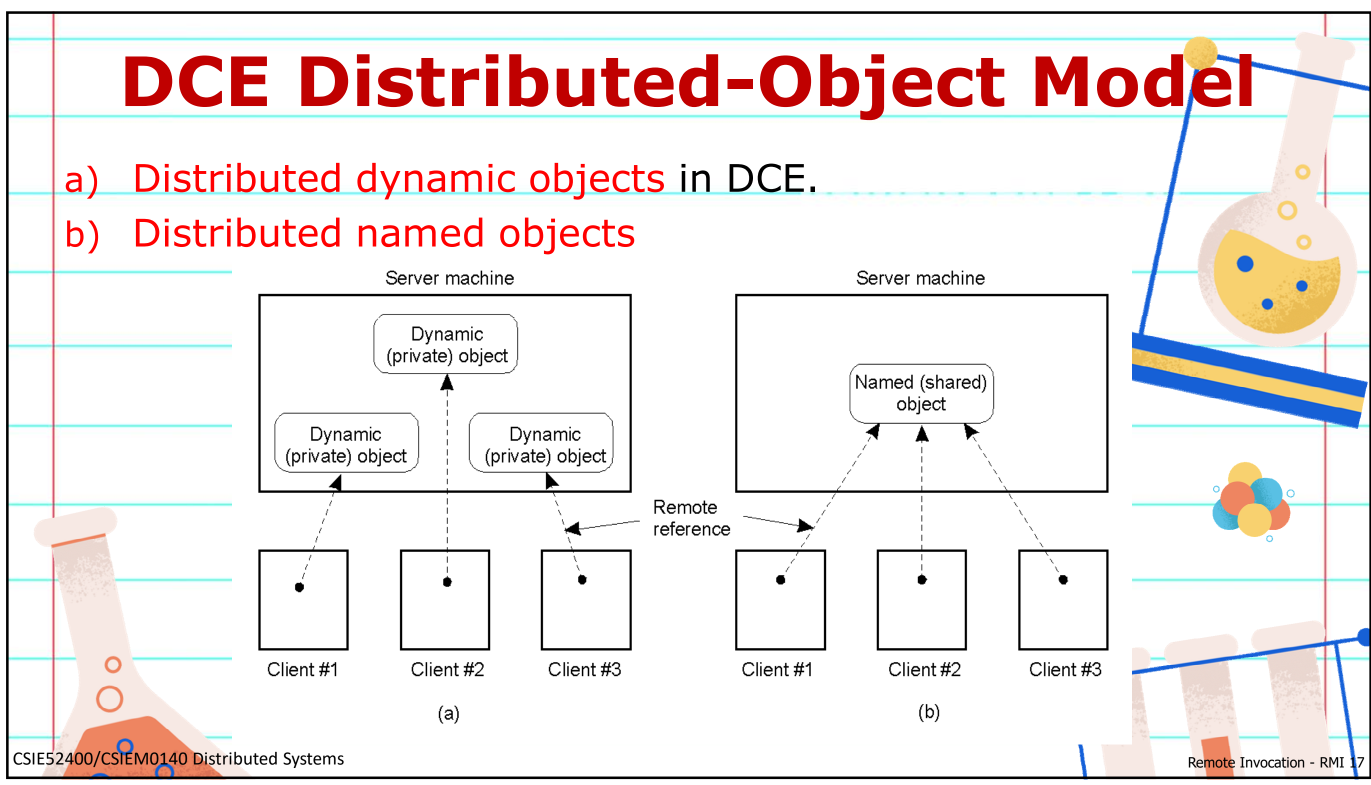

Mechanisms for RMI Delivery Guarantees
Retry request message
Duplicate filtering
Retransmission of results
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 12

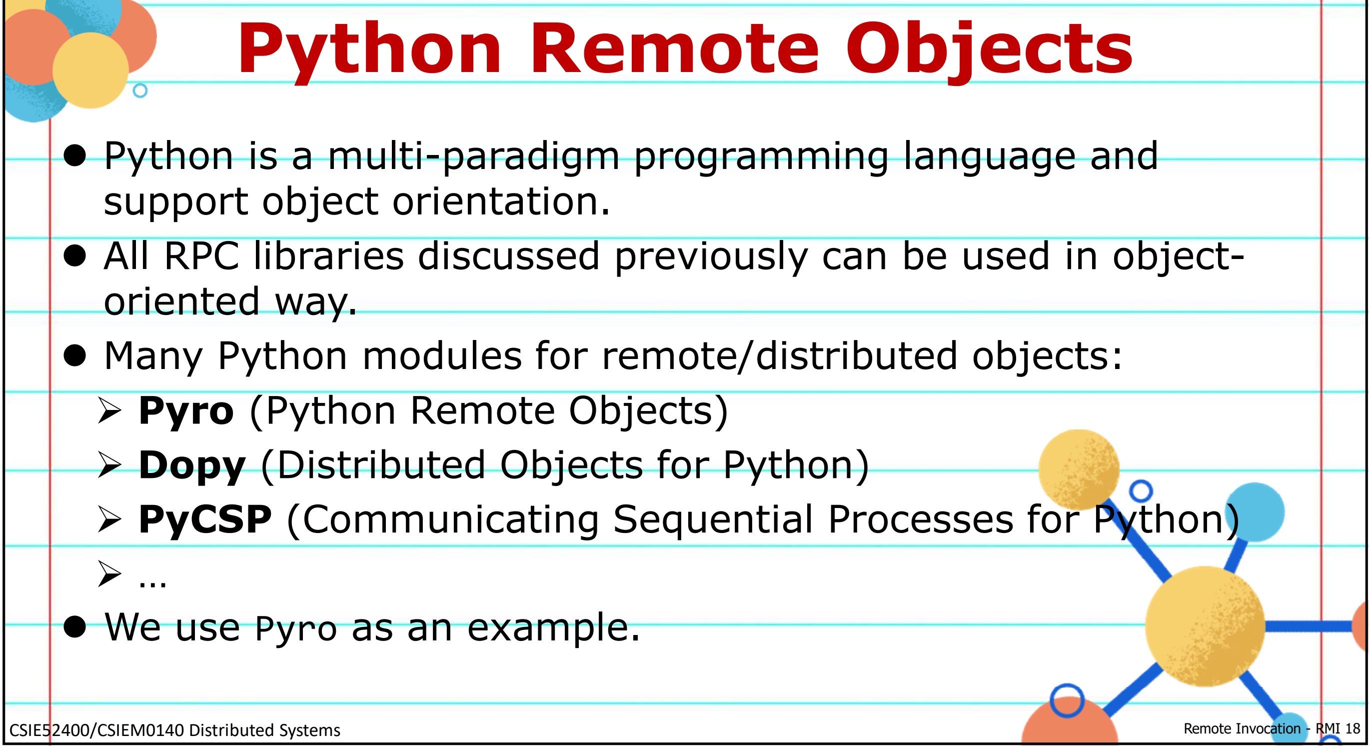

Invocation Semantics
Fault tolerance measures
Invocation semantics
Retransmit request message
Duplicate filtering
Re-execute procedure or retransmit reply
No
Not applicable
Not applicable
Maybe
Yes
No
Re-execute procedure
At-least-once
Yes
Yes
Retransmit reply
At-most-once
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 13

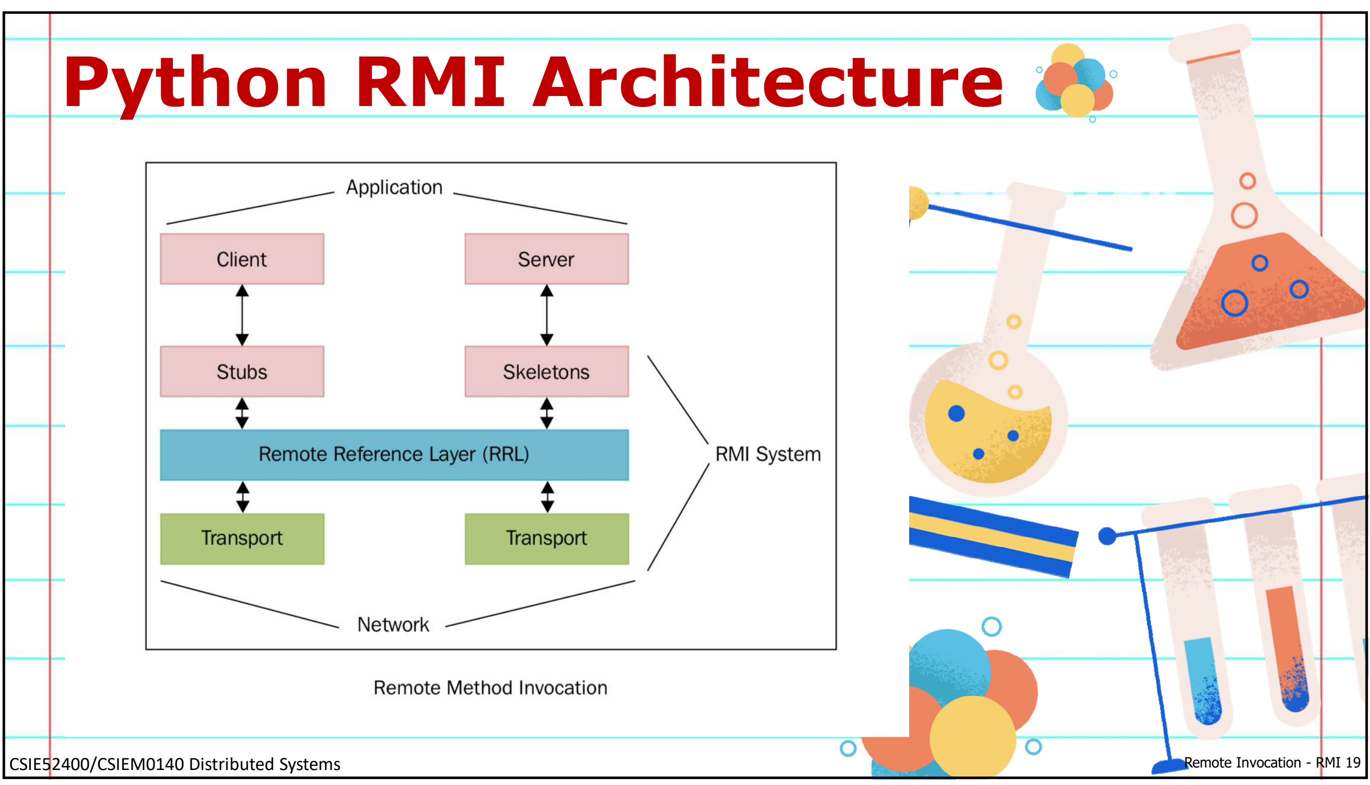

Question:
Should remote invocations provide transparency? Why or why not?
Current consensus:
remote invocation should be made transparent on the syntax level
the difference between local and remote objects should be expressed in their interfaces
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 14


Parameter Passing
The situation when passing an object by reference(remote) or by value(local).
Machine A
Local object O1
Local reference L1
Remote reference R1
Machine B
Remote object O2
Client code with RMI to server at C (proxy)
New local reference
Copy of O1
Remote invocation with L1 and R1 as parameters
Copy of R1 to O2
Server code (method implementation)
Machine C
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 15


DCE Remote Objects
DCE is a RPC-based system which does not have object support in the first place.
Remote objects were added to catch up with the object movement.
However, remote object invocation is still done by means of an RPC.
Lacking a proper systemwide object reference mechanism makes parameter passing in DCE harder.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 16


DCE Distributed-Object Model
a) Distributed dynamic objects in DCE.
b) Distributed named objects
Server machine
Dynamic (private) object
Dynamic (private) object
Dynamic (private) object
Server machine
Named (shared) object
Remote reference
Client #1
Client #2
Client #3
(a)
Client #1
Client #2
Client #3
(b)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 17


Python Remote Objects
Python is a multi-paradigm programming language and support object orientation.
All RPC libraries discussed previously can be used in object-oriented way.
Many Python modules for remote/distributed objects:
Pyro (Python Remote Objects)
Dopy (Distributed Objects for Python)
PyCSP (Communicating Sequential Processes for Python)
...
We use Pyro as an example.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 18



Pyro

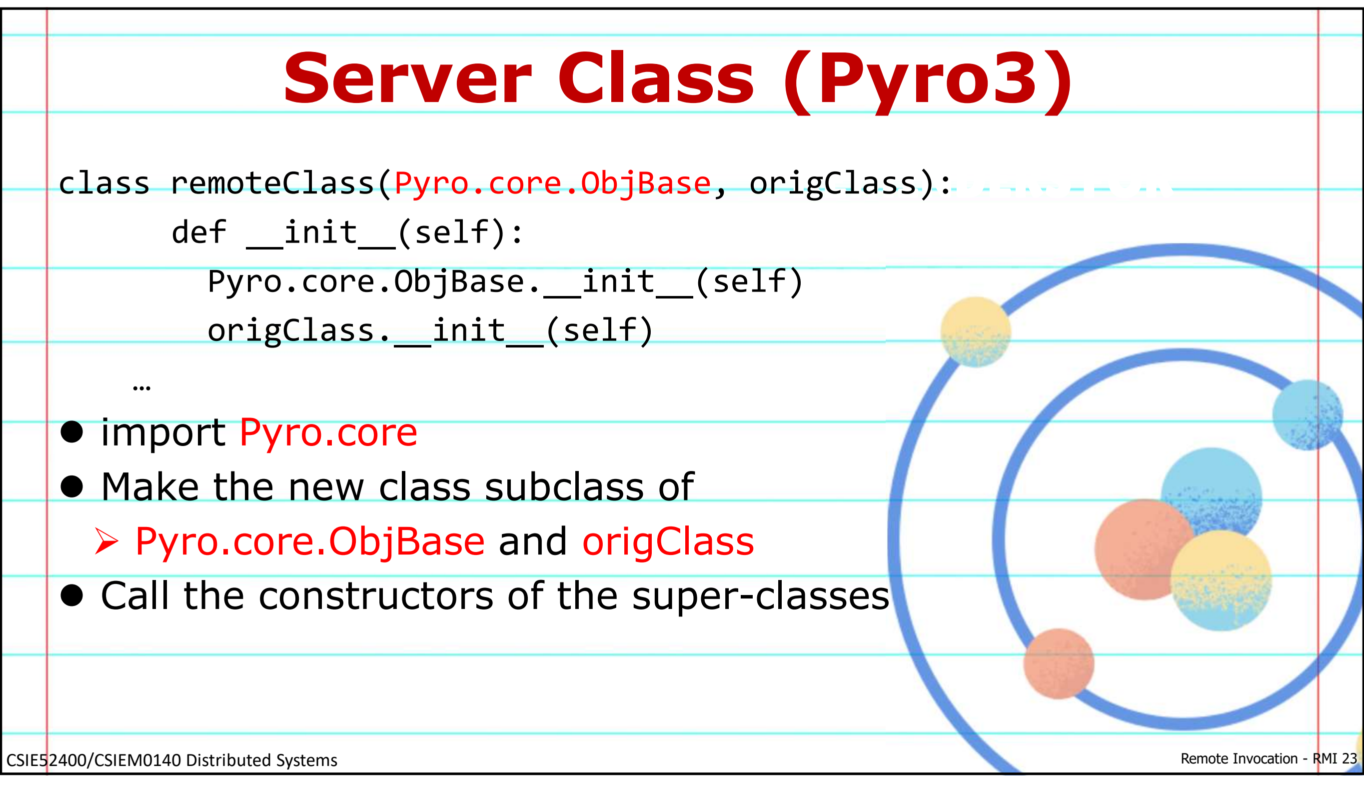
- Python Remote Objects
  - Pyro3 - https://pythonhosted.org/Pyro/ (Pyro 3.16)
  - Pyro4 - https://pythonhosted.org/Pyro4/ (Pyro4 4.82)
  - Pyro5 - https://pypi.org/project/Pyro5/ (Pyro5 5.15)
- Distributed Object Technology
  - RMI
  - Mobile code
- 100% pure Python, comm between diff Python versions
- Name server, IPv4/IPv6, SSL/TLS, remote exceptions, …
- Should use Pyro5 whenever possible.

