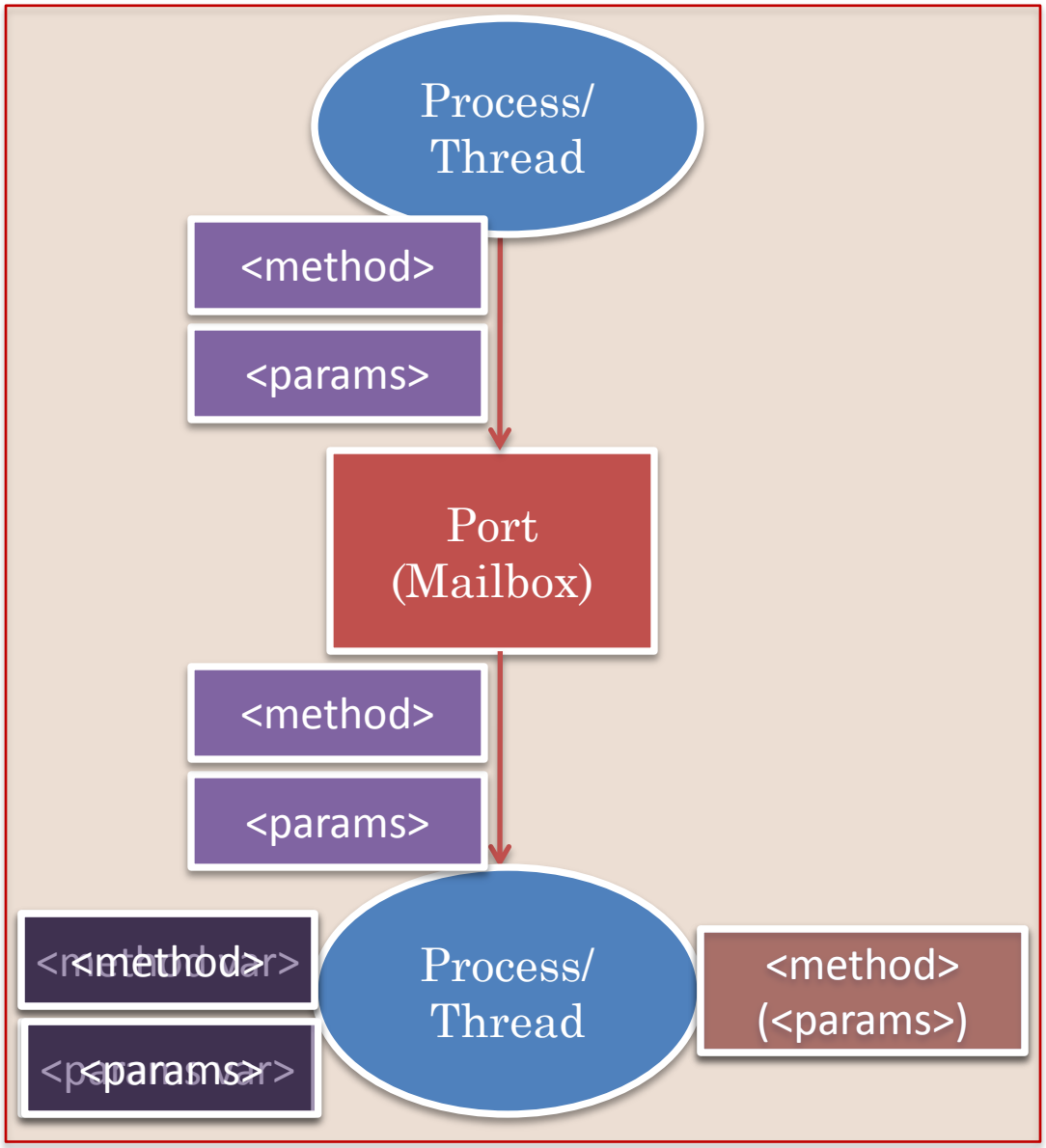
Issues in Design and Implementation of RMI

Instructor: Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu

Simulating Remote Procedure Call



Goal is to call a method with params

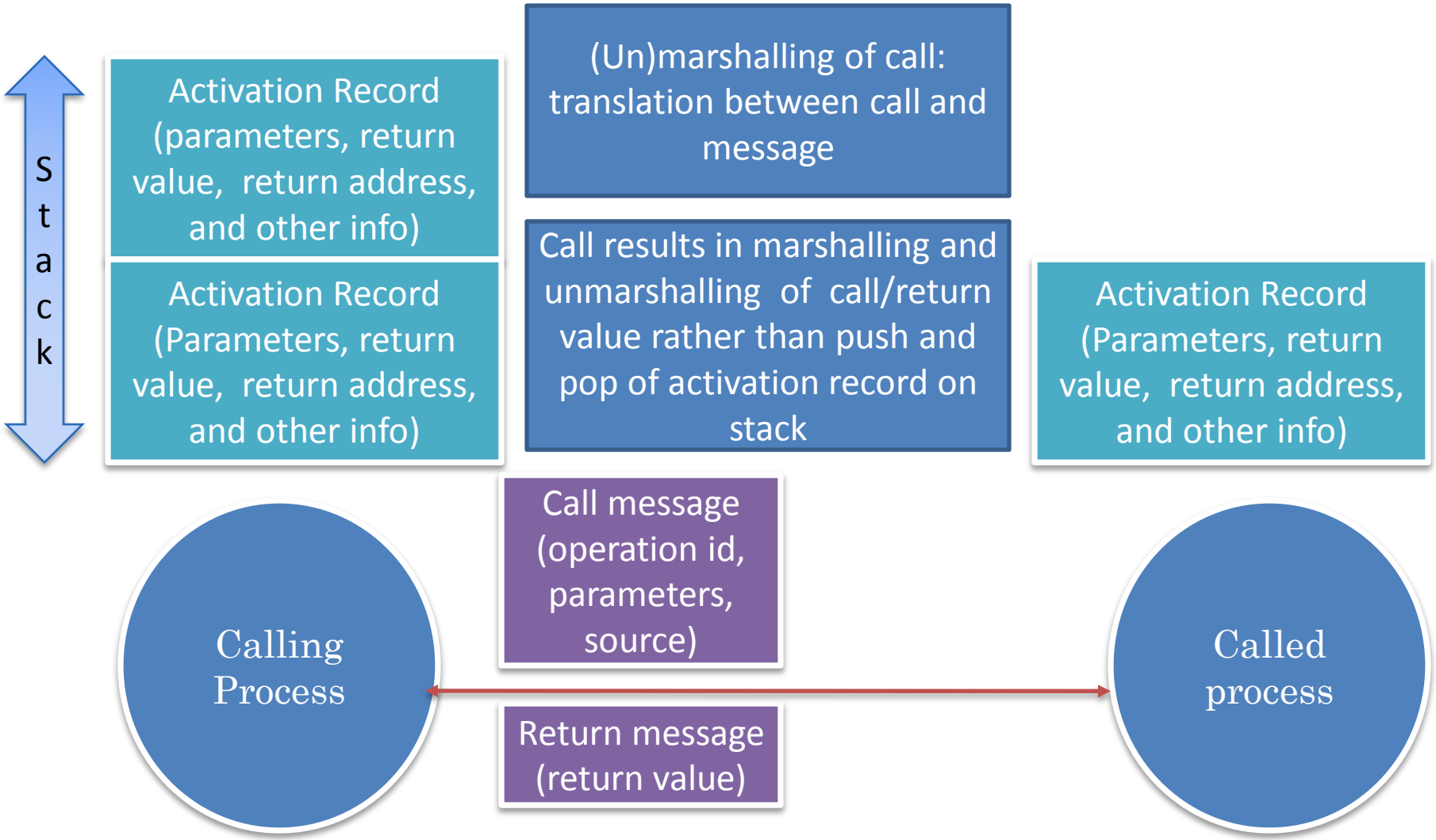
Send expressions encoding method and parameters

These are assigned to corresponding remote variables

Side effect of assignment is to call method with parameters

vs. local call?

Activation Record vs. RPC Messages



Data Set up

Server exports external description of data (RA) port

Statically or dynamically

Server starts and makes data port connectable

Client determines external description

Statically or dynamically

Client starts and establishes connection

RPC Set Up

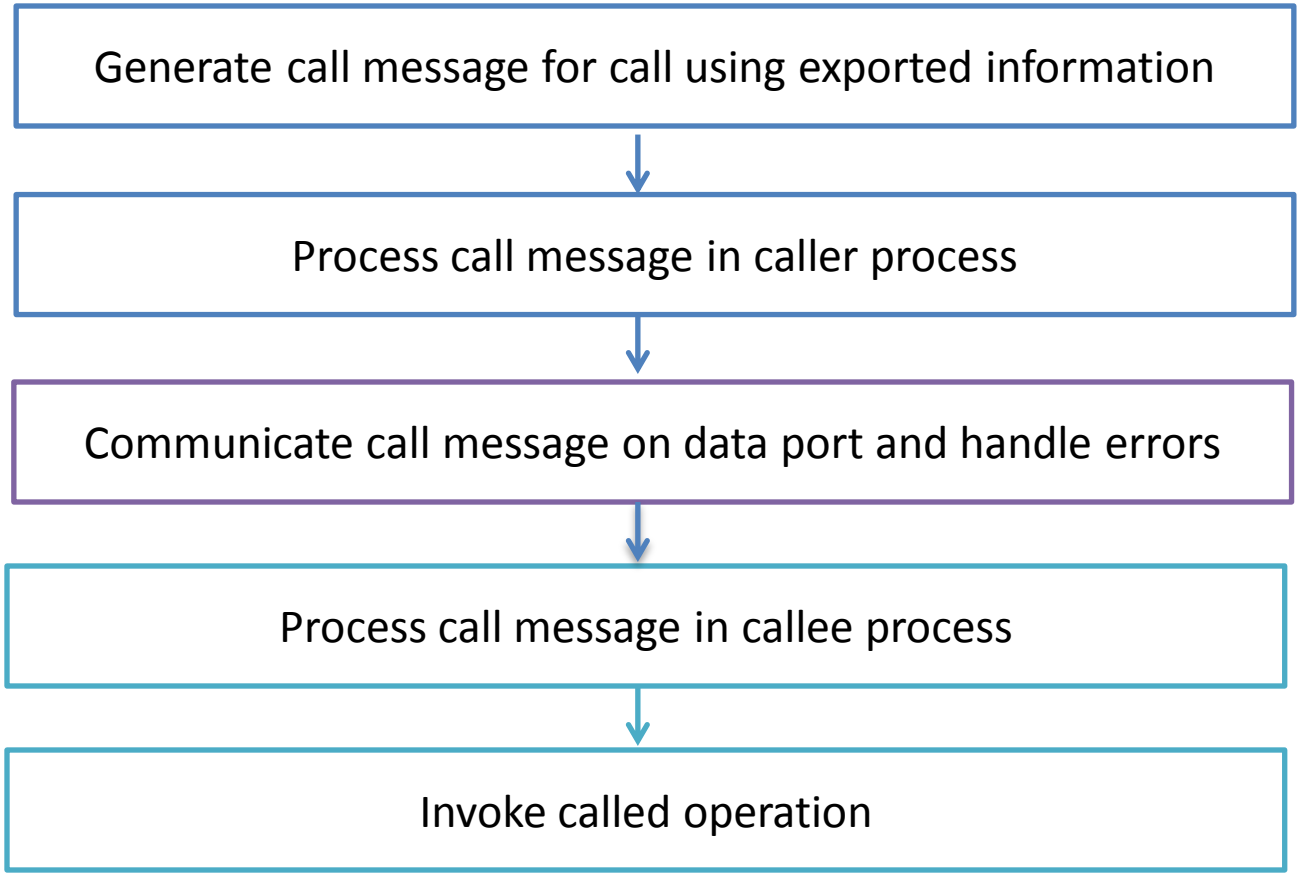
Server exports external description of calls serviced on RA (data) port

Statically or dynamically

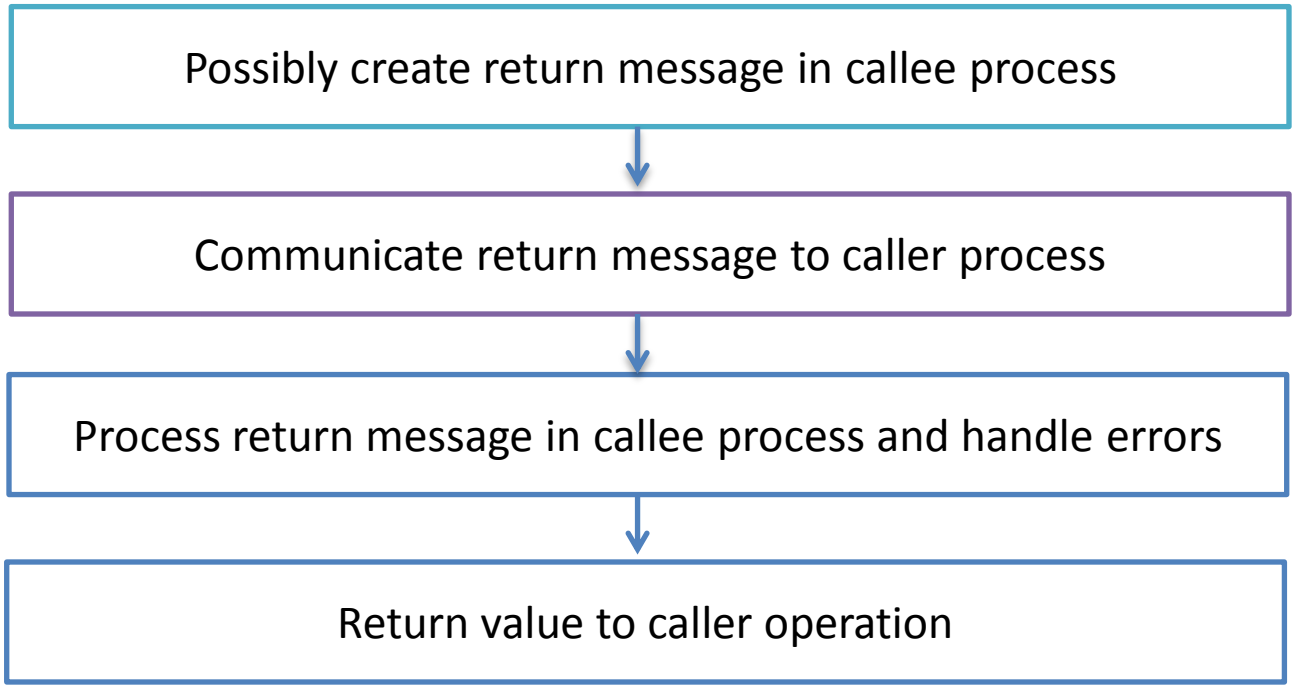
Client finds this information

Statically or dynamically

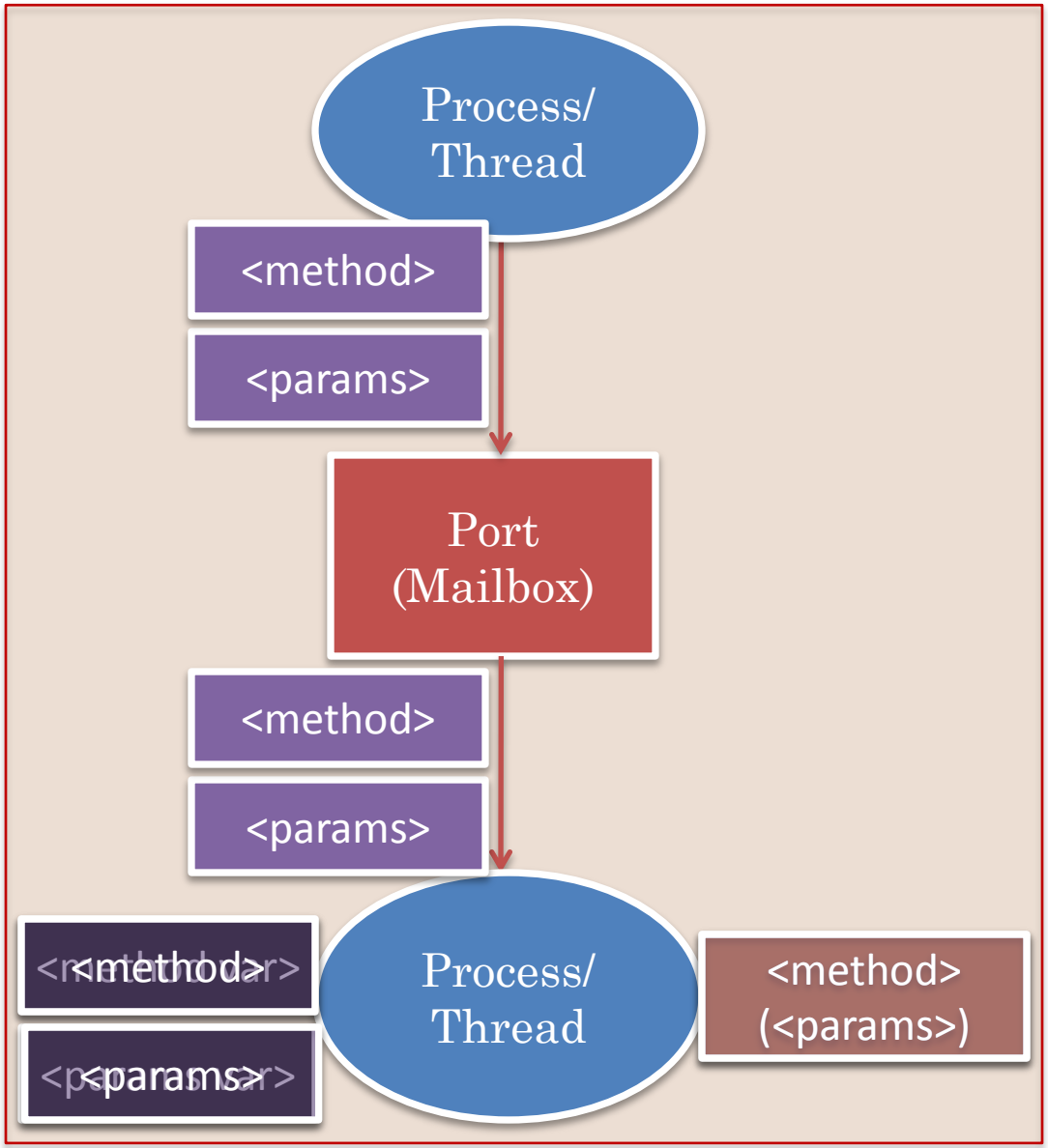
Processing of Call Message and Return Value



Processing of Call Message and Return Value



Automation of Remote Procedure Call



App @ Calling Site

RPC System@Calling Site

Division of labor?

App @Calleee Site

RPC System @Callee Site

RPC Awareness in Application: Two Extremes

RPC clients and servers handle all aspects of RPC and are completely aware of all of the steps listed below

RPC clients and servers are completely unaware of distribution details

Java RMI falls close to complete transparency

GIPC RMI supports a wider spectrum that falls closer and further

Can one have complete transparency? (Answered at the end)

RPC vs. RMI

RPC: O-O or Conventional

RMI: O-O

Main O-O Examples

Java RMI

GIPC RMI

Will use GIPC as an example of detailed implementation

Main O-O Examples (Review)

Java RMI

GIPC RMI

Will use GIPC as an example of detailed implementation

GIPC-RPC vs. RMI : Scope

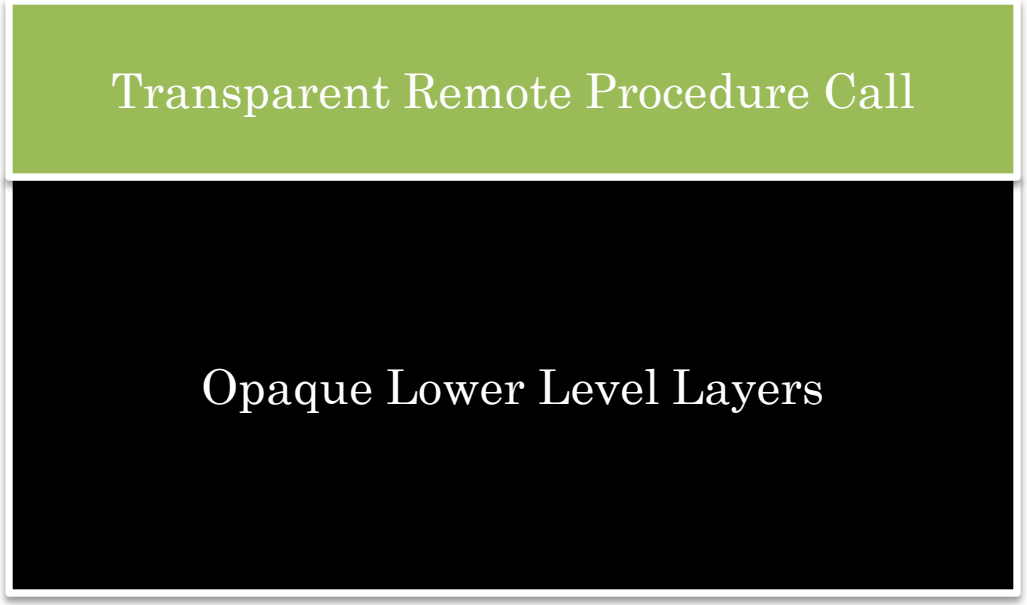
RMI

- Designed for duplex communication
- Has some features allowing its customization

GIPC

- Designed for teaching duplex and group communication
- (Only?) Open source code
- Designed for source-code independent extensibility
- Should work on mobile computers (Android)

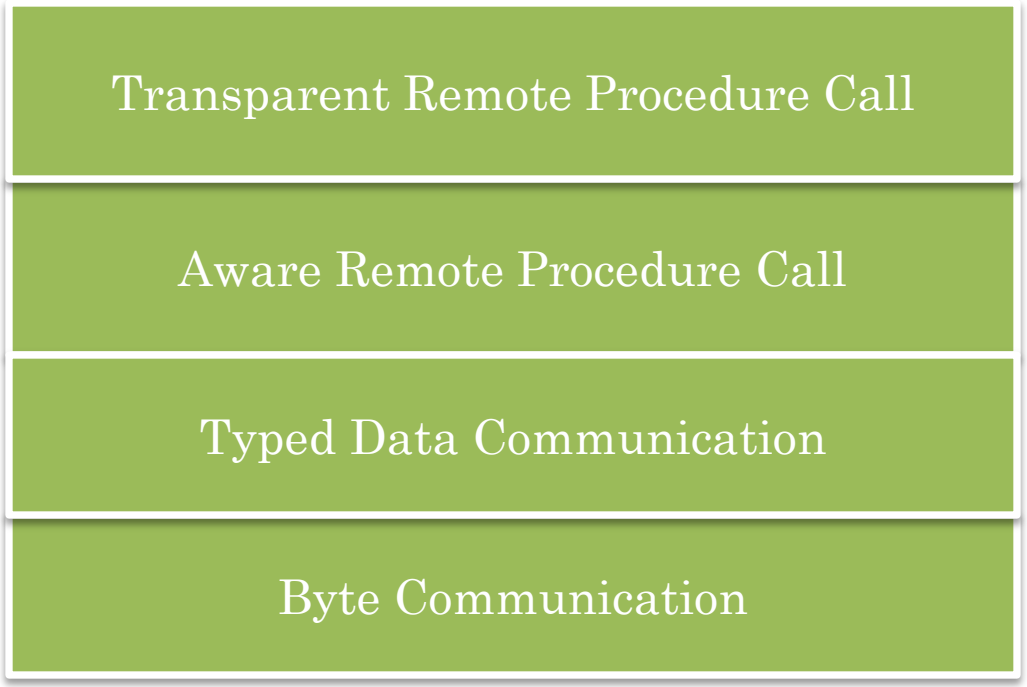
RMI Layer Visibility



Lower level layers API is hidden by the RMI Layer

Though it allows socket to be passed to RMI layer

GIPC Layer Visibility



API of lower level layers (except byte communication) is visible

Designed for customization (for teaching)

GIPC Layer Visibility

Allow remote assignment and remote procedure call on same port

Notion of connected client and server port visible

RMI vs. GIPC Synchrony

RMI

RMI is integrated with local procedure call in that all calls are blocking

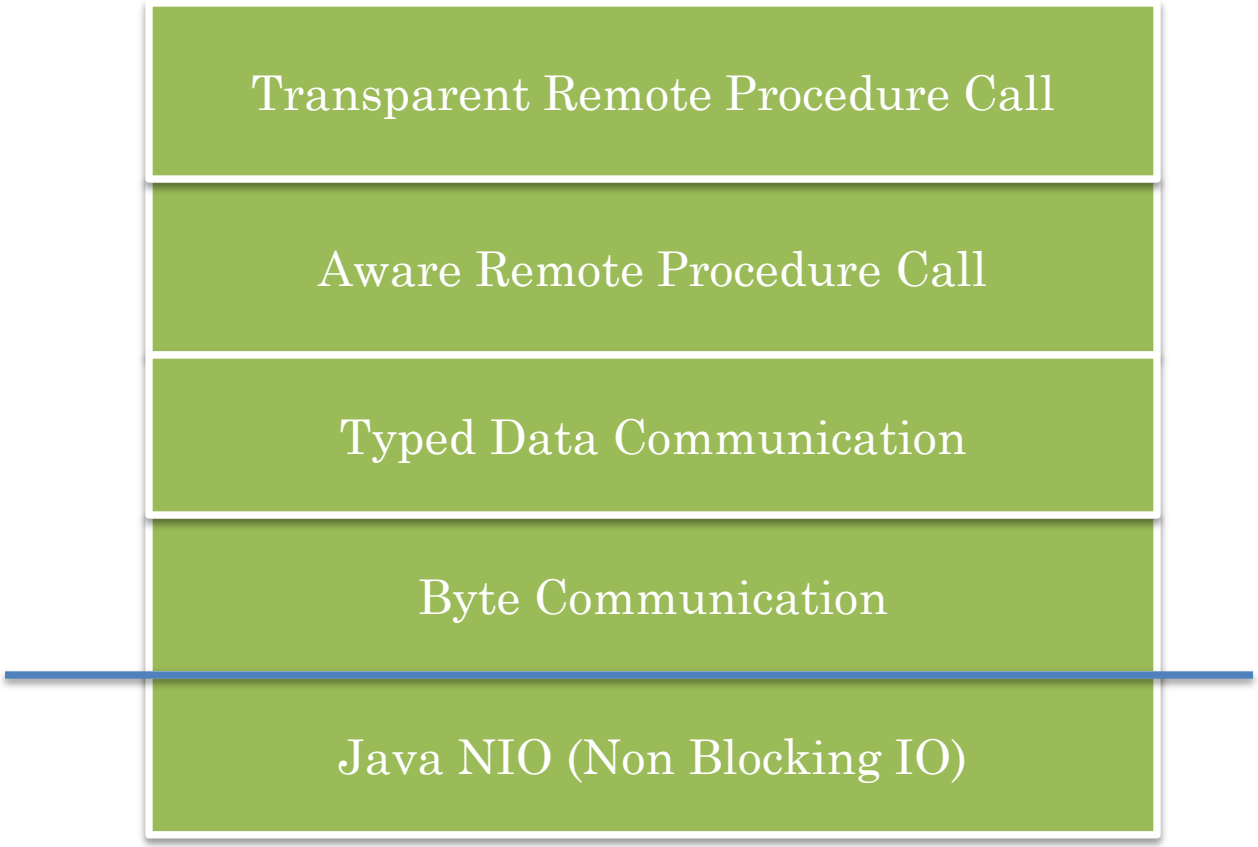
GIPC

An operation does not have to wait unless it has to

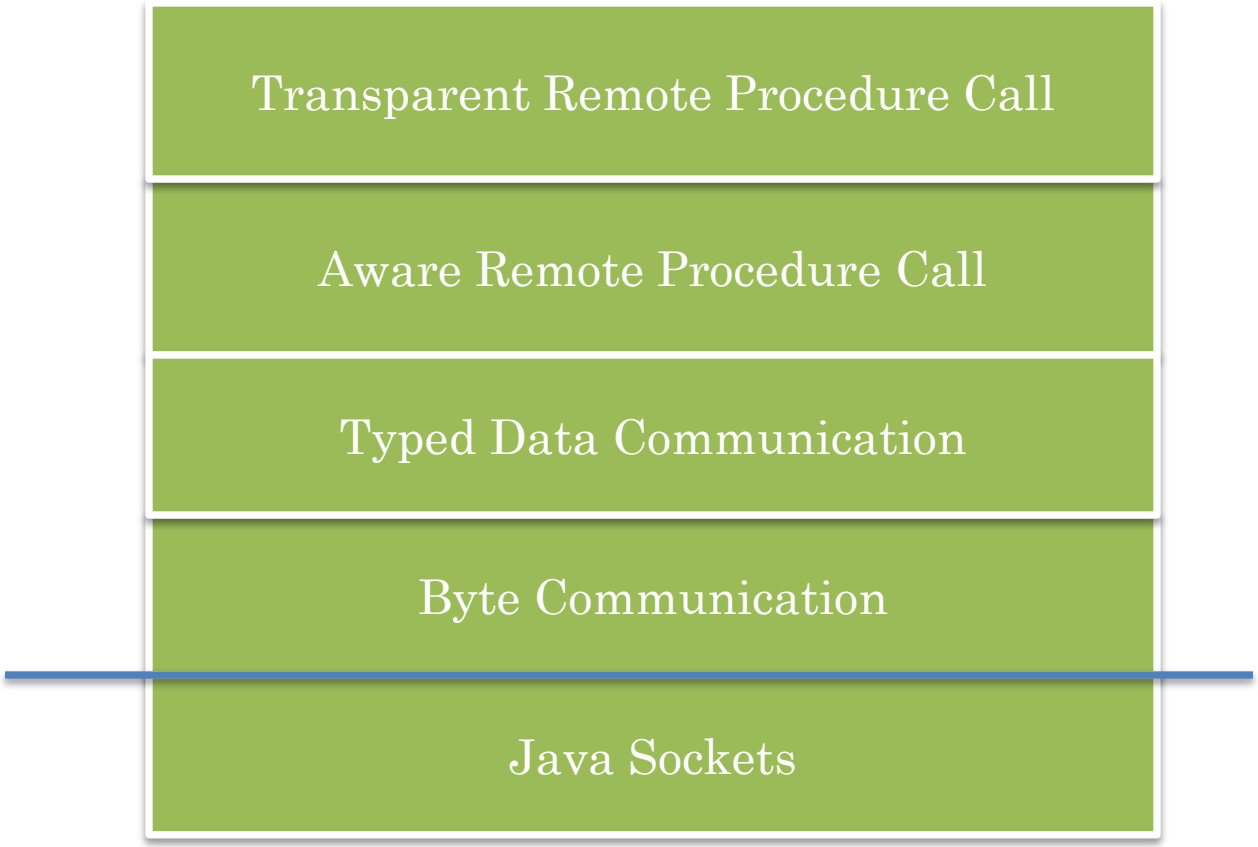
Remote function calls block (depending on configuration)

Remote procedures, data and byte communication may or may not block depending on configuration

Blocking Degree in Default Layer



Replaceable Lowest Layer



Summary: Synchrony

- RMI preserves local procedure call semantics by making the caller wait until the callee finished servicing the request
- GIPC makes the caller wait only if the remote request returns a value
- This is consistent with collaboration systems, which are designed to not block the inputting user
  - The goal is to cure rather than prevent problems caused by not blocking
- It is also consistent with the new Java NIO (non blocking I/O) byte communication layer
- In fact the default implementation of GIPC uses this layer as the underlying **channel** (port provided by some other system to communicate messages on which GIPC ports are built)
  - Extensibility allows this channel layer to be replaced by, for instance, the blocking socket layer

Java RMI: Reusability Error Checking Tradeoff

Caller and callee are distribution-aware because of requirement to handle special remote exception.