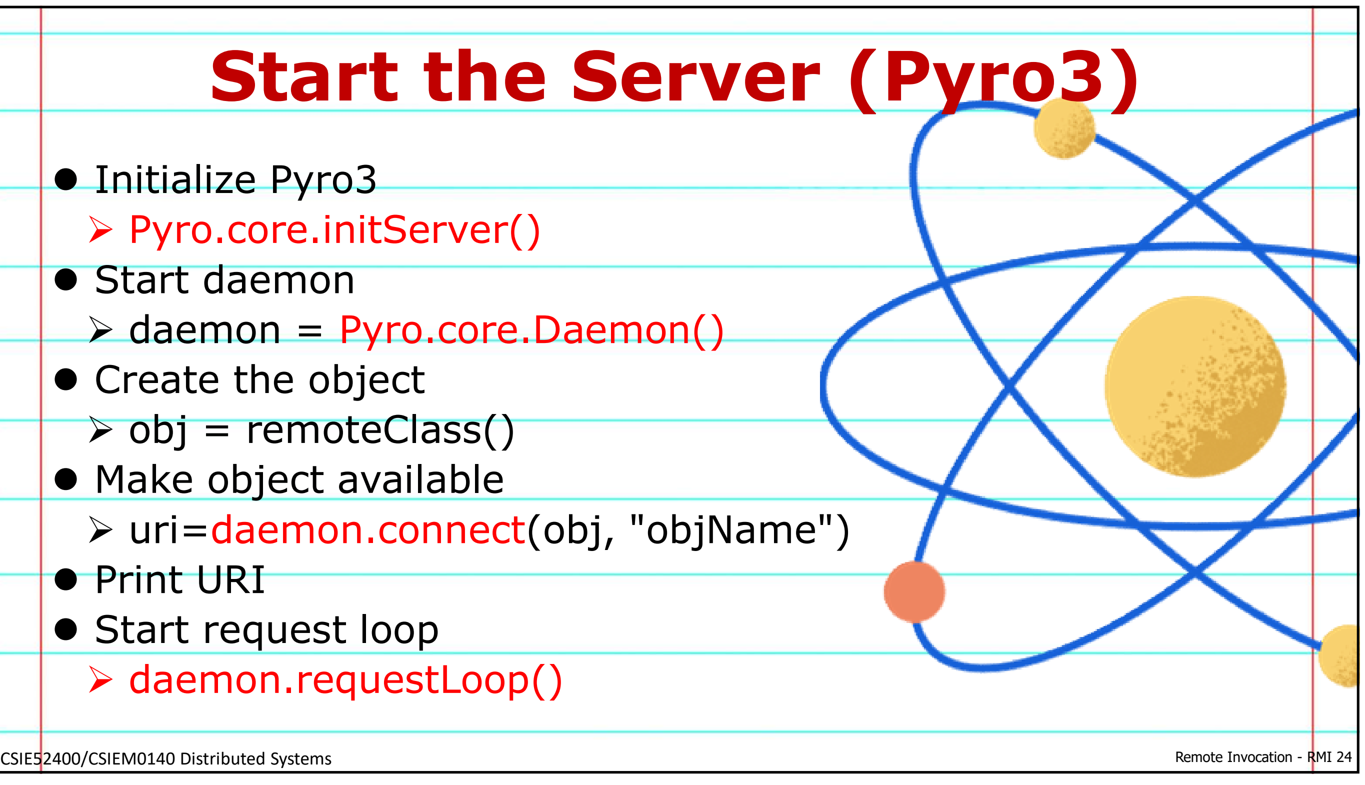
Pyro Overview

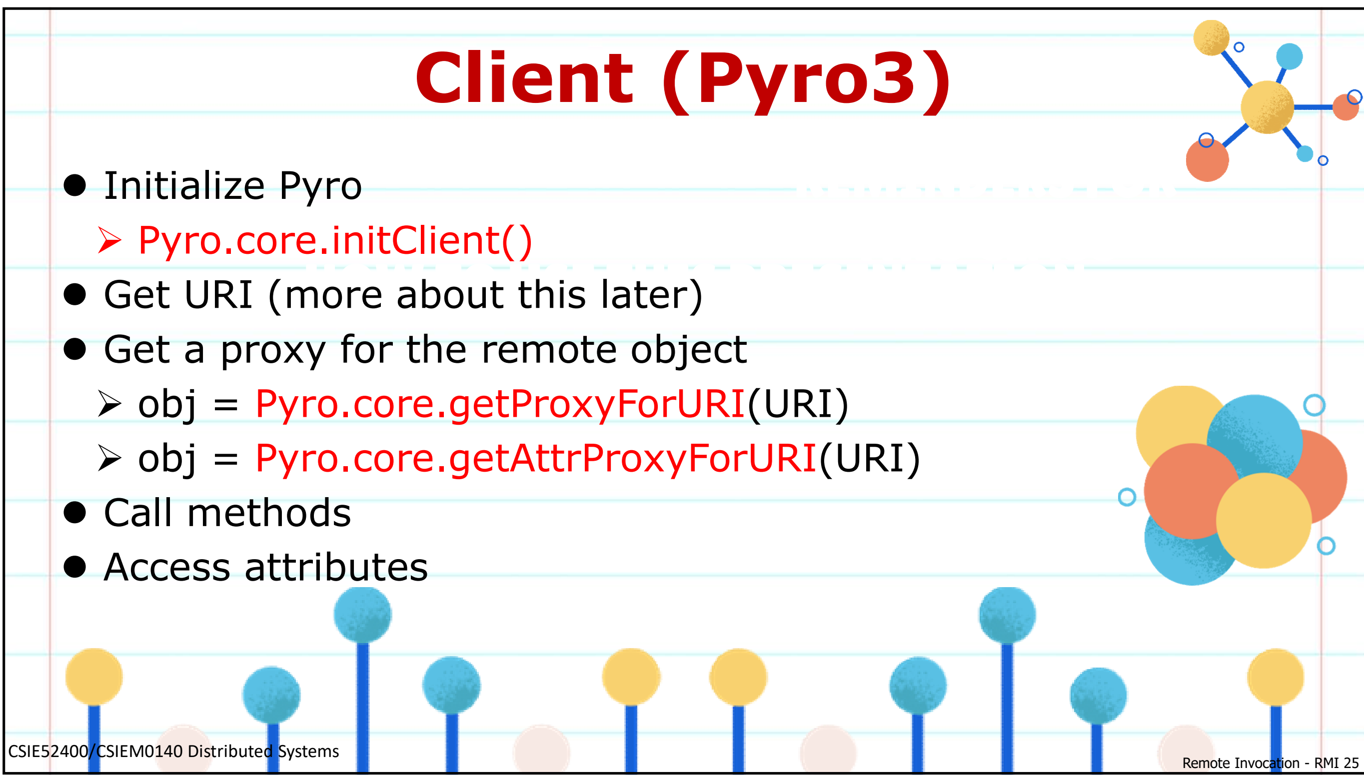
- Server
  - Write a server program
    - containing a class 'serverClass'
  - Expose the server class 'serverClass'
  - Register the class with Pyro daemon (returns an URI) as a Pyro object
  - [Register the object(URI) with a name in the Name Server]
  - Start the request loop
- Client
  - Queries the Name Server for the server object(s)
    - returns Pyro URI (Universal Resource Identifier) for them.
  - Create proxie(s) for the remote object(s).
    - Proxy mimics the real 'serverClass',
  - Invoke methods on the remote objects.
    - The proxy will forward the method invocations and return the results, just as if it was the local object itself.

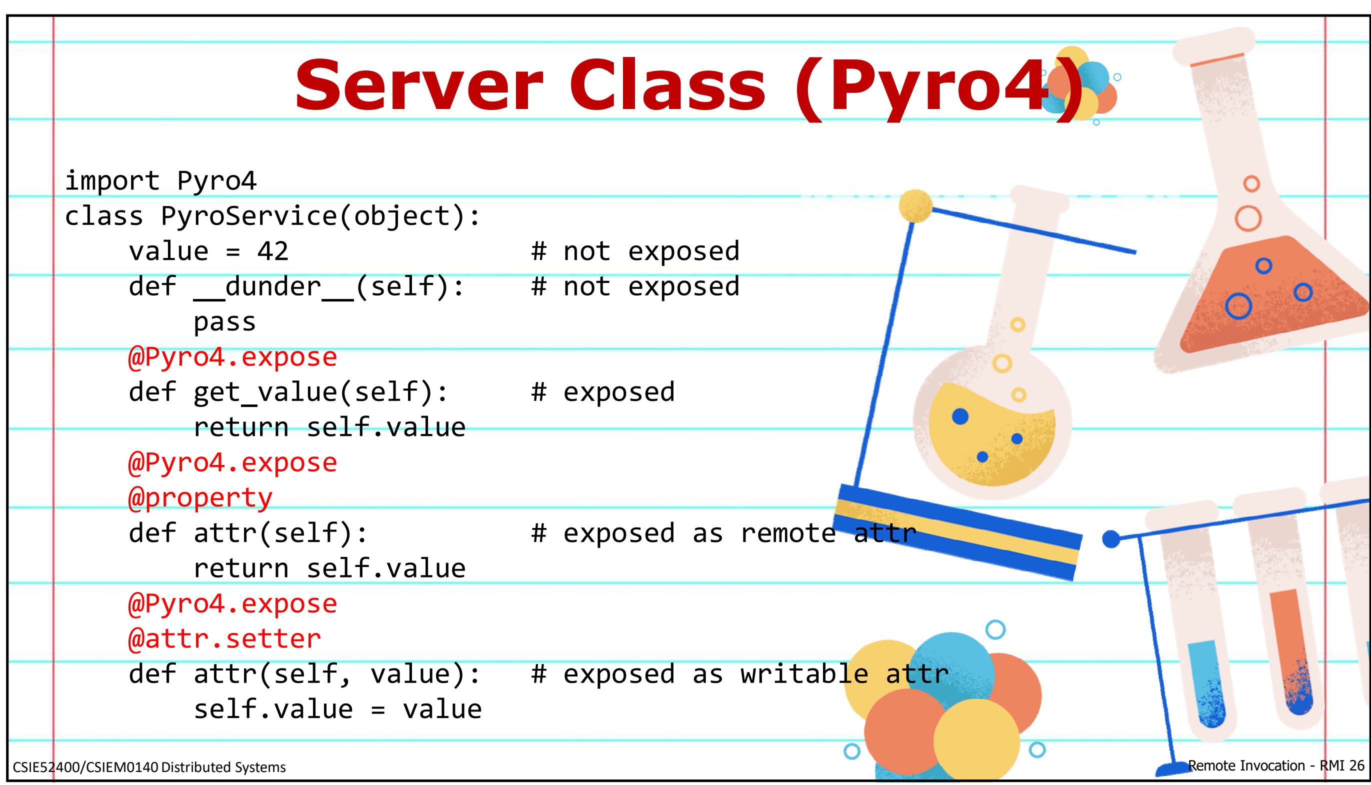
Server

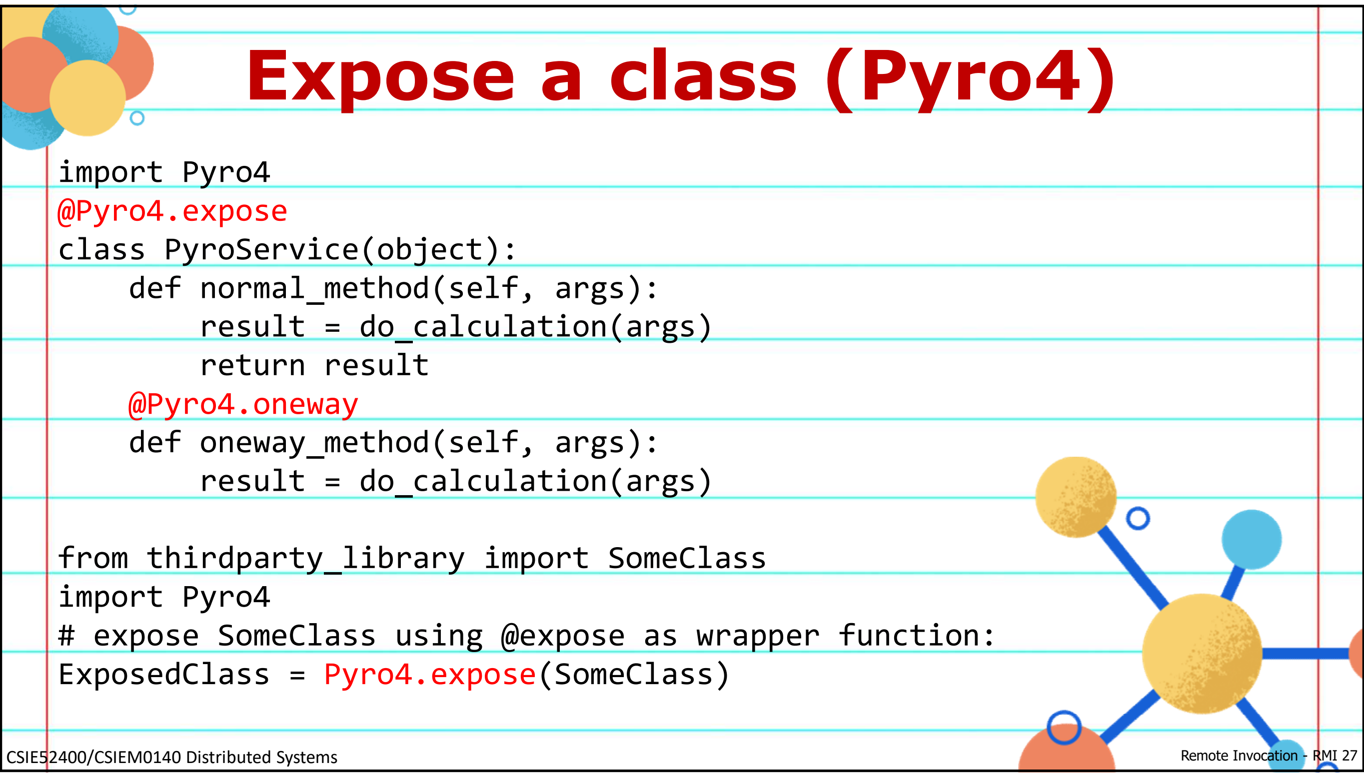
- Implement a server class to be accessed remotely with methods and attributes.
- Make it "remotable"
  - Pyro3
    - Make it a subclass of Pyro.core.ObjBase
    - Derive a new class
  - Pyro4
    - Expose methods: @Pyro4.expose
    - Create a new "exposed" class:
      - ✓ ExposedClass = Pyro4.expose(SomeClassFromLibrary)
  - Pyro5
    - Expose the server class: @Pyro5.api.expose
    - Can give it a name through Name Server

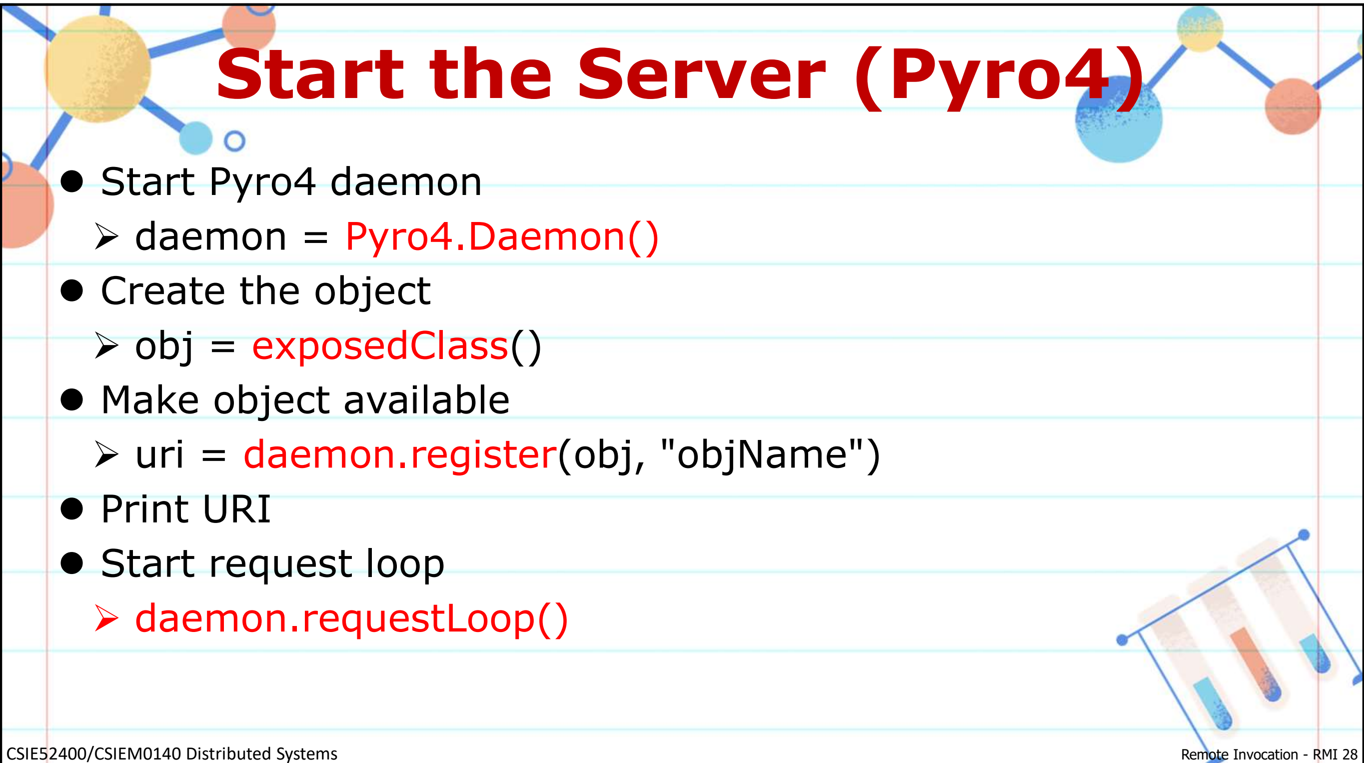

Server Class (Pyro3)
class remoteClass(Pyro.core.ObjBase, origClass):
def __init__(self):
Pyro.core.ObjBase.__init__(self)
origClass.__init__(self)
…
import Pyro.core
Make the new class subclass of
Pyro.core.ObjBase and origClass
Call the constructors of the super-classes
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 23

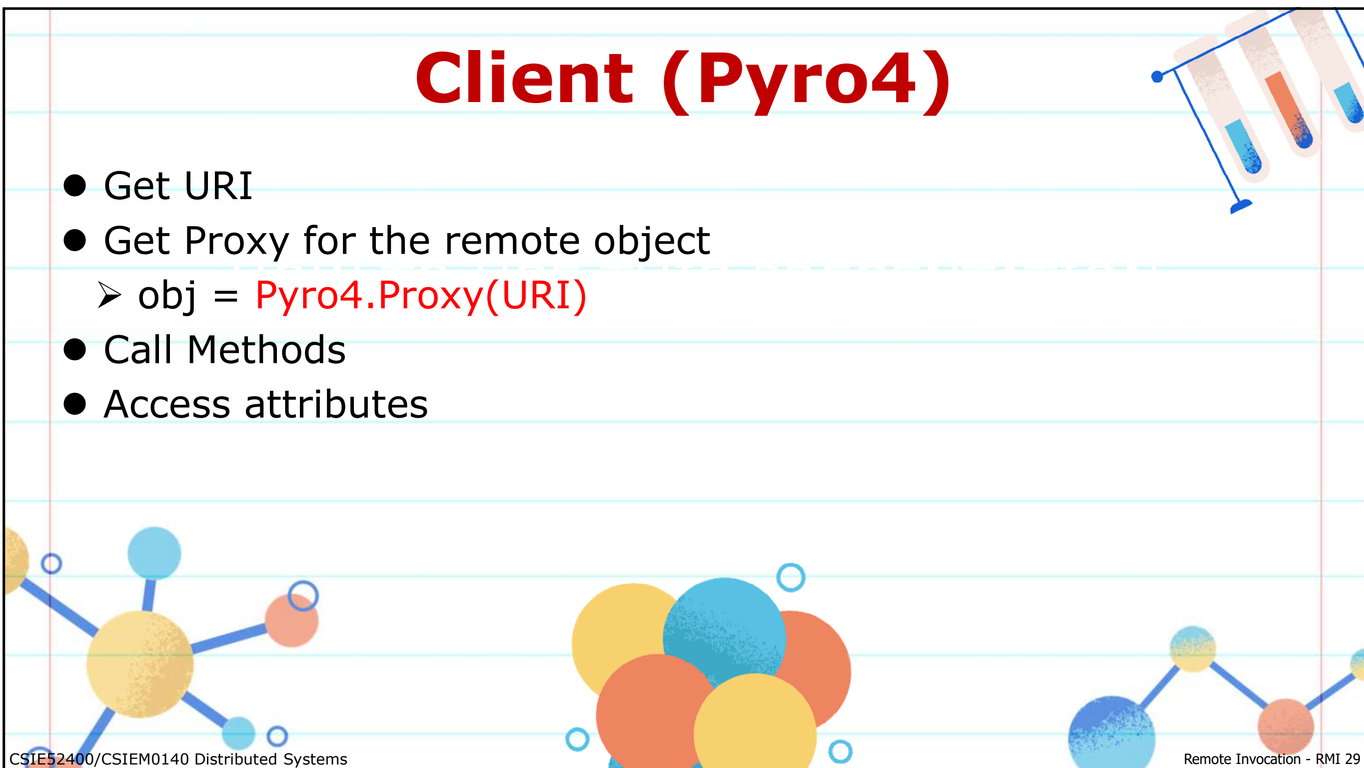

Start the Server (Pyro3)
Initialize Pyro3
Pyro.core.initServer()
Start daemon
daemon = Pyro.core.Daemon()
Create the object
obj = remoteClass()
Make object available
uri=daemon.connect(obj, "objName")
Print URI
Start request loop
daemon.requestLoop()
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 24

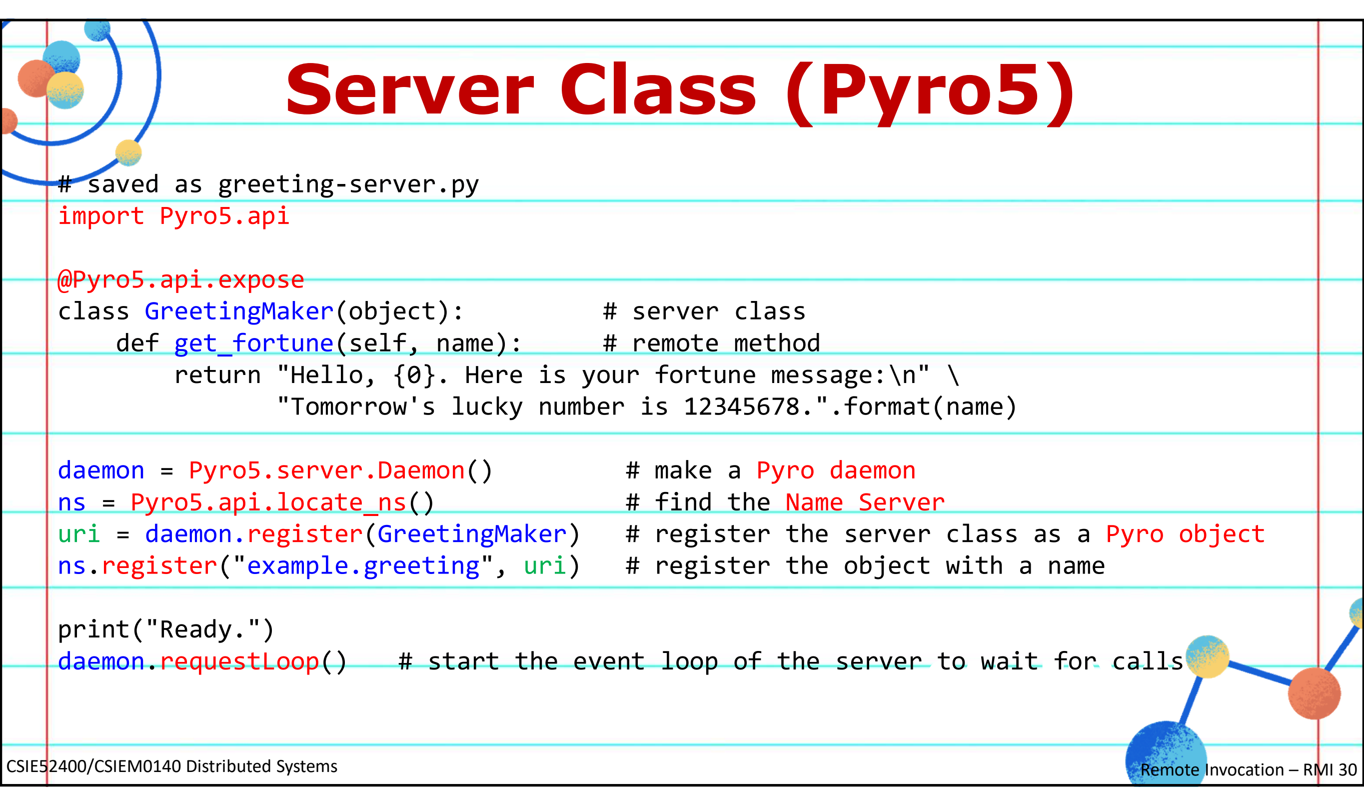

Client (Pyro3)
Initialize Pyro
Pyro.core.initClient()
Get URI (more about this later)
Get a proxy for the remote object
obj = Pyro.core.getProxyForURI(URI)
obj = Pyro.core.getAttrProxyForURI(URI)
Call methods
Access attributes
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 25

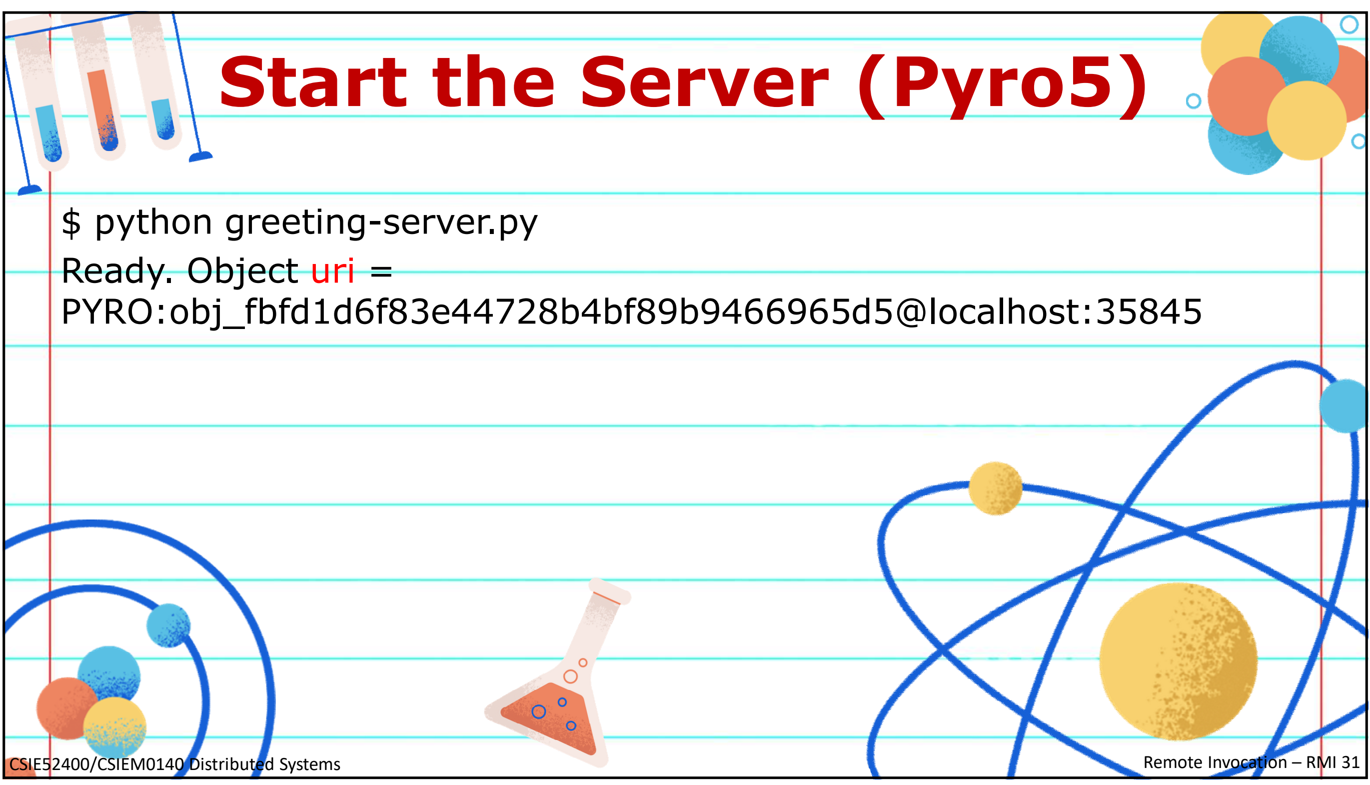

Server Class (Pyro4)
import Pyro4
class PyroService(object):
    value = 42   # not exposed
    def __dunder__(self):   # not exposed
        pass
    @Pyro4.expose
    def get_value(self):   # exposed
        return self.value
    @Pyro4.expose
    @property
    def attr(self):   # exposed as remote attr
        return self.value
    @Pyro4.expose
    @attr.setter
    def attr(self, value):   # exposed as writable attr
        self.value = value
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 26

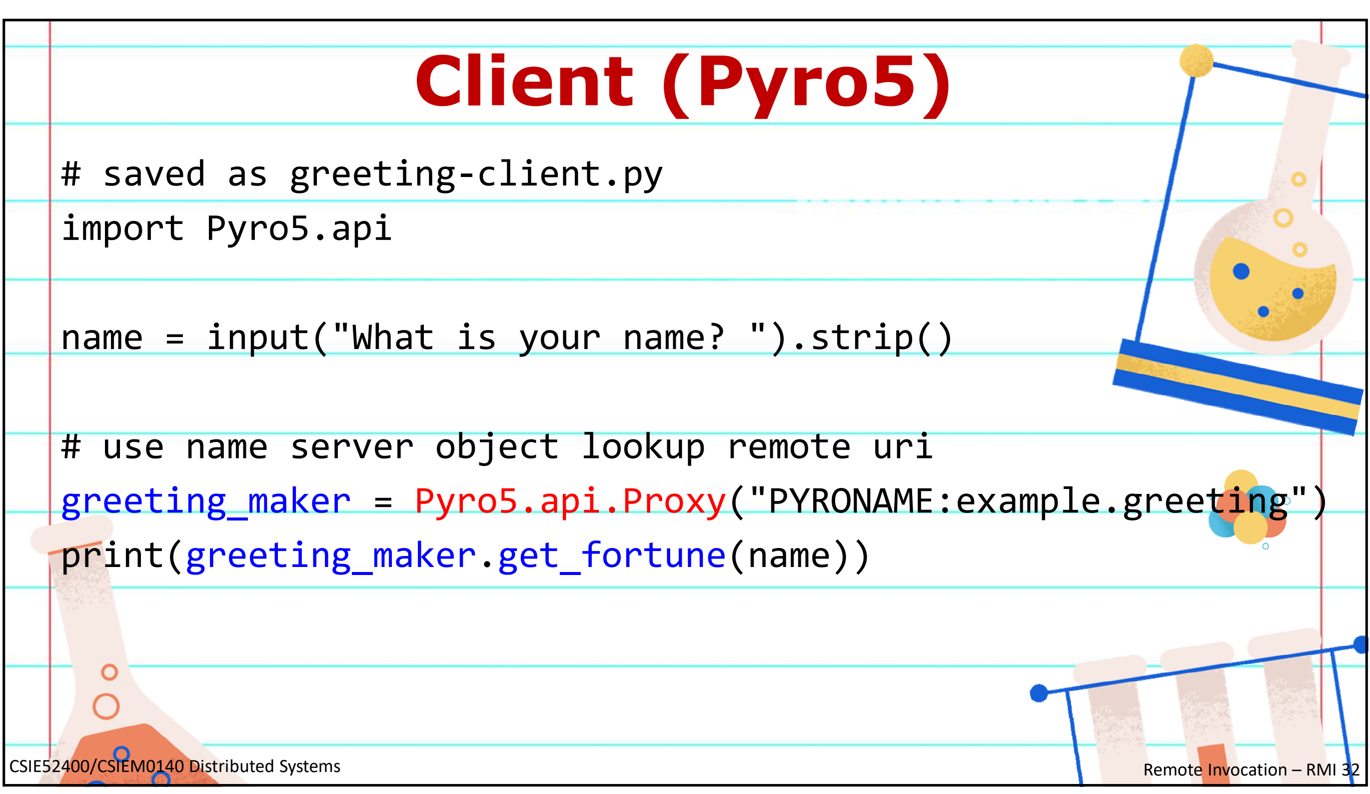

Expose a class (Pyro4)
import Pyro4
@Pyro4.expose
class PyroService(object):
def normal_method(self, args):
result = do_calculation(args)
return result
@Pyro4.oneway
def oneway_method(self, args):
result = do_calculation(args)
from thirdparty_library import SomeClass
import Pyro4
# expose SomeClass using @expose as wrapper function:
ExposedClass = Pyro4.expose(SomeClass)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 27


Start the Server (Pyro4)
Start Pyro4 daemon
daemon = Pyro4.Daemon()
Create the object
obj = exposedClass()
Make object available
uri = daemon.register(obj, "objName")
Print URI
Start request loop
daemon.requestLoop()
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 28


Client (Pyro4)
● Get URI
● Get Proxy for the remote object
➢ obj = Pyro4.Proxy(URI)
● Call Methods
● Access attributes
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 29

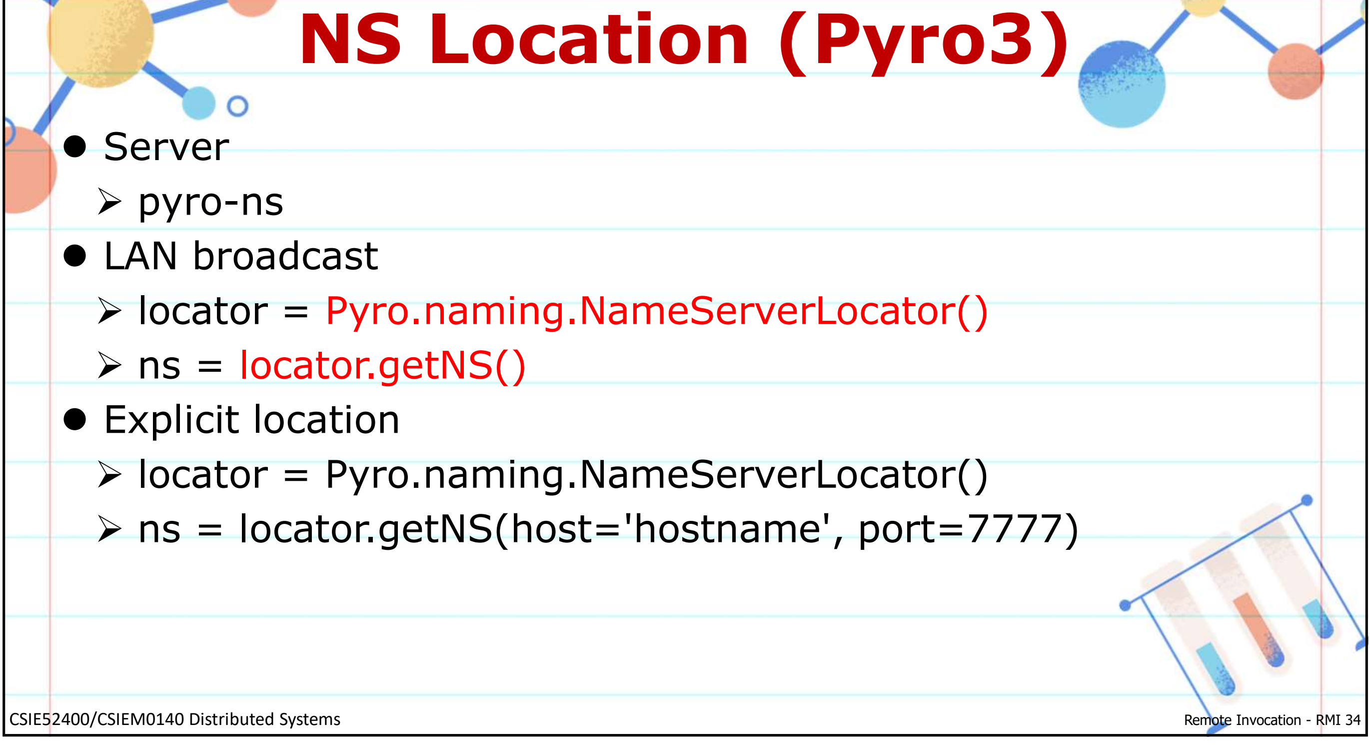

Server Class (Pyro5)
# saved as greeting-server.py
import Pyro5.api
@Pyro5.api.expose
class GreetingMaker(object): # server class
def get_fortune(self, name): # remote method
return "Hello, {0}. Here is your fortune message:\n" \
"Tomorrow's lucky number is 12345678.".format(name)
daemon = Pyro5.server.Daemon() # make a Pyro daemon
ns = Pyro5.api.locate_ns() # find the Name Server
uri = daemon.register(GreetingMaker) # register the server class as a Pyro object
ns.register("example.greeting", uri) # register the object with a name
print("Ready.")
daemon.requestLoop() # start the event loop of the server to wait for calls
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RMI 30