Proxies can be generated only for Remote methods and these must acknowledge RemoteException in header

Unnecessary awareness if all the caller does is print stack trace

Cannot reuse existing software without changing it

Java RMI

Cannot call Object methods remotely
e.g. equals()
e.g. toString() (ObjectEditor uses it extensively)

GIPC Solution

GIPC: Programmer can make the tradeoff between reusability and error checking

Proxies can be generated for all methods of an object

The proxy generation call takes as an argument a type which defines the set of remote methods of the object . Proxies are generated only for methods declared, implemented or inherited by the type.

If a remote method is declared in a Remote interface and does not acknowledge RemoteException in its header, then a warning is given

Currently proxy methods are generated for equals() and toString() without option to disable this

Not hashcode(), wait(), notify()

Summary

- In RMI
  - the proxy generation call takes as an argument only the object.
  - proxies are generated for all remote methods of the object, which by definition do not include Object methods.
  - a method of a class is remote if the class implements Remote or if the method is declared in an interface implemented by the class that extends Remote
- In GIPC
  - The proxy generation call takes as an argument not only the object but some type (class or interface) describing the object
    - Object **instanceof** type must be true
  - Proxies are generated for all methods declared, implemented or inherited by the type
    - Which can include Object methods
  - If one of these methods is declared in a Remote interface, then a warning is given

RMI: Application Supported Group Communication

```
public class ARelayingCollaborativeRMIUpperCaser
      extends ACollaborativeRMIUpperCaser
      implements RelayingCollaborativeRMIUpperCaser {
  protected Map<String, DistributedRMIEchoer> nameToEchoer = new HashMap();
  public void relayToOthers(String aString, String aCallerName) {
    for (String aClient : nameToEchoer.keySet()) {
      DistributedRMIEchoer echoer = nameToEchoer.get(aClient);
      if (!aClient.equals(aCallerName))
      try {
        echoer.echo(aString);
      } catch (Exception e) {
         e.printStackTrace();
      }
    }
  }
  public void addListener(String aName, DistributedRMIEchoer anEchoer) {
    nameToEchoer.put(aName, anEchoer);
  }
}
```

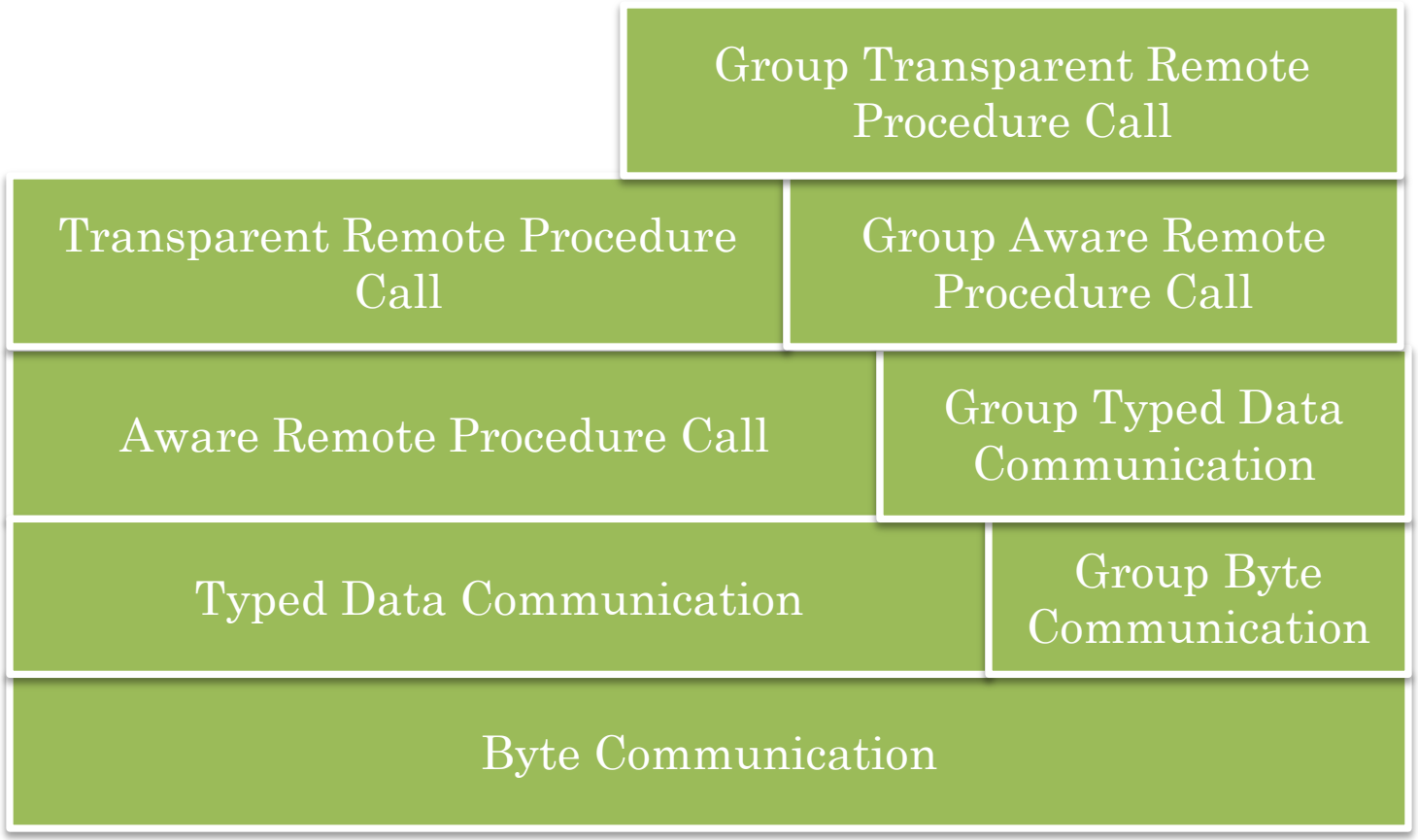
Free programmer from group communication?

Java RMI: Duplex vs. Group Communication

Java RMI designed for duplex communication

Programmer must implement group communication on top of it

Group vs. Non Group (Simplex, Duplex) IPC



GIPC has group communication for each communication abstraction

Allows server to broadcast or multicast messages (data or service requests) to its clients

Simulating ByteBuffer and Object Send

Sometimes we need to send byte buffer or object to a group of computers

- An IM message to all other users
- A file chunk to all DropBox files

Can simulate data transfer through procedure call

```
byteBufferProxy.newFileChunk(byteBuffer)
```

```
objectProxy.newMessage(message)
```

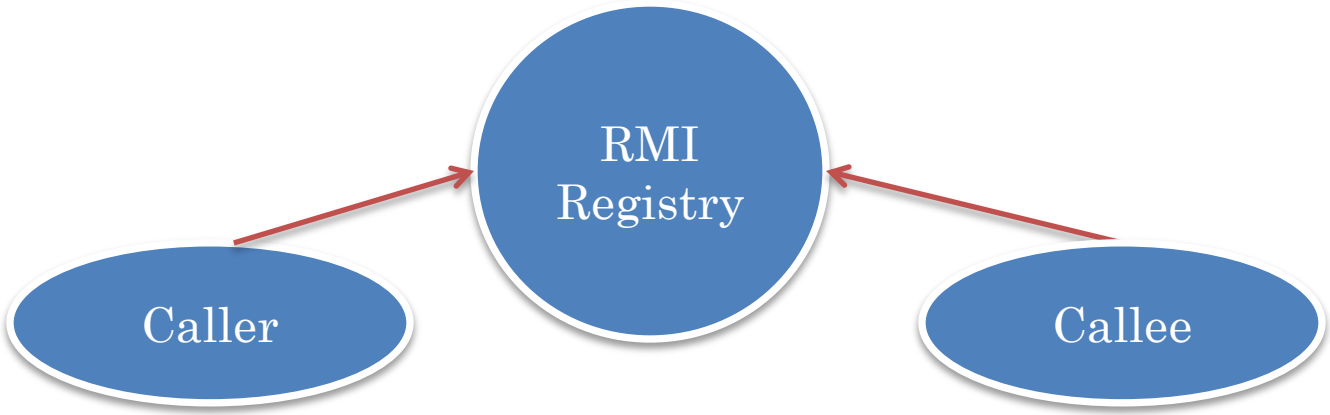
Extra cost converting (marshaling) procedure call to data and unmarshaling data to procedure call

Makes it hard to create relayer of procedure calls (later)

Summary

- Existing data communication and rpc systems support duplex input ports that is ports that allow a server to send messages to clients.
- However messages must be addressed individually to the clients which increases
  - programmer overhead as a loop must be written, which in turn increases collaboration awareness
  - processing overhead as serialization/marshalling must be multiple times rather than once
- Each IPC layer of GIPC allows messages to be directly broadcast or multicast to clients to avoid the problems above

Java RMI Required Architecture



Registry idea inconsistent with (peer to peer) groups

GIPC Required Architecture

Proxy can be created before callee object

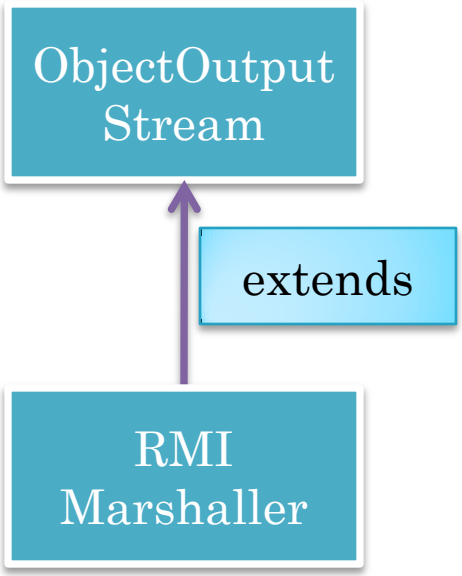
Unlike Java RMI, target object name may be invalid, as can the arguments

Supports (peer to peer) sessions with late binding of members

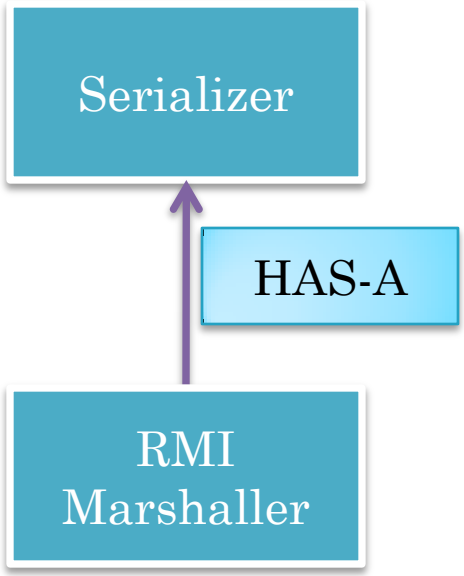
Server is optional and can be built using basic primitives

Consistent with teaching goal

Java RMI: Serialization →Marshalling



GIPC: Serialization →Marshalling



GIPC API

API of lower level layers (except byte communication) is visible

Only remote function calls have to wait

Proxies can be generated for all methods of an object

Server method can determine the host and client who made the call (from data layers)

Special call made by client to pass its name to the RPC system

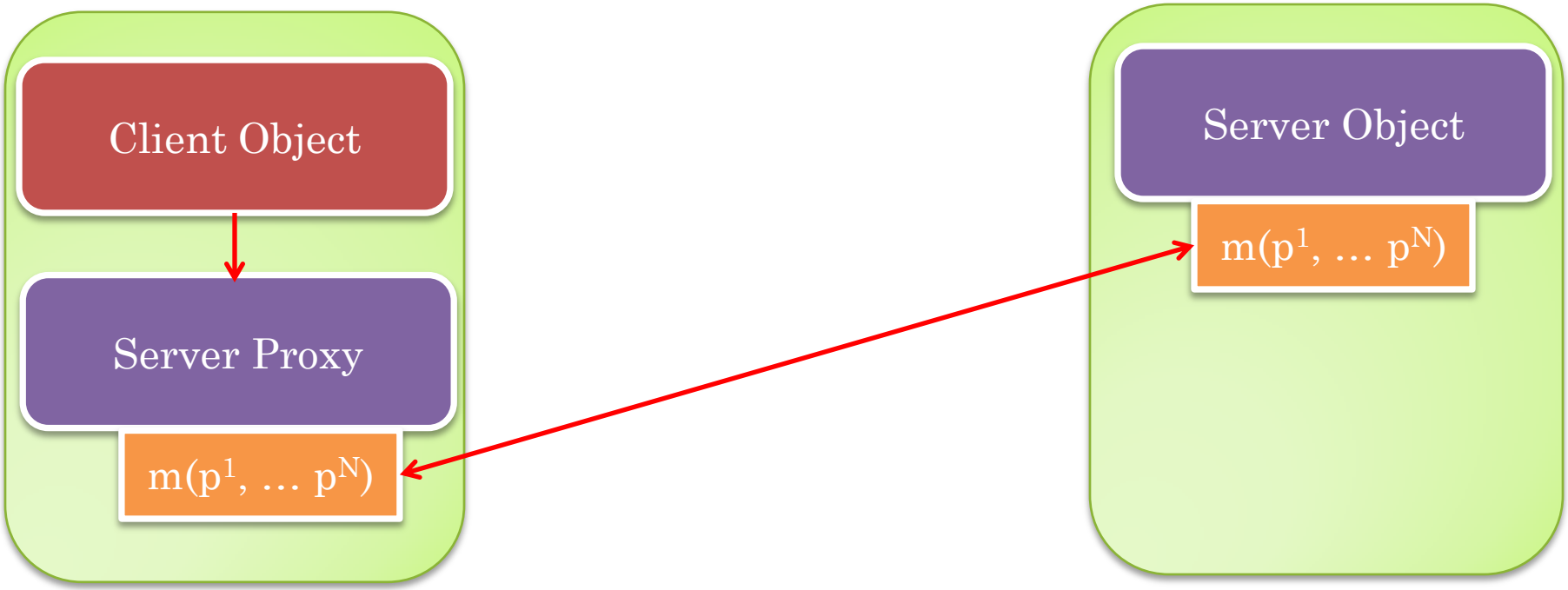
GIPC has group communication for each communication abstraction

Proxy can be created before target remote object

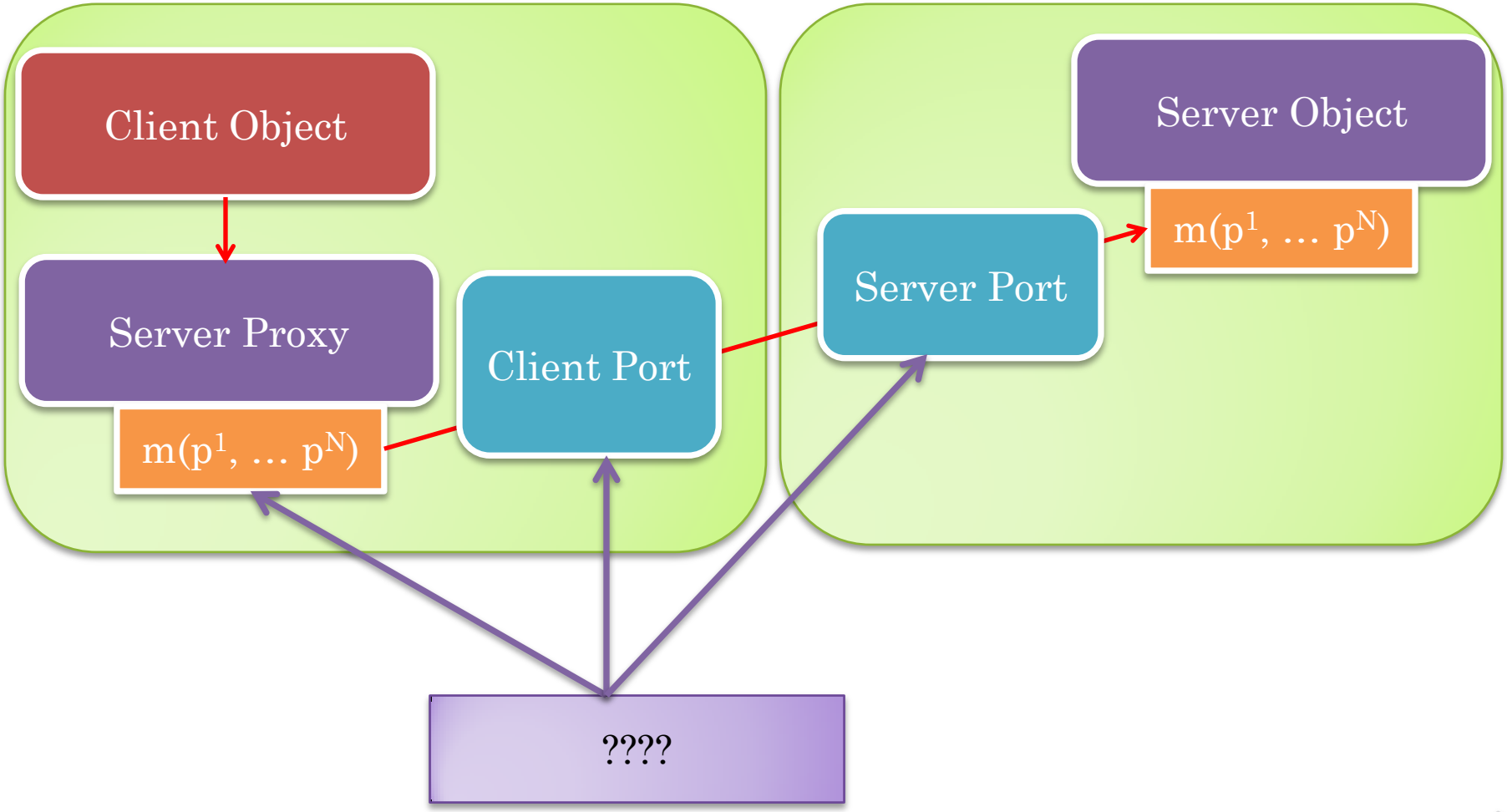
GIPC vs. RMI Extensibility

Layered, extendible
Aware and transparent layers allowing control
No global registry but such a registry can be built
Unlike Java RMI, target object name may be invalid, as can the arguments
Communication with registry is not magic predefined code
Serialization and marshaling linked by HAS-A and not IS-A
Synchronizing (and other mechanisms) not hardwired

RMI



Port-based RMI Architecture



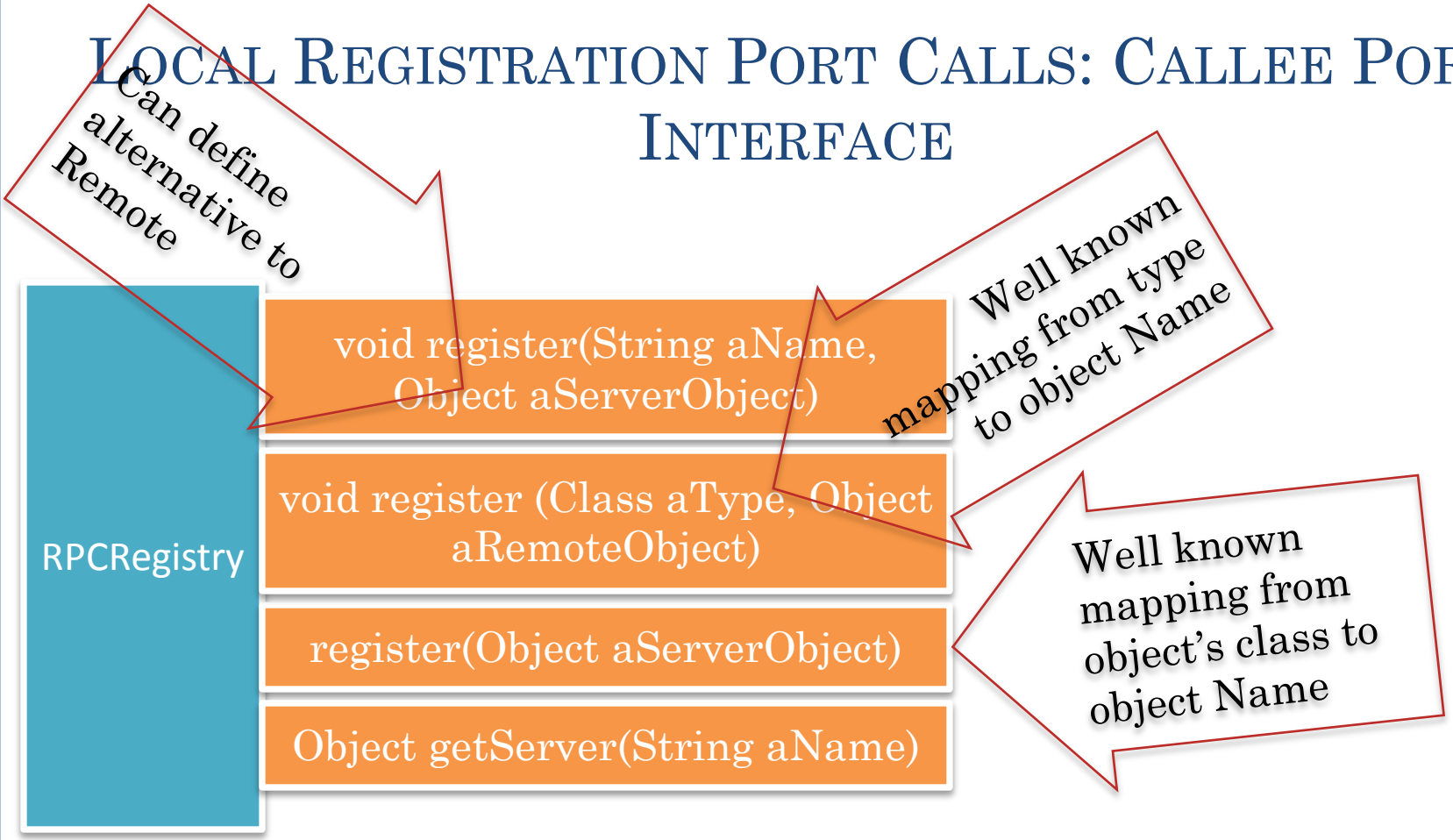
RPC Set Up

Server exports external description of calls serviced on RA (data) port

Statically or dynamically

No global registry but such a registry can be built

Local Registration Port Calls: Callee Port Interface

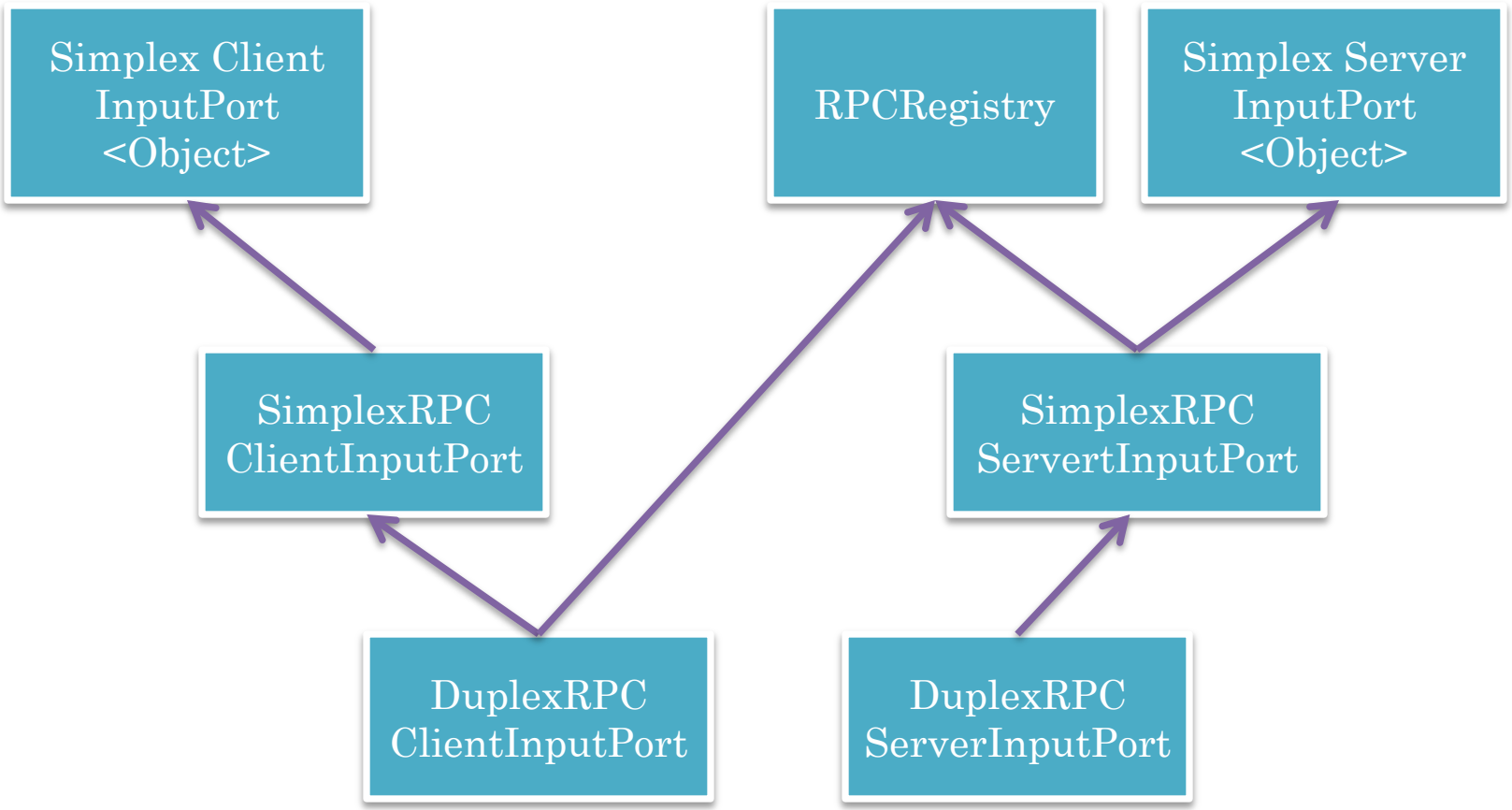


RPC Registry Implementation

```
public class AnRPCRegistry implements RPCRegistry {
  Map<String, Object> nameToServer = new HashMap();
  public void register(Class aType, Object aServerObject) {
    nameToServer.put(aType.getName(), aServerObject);
  }
  public void register(String aName, Object aServerObject) {
    nameToServer.put(aName, aServerObject);
  }
  public void register(Object aServerObject) {
    nameToServer.put(aServerObject.getClass().getName(),
        aServerObject);
  }
  public Object getServer(String aName) {
    return nameToServer.get(aName);
  }
}
```

Instance kept by port of callee

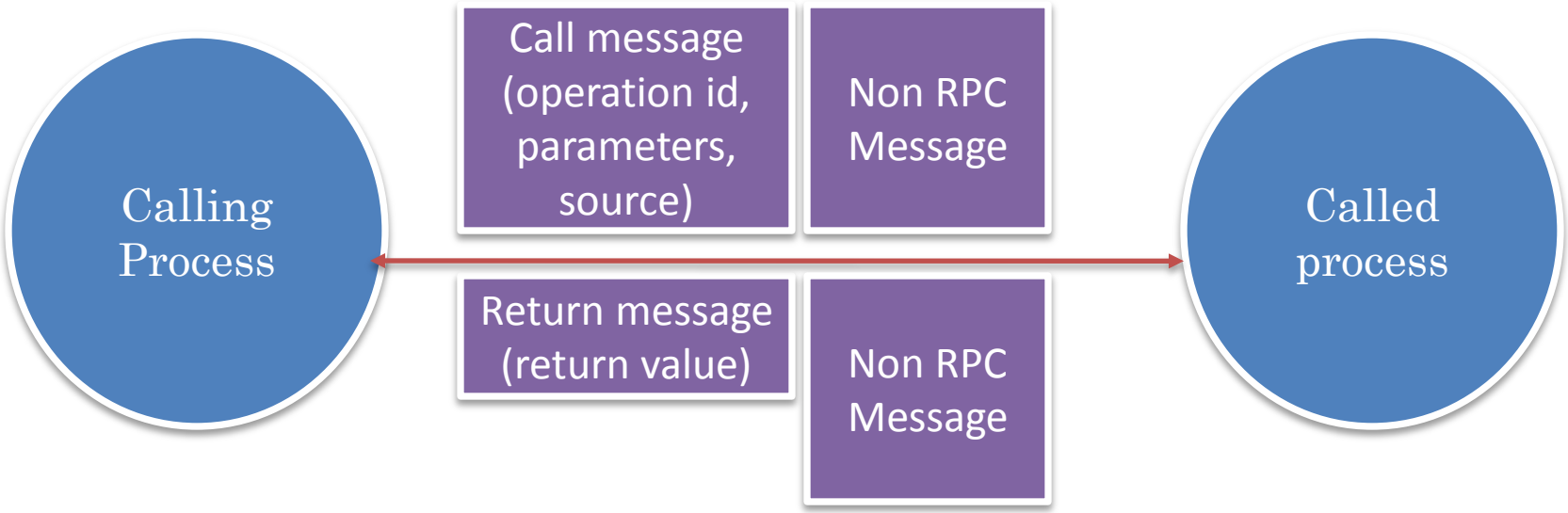
Interface Extensions



GIPC vs. RMI

Allow remote assignment and remote procedure call on same port

Notion of connected client and server port visible



Distinguishing between regular and RPC messages?

Distinguishing RPC and Regular Messages

Use special GIPC types for RPC Messages

GIPC RPC traps instances of these types and passes others to application listeners of underlying data ports

Applications cannot pass instances of these types directly

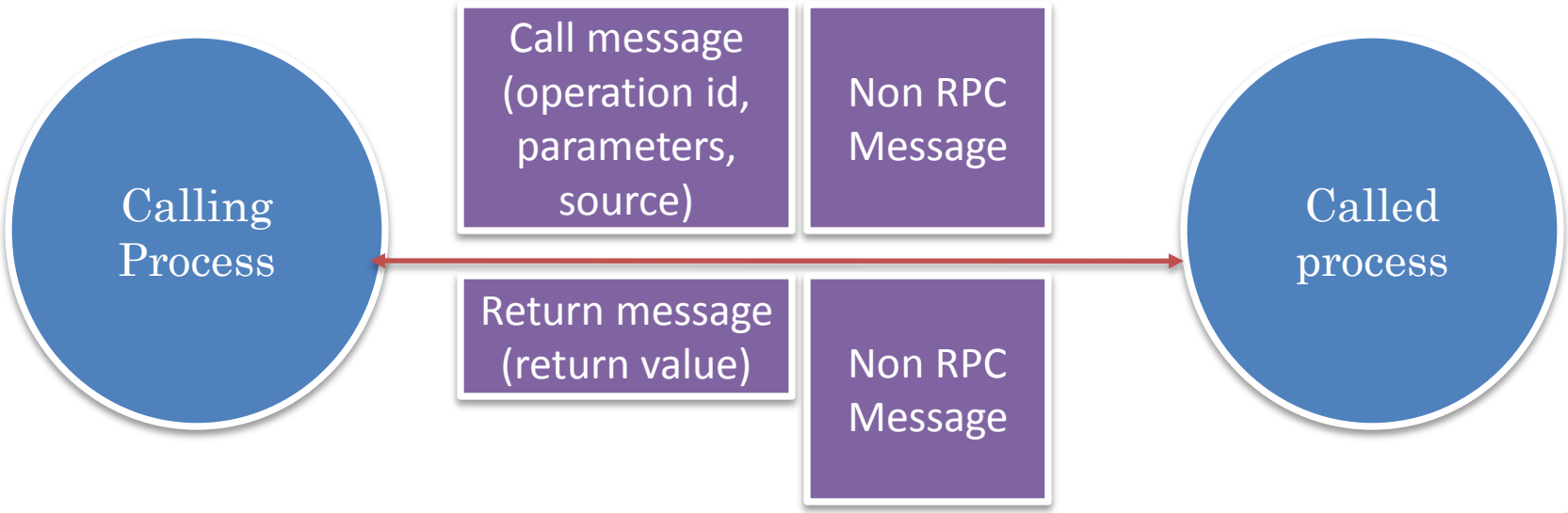
Should allow GIPC developers to change the nature and processing of of these messages without changing the complete RPC system

Nature of these messages?