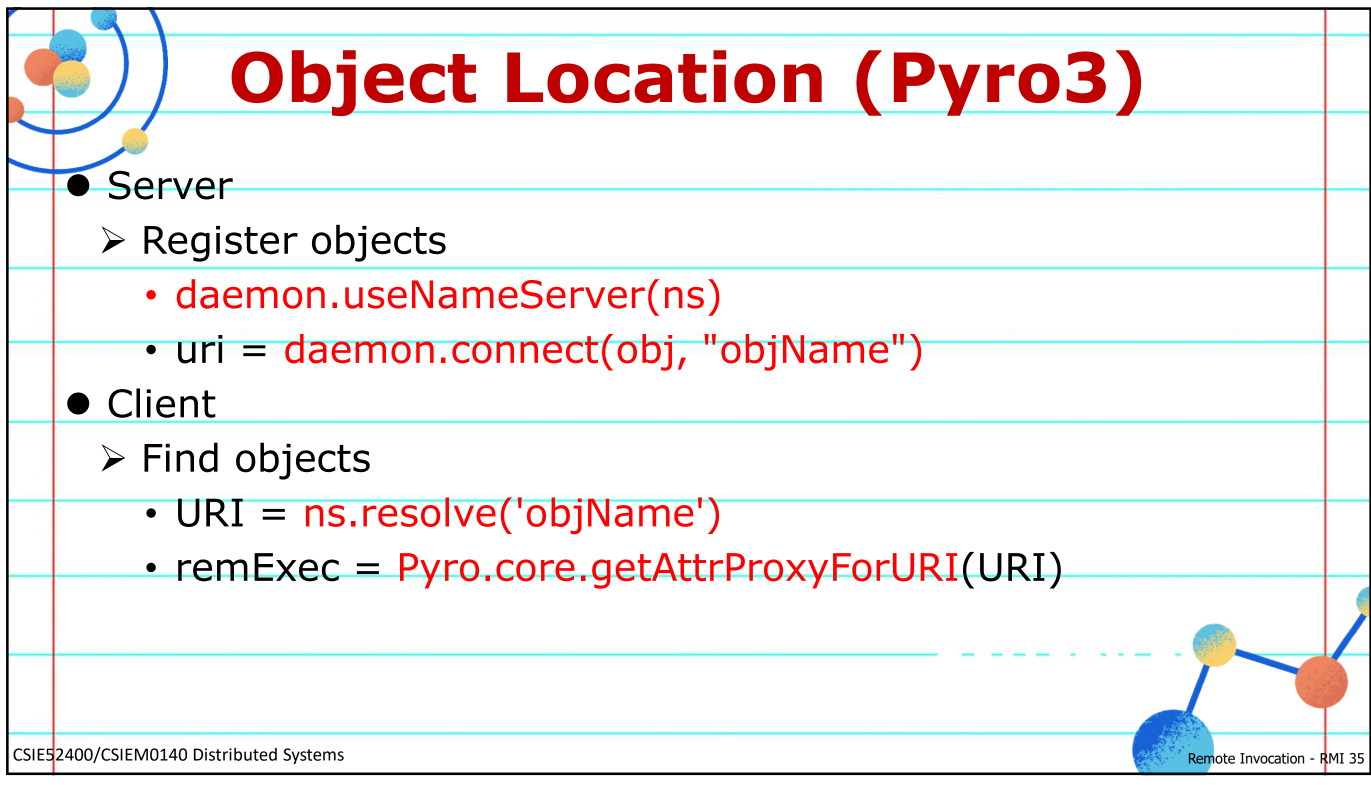

Start the Server (Pyro5)
$ python greeting-server.py
Ready. Object uri =
PYRO:obj_fbfd1d6f83e44728b4bf89b9466965d5@localhost:35845
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RMI 31

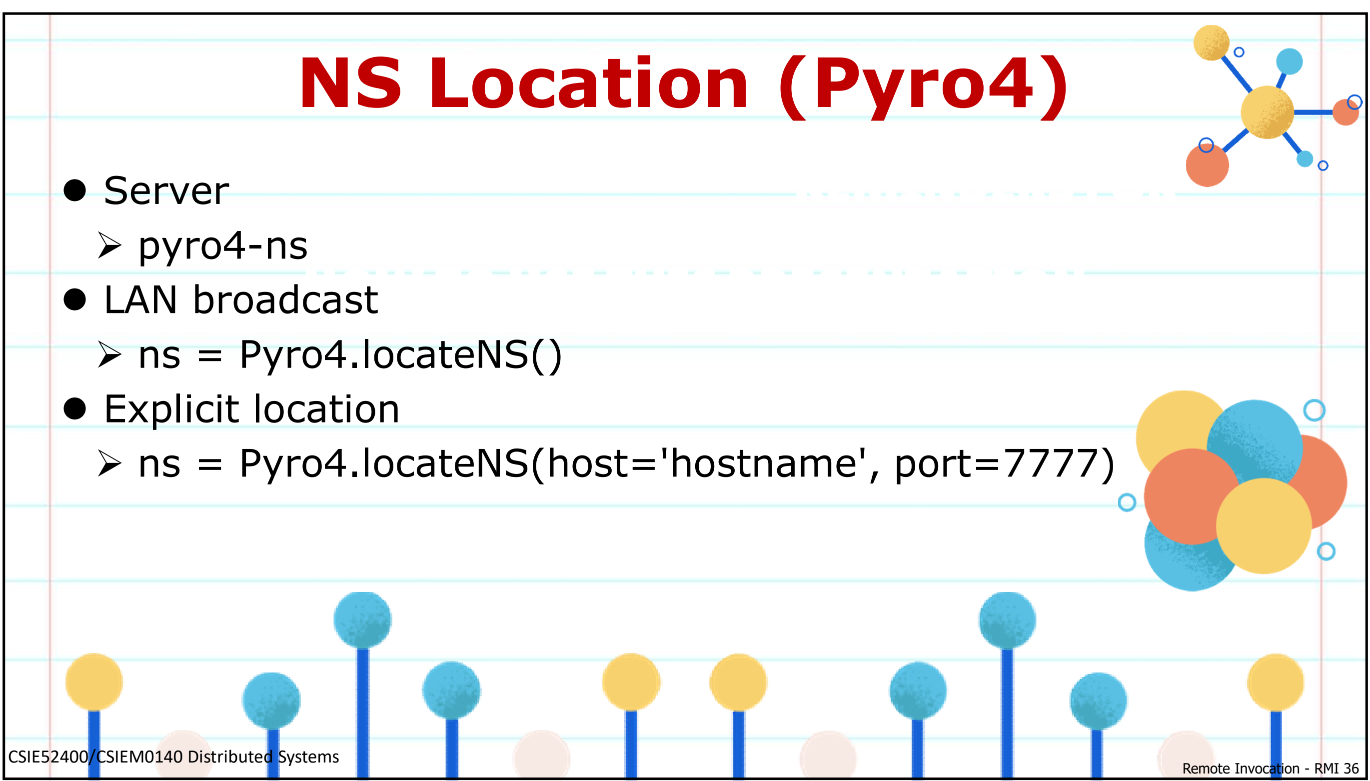

Client (Pyro5)
# saved as greeting-client.py
import Pyro5.api
name = input("What is your name? ").strip()
# use name server object lookup remote uri
greeting_maker = Pyro5.api.Proxy("PYRONAME:example.greeting")
print(greeting_maker.get_fortune(name))
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RMI 32

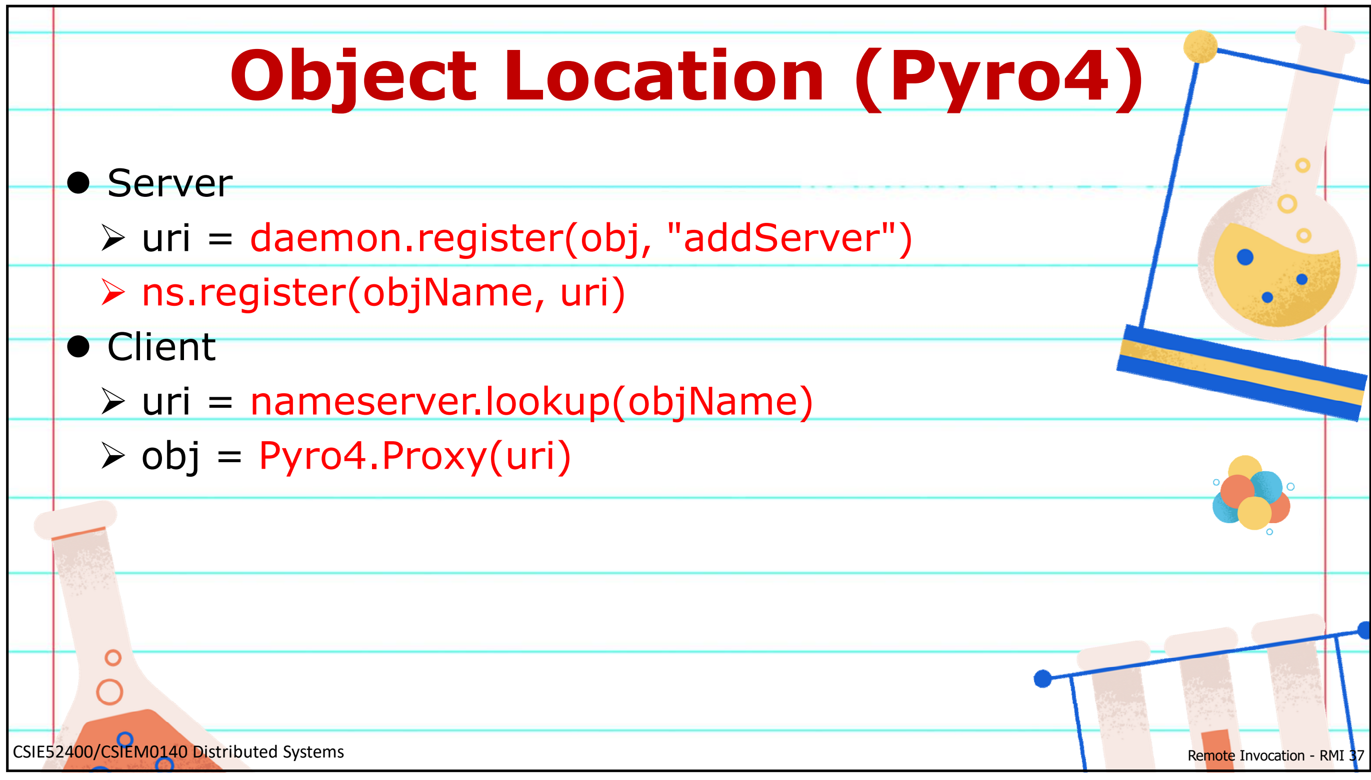
Naming Services

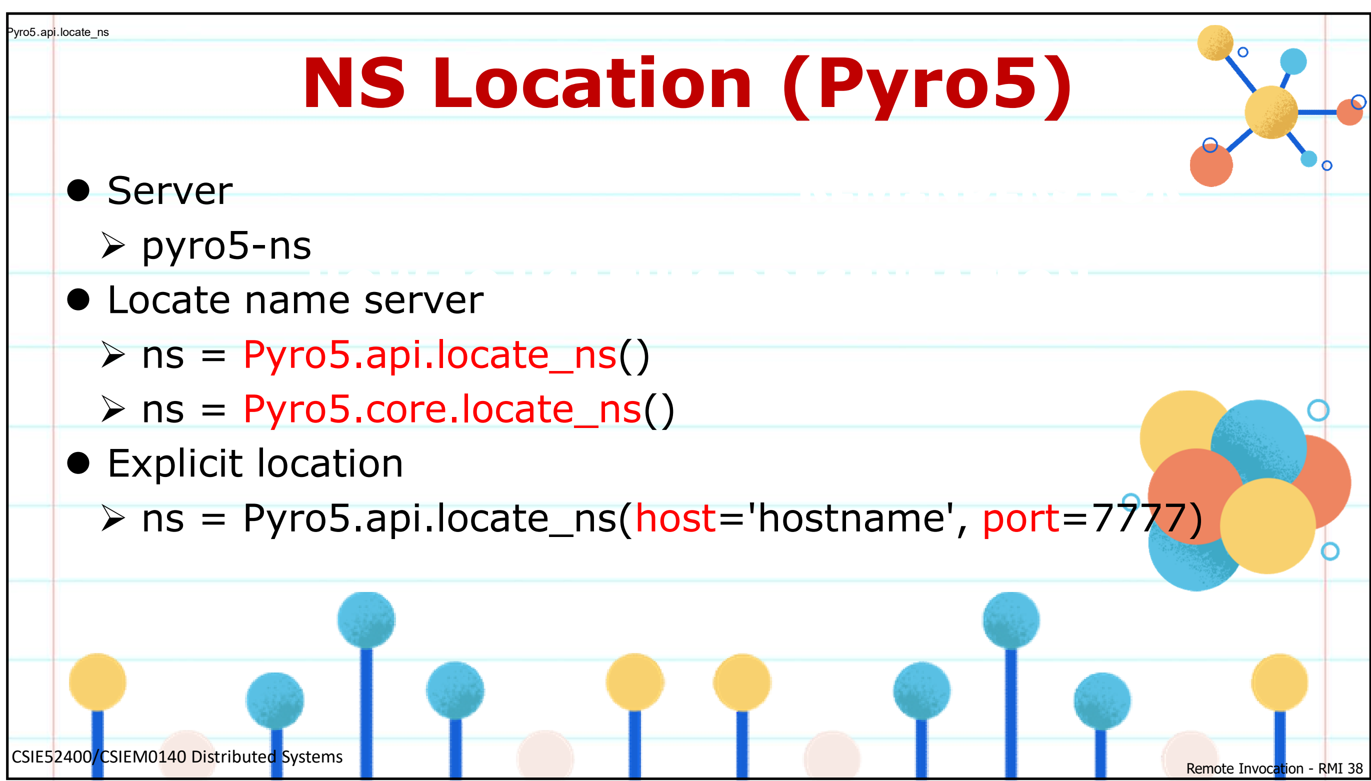
- URI is not user friendly
  - PYRO://134.208.2.15:7766/92c1290f512e20e1b13888fdd50 4a238d5
  - PYRO:addServer@localhost:51989
- Objects can be scattered on the network
- Name service should handle translations
  - Text name → URI
- Starting a name server(NS)
  - python -m Pyro4.naming or simply: pyro4-ns
  - python –m Pyro5.nameserver or simply: pyro5-ns