Processing of Call Message and Return Value

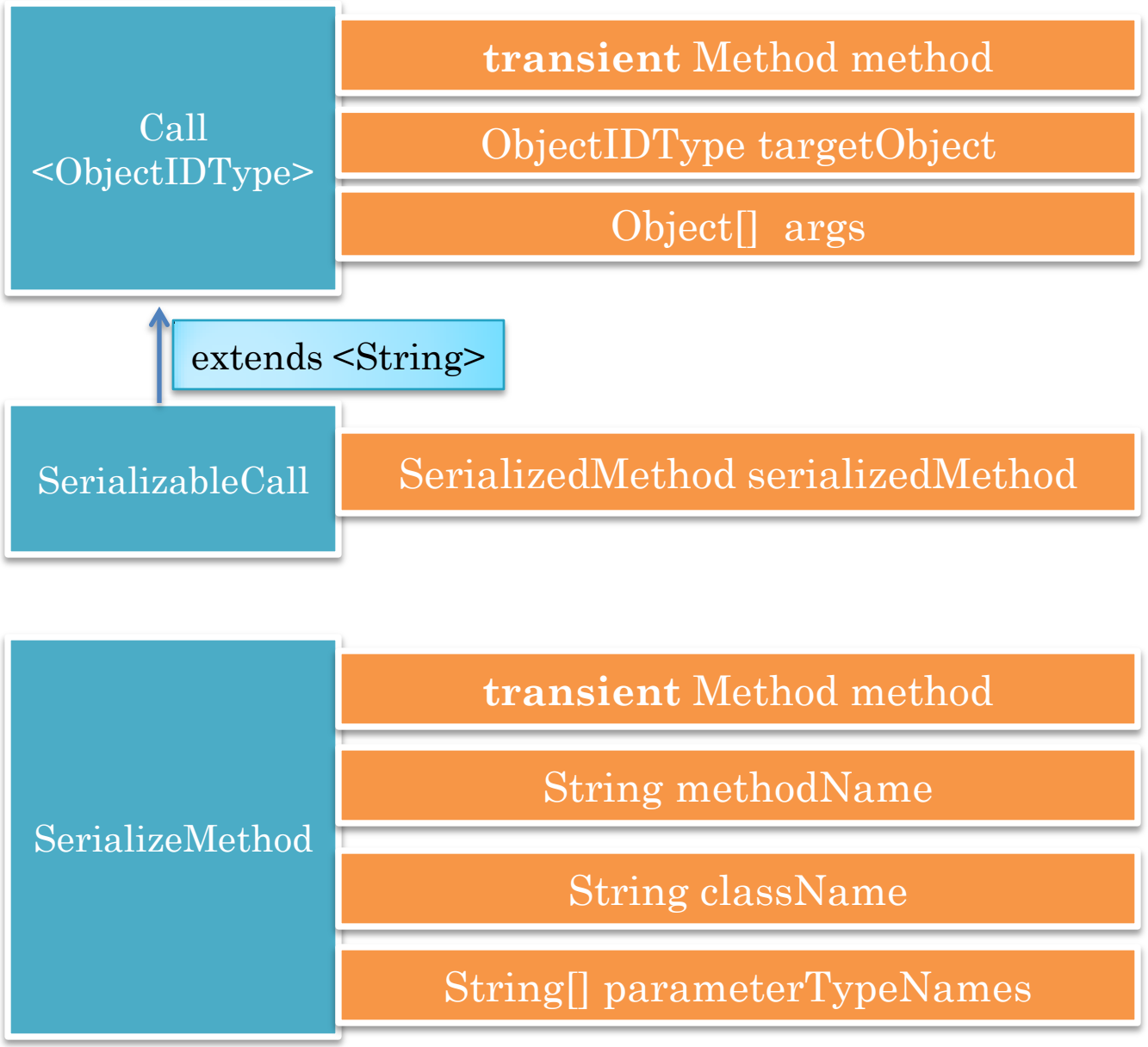
Generate call message for call using exported information

Possibly create return message in callee process

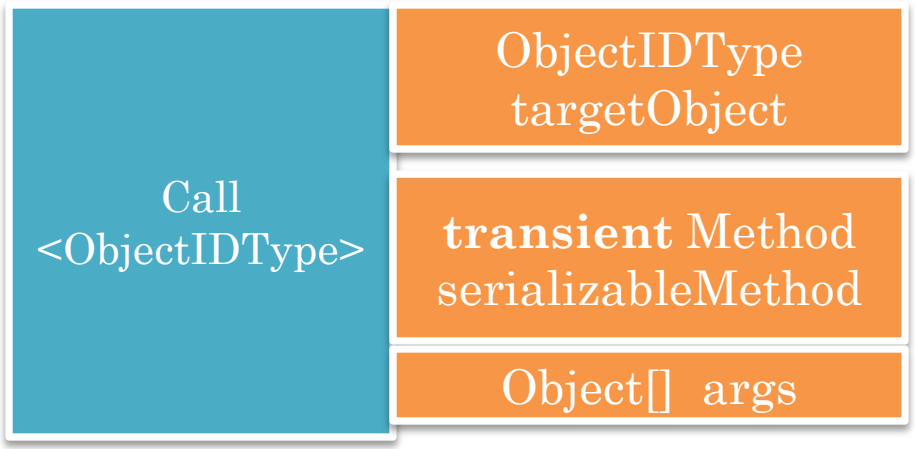


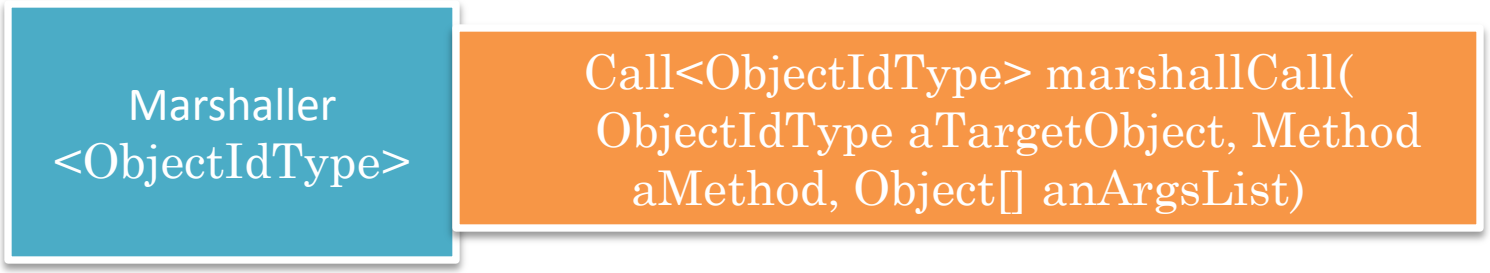
Method Return Value

Call Message

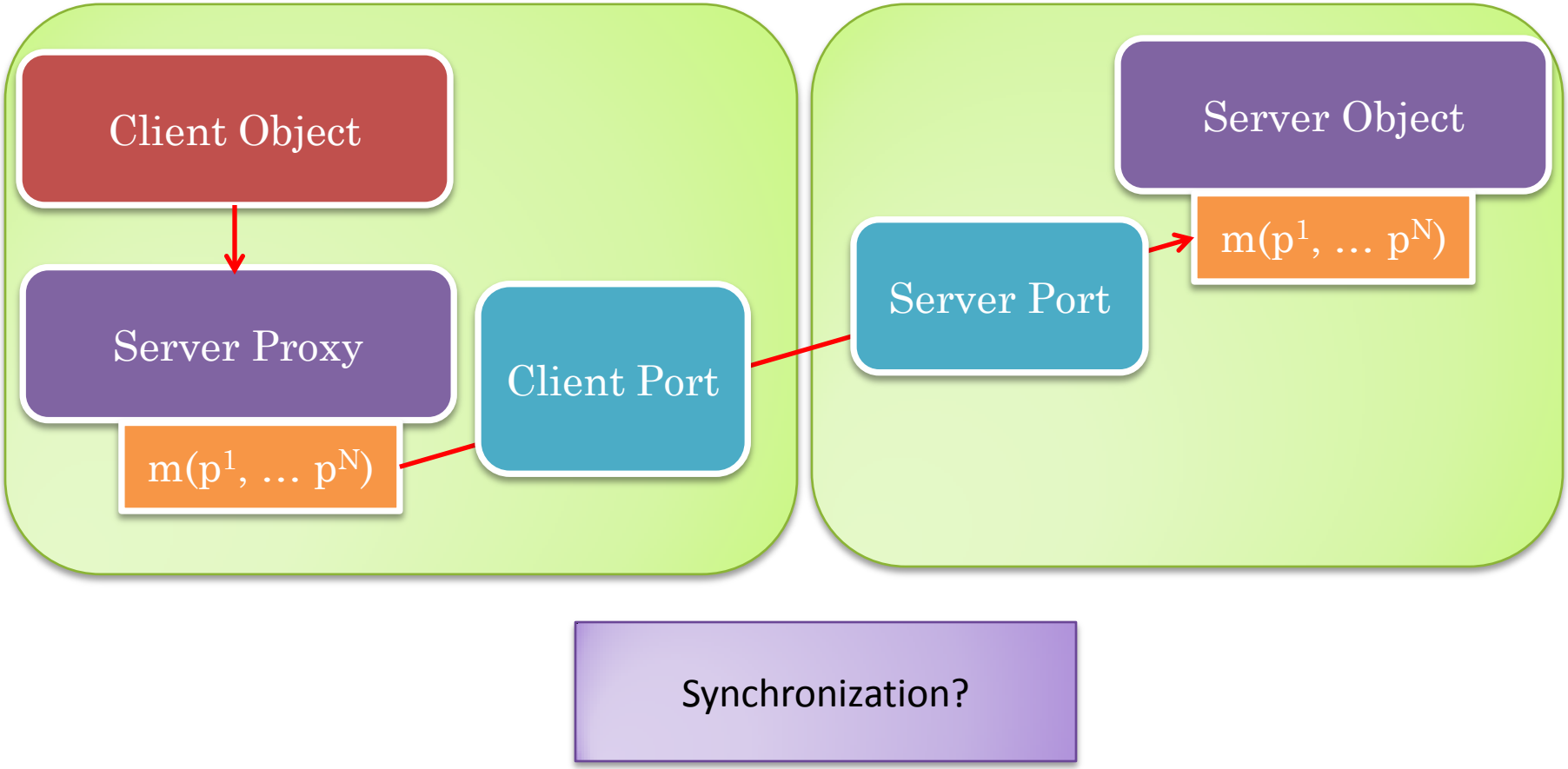


Serializing a Method

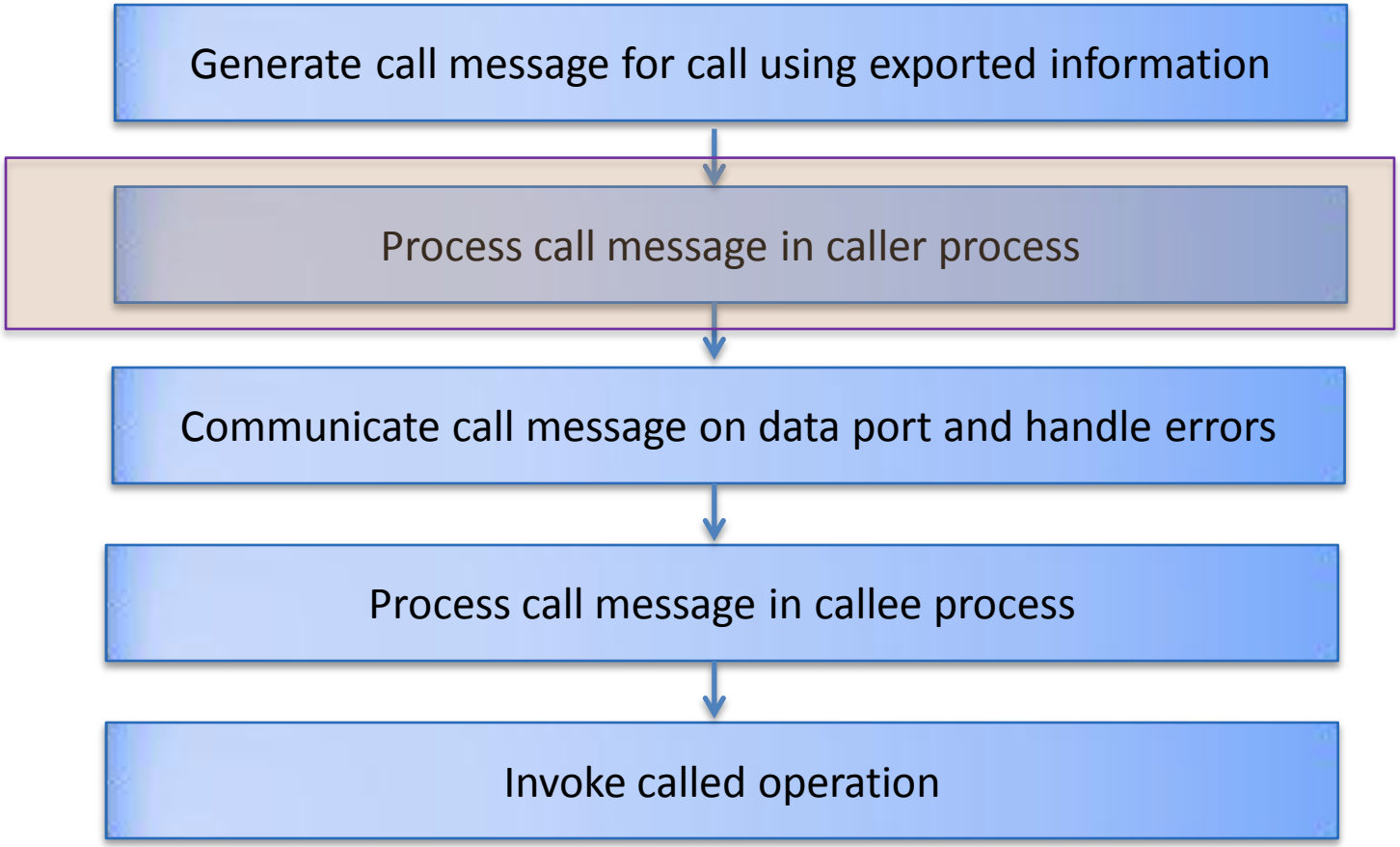




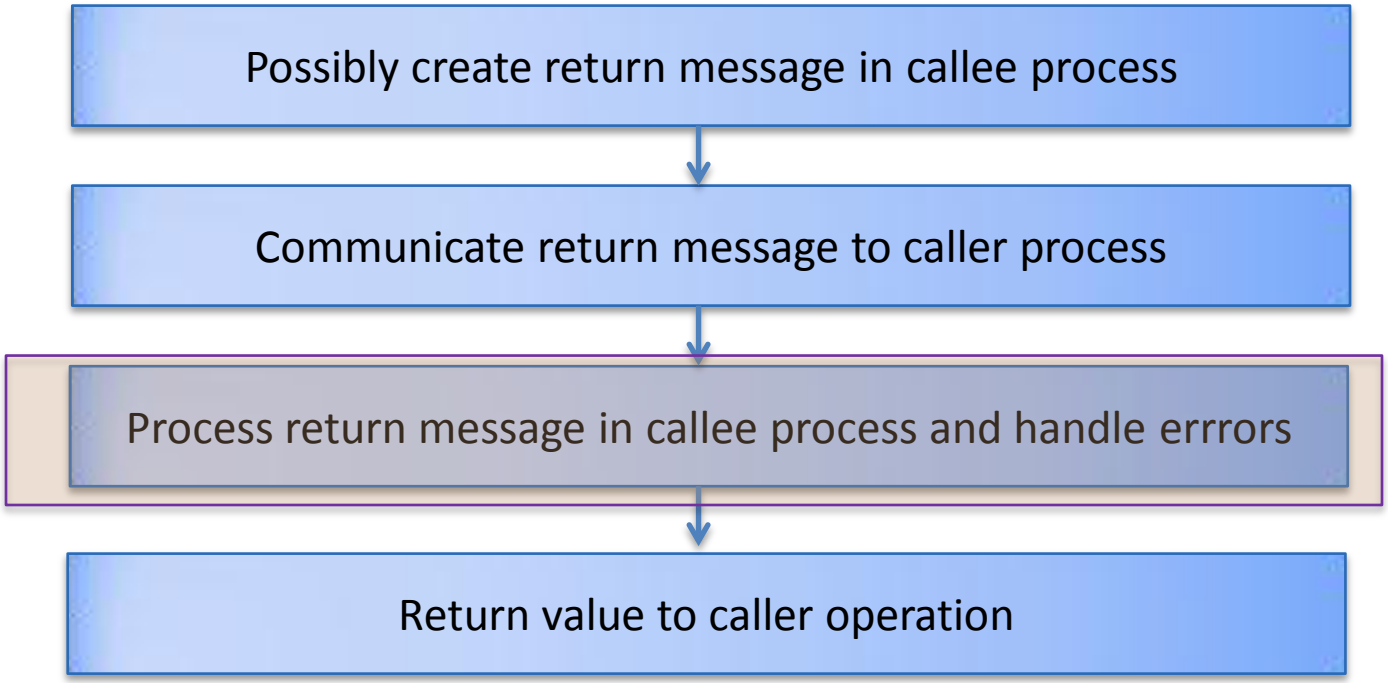
Port vs. GIPC-RMI



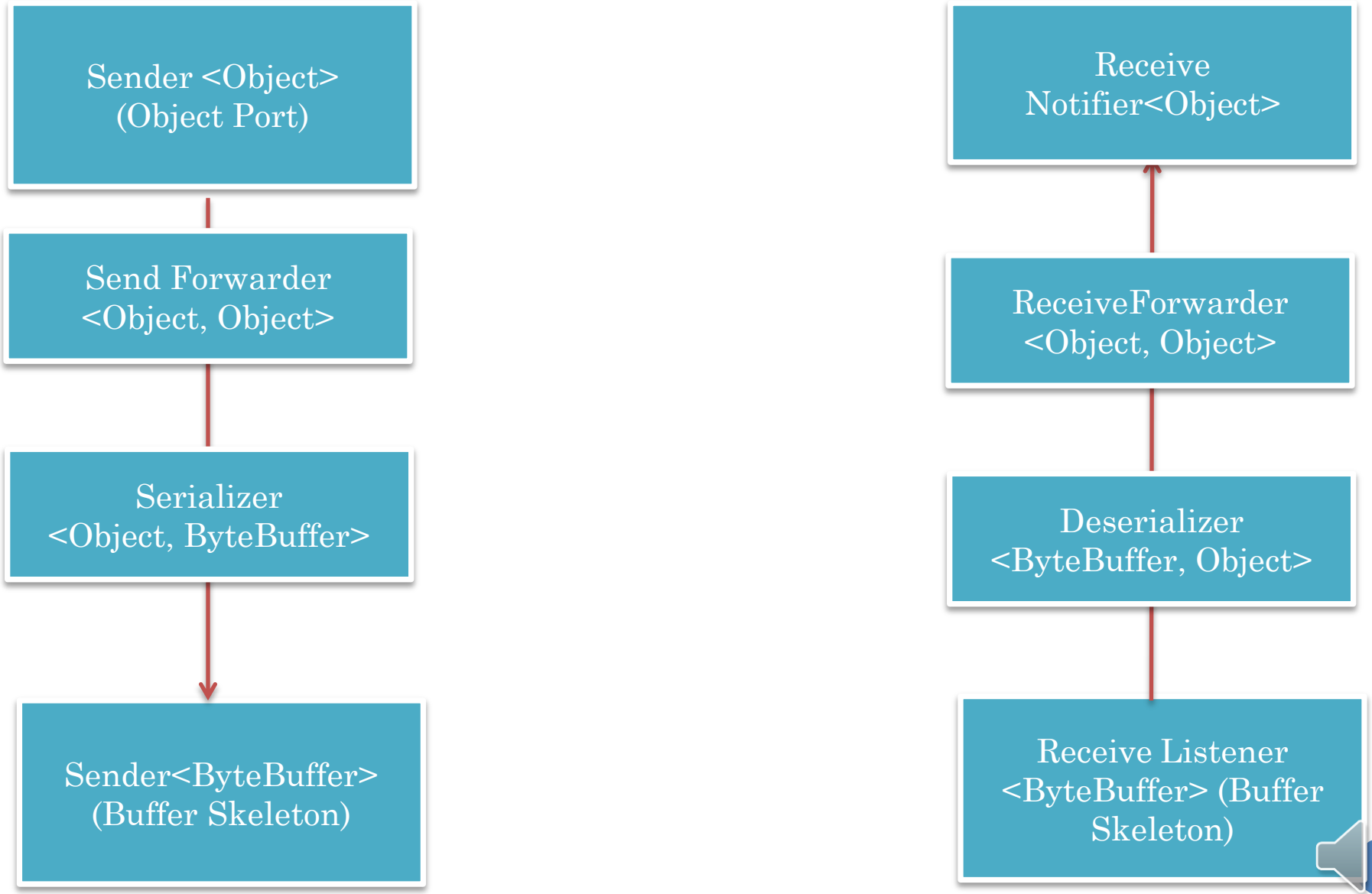
Processing of Call Message and Return Value



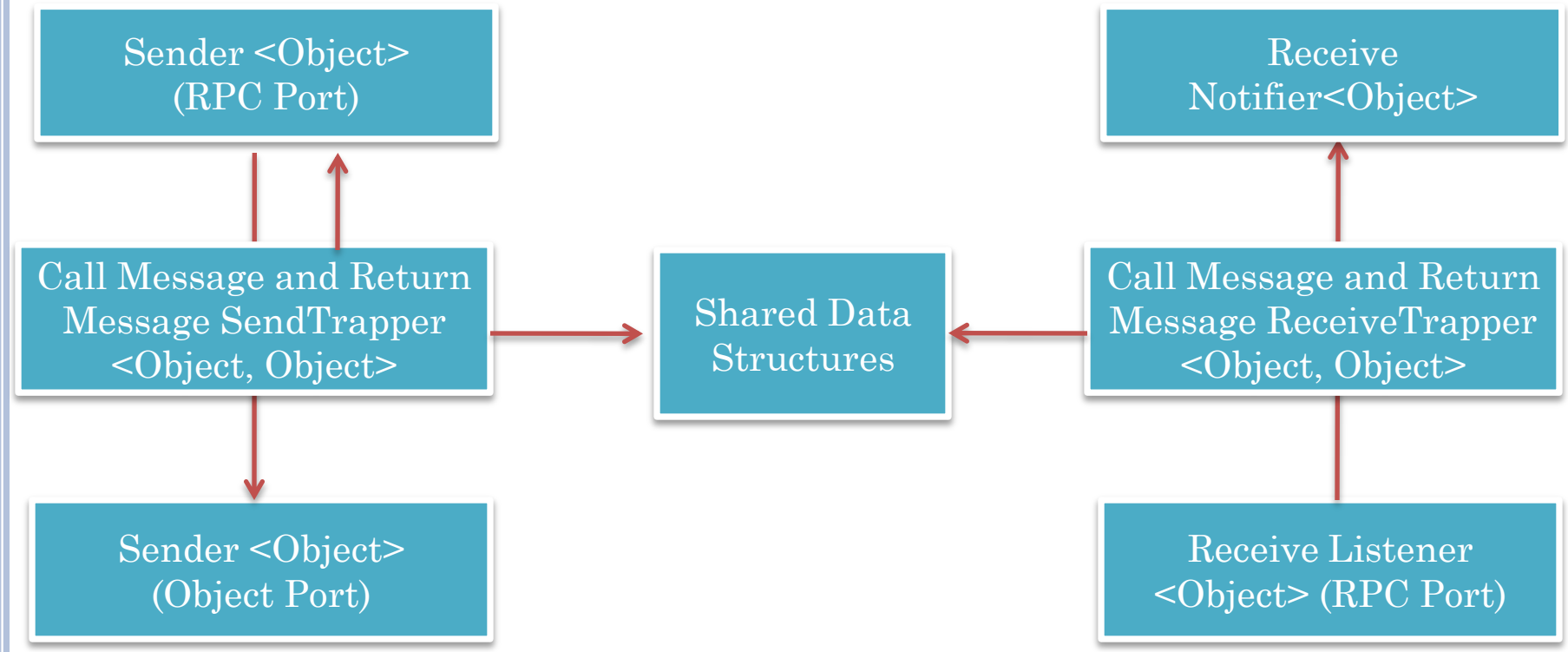
Processing of Call Message and Return Value



Object Forwarders and Serializers



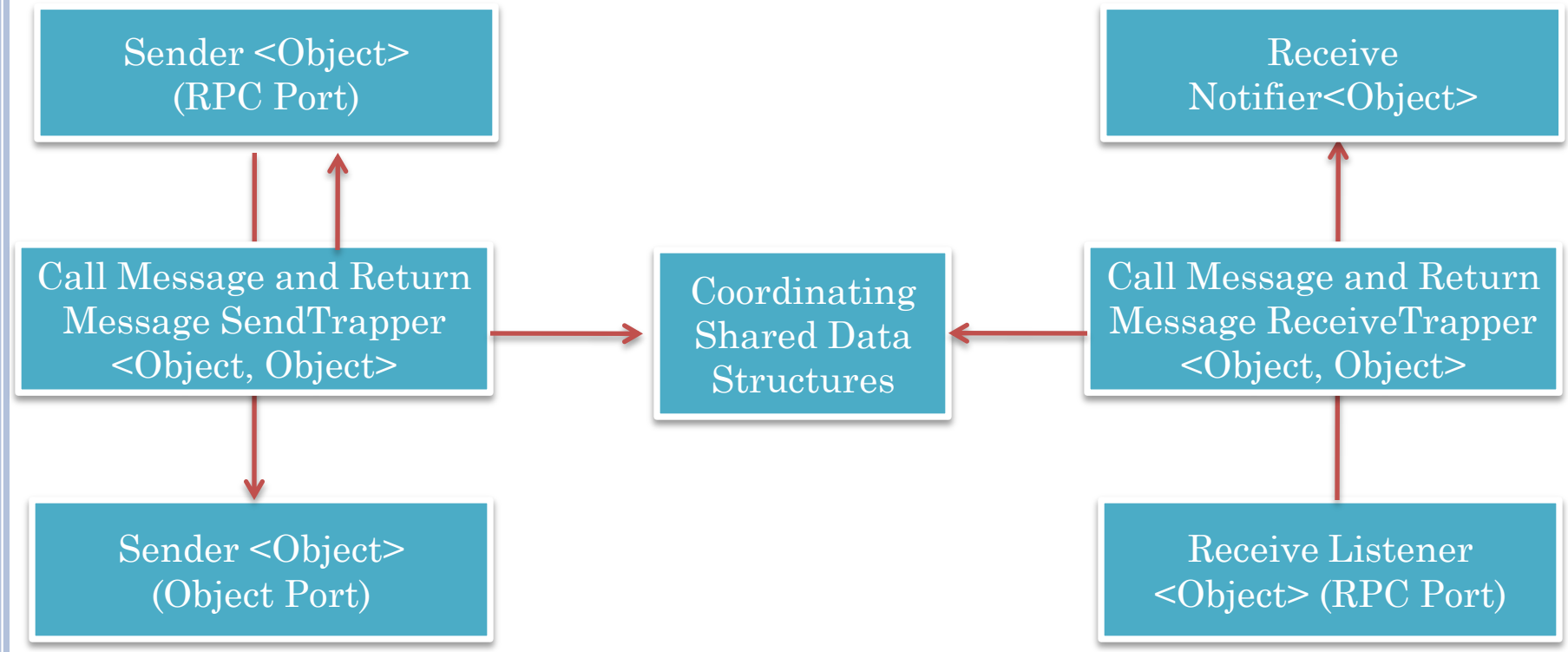
Object Forwarders and Serializers



Send trapper may need to block sender to implement synchronous operation

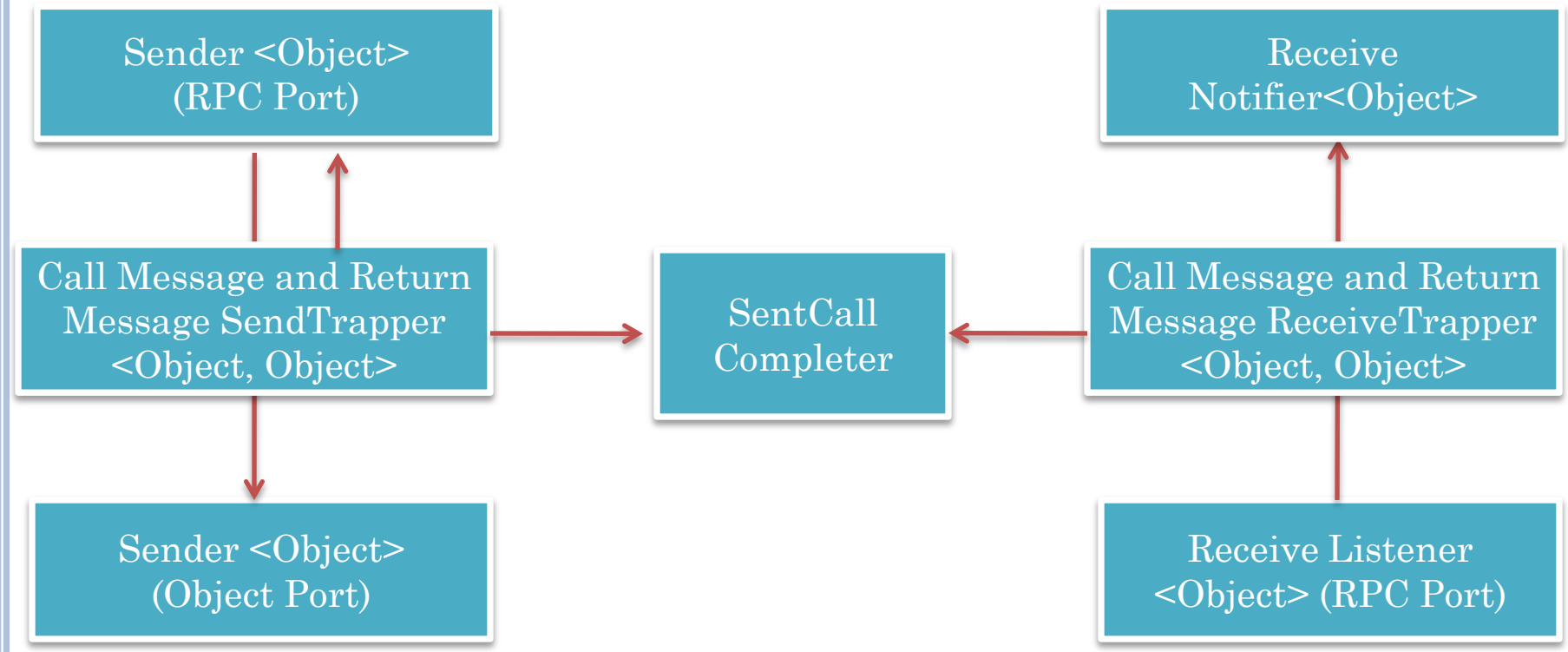
Send and receive trapper may need to share information about remote references sent and received

Object Forwarders and Serializers (Review)



Send trapper may need to block sender to implement synchronous operation

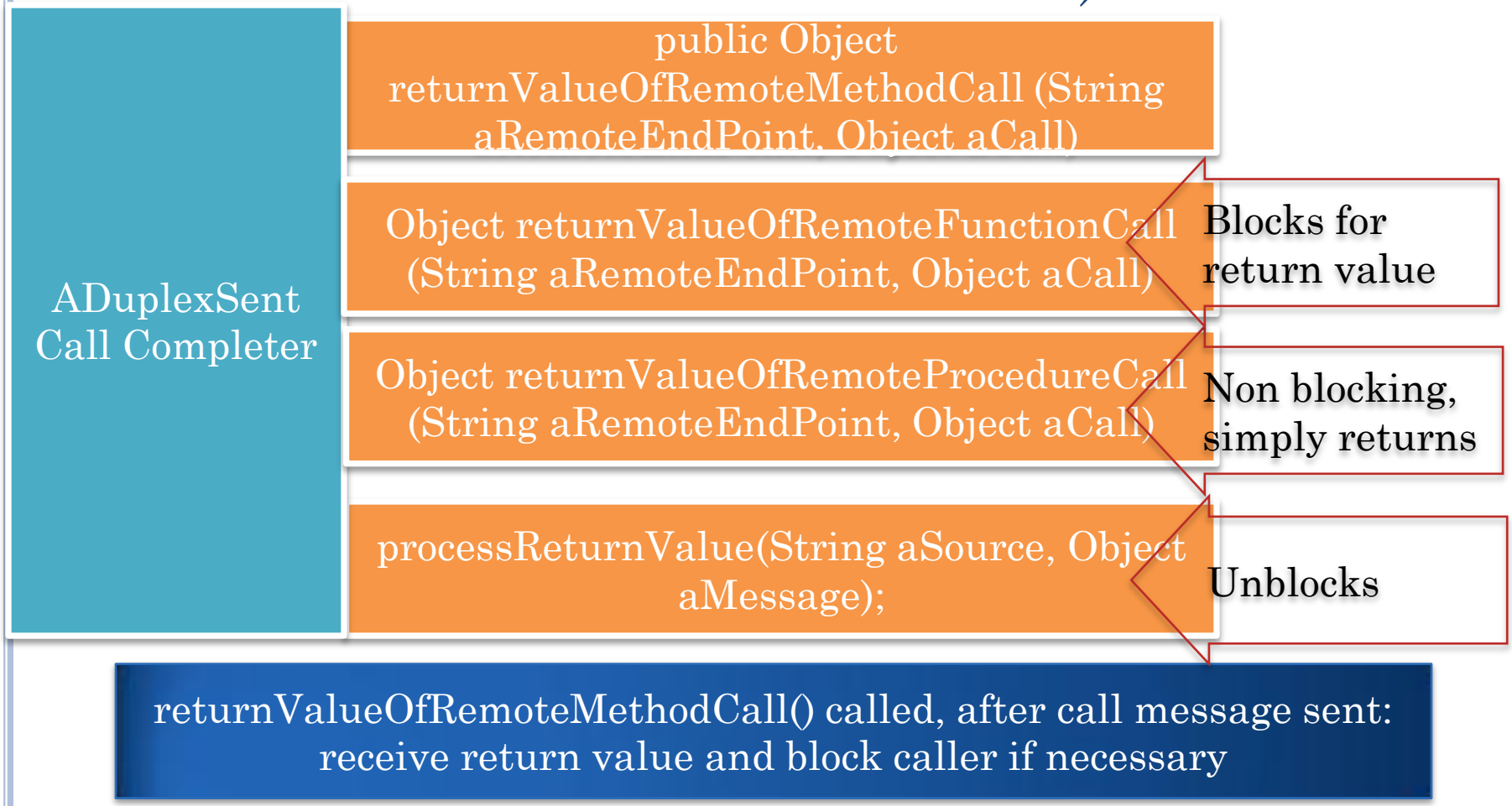
Object Forwarders and Serializers (Review)



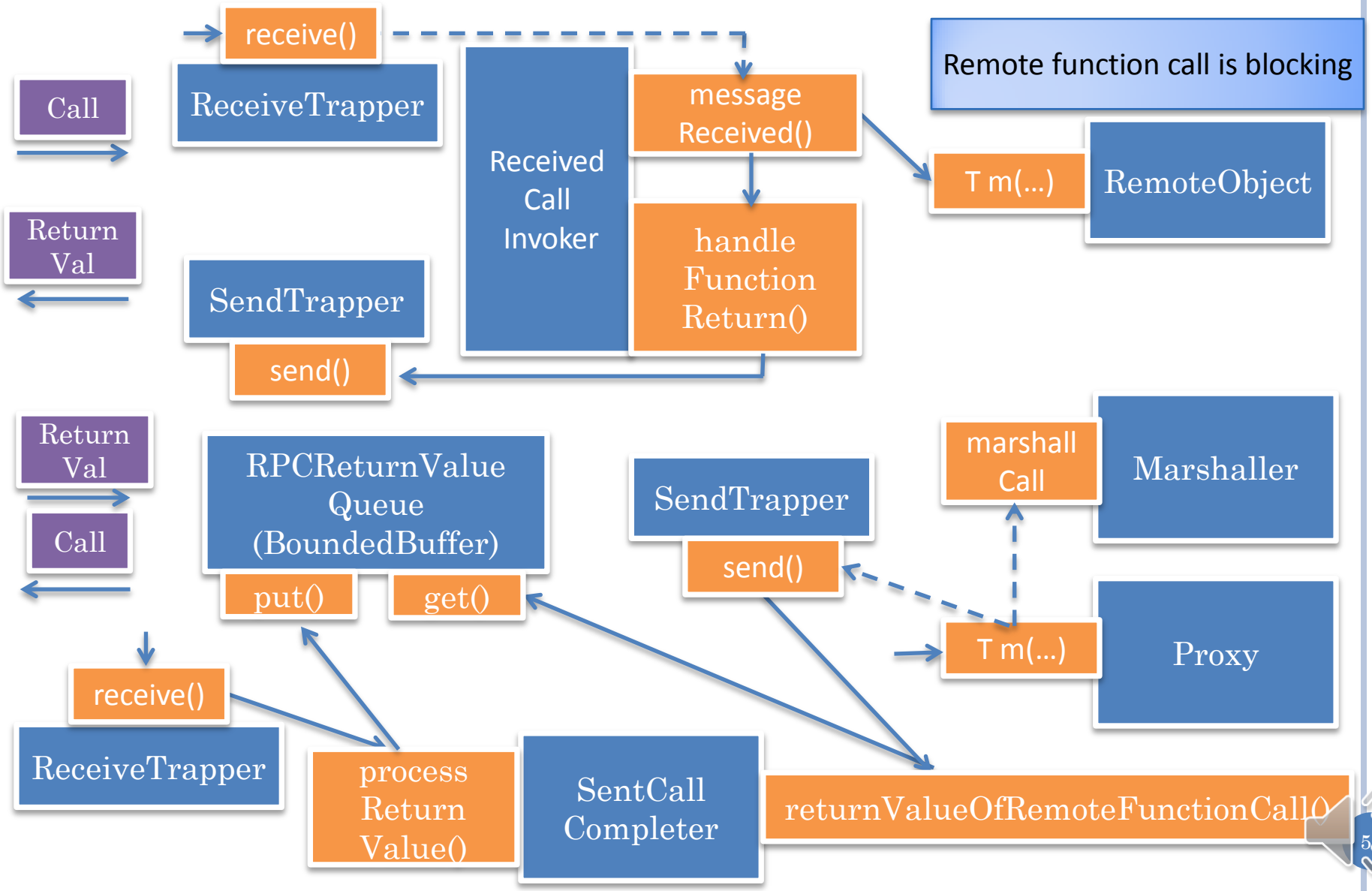
SendTrapper and ReceiveTrapper at Caller Share an Instance of Synchronizing SentCallCompleter

Coordination between Caller and Called Site?

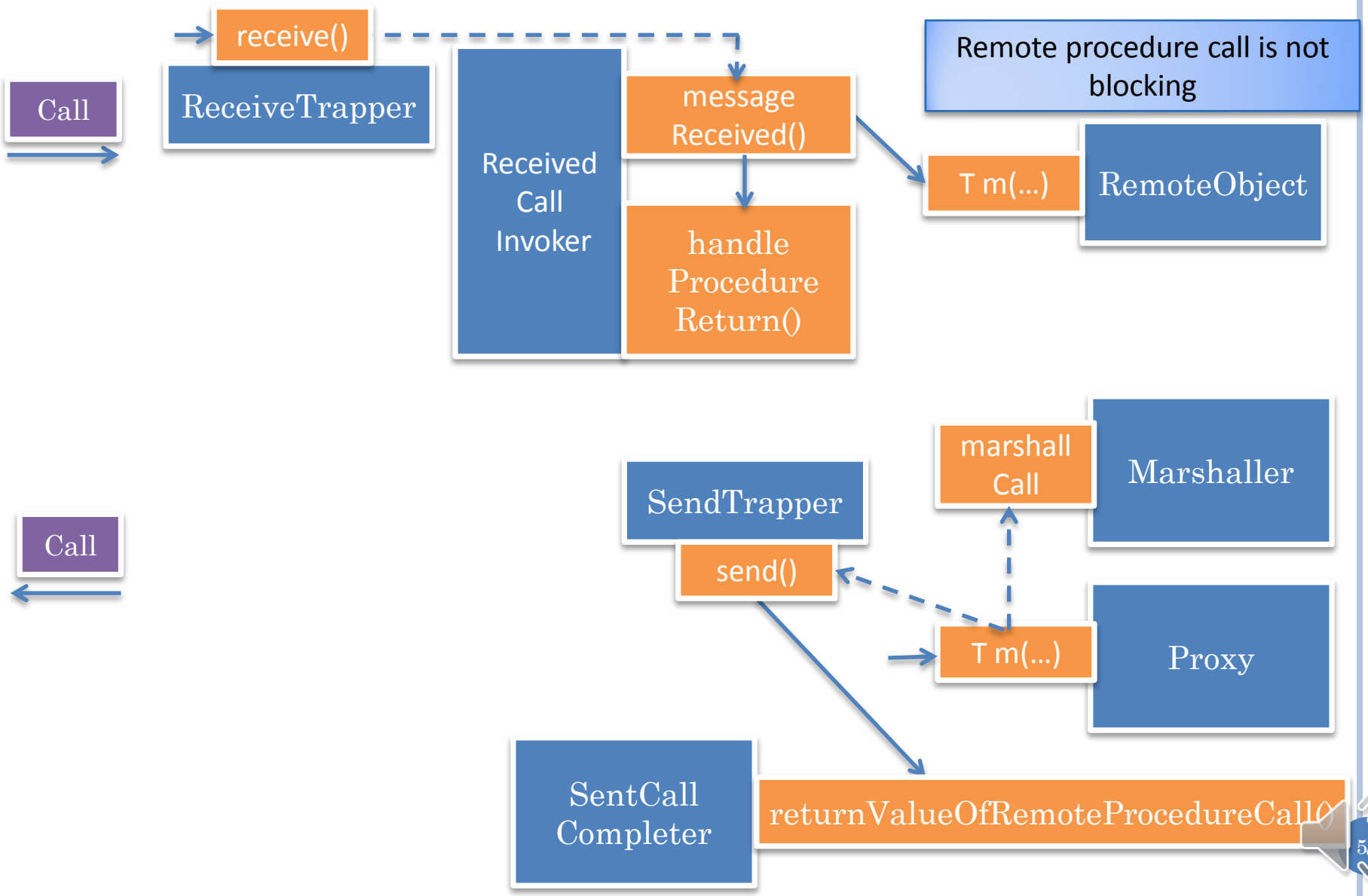
Duplex Sent Call Completer (Public and Non Public Methods)



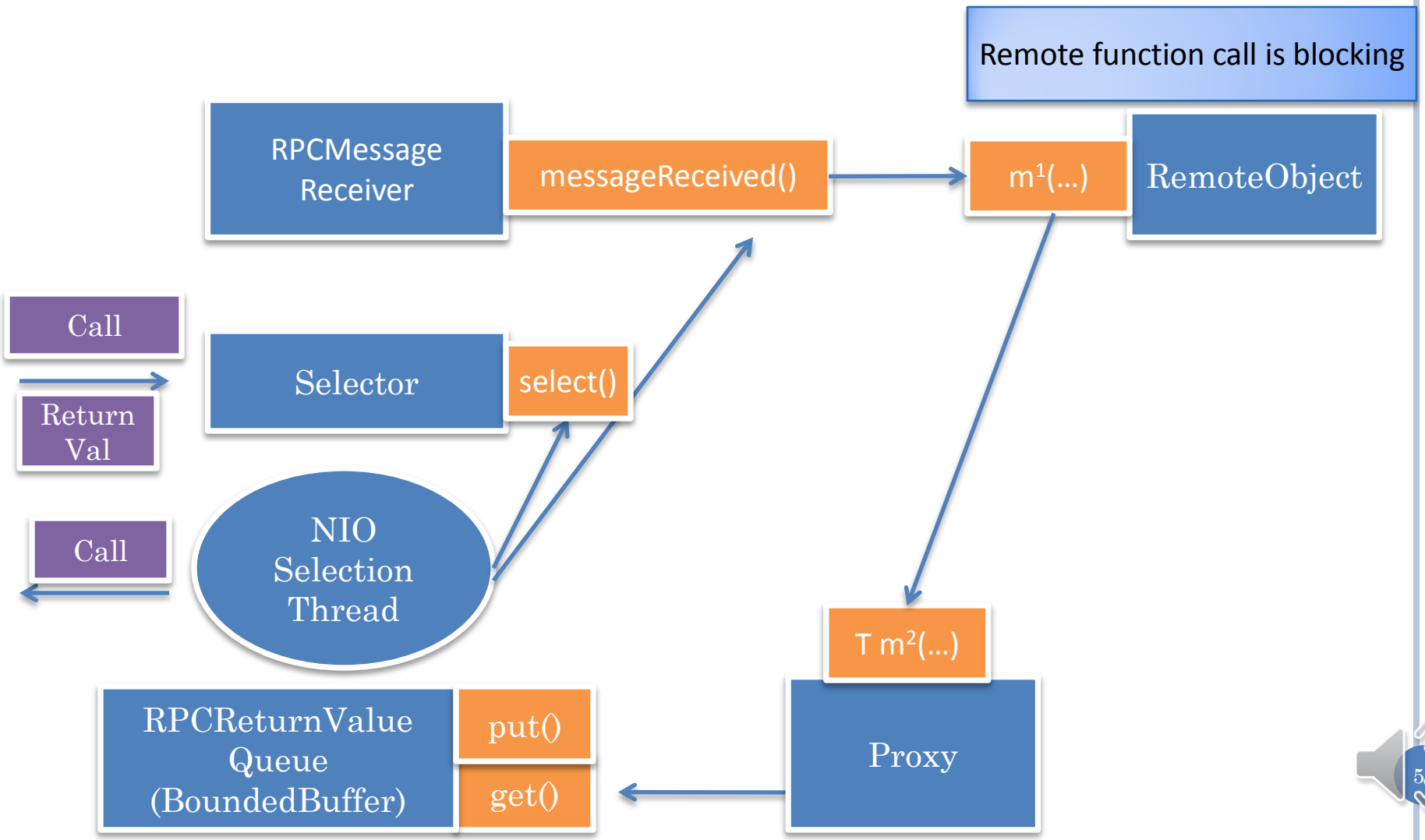
Function Call Synchronization and Matched Client/Server Objects



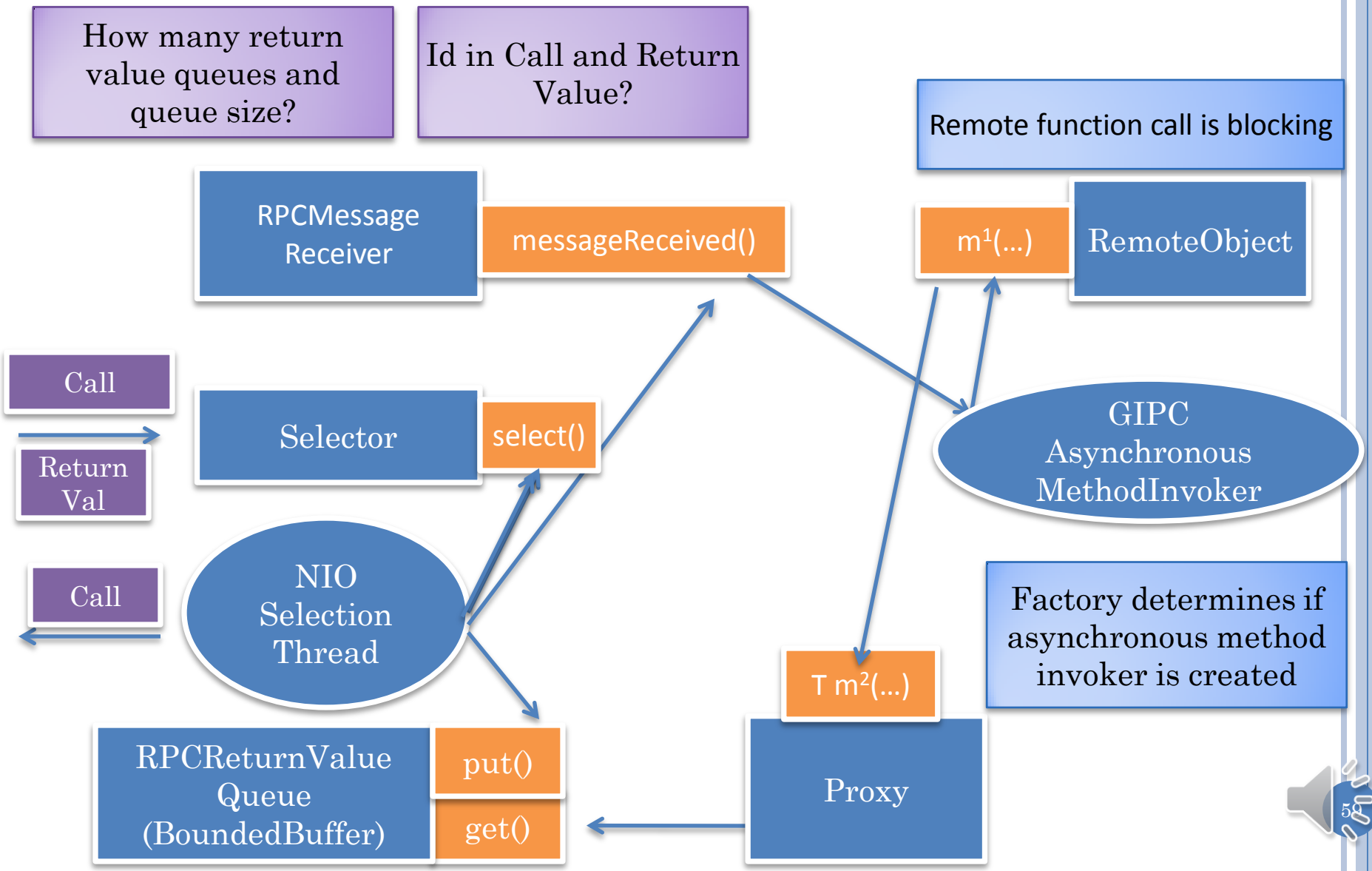
Procedure Call Synchronization and Matched Client/Server Objects



Synchronous Callback in Remote Call



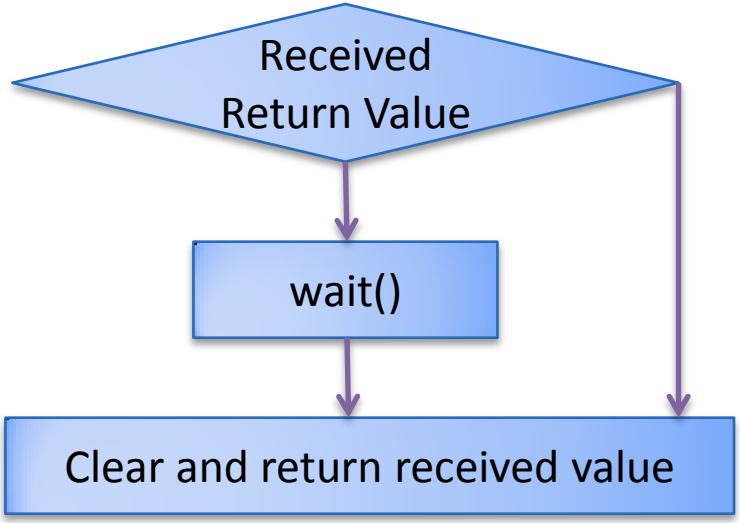
GIPC Asynchronous Invoker (Factory Selected)



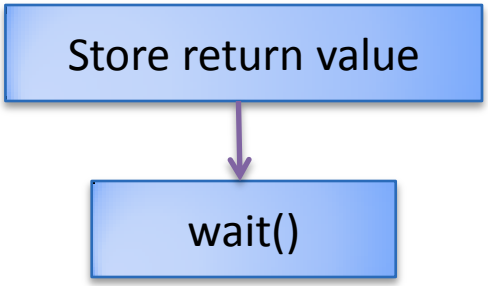
Global Received Value Instead of Buffer

Object returnValueOfRemoteFunctionCall(String aSource, Call aCall)

Multiple calls can be made concurrently to a server by different threads



processReturnValue(String aSource, Object aMessage)



Global Bounded Buffer Servers

Object returnValueOfRemoteFunctionCall(String aSource, Call aCall)

Calls return in same order as they wait

return next value in global (synchronized) bounded buffer

These would occur in same order at a particular callee

As long as caller and callee sites serialize them

processReturnValue(String aSource, Object aMessage)

Concurrent calls can be made to different callees

Put return value in global bounded buffer?

Per Callee Bounded Buffer

Object returnValueOfRemoteFunctionCall(String aSource, Call aCall)

Calls return in same order as they wait

Local bounded buffer ← lookup(aDestination)

return next value in local bounded buffer

These would occur in same order at a particular callee

As long as caller and callee sites serialize them

processReturnValue(String aSource, Object aMessage)

Concurrent calls can be made to different callees

Local bounded buffer ← lookup(aSource)

Put return value in local bounded buffer

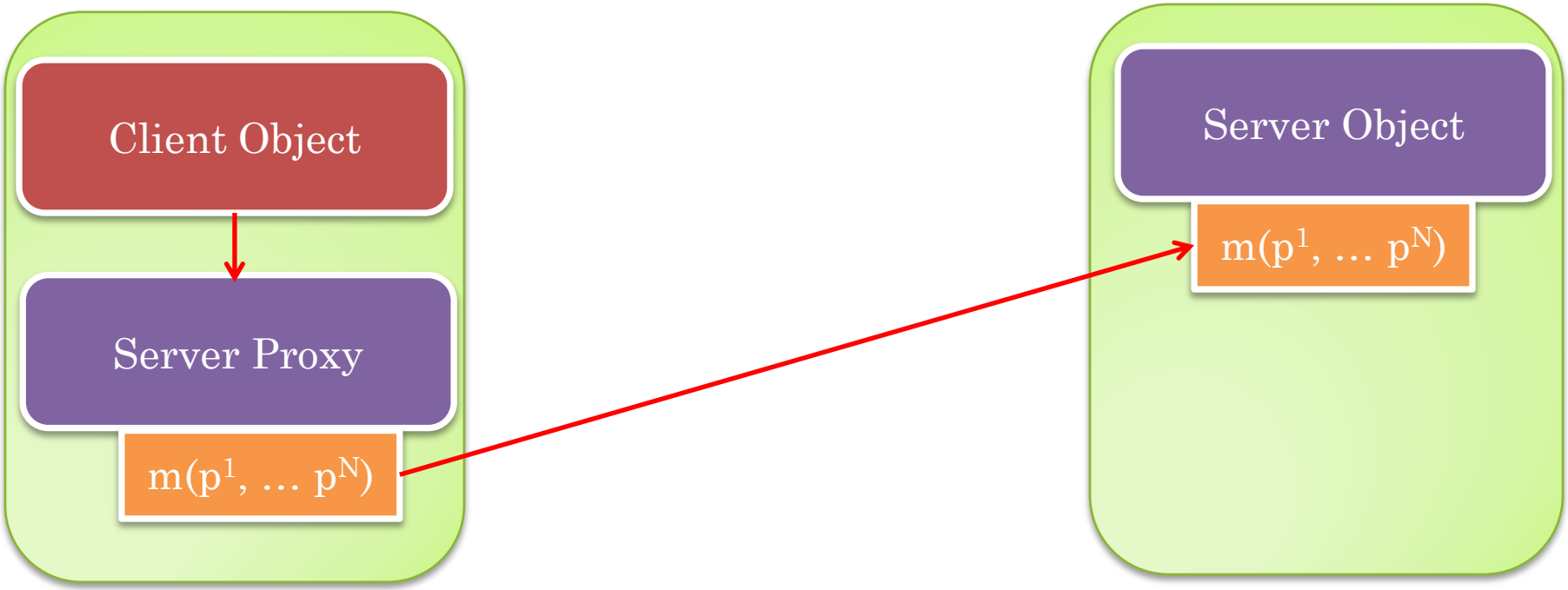
Method Return Value

RPCReturnValue

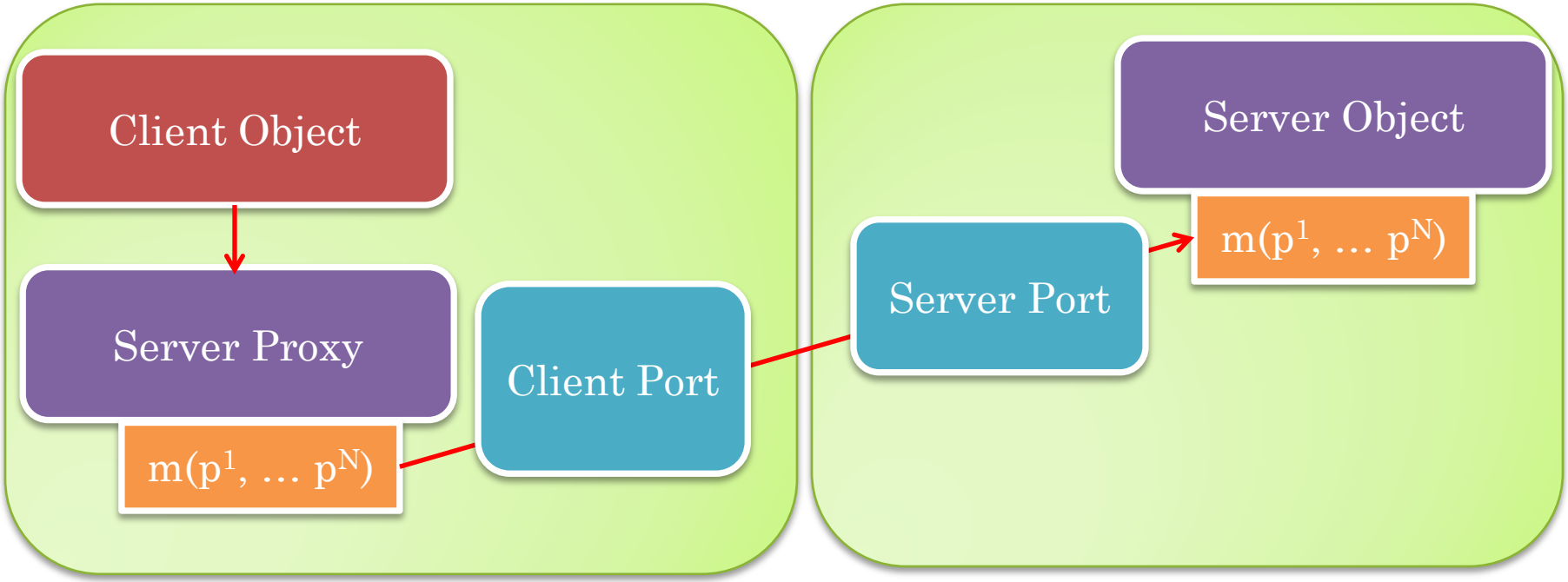
Object returnValue

No Id sent to match with call

Proxy?