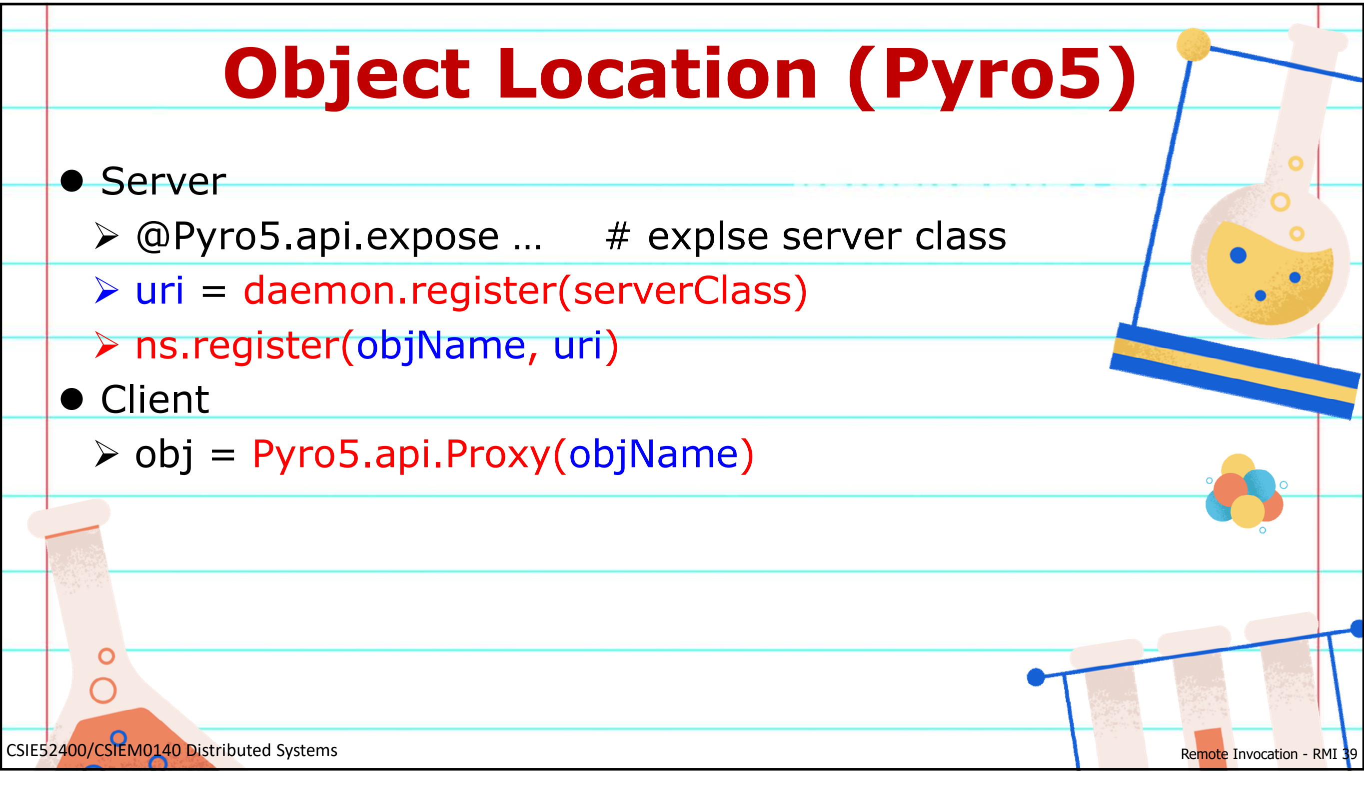


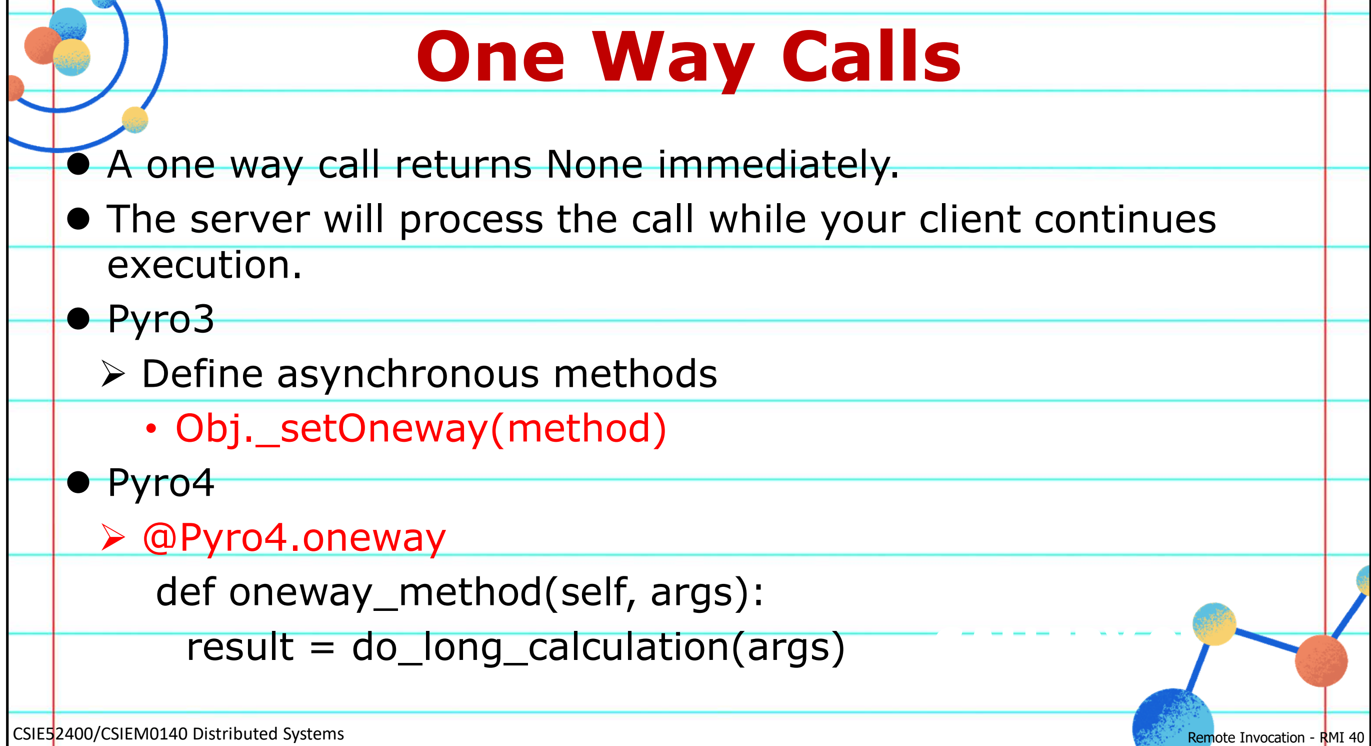

Object Location (Pyro3)
● Server
➢ Register objects
• daemon.useNameServer(ns)
• uri = daemon.connect(obj, "objName")
● Client
➢ Find objects
• URI = ns.resolve('objName')
• remExec = Pyro.core.getAttrProxyForURI(URI)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 35

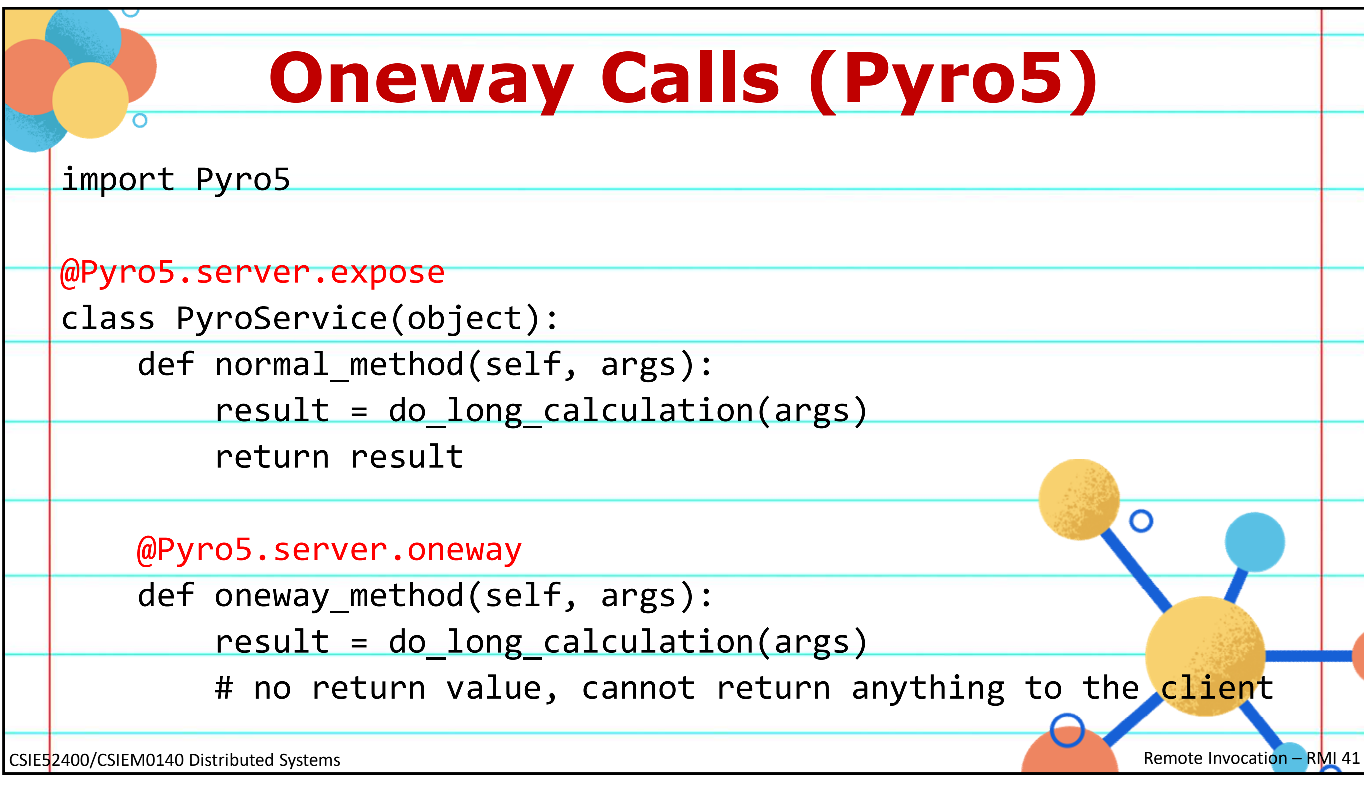

NS Location (Pyro4)
● Server
➢ pyro4-ns
● LAN broadcast
➢ ns = Pyro4.locateNS()
● Explicit location
➢ ns = Pyro4.locateNS(host='hostname', port=7777)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 36

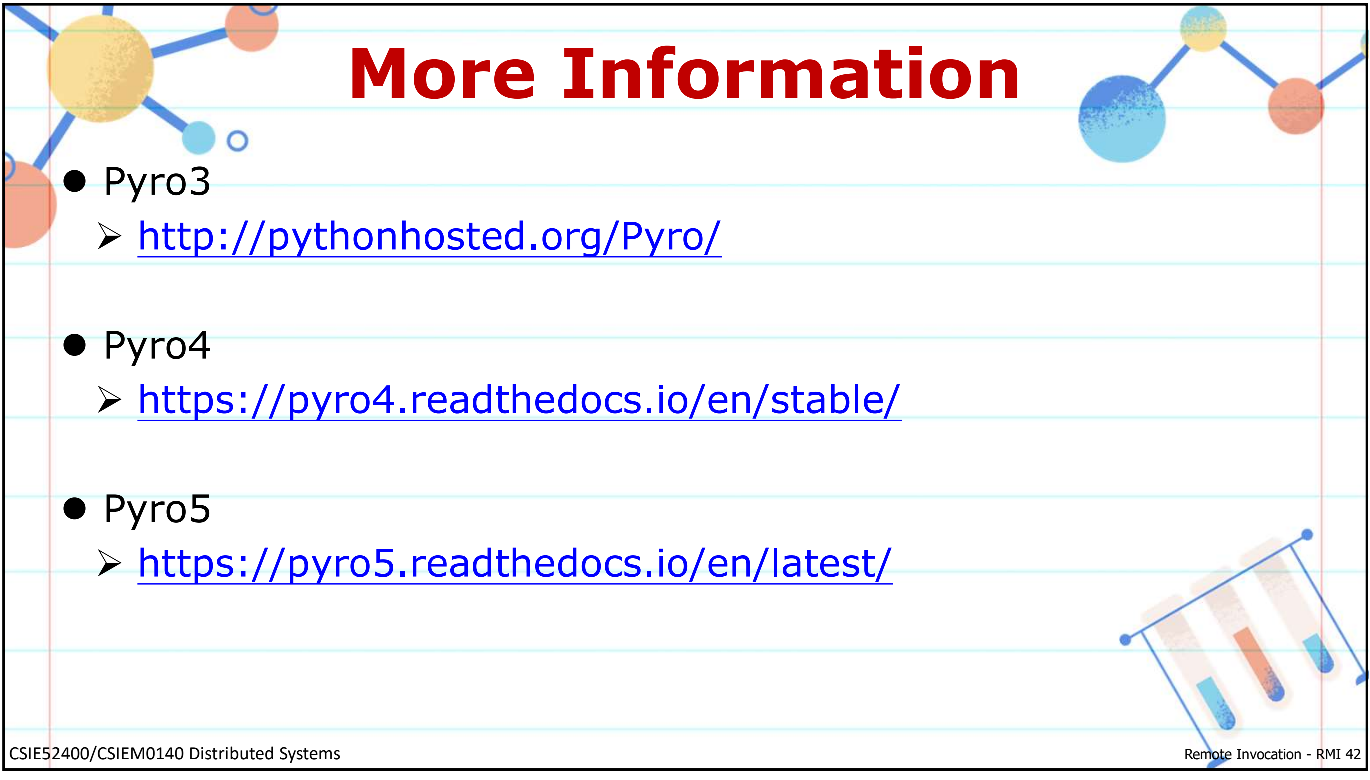

Object Location (Pyro4)
Server
uri = daemon.register(obj, "addServer")
ns.register(objName, uri)
Client
uri = nameserver.lookup(objName)
obj = Pyro4.Proxy(uri)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 37


Pyro5.api.locate_ns
NS Location (Pyro5)
Server
pyro5-ns
Locate name server
ns = Pyro5.api.locate_ns()
ns = Pyro5.core.locate_ns()
Explicit location
ns = Pyro5.api.locate_ns(host='hostname', port=7777)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 38


Object Location (Pyro5)
Server
@Pyro5.api.expose … # explse server class
uri = daemon.register(serverClass)
ns.register(objName, uri)
Client
obj = Pyro5.api.Proxy(objName)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 39


One Way Calls
A one way call returns None immediately.
The server will process the call while your client continues execution.
Pyro3
Define asynchronous methods
Obj._setOneway(method)
Pyro4
@Pyro4.oneway
def oneway_method(self, args):
result = do_long_calculation(args)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 40


Oneway Calls (Pyro5)
import Pyro5
@Pyro5.server.expose
class PyroService(object):
def normal_method(self, args):
result = do_long_calculation(args)
return result
@Pyro5.server.oneway
def oneway_method(self, args):
result = do_long_calculation(args)
# no return value, cannot return anything to the client
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RMI 41

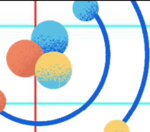

More Information
Pyro3
http://pythonhosted.org/Pyro/
Pyro4
https://pyro4.readthedocs.io/en/stable/
Pyro5
https://pyro5.readthedocs.io/en/latest/
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 42

Java Distributed Object Model

- Java has built in support for distributed objects using the Java RMI.
- Strictly speaking, only remote objects are supported (i.e. an object's state always resides on a single machine, but whose interfaces can be made available to remote processes).
- All related classes are in the `java.rmi` package.

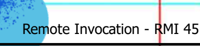
Java RMI

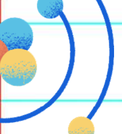
- Local and remote objects are almost the same at the language level.
- All serializable data (i.e. can be marshaled) can be passed as a parameter to an RMI.
- Local objects are passed by value whereas remote objects are passed by reference.
- A remote object is built from two different classes.
  - Server class. Contain the object's state and methods and the skeleton generated from interface specifications.
  - Client class. Contain the client code and the proxy also generated from interface specification.



Participating Processes

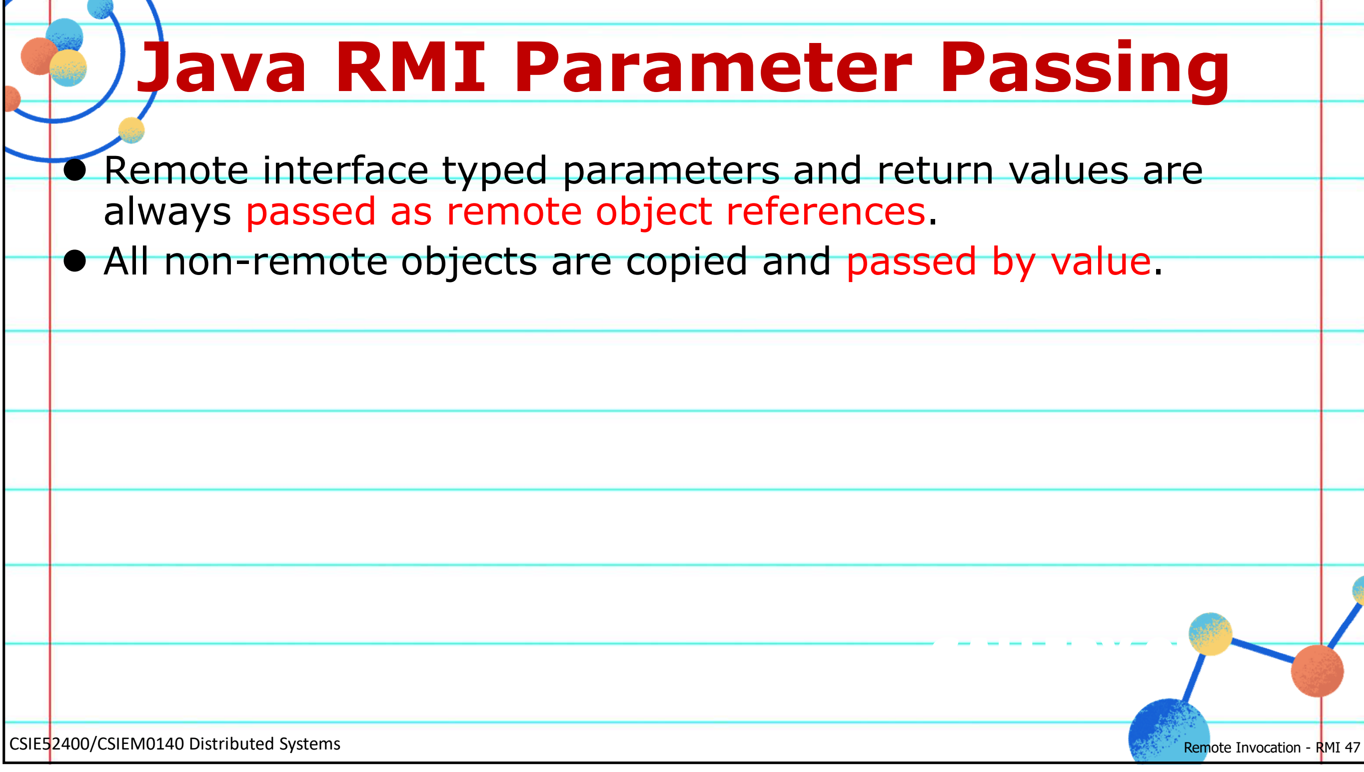
- **Client**
  - Process that is invoking a method on a remote object
- **Server**
  - Process that owns the remote object
  - To the server, this is a local object
- **Object Registry** (rmiregistry)
  - Name server that associates objects with names
  - A server registers an object with rmiregistry
  - URL namespace
    - `rmi://hostname:port/pathname`
    - e.g.: `rmi://crapper.pk.org:12345/MyServer`

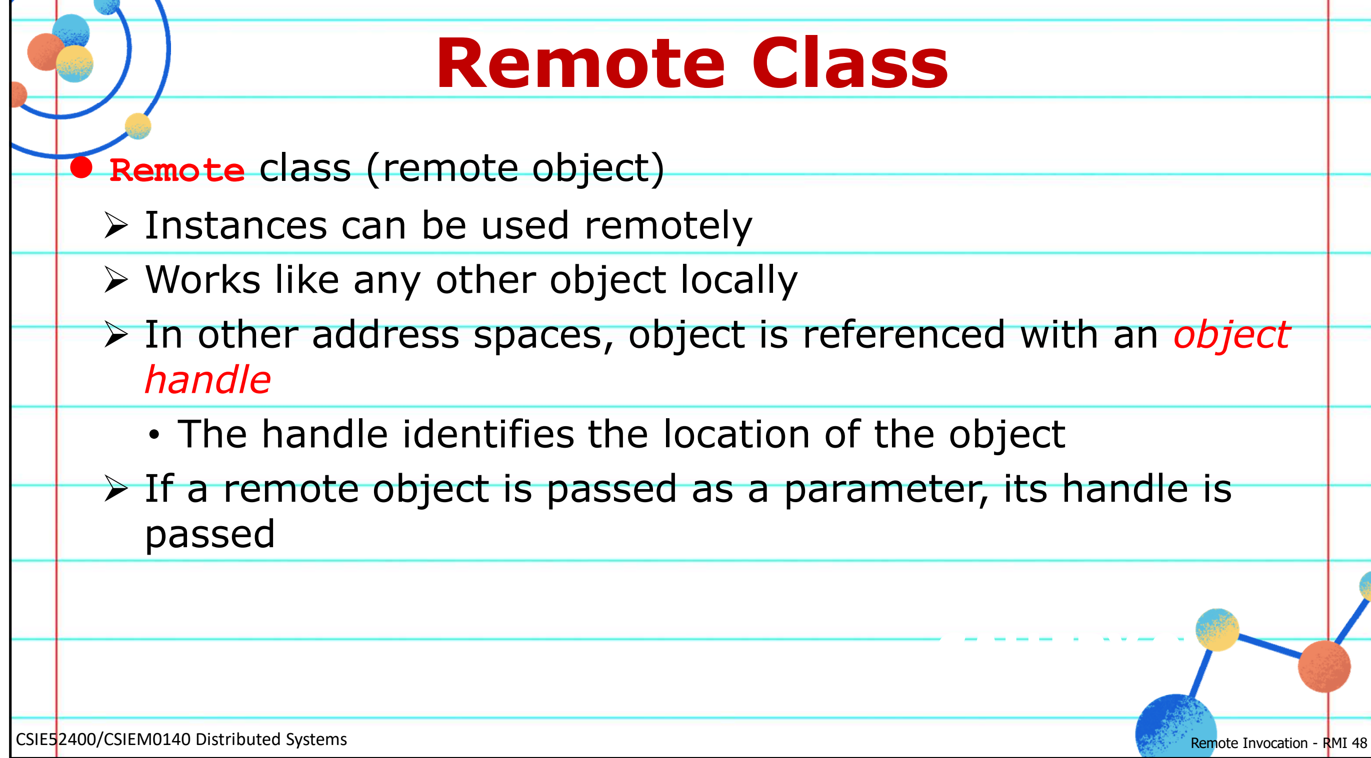


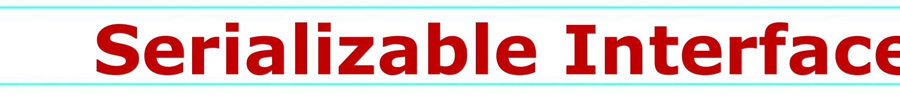


Java RMI Interface

- User defined remote interfaces must extend the `Remote` interface (in `java.rmi`).
- Remote interface methods must throw `RemoteException`.
- Any serializable object (i.e. implements the `Serializable` interface) can be passed as an argument or result.
- The `UnicastRemoteObject`: used to export a remote object reference or obtain a stub for a remote object
- `Naming`: methods to interact with the registry


Java RMI Parameter Passing
Remote interface typed parameters and return values are always passed as remote object references.
All non-remote objects are copied and passed by value.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 47


Remote Class
Remote class (remote object)
Instances can be used remotely
Works like any other object locally
In other address spaces, object is referenced with an object handle
The handle identifies the location of the object
If a remote object is passed as a parameter, its handle is passed
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 48

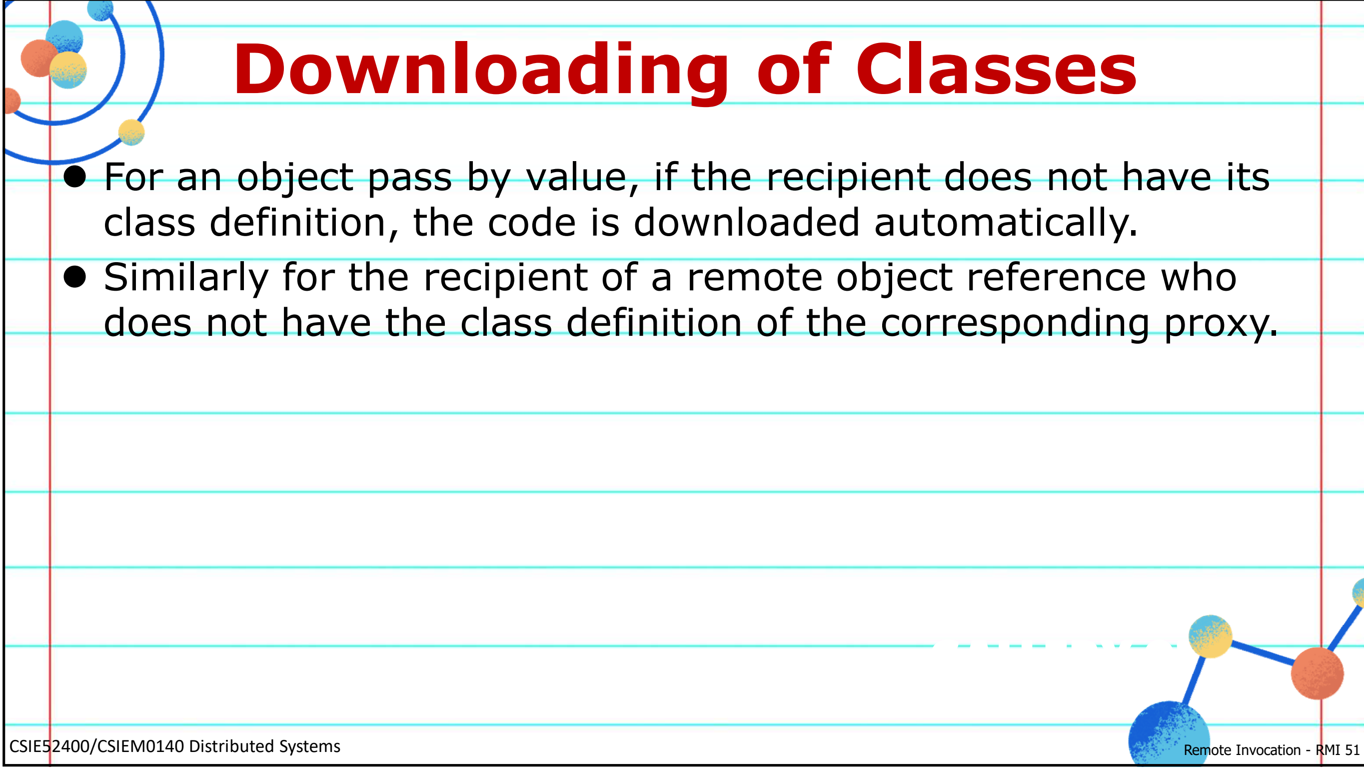


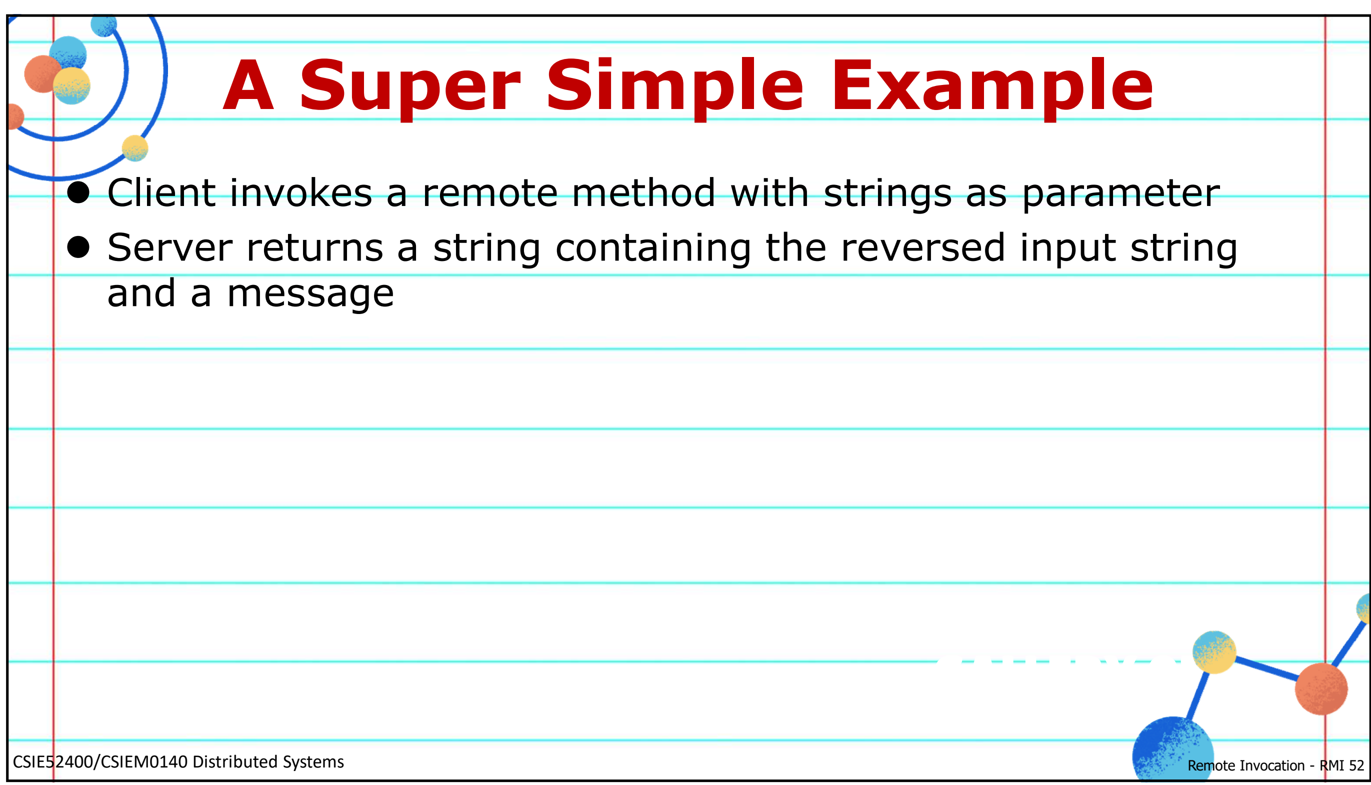
Serializable Interface

- `java.io.Serializable` interface (serializable object)
  - Allows an object to be represented as a sequence of bytes (marshaled)
  - Allows instances of objects to be copied between address spaces
    - Can be passed as a parameter or be a return value to a remote object
    - Value of object is copied (pass by value)
  - Any objects that may be passed as parameters should be defined to implement the `java.io.Serializable` interface
    - Good news: you rarely need to implement anything
    - All core Java types already implement the interface
    - For your classes, the interface will serialize each variable iteratively

Remote Classes

- Classes that will be accessed remotely have 2 parts:
  1. interface definition
  2. class definition
- Remote interface
  - This will be the basis for the creation of stub functions
  - Must be public
  - Must extend `java.rmi.Remote`
  - Every method in the interface must declare that it throws `java.rmi.RemoteException`
- **Remote class**
  - implements `Remote` interface
  - extends `java.rmi.server.UnicastRemoteObject`


Downloading of Classes
For an object pass by value, if the recipient does not have its class definition, the code is downloaded automatically.
Similarly for the recipient of a remote object reference who does not have the class definition of the corresponding proxy.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 51

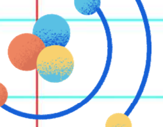

A Super Simple Example
Client invokes a remote method with strings as parameter
Server returns a string containing the reversed input string and a message
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 52

Define the remote interface

SampleInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface SampleInterface extends Remote {
  public String invert(String msg) throws RemoteException;
}
```

- Interface is public
- Extends the Remote interface
- Defines methods that will be accessed remotely
  - We have just one method here: *invert*
- Each method must throw a RemoteException
  - In case things go wrong in the remote method invocation



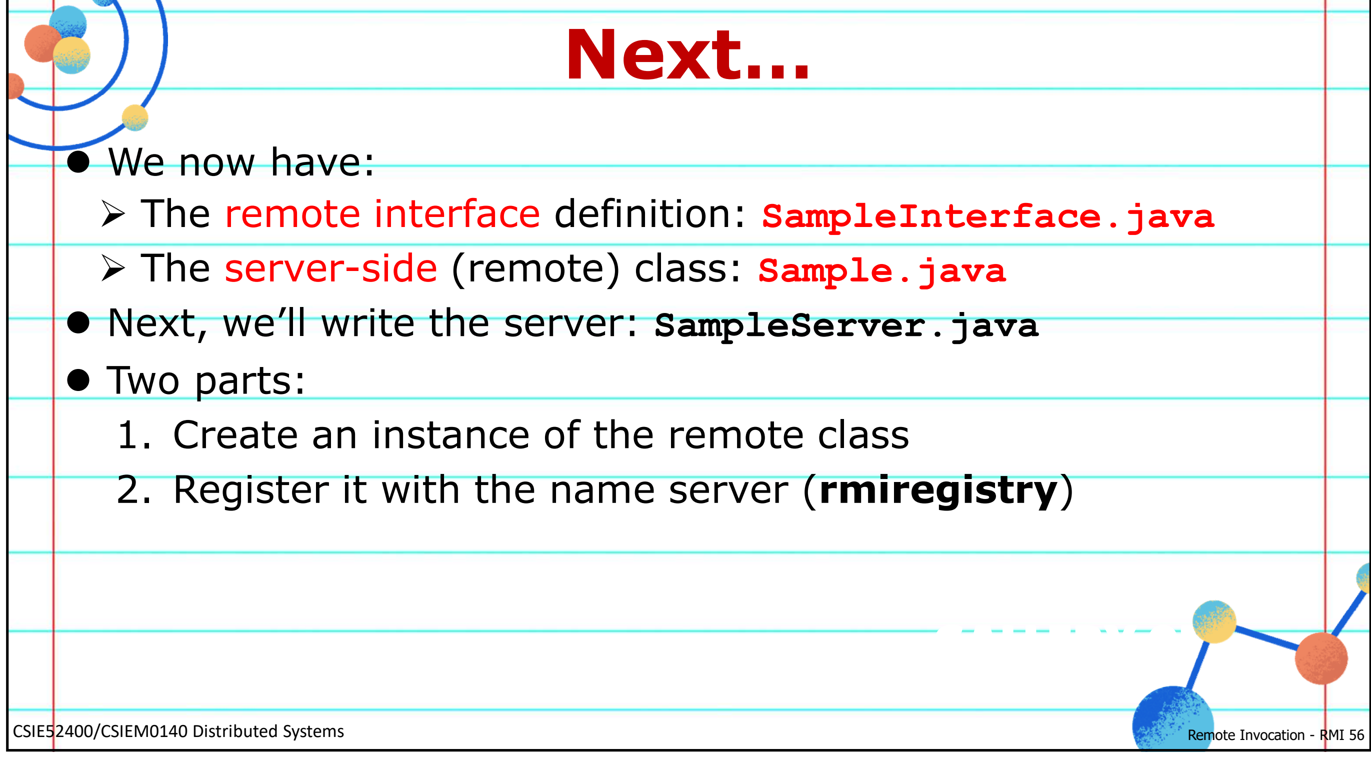
Define the remote class

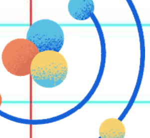
- Defines the implementation of the remote methods
- It implements the interface we defined
- It extends the `java.rmi.server.UnicastRemoteObject` class
  - Defines a unicast remote object whose references are valid only while the server process is alive.

Sample.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.*;

public class Sample extends UnicastRemoteObject
                   implements SampleInterface {
  public Sample() throws RemoteException { }
  public String invert(String m) throws RemoteException {
    // return input message with characters reversed
    return new StringBuffer(m).reverse().toString();
  }
}
```



Java RMIregistry

- The binder for Java RMI.
- Maintains a table mapping textual names to remote object references.
  //computerName:port/objectName
- Accessed by methods of the **`Naming`** class.
- Can also use the **`LocateRegistry`** class to get a **`Registry`** object and then use methods similar to the **`Naming`** class.(See the API doc)
- Clients must direct their lookup enquires to particular hosts.

The *Naming* Class of Java RMIregistry

void rebind (String name, Remote obj)
Used by a server to register the identifier of remote object by name.
void bind (String name, Remote obj)
Used by a server to register a remote object by name, but if the name is already bound, an exception is thrown.
void unbind (String name)
This method removes a binding.
Remote lookup (String name)
Used by clients to look up a remote object by name. A remote object reference is returned.
String[] list(String name)
This method returns an array of Strings containing the names bound in the registry.