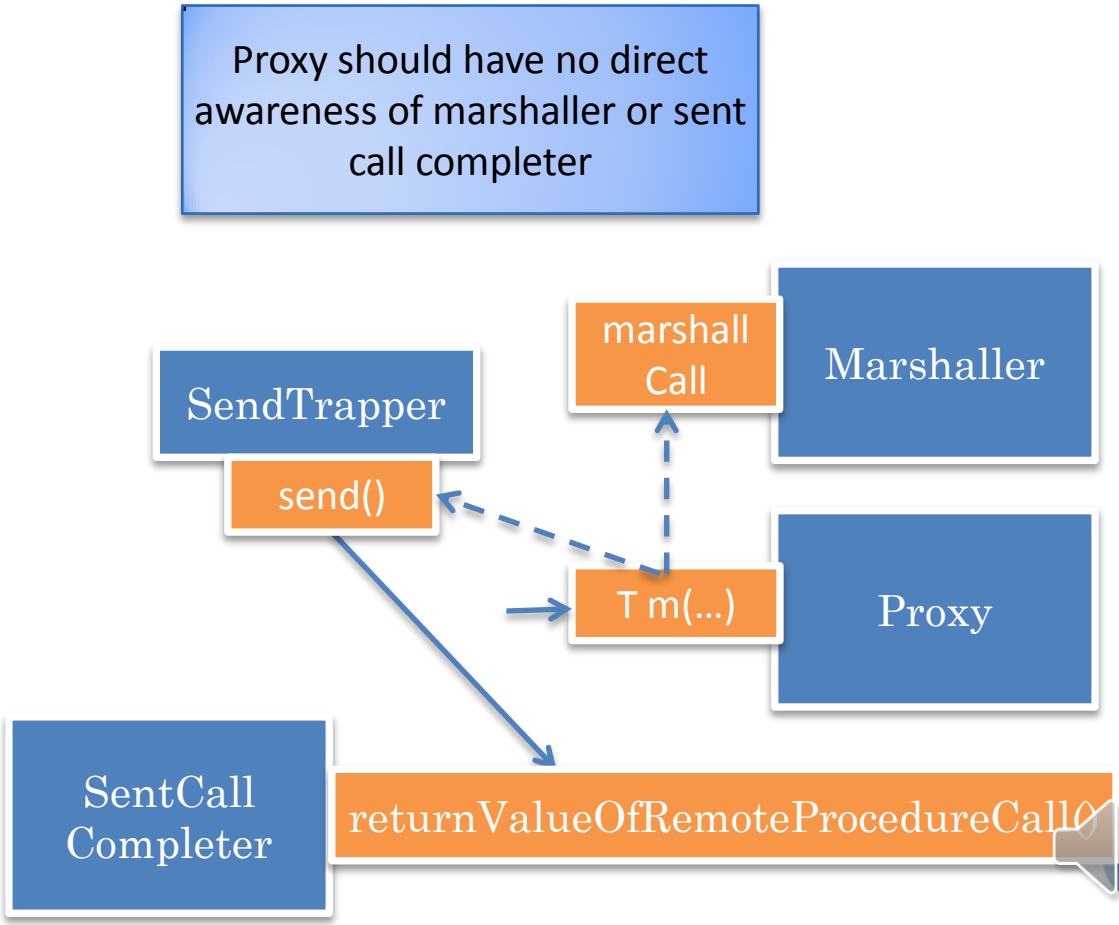
Proxy with Underlying Data Port



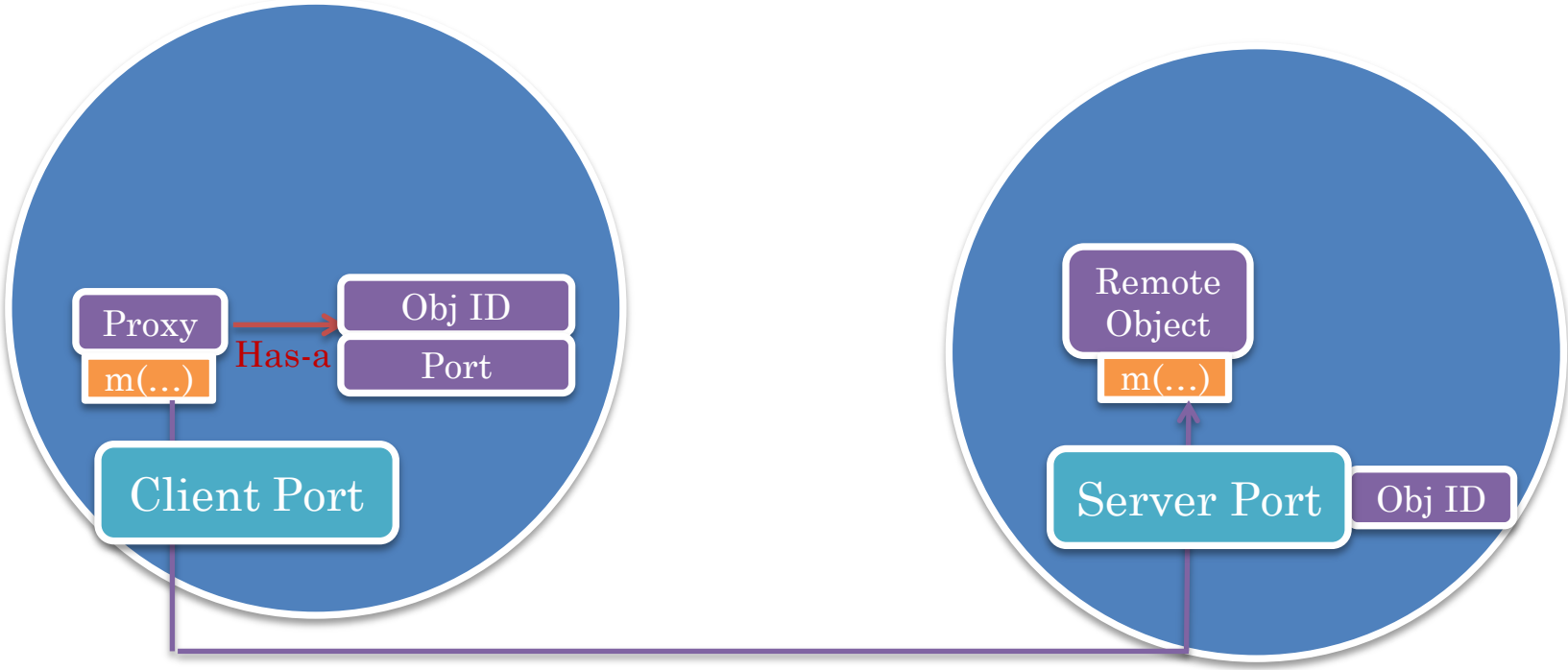
How much work should the proxy do?

Separation of concerns –do only forwarding.

Function Call Synchronization and Matched Client/Server Objects

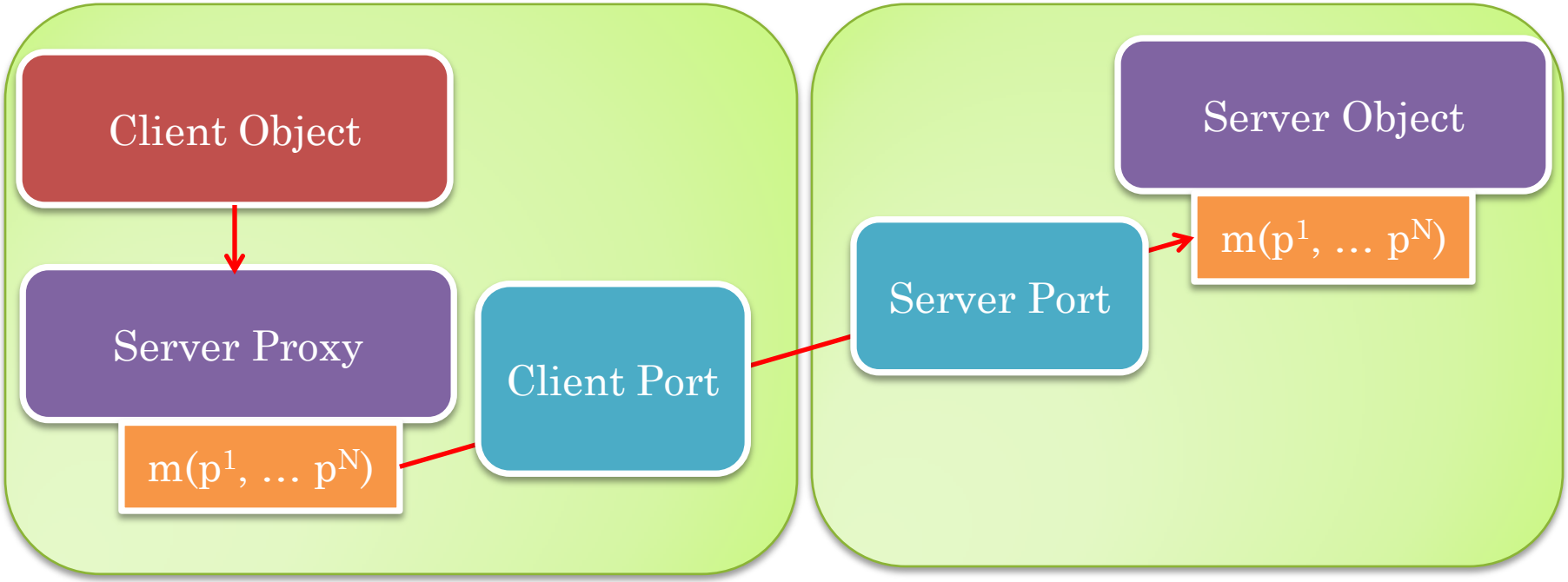


Minimum Proxy State



Stub object keeps forwarding information (remote server and remote object)

Port vs. GIPC-RMI

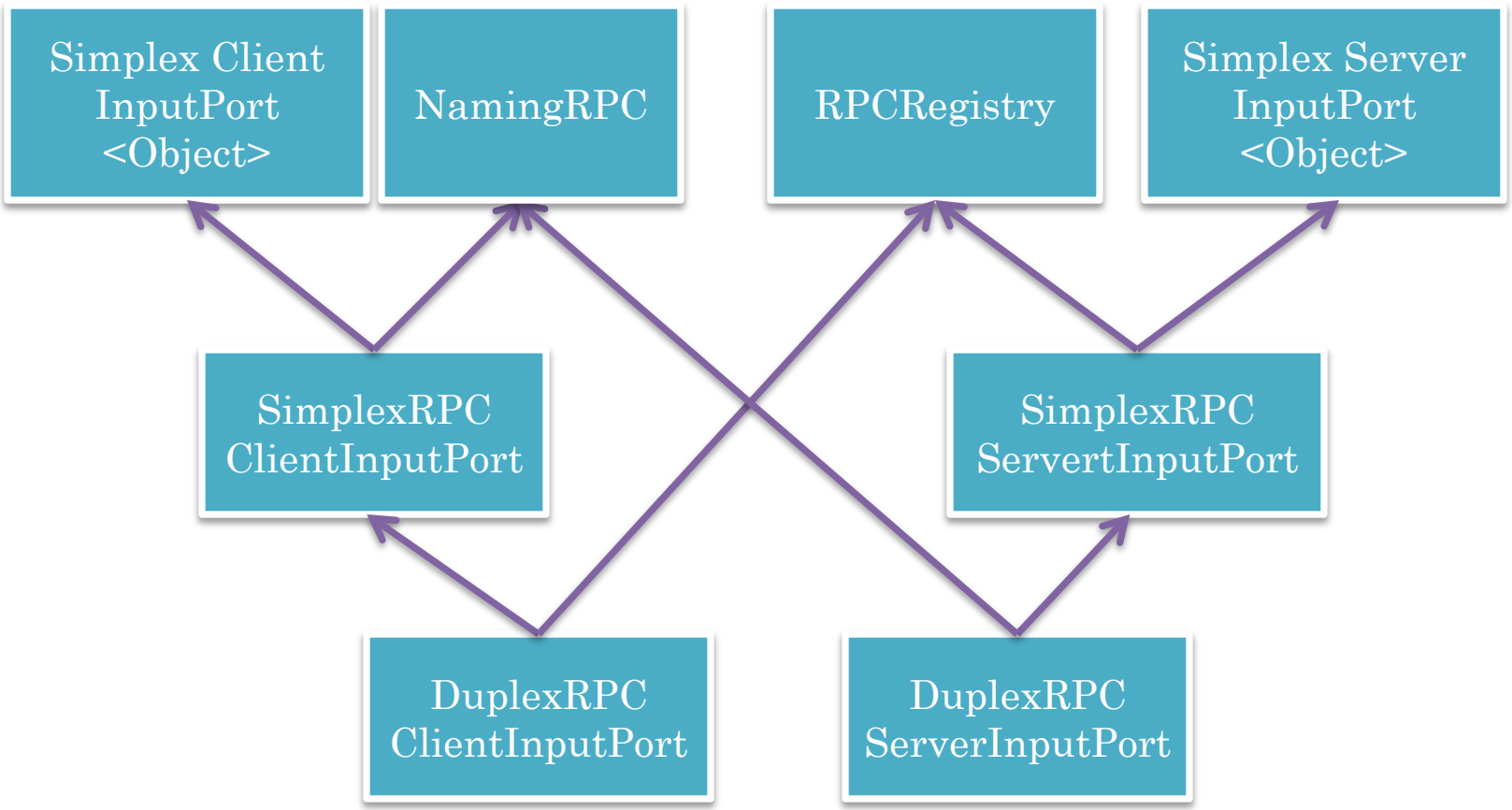


How much work should the proxy do?

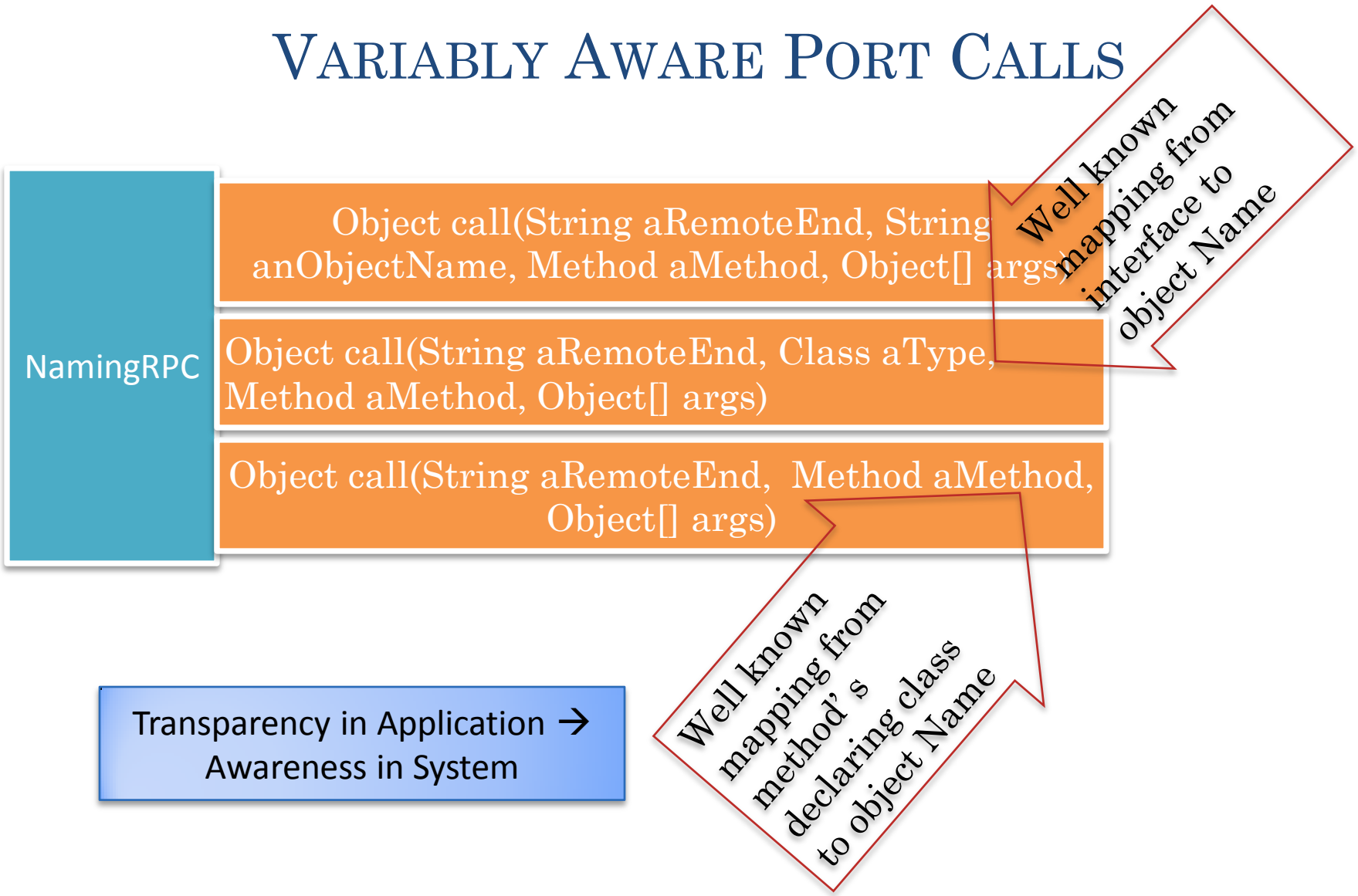
Separation of concerns –do only forwarding.

Interface between proxy and data port assuming minimal code?

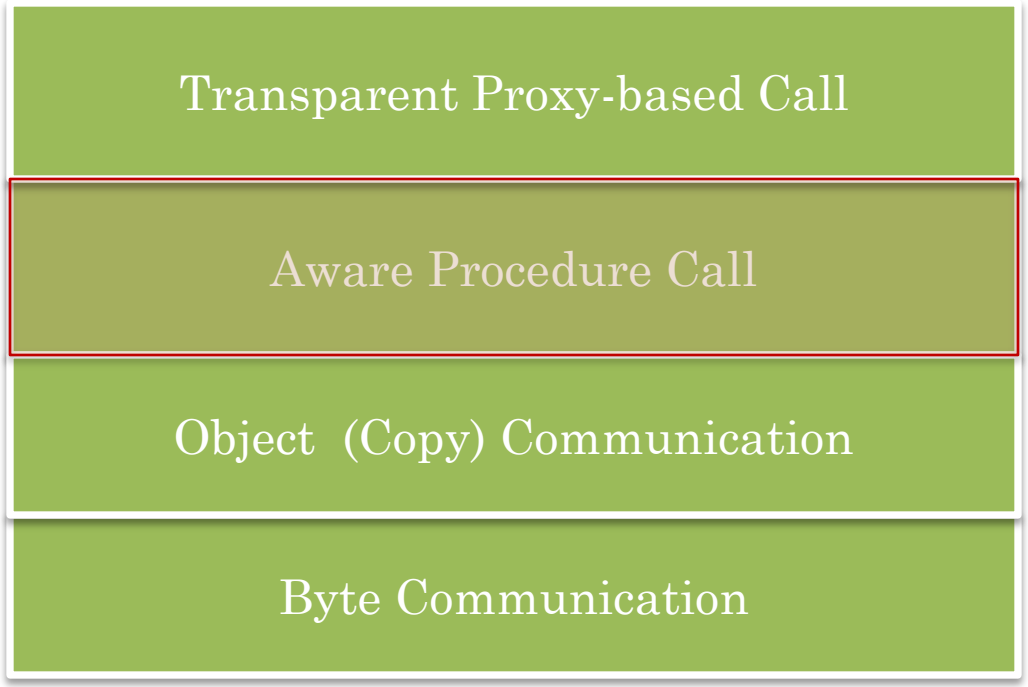
Naming RPC Port



Variably Aware Port Calls



Aware/Transparent Layers



Logic of Proxy Class

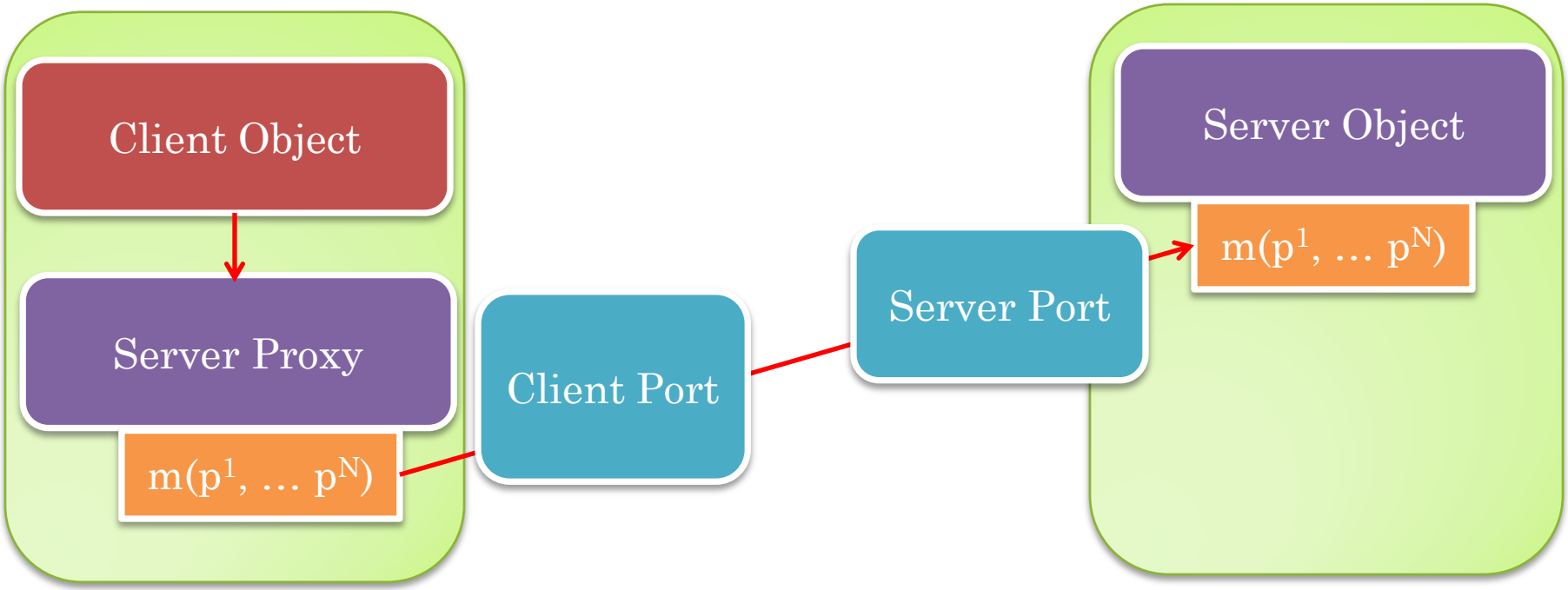
```
public class CounterStub implements Counter {
  NamingRPC rpcPort;
  String destination;
  public void init (NamingRPC anRPCPort, String aDestination) {
    rpcPort = anRPCPort;
    destination = aDestination;
  }
  public int getValue() {
    try {
      Method method = Counter.class.getMethod("getValue");
      Object[] args = {};
      return (Integer) rpcPort.call(destination,
             Counter.class.getName(), method, args);
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }
```

Logic of Proxy Class

```
public void increment(int val) throws RemoteException {
  try {
    Method method = Counter.class.getMethod("increment",
                        Integer.TYPE);
    Object[] args = {val};
    rpcPort.call(destination, Counter.class.getName(),
                    method, args);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
  Counter counter = new CounterStub();
  counter.init(rpcPort, destination);
  try {
    counter.getValue();
    counter.increment(5);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Proxy Creation?



Given by system to programmer

How?

Created by programmer

Proxy Creation

Proxies may be created from name of server object

May be obtained in arguments to method calls

In either case, we somehow need interface of remote object at the client

```
public void increment(int val) throws RemoteException {
  try {
    Method method = Counter.class.getMethod("increment",
                        Integer.TYPE);
    Object[] args = {val};
    rpcPort.call(destination, Counter.class.getName(),
                    method, args);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Generating Proxy Class: Compilation

```
set javabin=D:\"Program Files"\Java\jdk1.6.0_03\bin
cd D:/dewan_backup/java/distTeaching/bin
%javabin%\rmic  rmi.examples. ADistributedInheritingRMICounter
```

```
Directory of D:\dewan_backup\Java\distTeaching\bin\rmi\examples
11/20/2011  09:12 AM    <DIR>          .
11/20/2011  09:12 AM    <DIR>          ..
11/19/2011  08:17 PM               933  ADistributedInheritingRMICounter.class
11/20/2011  09:12 AM             1,977  ADistributedInheritingRMICounter_Stub.class
11/20/2011  09:13 AM               264  DistributedRMICounter.class
11/19/2011  07:35 PM             1,112  DistributedRMICounterClient.class
11/19/2011  06:17 PM             1,154  DistributedRMICounterServer.class
11/19/2011  08:14 PM               908  RMIRegistryStarter.class
               6 File(s)          6,348 bytes
               2 Dir(s)  125,598,871,552 bytes free
```

Pre-compiler works from object code and produces object stub code

Eclipse will delete object code it has not generated

Java RMI: Interpretive Reflection-based Class and Proxy Creation

UnicastRemote Object

static exportObject(Remote object, int port)

Proxy (Counter_Stub instance)

m(…)

Proxy Class (Counter_Stub)

m(…)

Remote Object ADistributedInheritingCounter

m(…)

Creates a proxy object for the remote object at the server end that can later be sent to client

If the stub class for the proxy had not been created so far, then it is conceptually created at runtime

Generated vs. Interpreted

Pure Generation	Interpretation
Pre(compiler) generates proxy class	At run time, a proxy class and object is created
Efficient	No compile-run-cycle delay
	No inconsistency as code evolves

Runtime Abstraction?

Interpretation

At run time, a proxy class and object is created

RMI system must generate application-specific proxy classes

Runtime abstraction for generation?

Higher-Level Abstraction for Proxy

```
public int getValue() {
    try {
      Method method = Counter.class.getMethod("getValue");
      Object[] args = {};
      return (Integer) rpcPort.call(destination,
              Counter.class.getName(), method, args);
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }
```

```
public void increment(int val) throws
  try {
    Method method = Counter.class.getM
                        Integer.TYPE);
    Object[] args = {val};
    rpcPort.call(destination, Counter.class.getName(),
                    method, args);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Reflection and dynamic class generation and loading?

Higher-level abstraction for proxy generation and loading?

Higher-Level Abstraction for Proxy

```
public int getValue() {
    try {
      Method method = Counter.class.getMethod("getValue");
      Object[] args = {};
      return (Integer) rpcPort.call(destination,
              Counter.class.getName(), method, args);
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }
```

```
public void increment(int val) throws RemoteException {
  try {
    Method method = Counter.class.getMethod("increment",
                      Integer.TYPE);
    Object[] args = {val};
    rpcPort.call(destination, Counter.class.getName(),
                  method, args);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Generatable Pattern

App-specific

Dispatch to an app-specific object

Dispatching Code

```
public int getValue() {
    try {
      Method method = Counter.class.getMethod("getValue");
      Object[] args = {};
      …
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }
```

Create method object representing the generated method

Dispatch the method to an App-Specific Object

```
public void increment(int val) throws Remot
  try {
    Method method = Counter.class.getMethod("increment",
                                Integer.TYPE);
    Object[] a
    …
} catch (Excep
    e.printSta
  }
}
```

Class

getMethod(String name, Class[] parameterTypes)

Invocation Handler

Object invoke(Object proxy, Method method, Object[] args)

Proxy and Invocation Handler

Proxy

static Class getProxyClass(ClassLoader loader, Class[] interfaces)

static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)

Internal name derived from interfaces so duplicate classes not created

Invocation Handler

Object invoke(Object proxy, Method method, Object[] args)

Why class loader?

Stores (unnamed) class object for this and later references

Simple Proxy Usage

```
Counter counter = (Counter)
        Proxy.newProxyInstance(
           Counter.class.getClassLoader(),
           new Class [] {Counter.class},
           invocationHandler );
```

Proxy for a list of interfaces

Invocation handler takes instance-specific constructor parameters

Proxy and Invocation Handler (Review)

Proxy

static Class getProxyClass(ClassLoader loader, Class[] interfaces)

static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)

Internal name derived from interfaces so duplicate classes not created

Invocation Handler

Object invoke(Object proxy, Method method, Object[] args)

Why class loader?

Stores (unnamed) class object for this and later references

Simple Proxy Usage (Review)

```
Counter counter = (Counter)
        Proxy.newProxyInstance(
           Counter.class.getClassLoader(),
           new Class [] {Counter.class},
           invocationHandler );
```

Proxy for a list of interfaces

Invocation handler takes instance-specific constructor parameters

AnAbstractRPCProxyInvocationHandler

```
public AnAbstractRPCProxyInvocationHandler(
                                        SimplexRPC anRPCPort,
  String aDestination, Class aType, String aName) {
  rpcInputPort = anRPCPort;
  destination = aDestination;
  remoteType = aType;
  name = aName;
}
public Object invoke(Object arg0, Method method,
                     Object[] args) {
  try {
    if (name != null)
      return call(destination, name, method, args);
    else if (remoteType != null) {
      return call(destination, remoteType, method, args);
    } else {
      return call(destination, method, args);
    }
  } catch (Exception e) {
    e.print
    return
  }
}
```

Transfers invoke to abstract port calls

AnRPCProxyInvocationHandler

```
protected Object call(String aDestination, String aName,
                      Method aMethod, Object[] args) {
    Object retVal = rpcInputPort.call(aDestination, aName,
                                      aMethod, args);
    return retVal;
}
```

Proxy Generator Example

```
public class StaticRPCProxyGenerator {
  public static Object generateRPCProxy(SimplexRPC aPort,
        String aDestination, Class aClass, String anObjectName) {
    Class[] remoteInterfaces =
         ReflectionUtility.getProxyInterfaces(aClass);
    InvocationHandler invocationHandler = new
         AnRPCProxyInvocationHandler(aPort, aDestination, aClass,
                                     anObjectName);
     return Proxy.newProxyInstance(aClass.getClassLoader(),
                         remoteInterfaces, invocationHandler);
  }
}
```

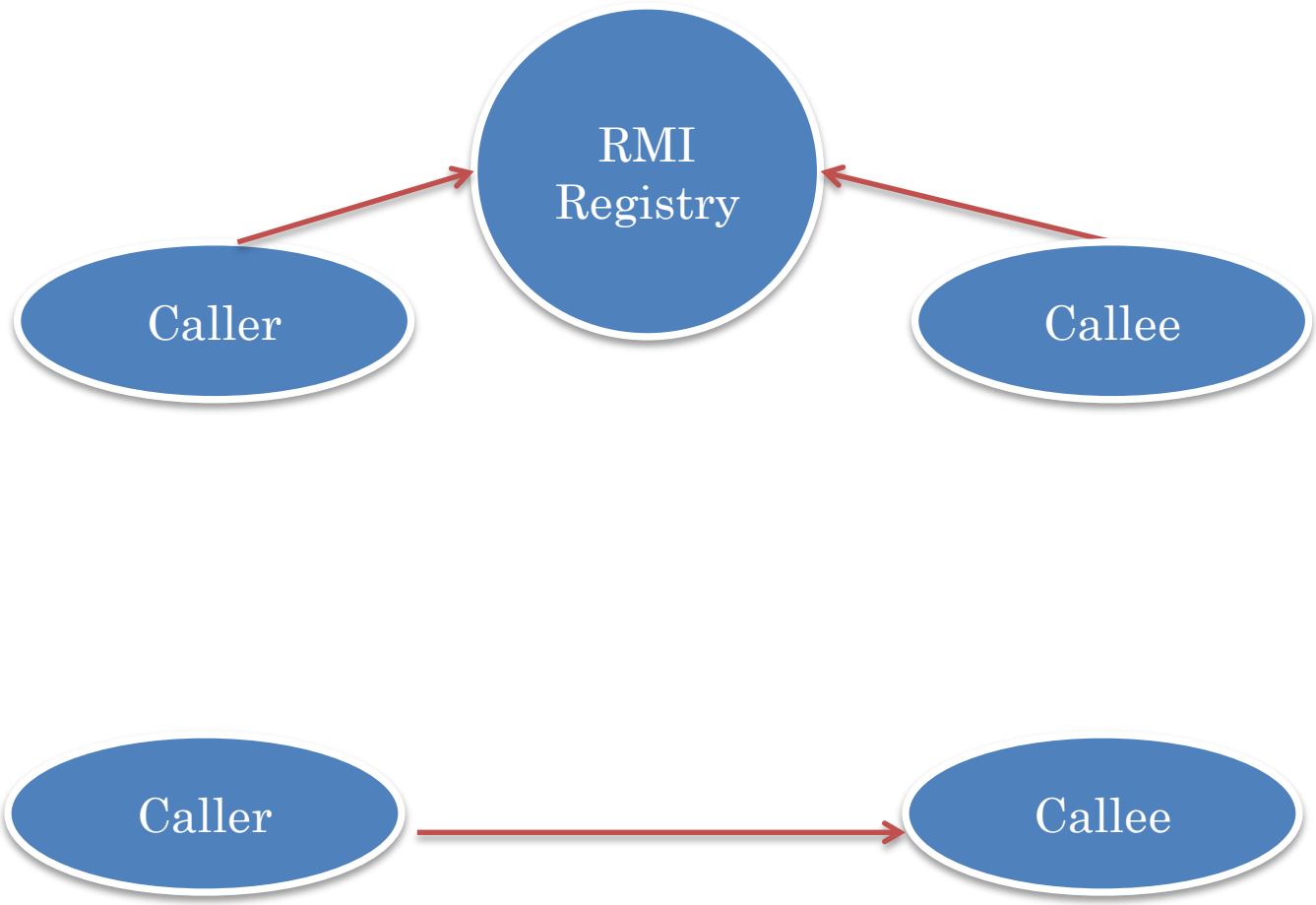
In GIPC, Proxy methods generated for all interfaces implemented by the class of the object and of the superclasses including Object