Server Code (SampleServer.java)

- Create the object
  `new Sample()`
- Register it with the name server (rmiregisty)
  `Naming.rebind("Sample”, new Sample())`
- *rmiregistry* runs on the server
  - The default port is 1099
  - The name is a URL format and can be prefixed with a hostname and port: `“//localhost:1099/Server”`

SampleServer.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class SampleServer {
  public static void main(String args[]) {
    if (args.length != 1) {
      System.err.println("usage: java SampleServer
rmi_port");
      System.exit(1);
    }
```

SampleServer.java

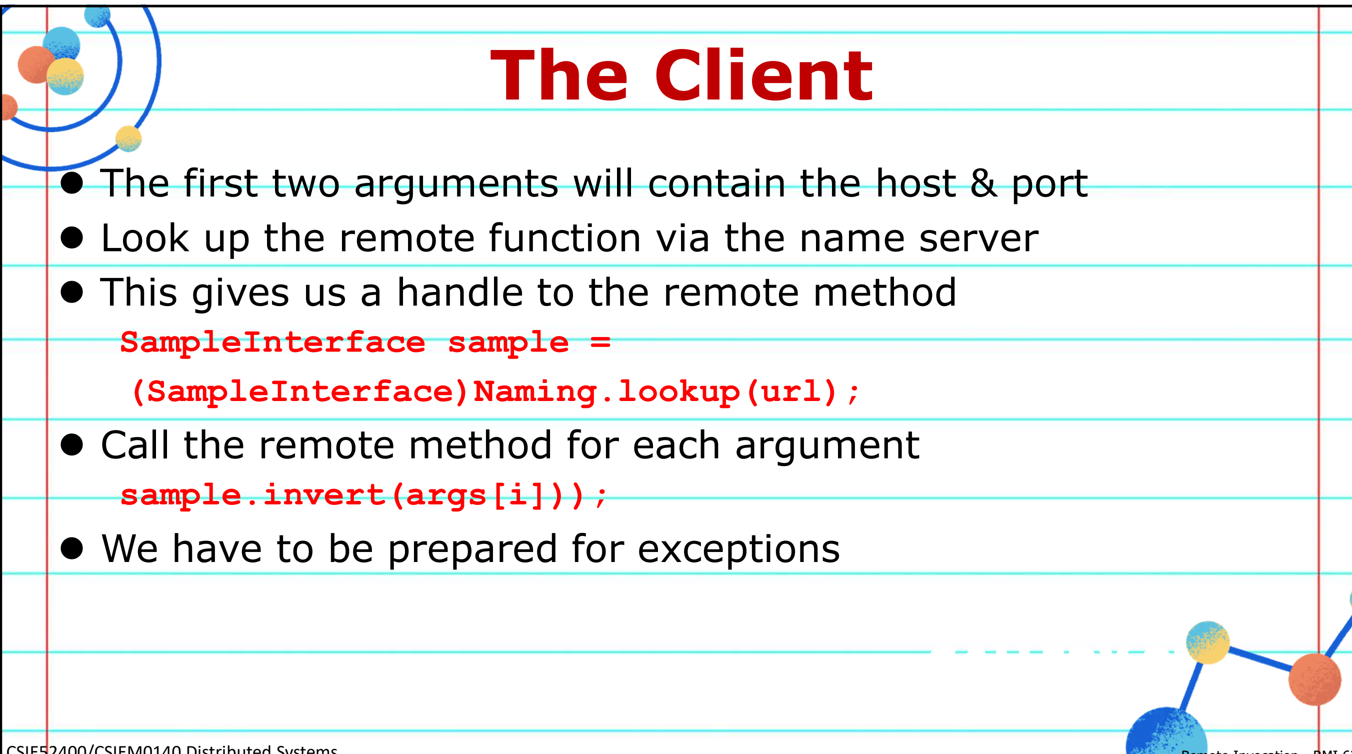
```
    try {
      // first command-line arg: the port of the rmiregistry
      int port = Integer.parseInt(args[0]);

      // create the URL to contact the rmiregistry
      String url = "//localhost:" + port + "/Sample";
      System.out.println("binding " + url);
      // register it with rmiregistry
      Naming.rebind(url, new Sample());
      // Naming.rebind("Sample", new Sample());
      System.out.println("server " + url + " is running...");
    }
    catch (Exception e) {
      System.out.println("Sample server failed:" + e.getMessage());
    }
  }
}
```

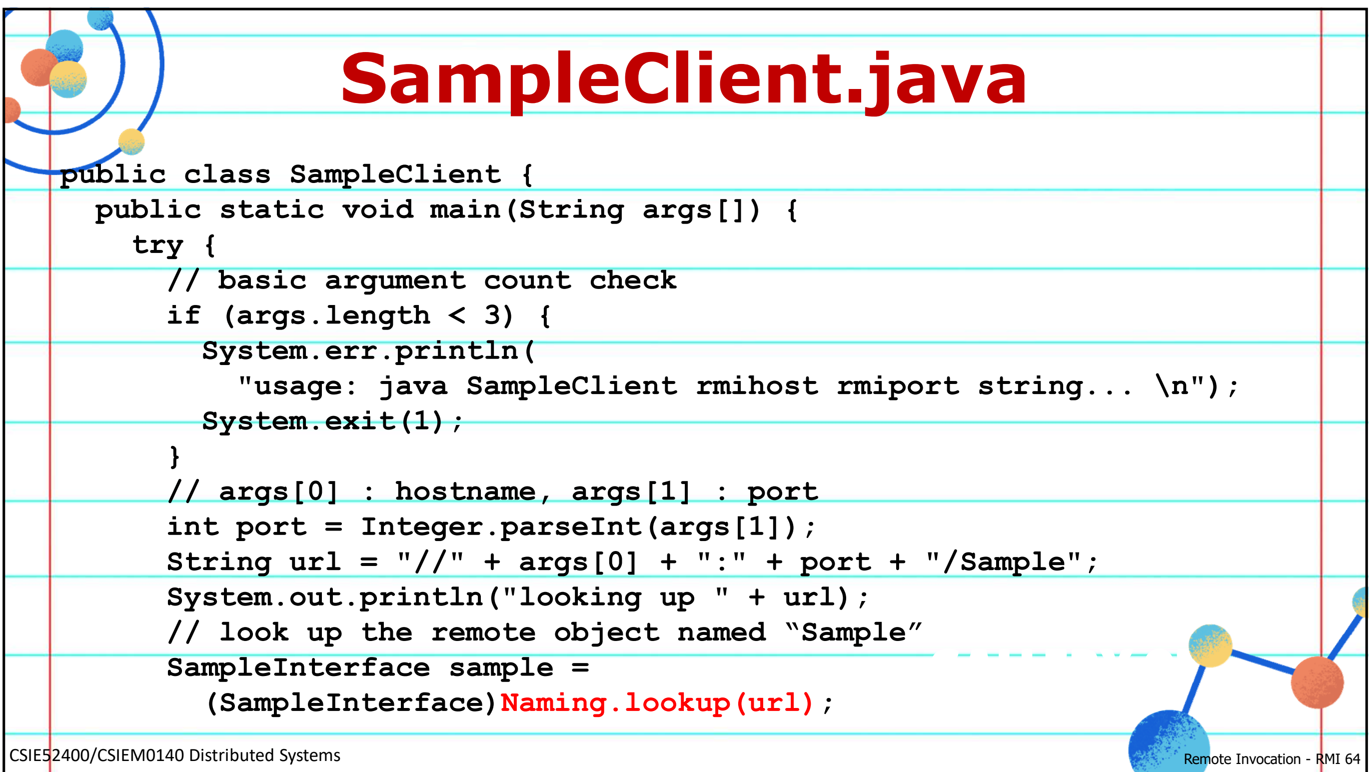

Policy File

- When we run the server, we need to specify security policies
- A security policy file specifies what permissions you grant to the program
- This simple one grants all permissions

```
grant {
  permission java.security.AllPermission;
};
```



The Client
● The first two arguments will contain the host & port
● Look up the remote function via the name server
● This gives us a handle to the remote method
SampleInterface sample =
(SampleInterface)Naming.lookup(url);
● Call the remote method for each argument
sample.invert(args[i]);
● We have to be prepared for exceptions
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 63



SampleClient.java
public class SampleClient {
public static void main(String args[]) {
try {
// basic argument count check
if (args.length < 3) {
System.err.println(
"usage: java SampleClient rmihost rmiport string... \n");
System.exit(1);
}
// args[0] : hostname, args[1] : port
int port = Integer.parseInt(args[1]);
String url = "//" + args[0] + ":" + port + "/Sample";
System.out.println("looking up " + url);
// look up the remote object named "Sample"
SampleInterface sample =
(SampleInterface)Naming.lookup(url);
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 64

SampleClient.java

```
      // args[2]… are the strings to reverse
      for (int i=2; i < args.length; ++i)
        // call remote method and print result
        System.out.println(sample.invert(args[i]));
    } catch(Exception e) {
     System.out.println("SampleClient exception: " + e);
    }
  }
}
```


Compile

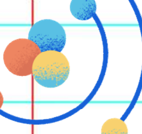
- Compile the interface and classes:

```
javac SampleInterface.java Sample.java
javac SampleServer.java
```

- And the client…