In RMI only the remote interfaces

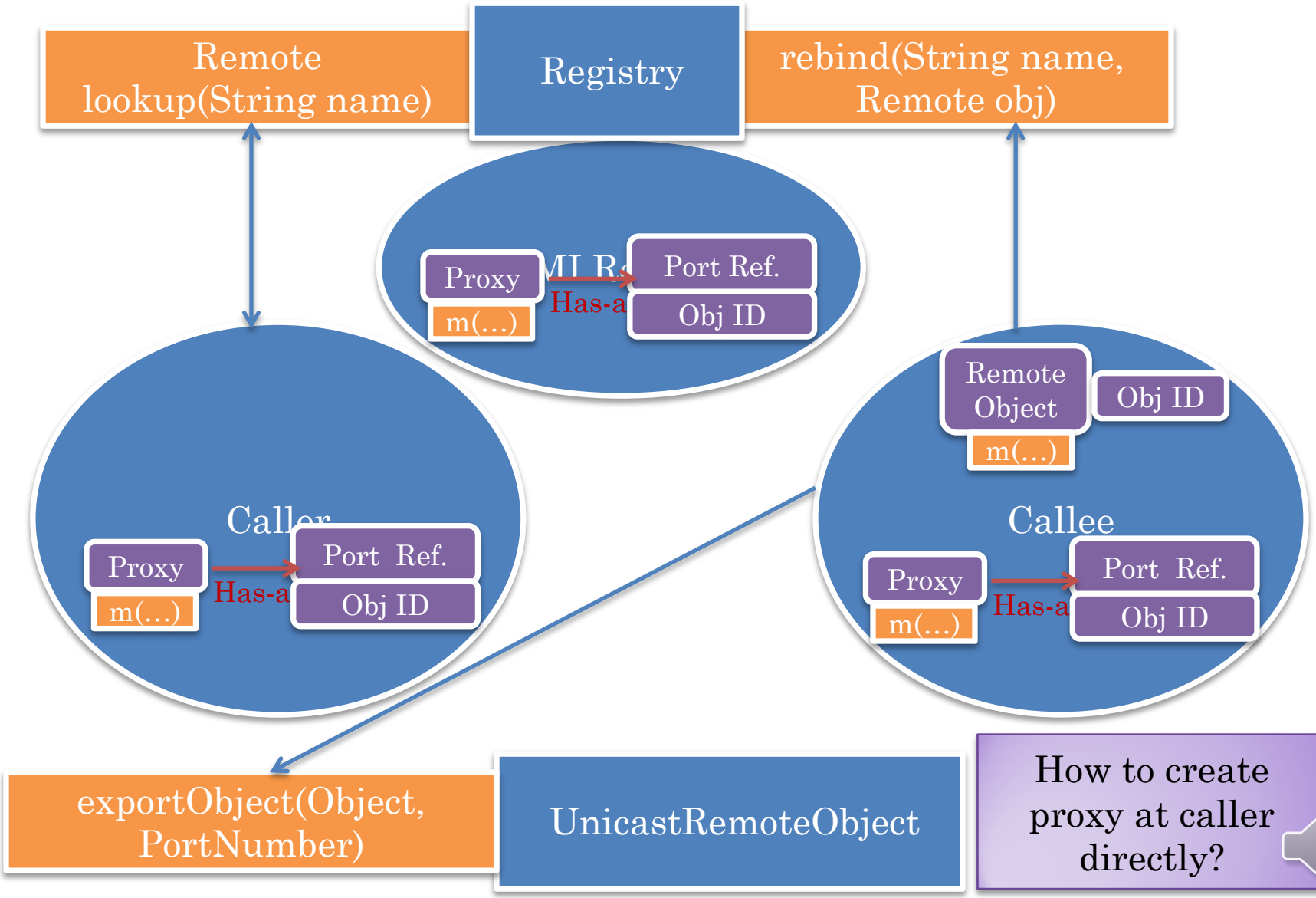
Basic idea of proxy generation the same

This code not to be directly used by programmer

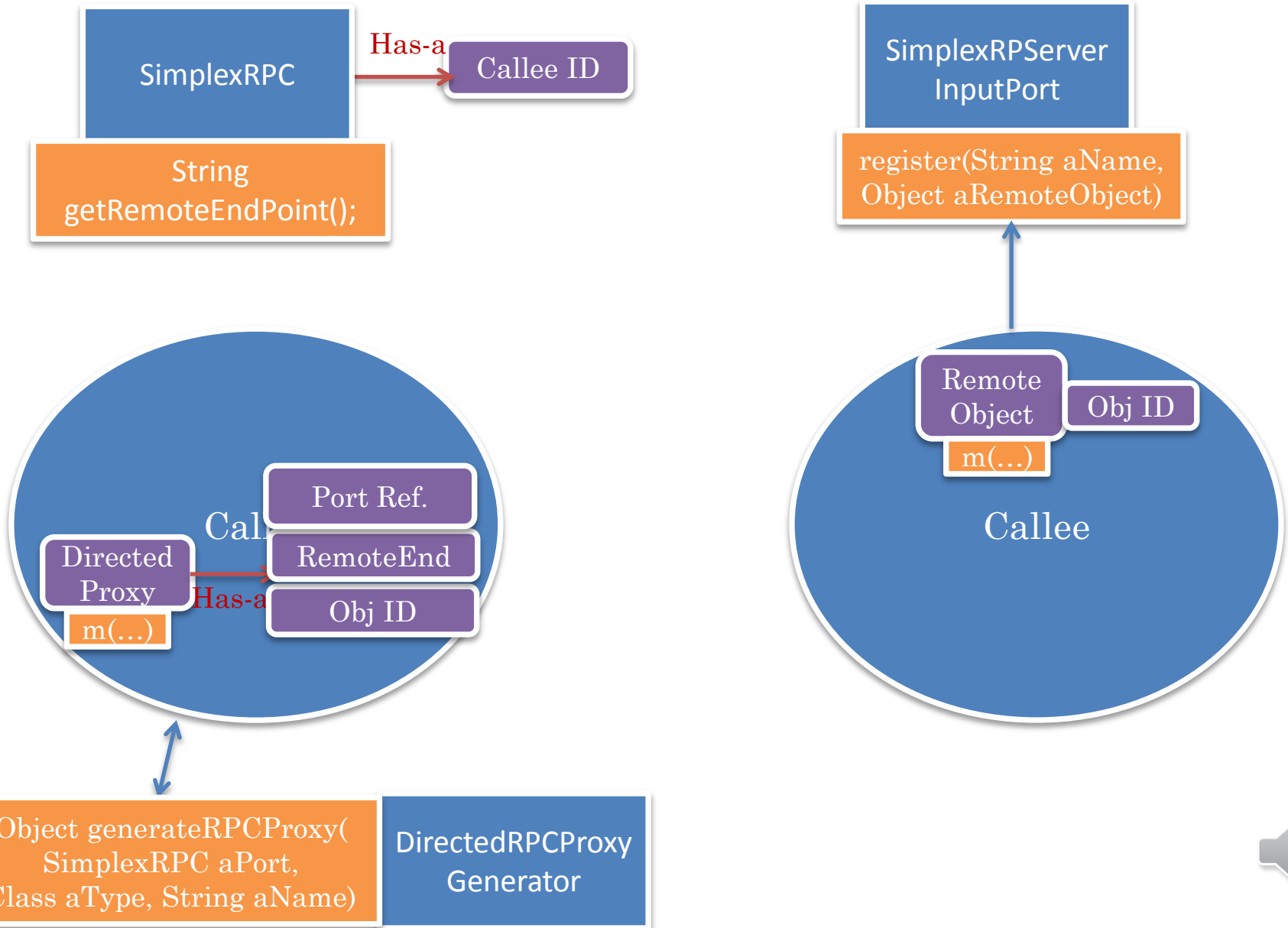
RMI vs. GIPC Binding Time



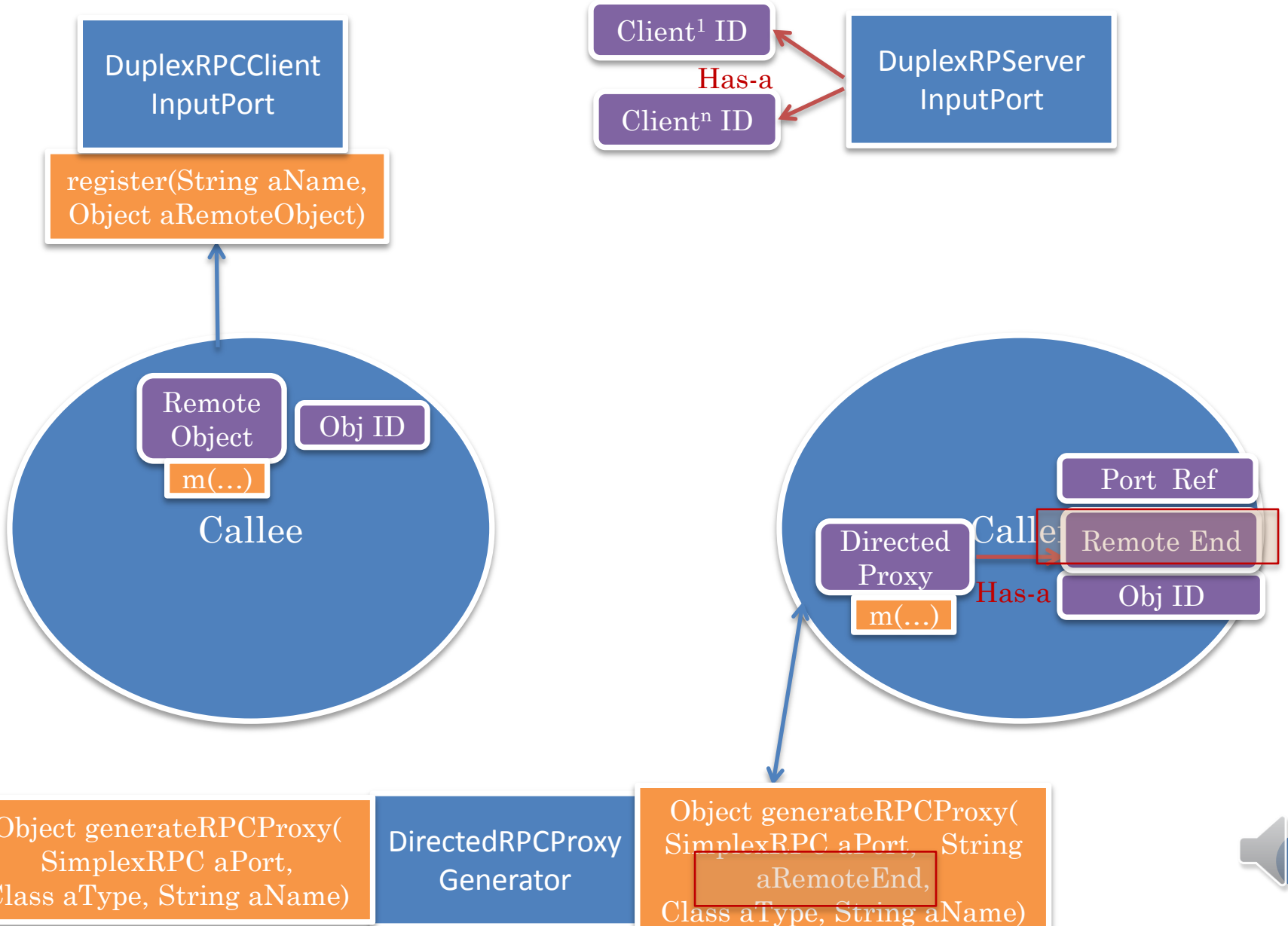
RMI Binding


Remote
lookup(String name)
Registry
rebind(String name,
Remote obj)
Proxy
m(…)
Has-a
Port Ref.
Obj ID
Caller
Proxy
m(…)
Has-a
Port Ref.
Obj ID
Remote
Object
m(…)
Obj ID
Callee
Proxy
m(…)
Has-a
Port Ref.
Obj ID
exportObject(Object,
PortNumber)
UnicastRemoteObject
How to create
proxy at caller
directly?

Direct Server Proxy



Directed Client Proxy



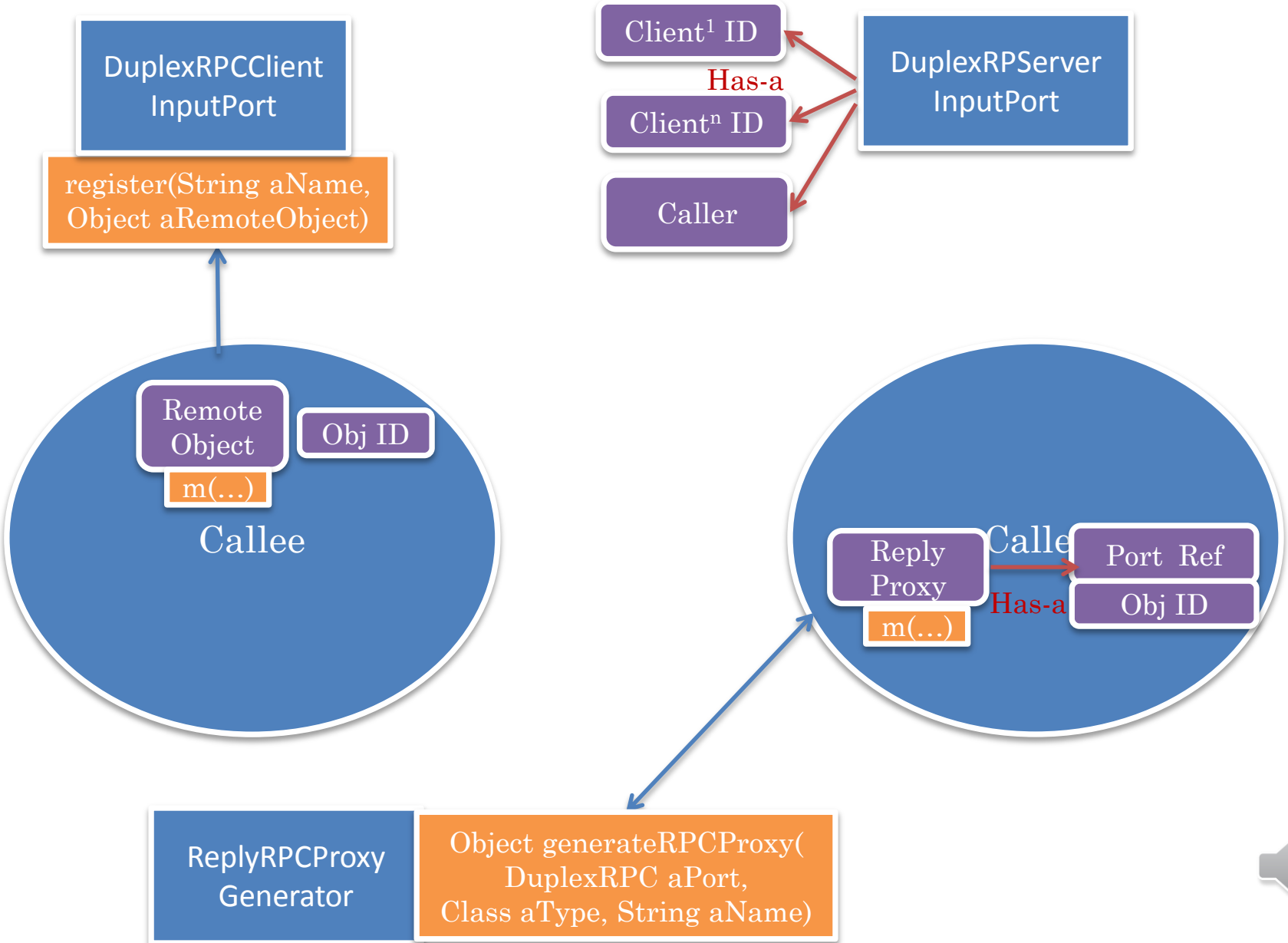
RMI: Replying to Explicitly Looked up Proxy

```
public class AMultiUserRMIUpperCaser extends ADistributedRMIUpperCaser
implements MultiUserRMIUpperCaser {
  protected Map<String, DistributedRMICounter> nameToCounter =
     new HashMap();
  protected DistributedRMICounter getCounterProxy(String aUserName) {
    try {
      DistributedRMICounter counter = nameToCounter.get(aUserName);
      if (counter == null) {
        Registry rmiRegistry = LocateRegistry.getRegistry();
        counter = (DistributedRMICounter) rmiRegistry.lookup(
          aUserName + DistributedRMICounter.class.getName());
        nameToCounter.put(aUserName, counter);
      }
      return counter;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }
```

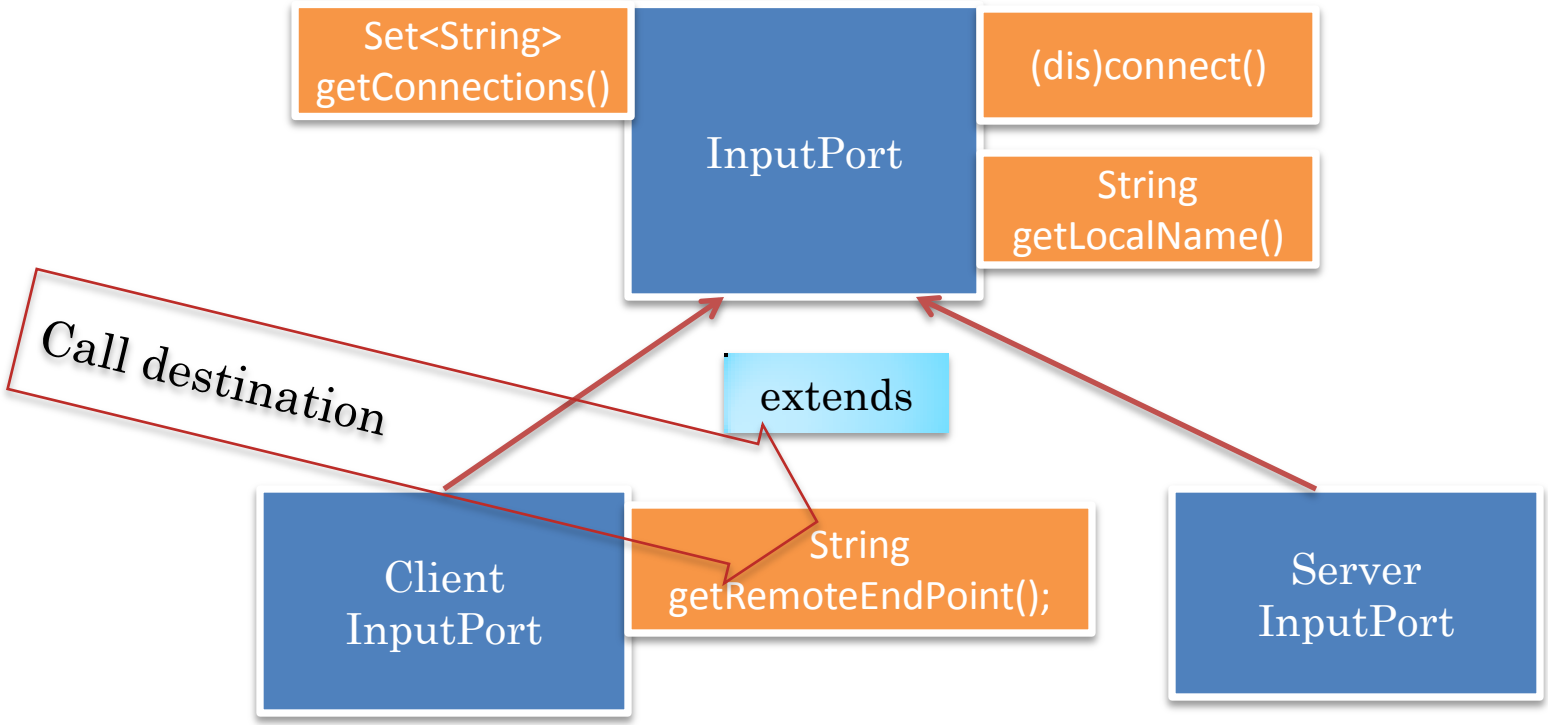
In RMI, user name must be passed to create proxy in server object to make a callback

In GIPC, proxy can be created by server launcher and bound to caller to make callback

Reply Proxy


DuplexRPCClient InputPort
register(String aName, Object aRemoteObject)
Client[1] ID
Has-a
Client[n] ID
Caller
DuplexRPServer InputPort
Remote Object
Obj ID
m(…)
Callee
Reply Proxy
Has-a
m(…)
Port Ref
Obj ID
ReplyRPCProxy Generator
Object generateRPCProxy(DuplexRPC aPort, Class aType, String aName)

Client Directed Server Generator

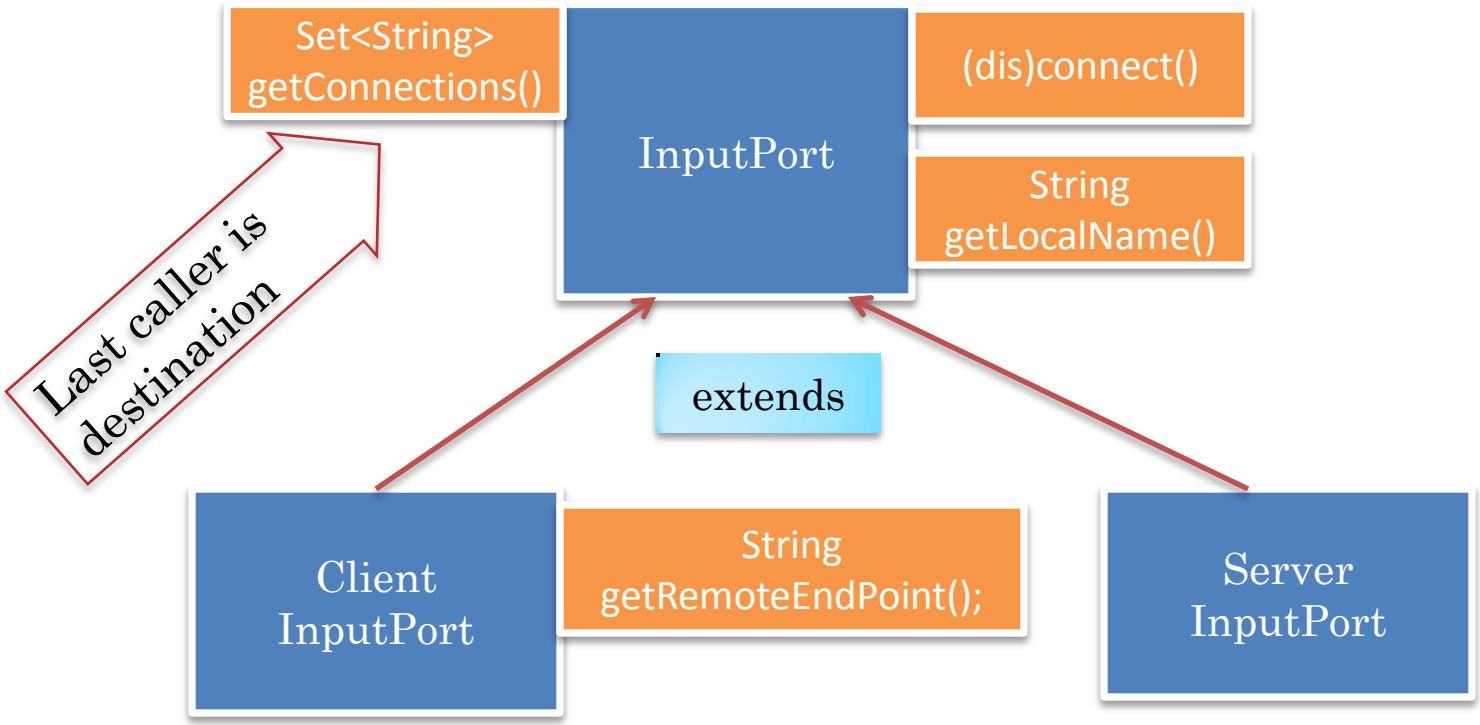


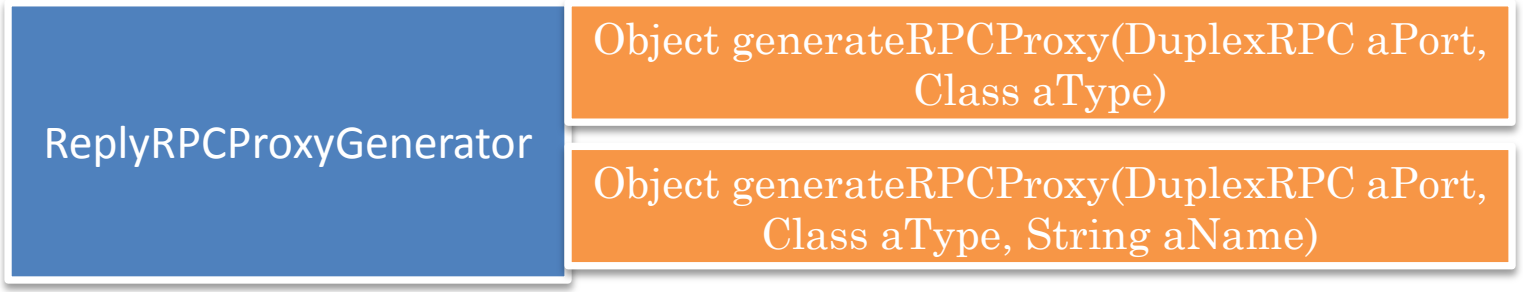
DirectedRPCProxy Generator

Object generateRPCProxy(SimplexRPC aPort, Class aType)

Object generateRPCProxy(SimplexRPC aPort, Class aType, String aName)

Server Reply Undirected Server Generator





Manual Group Communication

```
public class ACollaborativeRMIUpperCaser extends AMultiUserRMIUpperCaser
             implements CollaborativeRMIUpperCaser {
  public void connect(String aClientName) {
    // need to register the counter
    getCounterProxy(aClientName};
  }
  protected Object[] getAllCounters
    Object[] retVal = new Object[na
    int index =0;
    for (String aClient:nameToCounter.keySet()) {
      try {
        retVal[index] = nameToCounter.get(aClient).getValue();
        index++;
     } catch (Exception e) {
        e.printStackTrace();
     }
    }
    return retVal;
  }
```

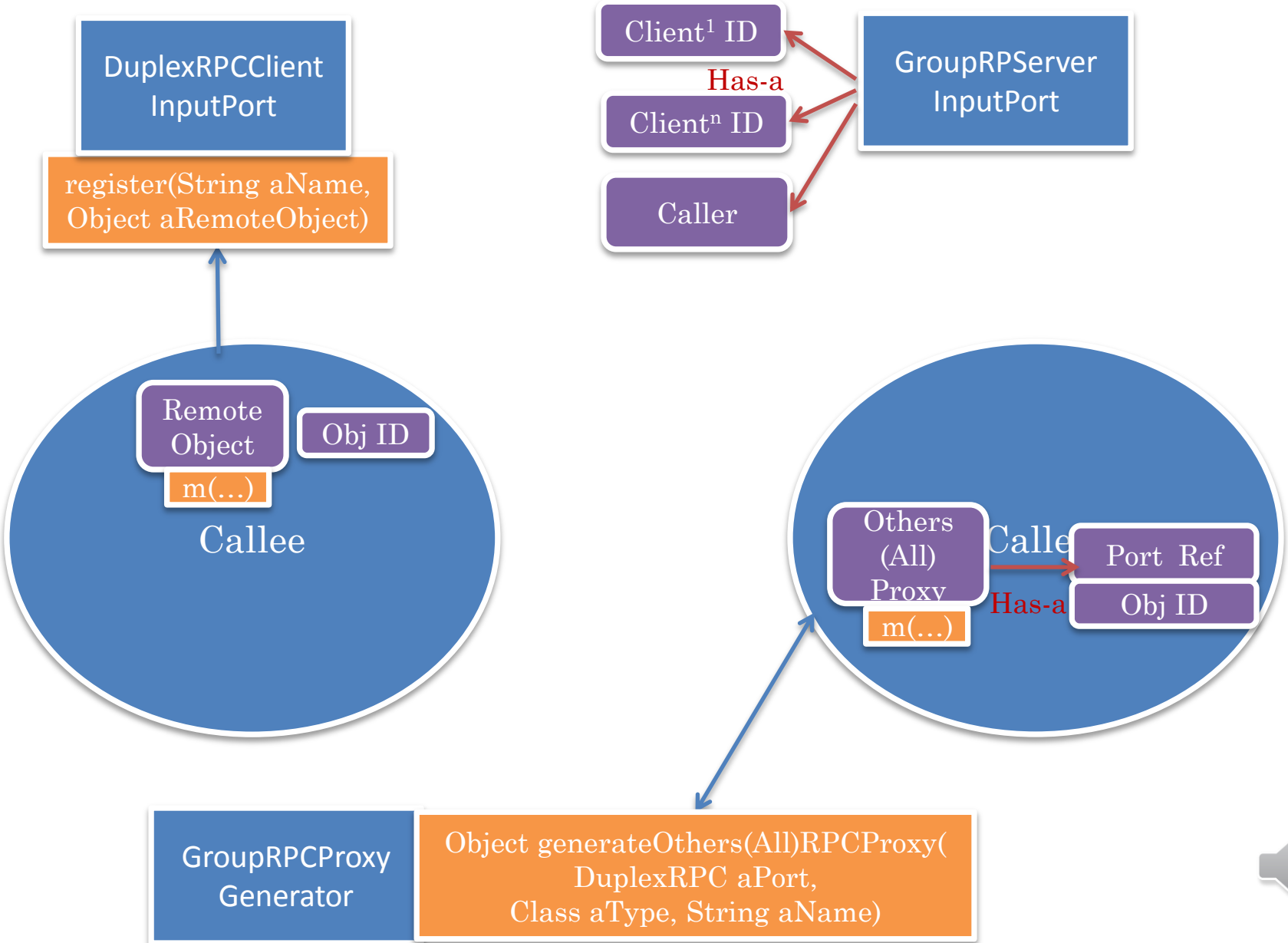
Application supports connect call

GIPC supports connect call

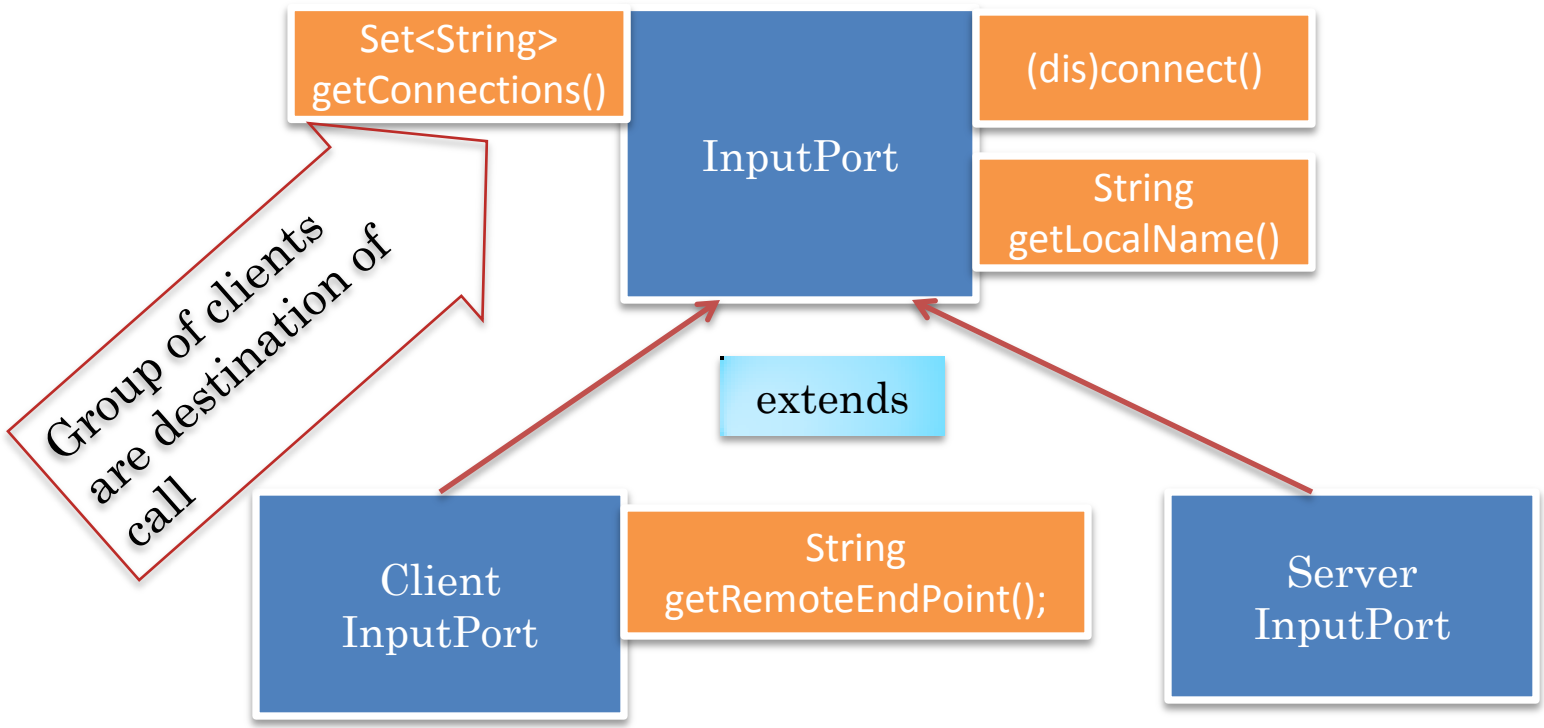
Application code to get all counter values