```
javac SampleClient.java
```

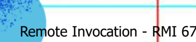
- (you can do it all on one command: **javac *.java**)
- Note – Java used to use a separate RMI compiler
  - Since Java 1.5, Java supports the dynamic generation of stub classes at runtime
  - In the past, one had to use an RMI compiler, *rmic*
  - If you want to, you can still use it but it's not needed

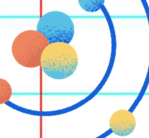


Run

- Start the object registry (in the background):
  `rmiregistry 12345 &`
  - *An argument overrides the default port 1099*
- Start the server (telling it the port of the rmi registry):
  `java -Djava.security.policy=policy SampleServer 12345`
- Run the client:
  `java SampleClient svrname 12345 testing abcdefgh`
  - Where svrname is the name of the server host
  - 12345 is the port number of the name server: rmiregistry, not the service!
- See the output:

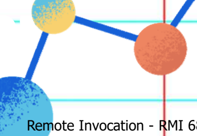
```
gnitset
hgfedcba
```

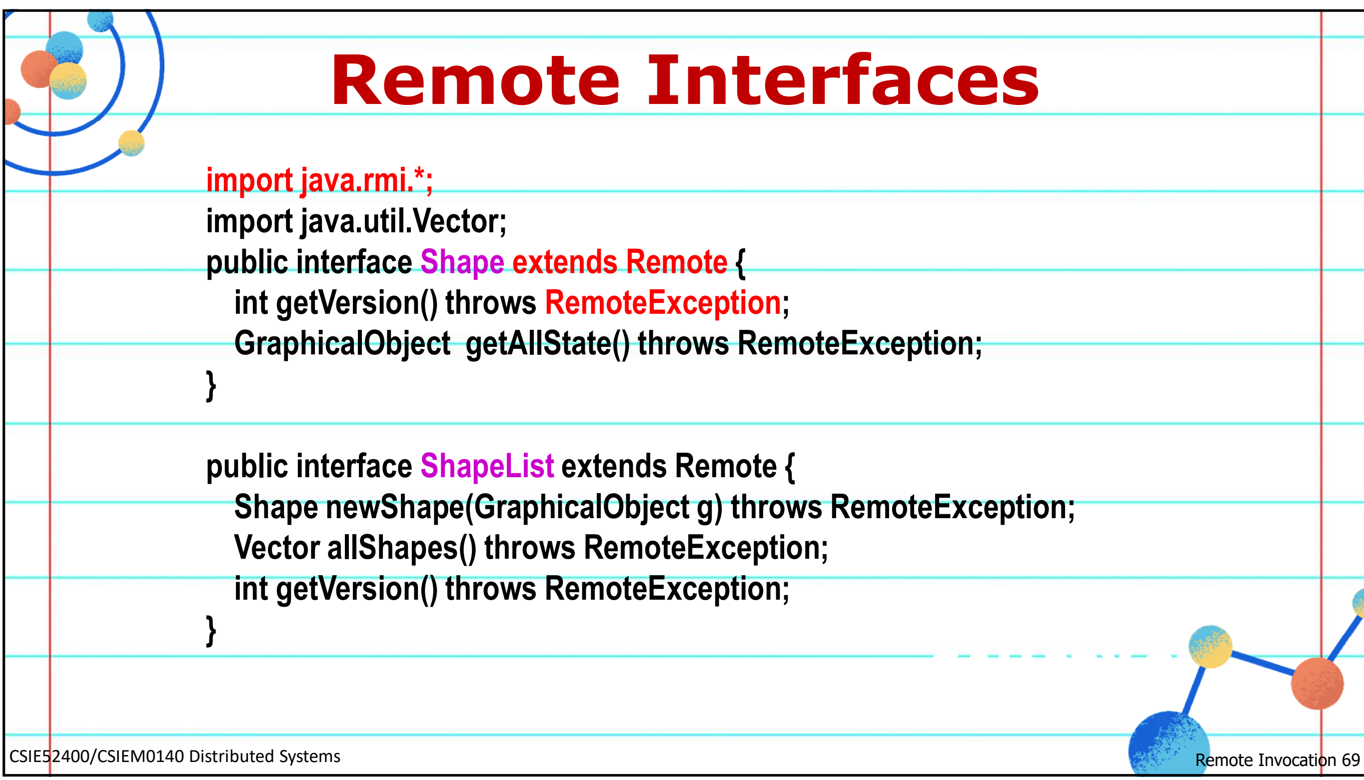


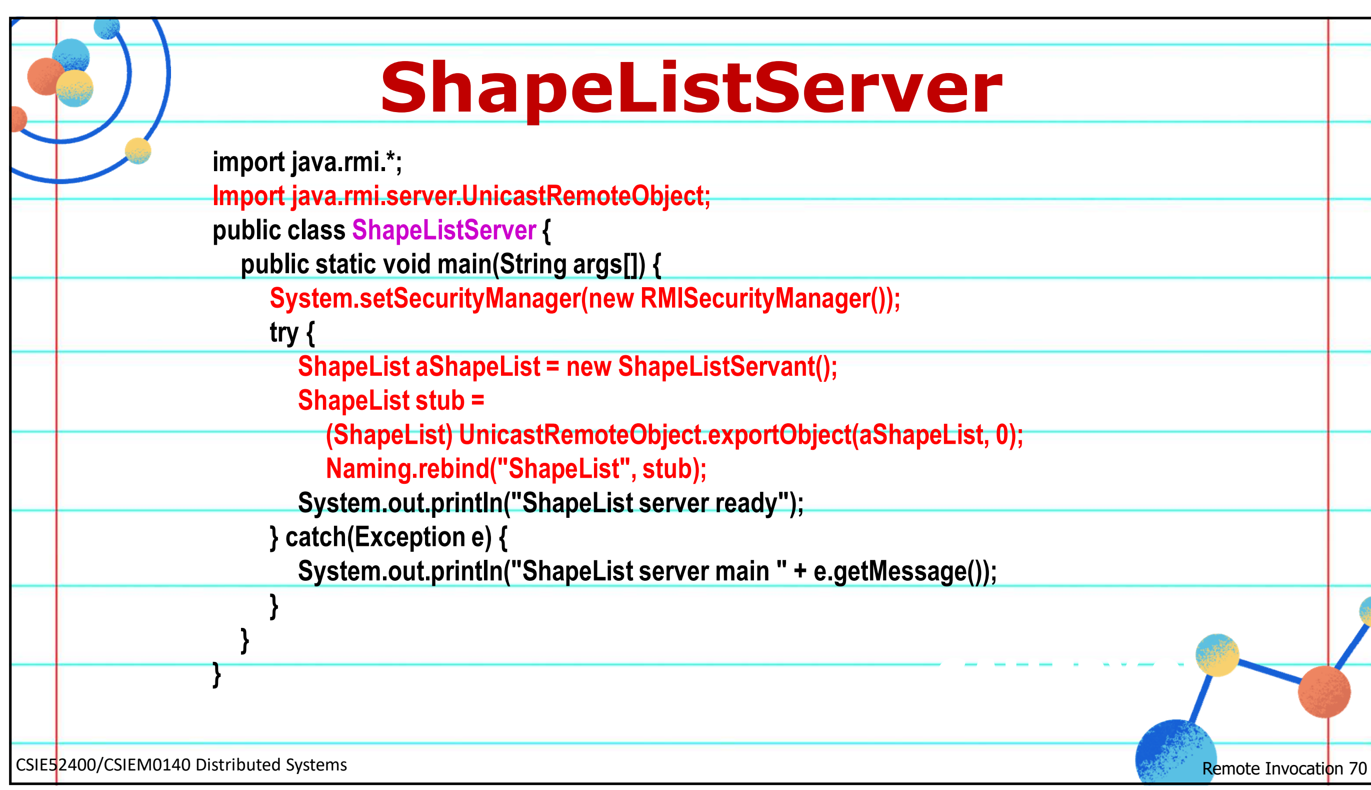


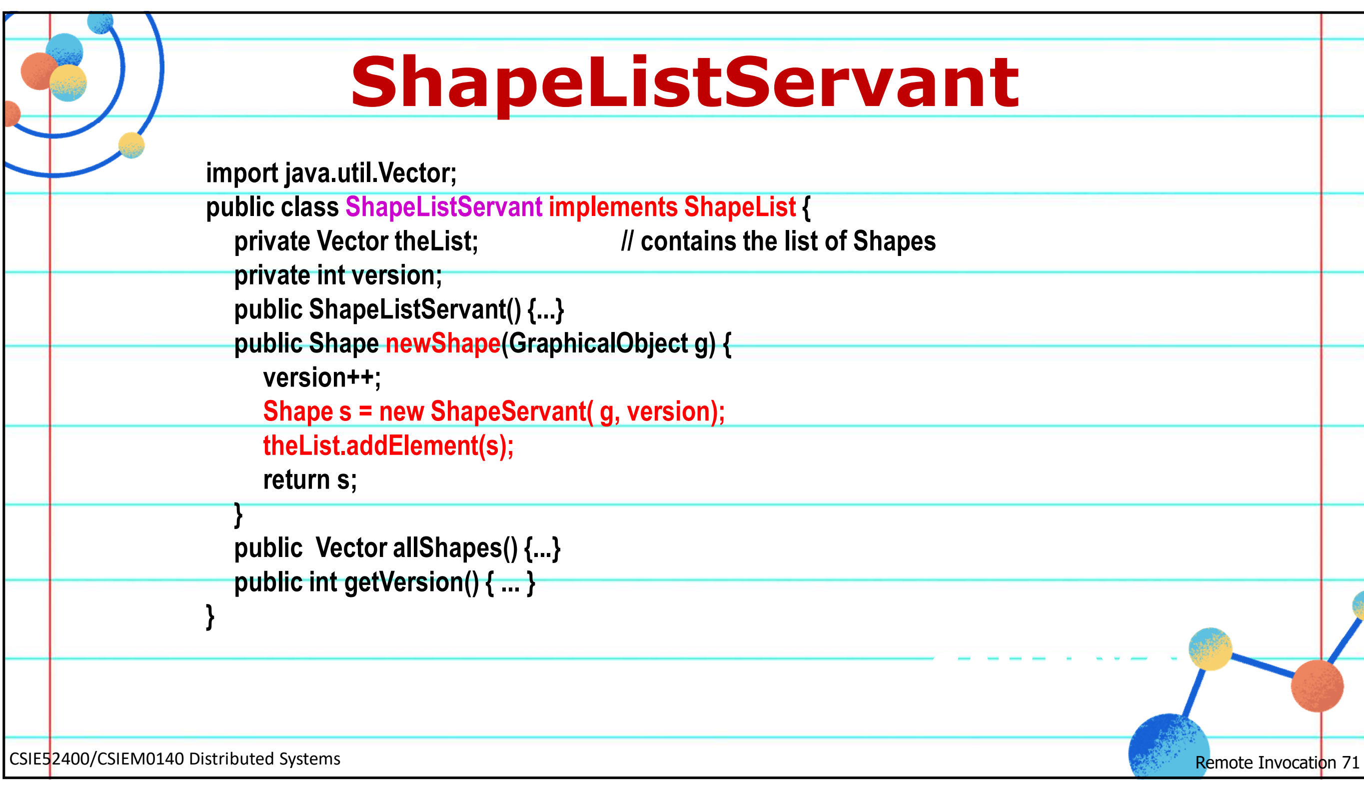
More Complex Case Study

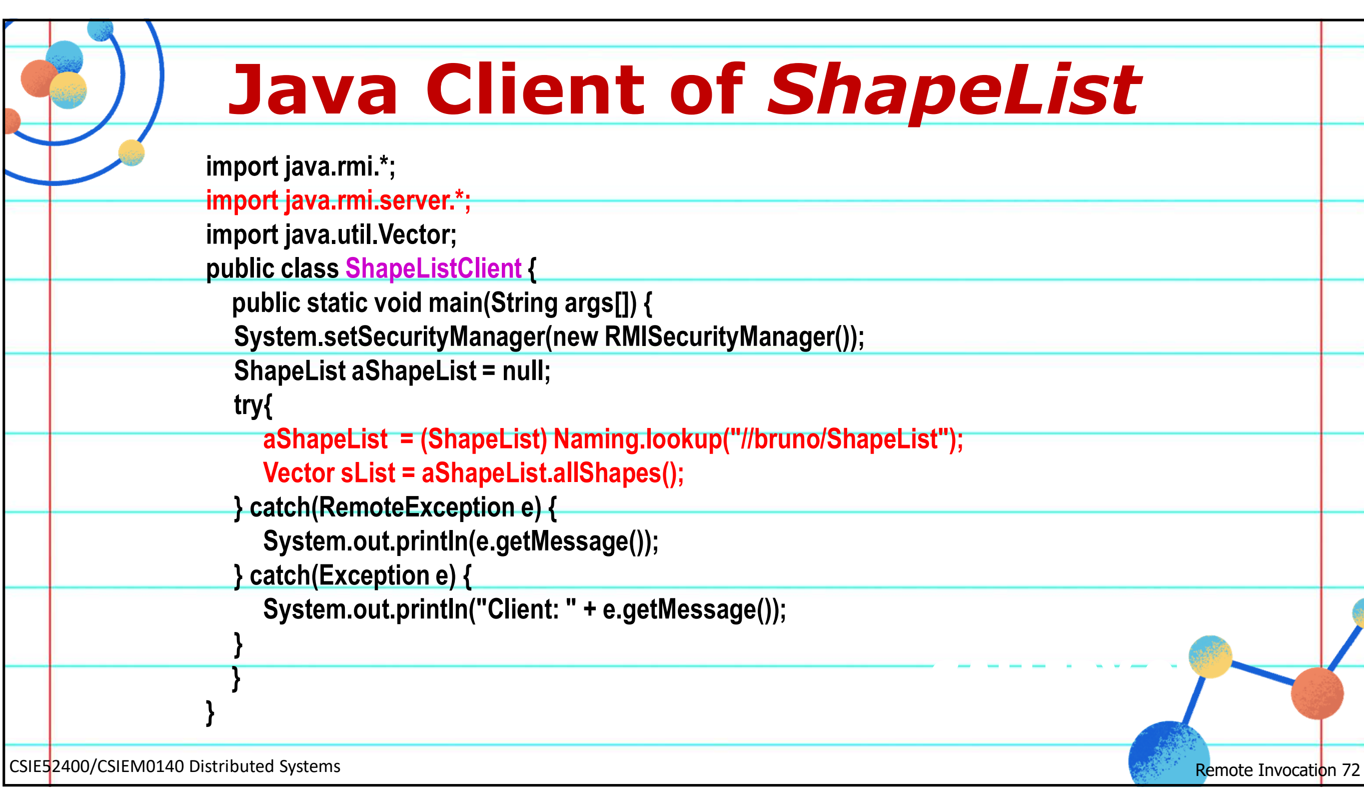
- We will use a shared whiteboard as example.
- A group of users shares a common view of a drawing surface containing graphical objects.
- The server maintains the current state of a drawing:
  - allows clients to submit the latest shapes drawn
  - keeps a record of all the shapes it has received
  - allows clients to retrieve the latest shapes
  - maintains a version number for each shape
  - allows clients to enquire info about version no.

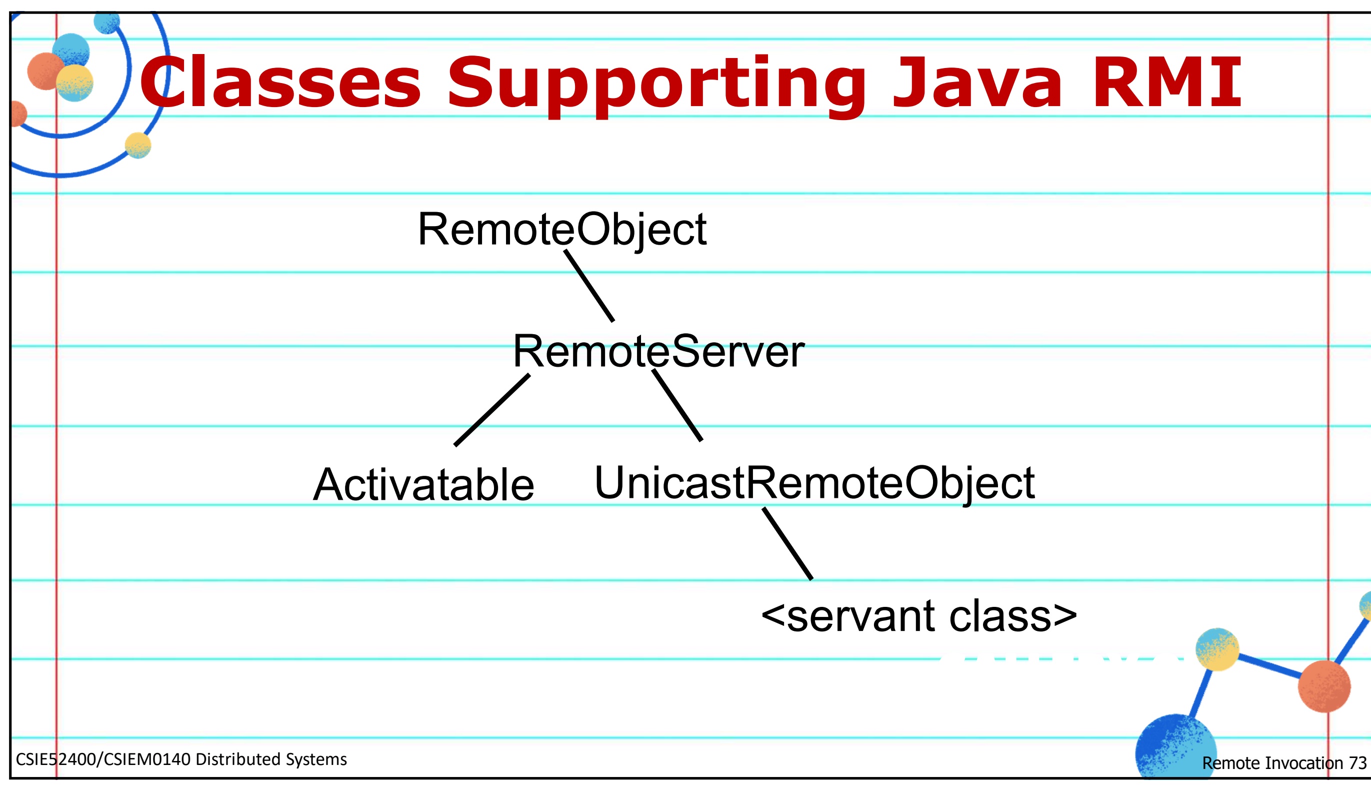


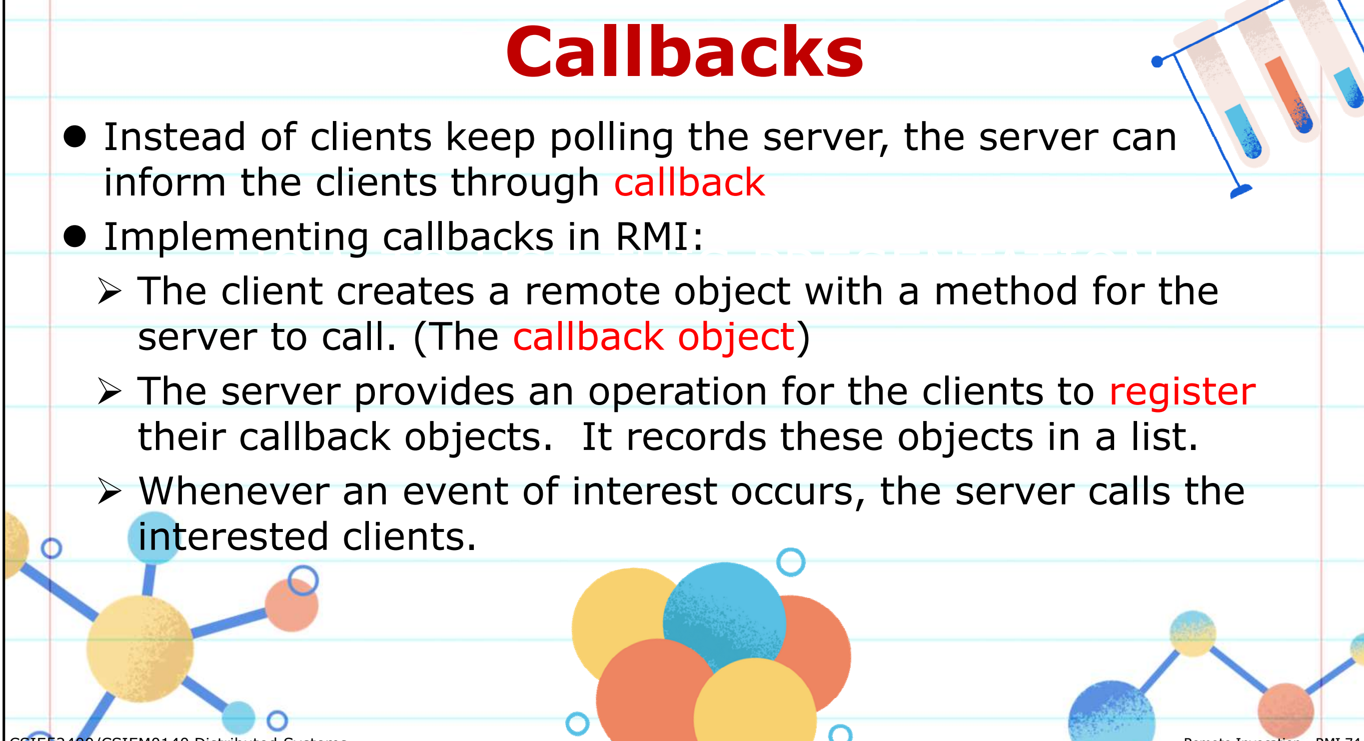

Remote Interfaces
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
int getVersion() throws RemoteException;
GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
Shape newShape(GraphicalObject g) throws RemoteException;
Vector allShapes() throws RemoteException;
int getVersion() throws RemoteException;
}
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 69

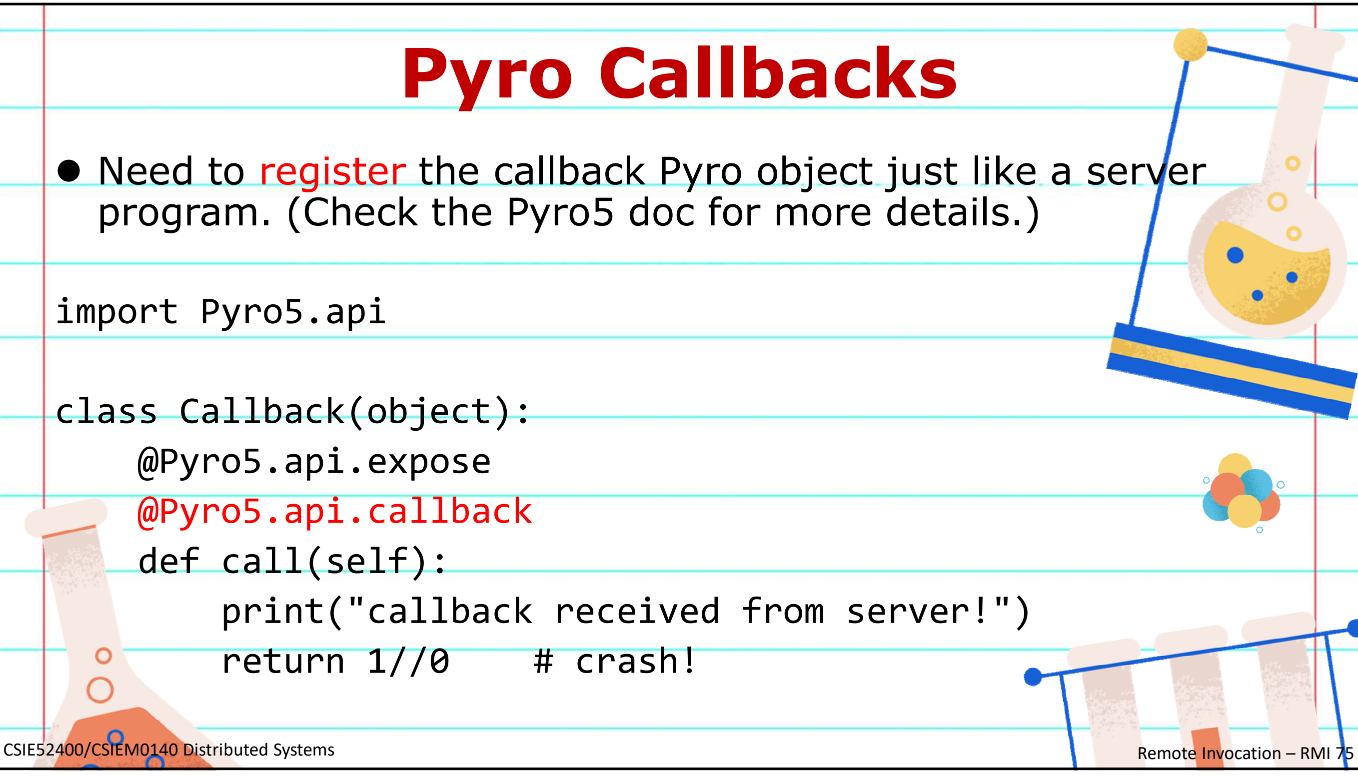

ShapeListServer
import java.rmi.*;
Import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer {
public static void main(String args[]) {
System.setSecurityManager(new RMISecurityManager());
try {
ShapeList aShapeList = new ShapeListServant();
ShapeList stub =
(ShapeList) UnicastRemoteObject.exportObject(aShapeList, 0);
Naming.rebind("ShapeList", stub);
System.out.println("ShapeList server ready");
} catch(Exception e) {
System.out.println("ShapeList server main " + e.getMessage());
}
}
}
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 70


ShapeListServant
import java.util.Vector;
public class ShapeListServant implements ShapeList {
private Vector theList; // contains the list of Shapes
private int version;
public ShapeListServant() {...}
public Shape newShape(GraphicalObject g) {
version++;
Shape s = new ShapeServant(g, version);
theList.addElement(s);
return s;
}
public Vector allShapes() {...}
public int getVersion() { ... }
}
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 71


Java Client of ShapeList
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient {
public static void main(String args[]) {
System.setSecurityManager(new RMISecurityManager());
ShapeList aShapeList = null;
try{
aShapeList = (ShapeList) Naming.lookup("//bruno/ShapeList");
Vector sList = aShapeList.allShapes();
} catch(RemoteException e) {
System.out.println(e.getMessage());
} catch(Exception e) {
System.out.println("Client: " + e.getMessage());
}
}
}
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 72


Classes Supporting Java RMI
RemoteObject
RemoteServer
Activatable
UnicastRemoteObject
<servant class>
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation 73


Callbacks
Instead of clients keep polling the server, the server can inform the clients through callback
Implementing callbacks in RMI:
The client creates a remote object with a method for the server to call. (The callback object)
The server provides an operation for the clients to register their callback objects. It records these objects in a list.
Whenever an event of interest occurs, the server calls the interested clients.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation - RMI 74



Assignment 4: RPC/RMI Exercises

1. Design a SciCalculator function and call it with RPC.
2. Design a SciCalculatorServer class to accept calculation requests from clients with RMI.
3. Both the function and server above should accept requests such as add, sub, mul, div, pow, sqr, log, sin, cos operations.
4. Define/expose remote interface/classes if necessary.
5. Design a SciCalculatorClient class to invoke the remote operations in a loop until exist.

- Due date: 3 weeks