GIPC late binding of proxy could be used to provide multicast

Others and All Proxy

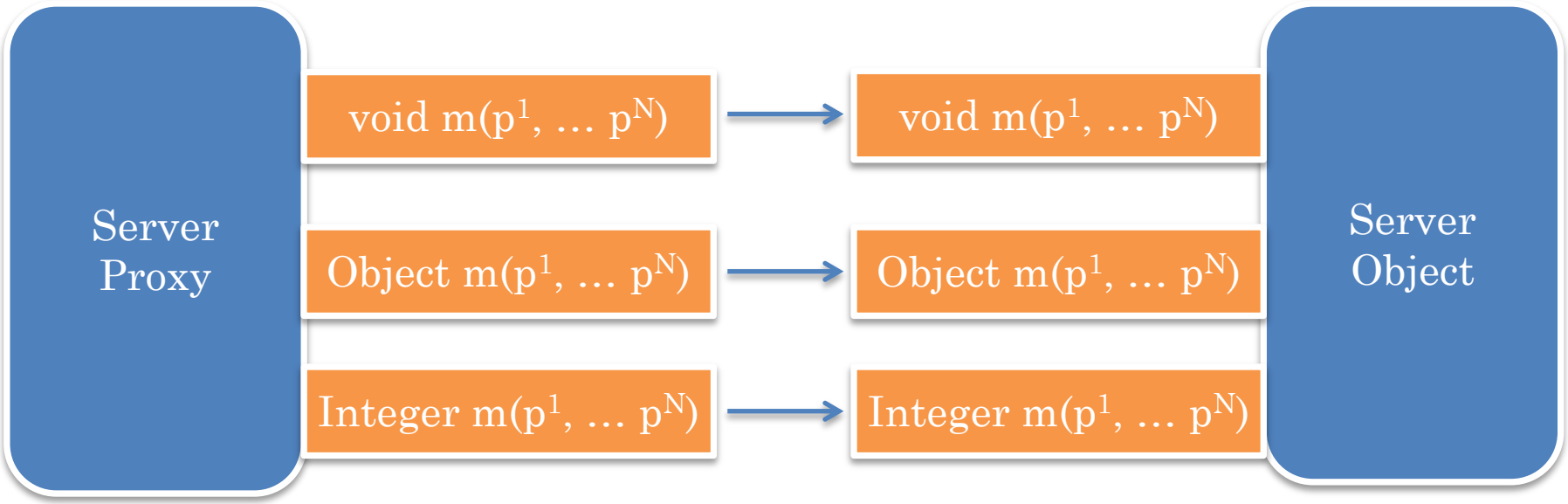


Server Reply Undirected Server Generator



GroupRPCProxy Generator	Object generateOthersRPCProxy(GroupRPC aPort, Class aType)
	Object generateAllRPCProxy(GroupxRPC aPort, Class aType, String aName)

Return value in Group Calls



GIPC returns an array of return values

Otherwise can be cast to Object[]

Exception if return value is not an Object

(Object[]) m(p^1, … p^N)

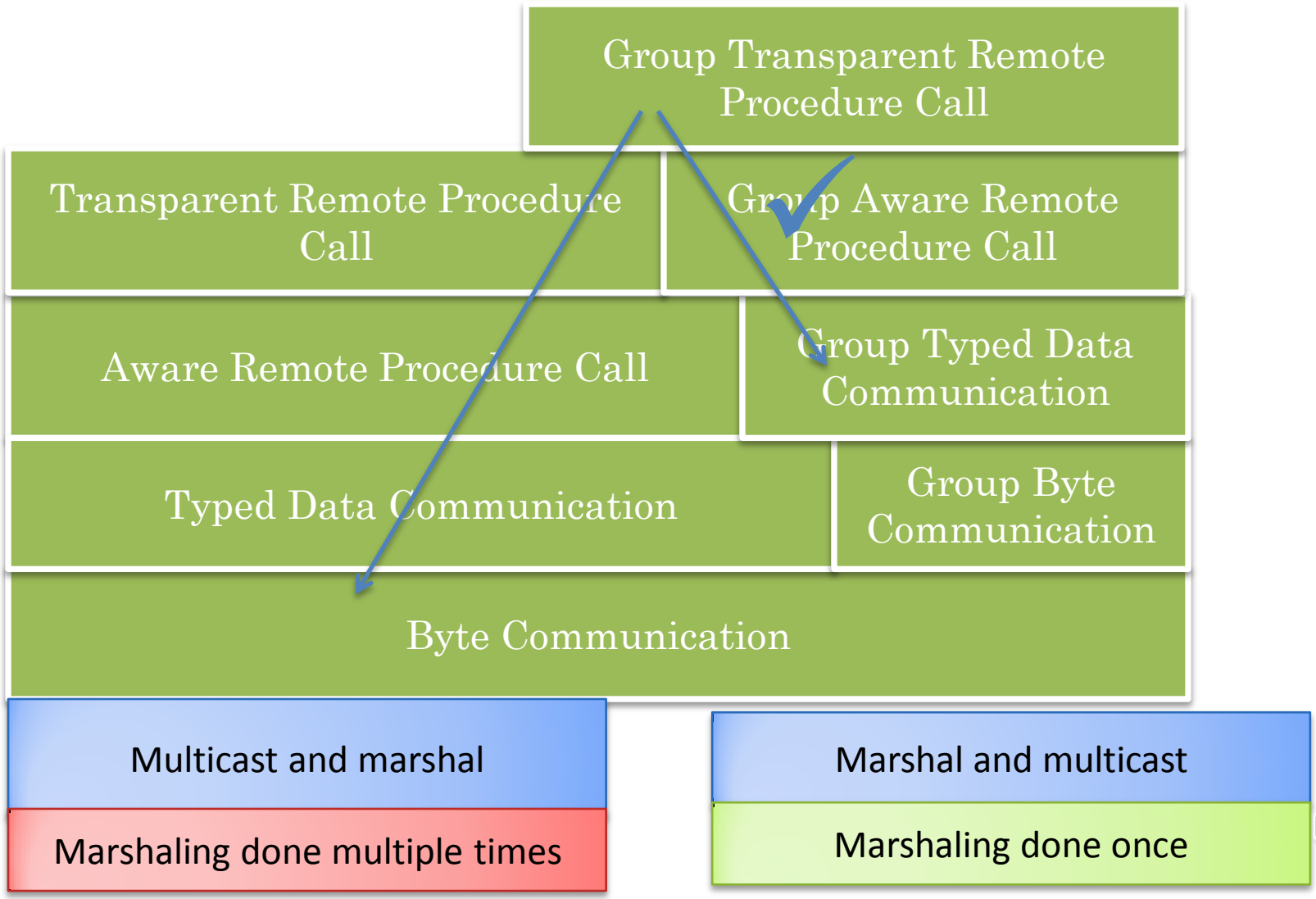
Summary

- In GIPC, a proxy for a remote object is generated by the caller rather than fetched from a server.
- Proxy can be generated before the remote object is registered.
- This late binding allows a client proxy:
  - referred by a remote call in a server object to be bound to the client who made the call. Such a proxy is a reply proxy.
  - referred by a remote call in a server object to be bound to all clients but the one who made the call. Such a proxy is an others proxy,
  - to be bound to be bound to all clients. Such a proxy is an all proxy.
  - To be bound to a specific client or server. Such a proxy is a directed proxy.
  - when a proxy is generated, the application must indicate the kind of proxy.
  - a non void call made in an others or group proxy returns an array of objects. Such a call must return an Object.
- A single communication channel between a client and server can be used for proxy calls in both directions. In RMI a separate communication channel must be created for the server and each client

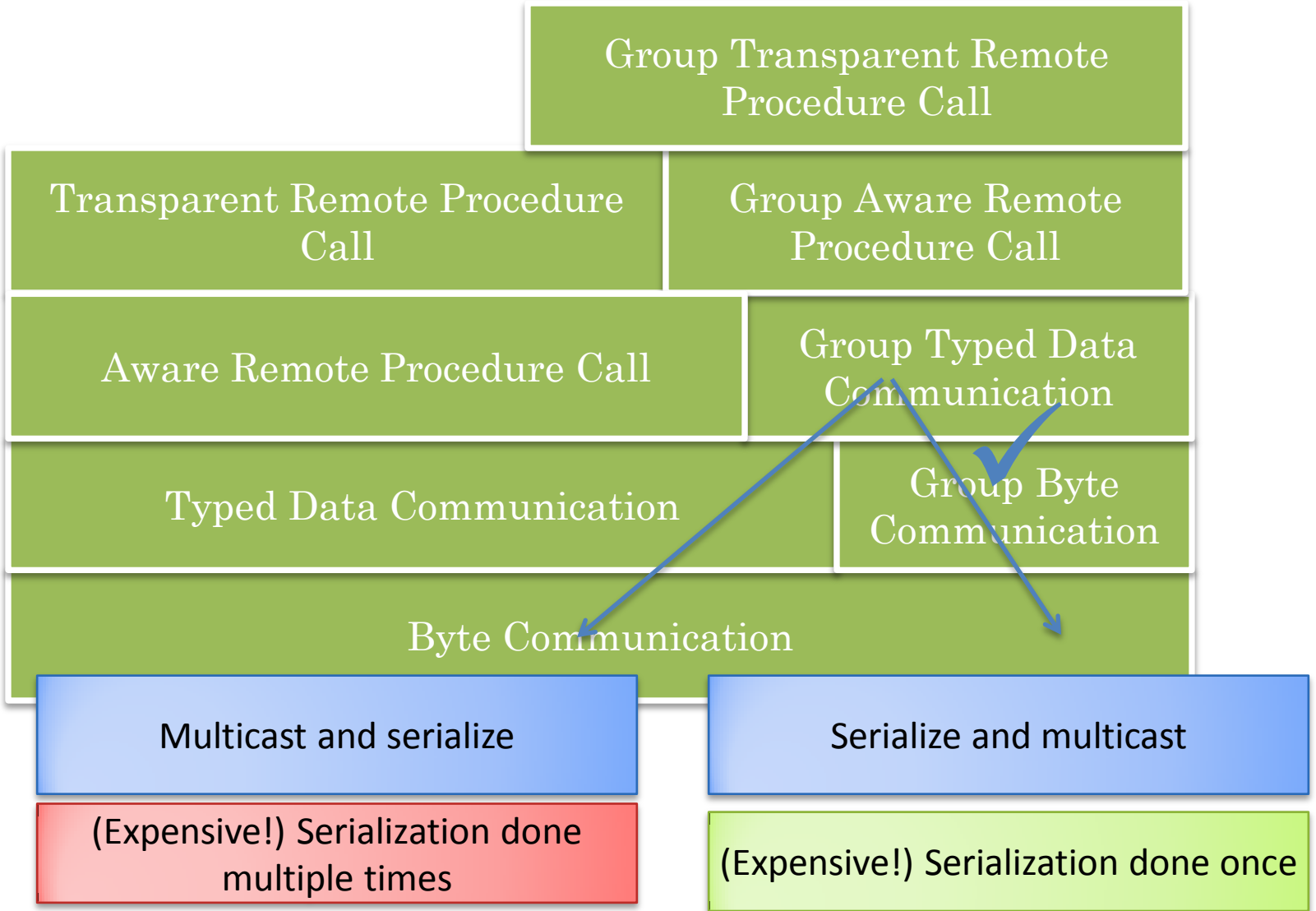
Implementing Group Calls in System

```
protected Object[] getAllCounters()  {
    Object[] retVal = new Object[nameToCounter.keySet().size()];
    int index =0;
    for (String aClient:nameToCounter.keySet()) {
      try {
        retVal[index] = nameToCounter.get(aClient).getValue();
        index++;
     } catch (Exception e) {
       e.printStackTrace();
     }
    }
    return retVal;
  }
```

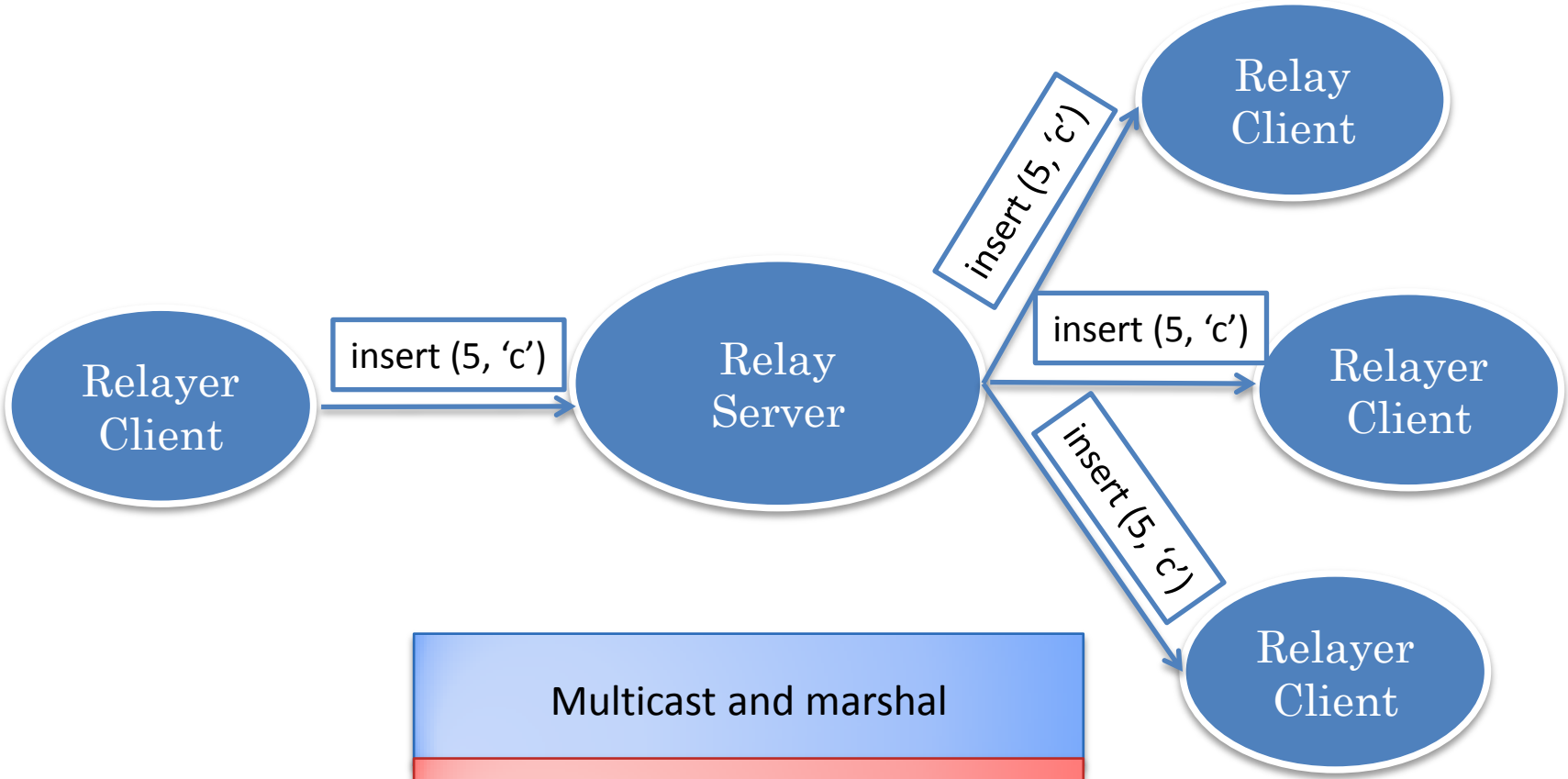
Implementation Paths



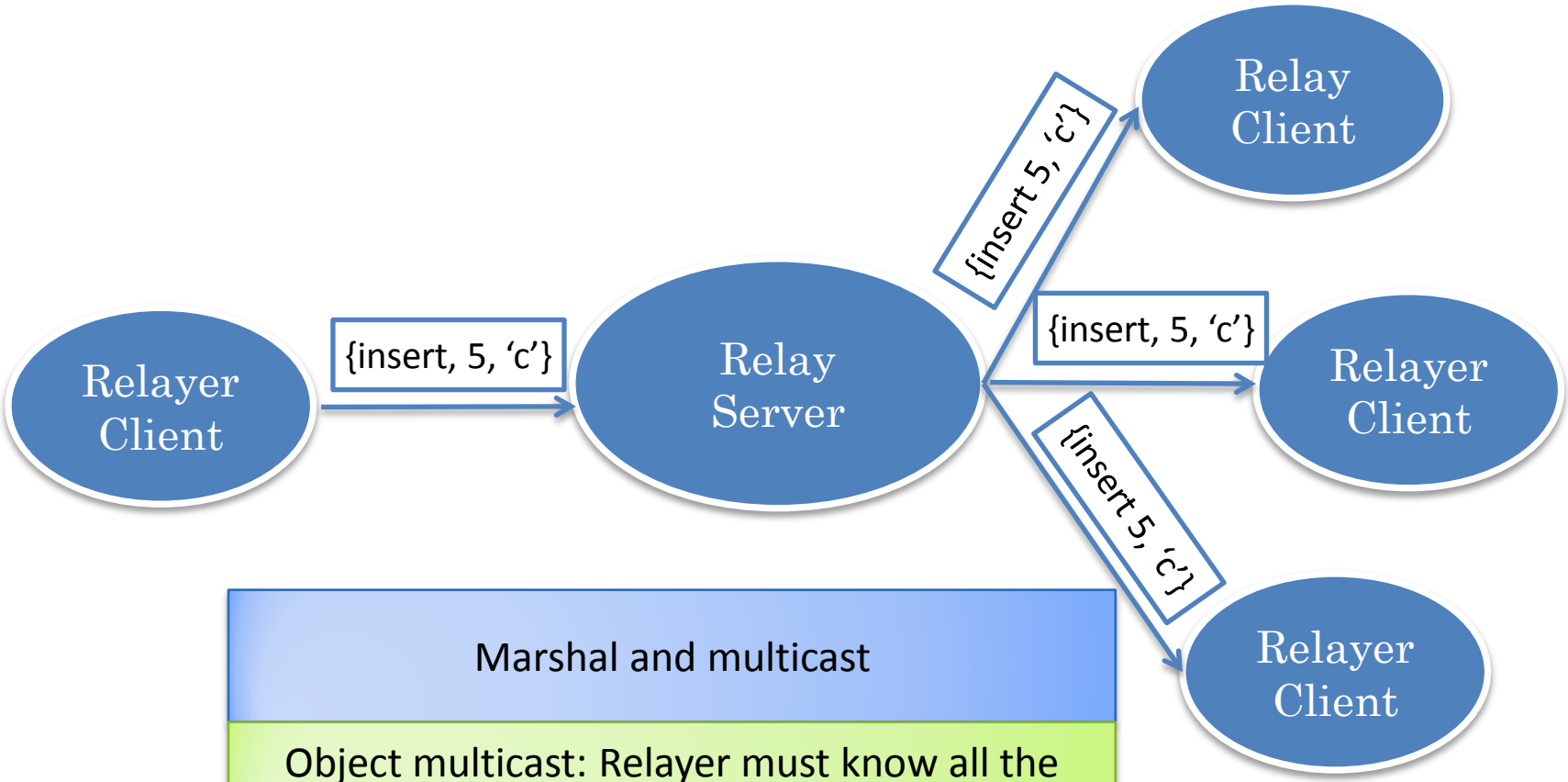
Multicast and Serialize



Relaying Method Calls



Relaying Method Calls



Marshal and multicast

Object multicast: Relayer must know all the parameter types for serialization

ByteBuffer multicast: Relayer can be truly generic

Group Communication

- A group rpc all is marshalled once by sending the marshalled data once through the group object communication layer.
- A group data object send is serialized once by sending the serialized value once through the group buffer communication layer

Proxy Creation

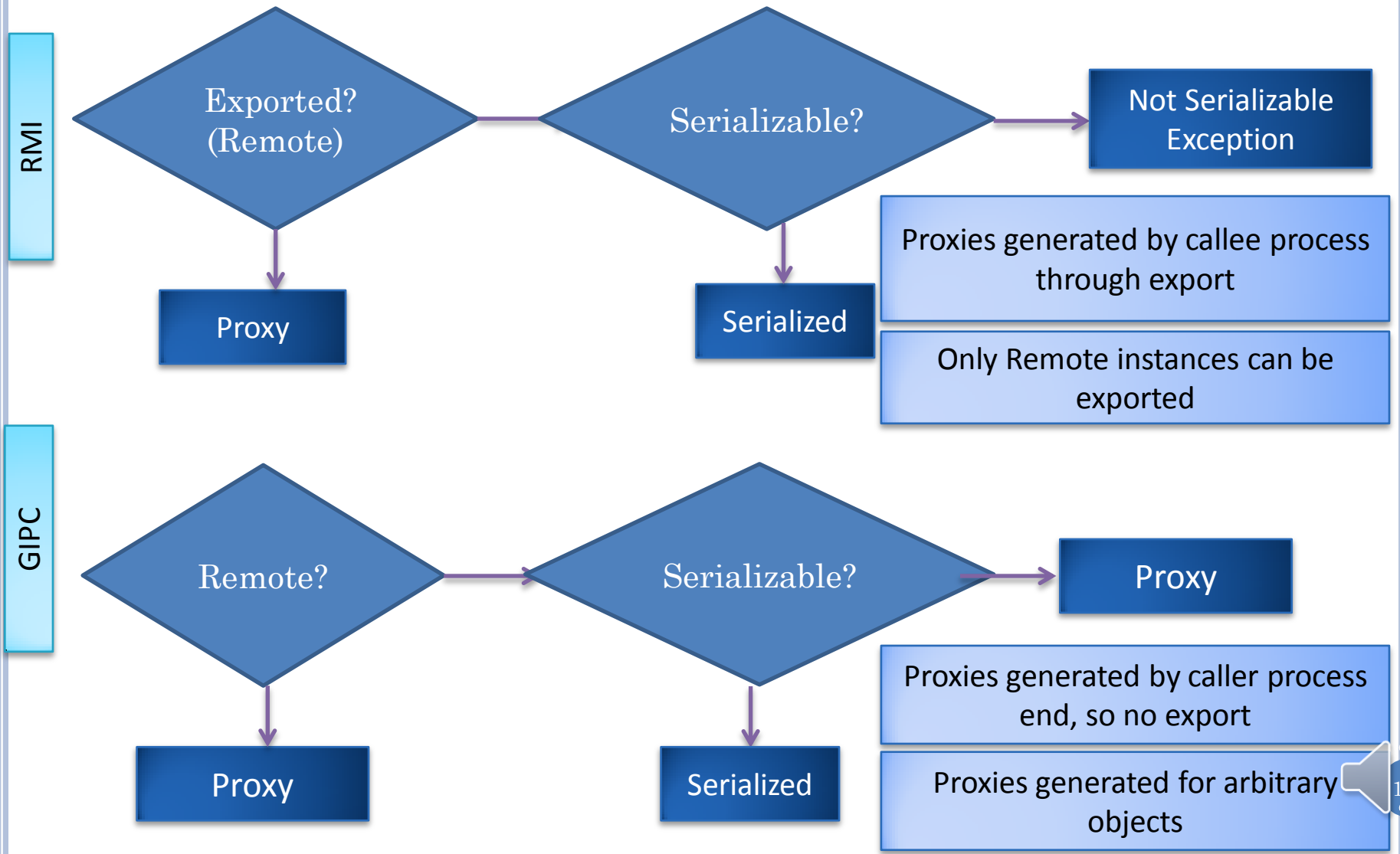
Proxies may be created from name of server object

May be obtained in arguments to method calls

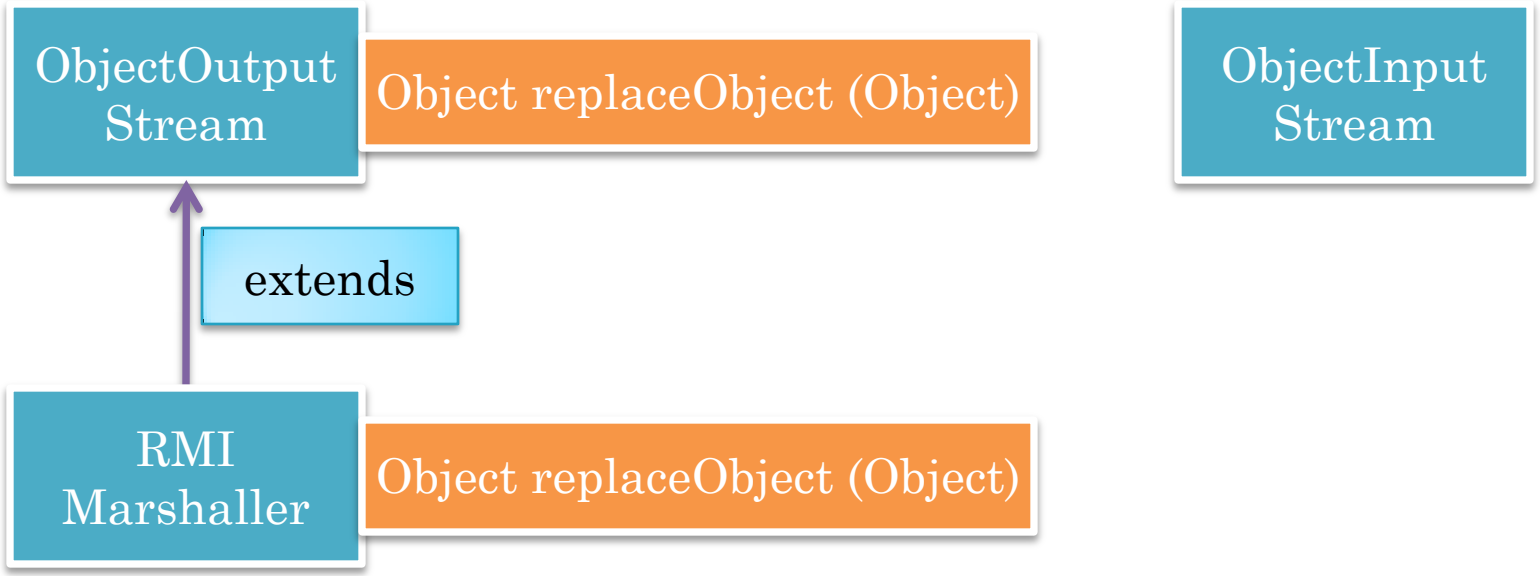
What should happen when a parameter is passed to a remote method or a return value is received?

Copy or Proxy? RMI vs. GIPC

Should a parameter to a remote method be passed as a reference or a serialized copy?



Serialization →Marshalling



ObjectOutputStream calls replaceObject(object) to determine what object is actually serialized and sent

RMI Marshaller returns stub if object IS-A Remote and has been exported (at compile or runtime)

ObjectInputStream uses stub or copy

Marshaller and Serializer are tied to each other through inheritance

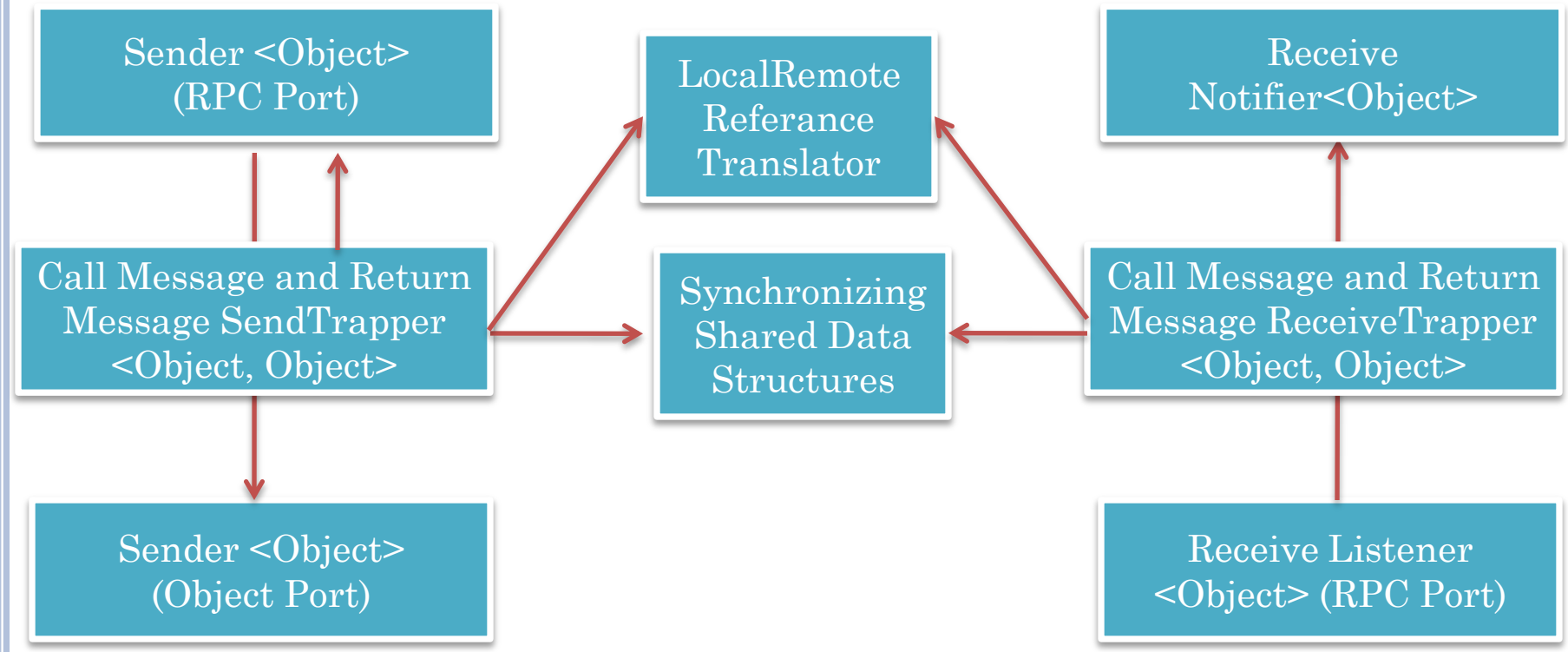
LocalRemoteReferenceTranslator

LocalRemote Reference Translator

- Object transformReceivedReference(Object possiblyRemote)
- Object transformSentReference (Object possiblyRemote)
- void transformReceivedReferences(Object[] args)
- void transformSentRemoteReferences(Object[] args)
- Object getProxy(Object remoteSerializable)
- connectRemoteAndRemoteSerializable(Object remoteSerializable, Object remote)
- Object createRemoteSerializable (String remoteEndName, String aClass, String anObjectName)
- RemoteSerializable getRemoteSerializable(Object getProxy)

Delegation analogue of Java ReplaceObject overriding

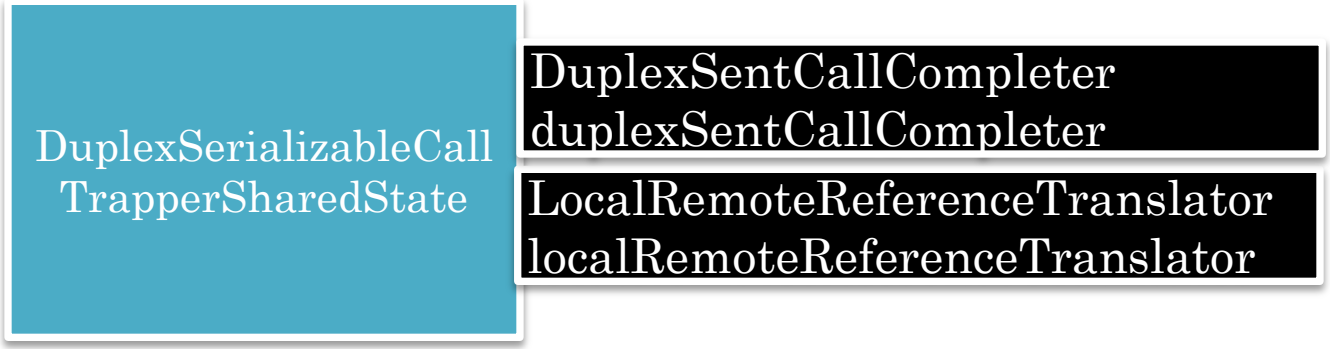
Object Forwarders and Serializers



Send trapper may need to block sender to implement synchronous operation

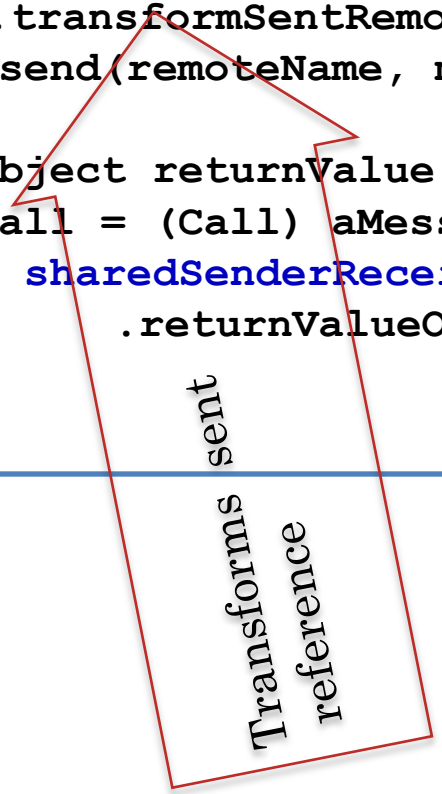
Send and receive trapper may need to share information about remote references sent and received

Duplex Shared State



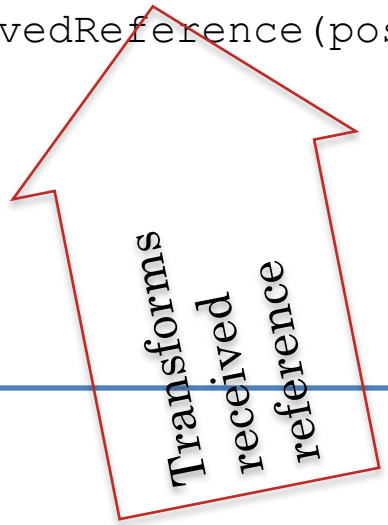
Duplex Send Trapper

```
public Object getSharedSenderReceiverState() {
  return sharedSenderReceiverState;
}
public void send(String remoteName, Object message) {
  Call call = (Call) message;
  sharedSenderReceiverState.localRemoteReferenceTranslator
        .transformSentRemoteReferences(call.getArgs());
  super.send(remoteName, message);
}
public Object returnValue(String aDestination, Object aMessage) {
  Call call = (Call) aMessage;
  return sharedSenderReceiverState.duplexSentCallCompleter
            .returnValueOfRemoteMethodCall(duplexRPCInputPort
                                                .getLastSender(), call);
}
```



Specialized Bounded Buffer: AnRPCReturnValueQueue

```
public void putReturnValue(RPCReturnValue message) {
  try {
    returnValueQueue.put(message);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
public Object takeReturnValue() {
  try {
    RPCReturnValue message = returnValueQueue.take();
    Object possiblyRemoteRetVal = message.getReturnValue();
    Object returnValue = localRemoteReferenceTranslator
            .transformReceivedReference(possiblyRemoteRetVal);
    return returnValue;
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }
}
```



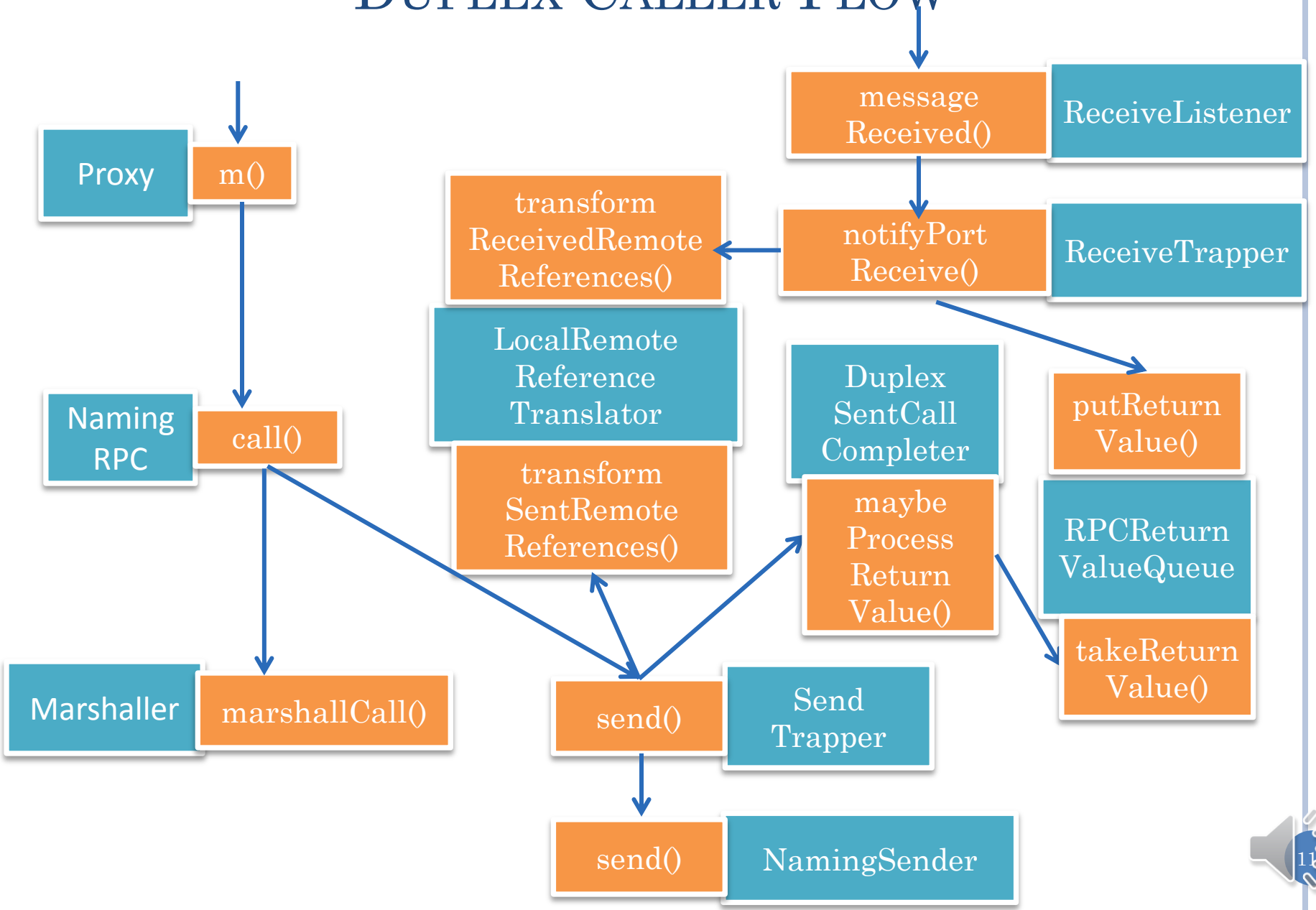
ADuplexReceivedCallInvoker

```
protected Object invokeMethod (Method method, Object targetObject,
Object[] args) {
  localRemoteReferenceTranslator.transformReceivedreferences(args);
  return super.invokeMethod(method, targetObject, args);
}
protected void handleFunctionReturn(String aSource, Object retVal) {
  Object possiblyTransformedRetVal =
      localRemoteReferenceTranslator.transformSentReference(retVal);
  replier.send (aSource,
    createRPCReturnValue(possiblyTransformedRetVal));
}
```

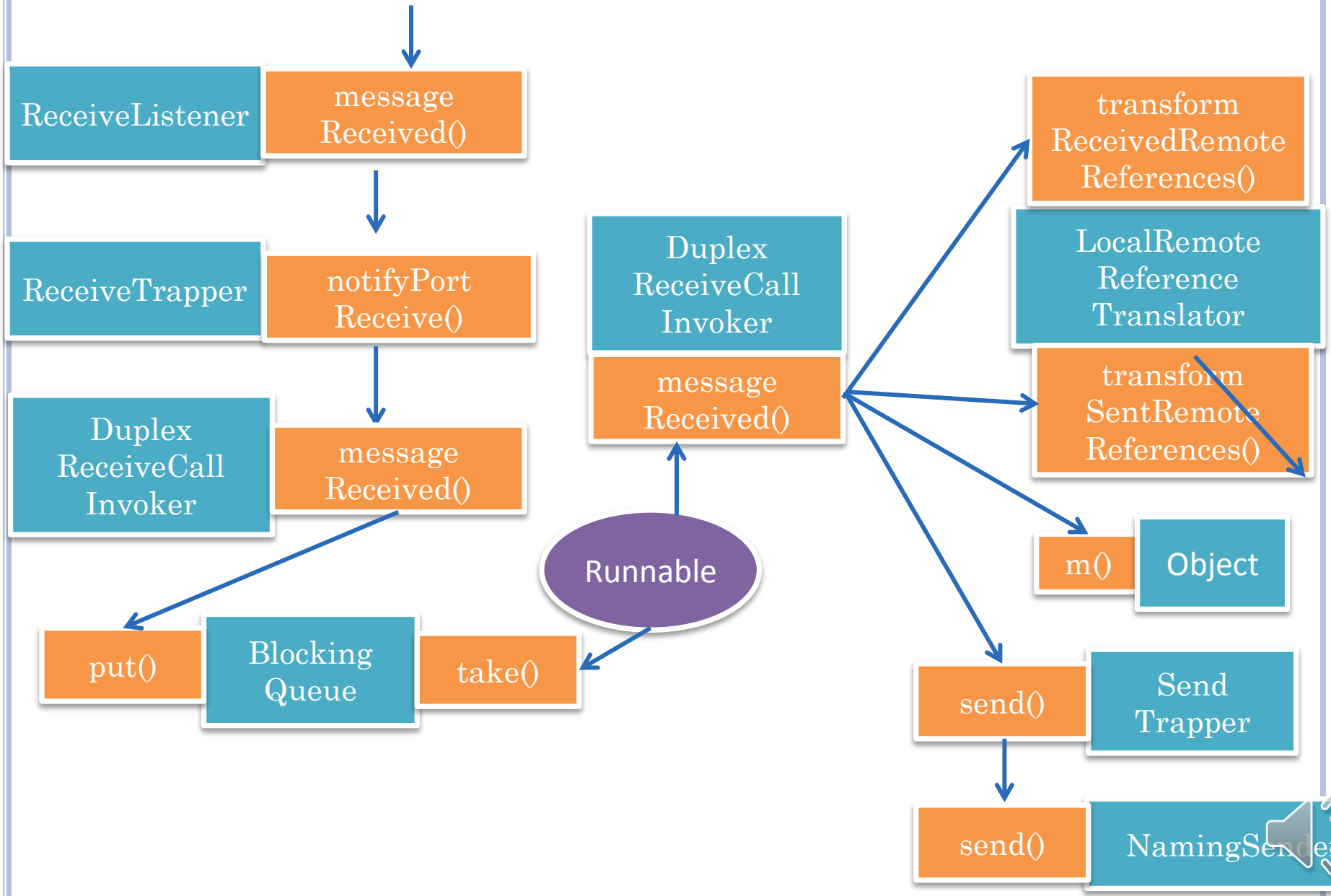
Transforms sent reference

Translates received reference

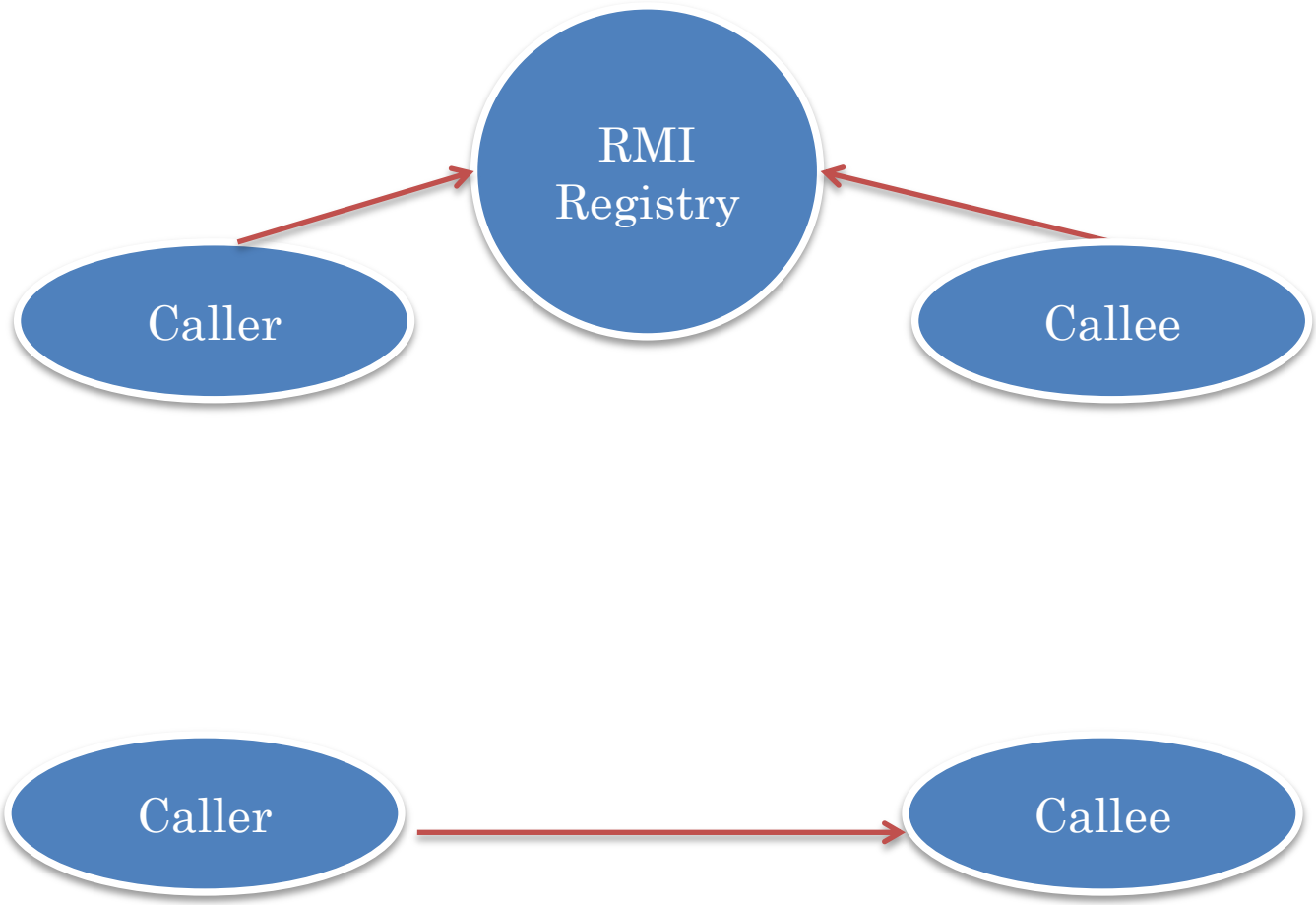
Duplex Caller Flow



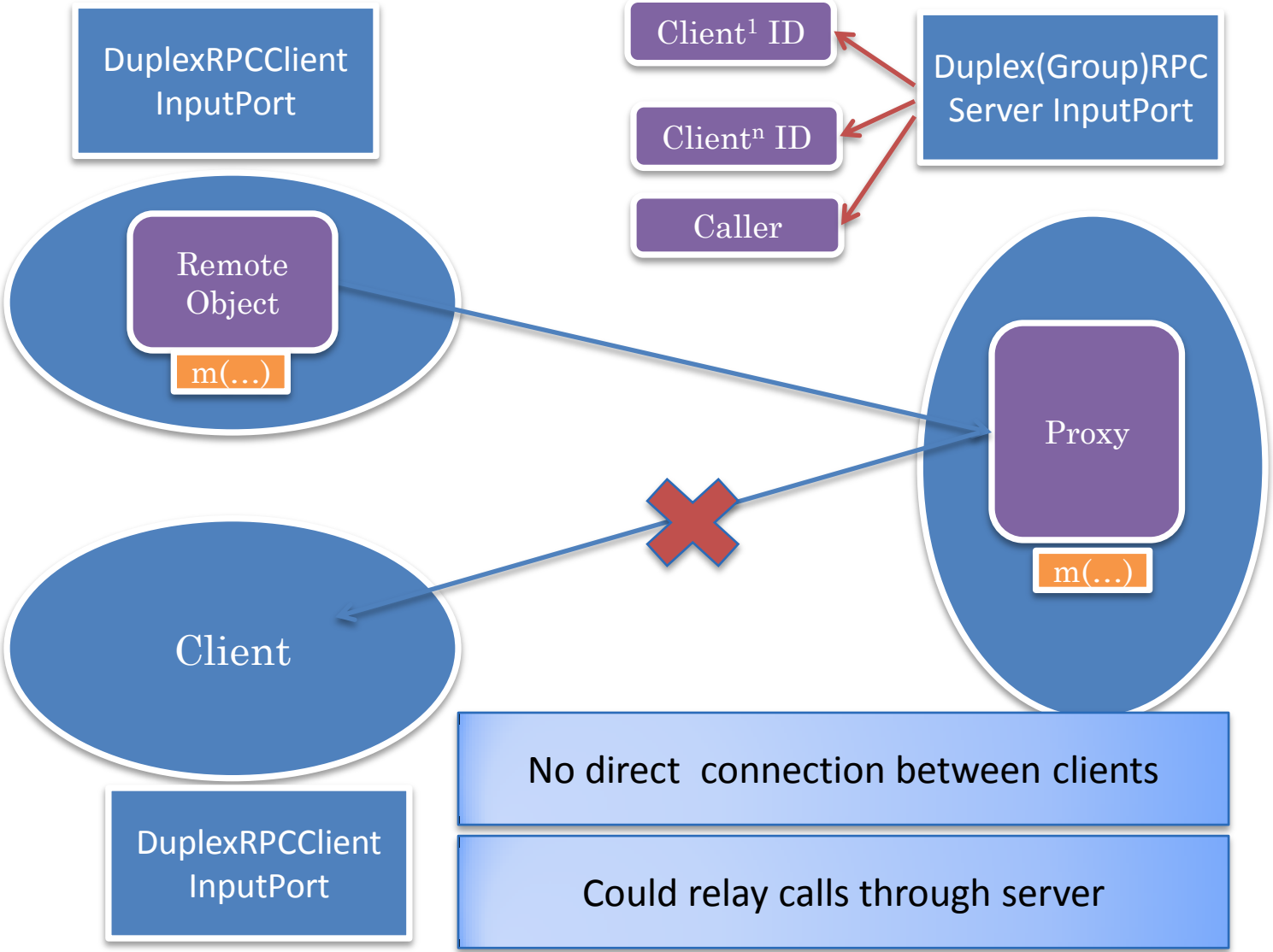
Duplex Callee Flow



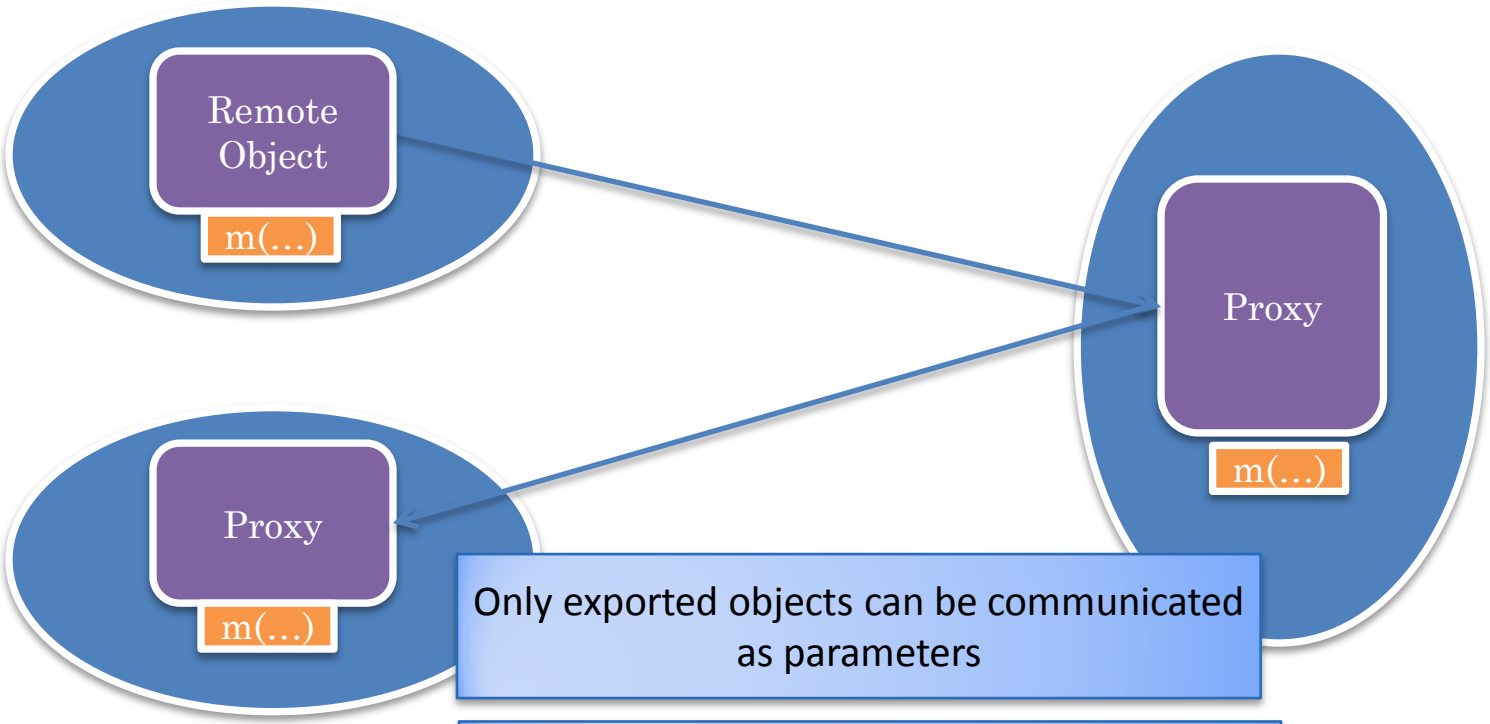
RMI vs. GIPC Binding Time



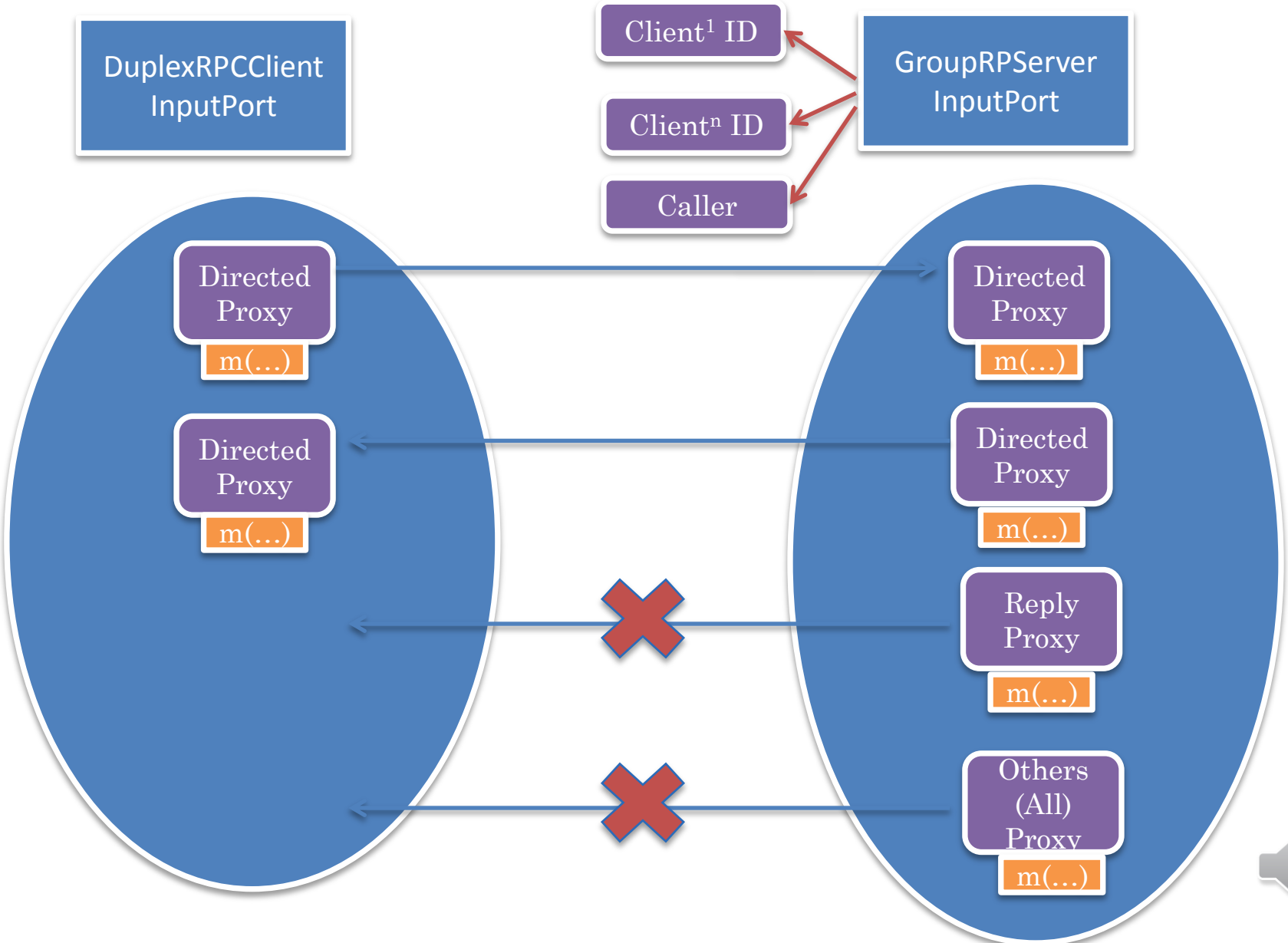
GIPC: Sharing with Other Clients?



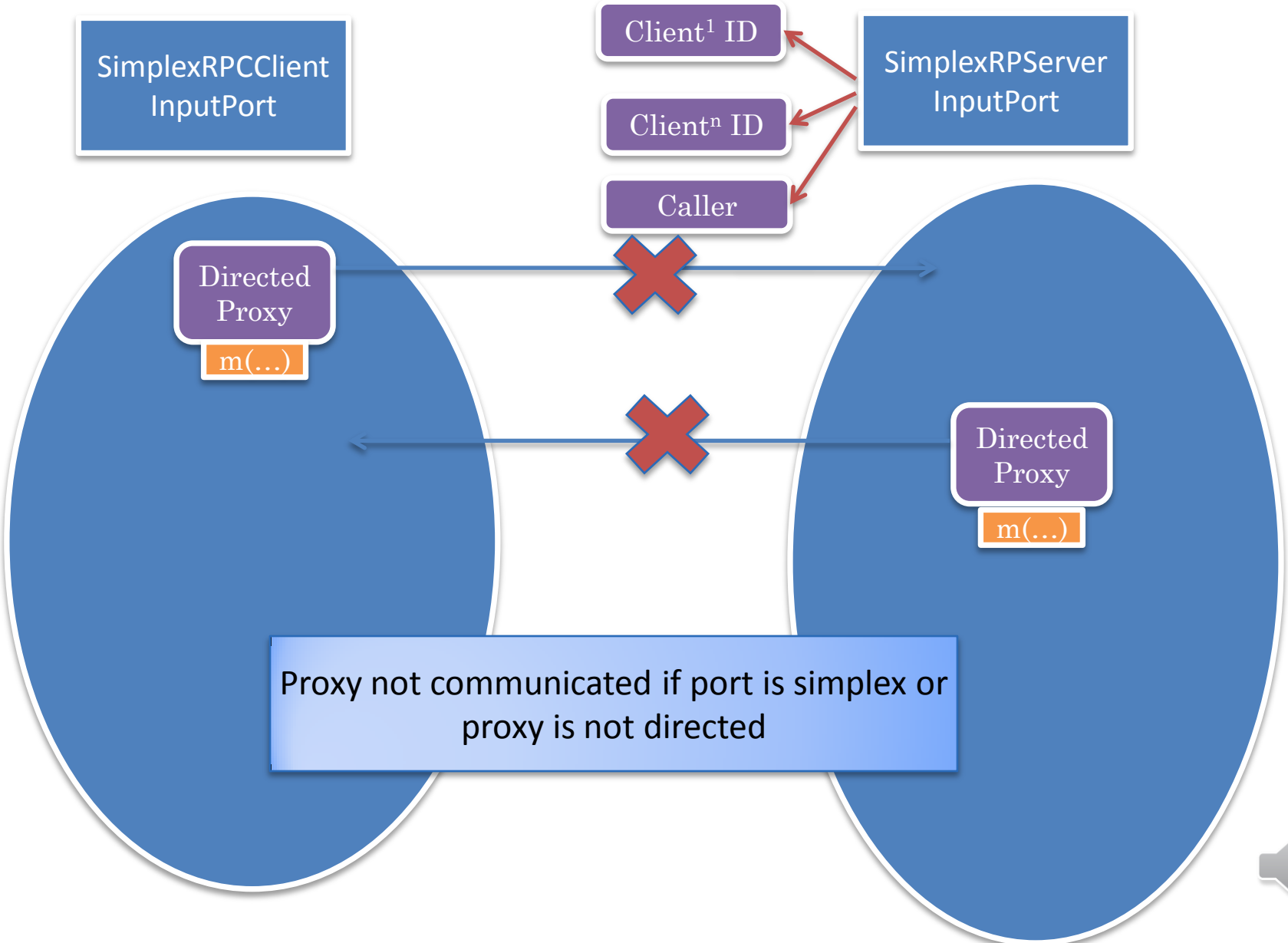
RMI: Sharing with Other Clients



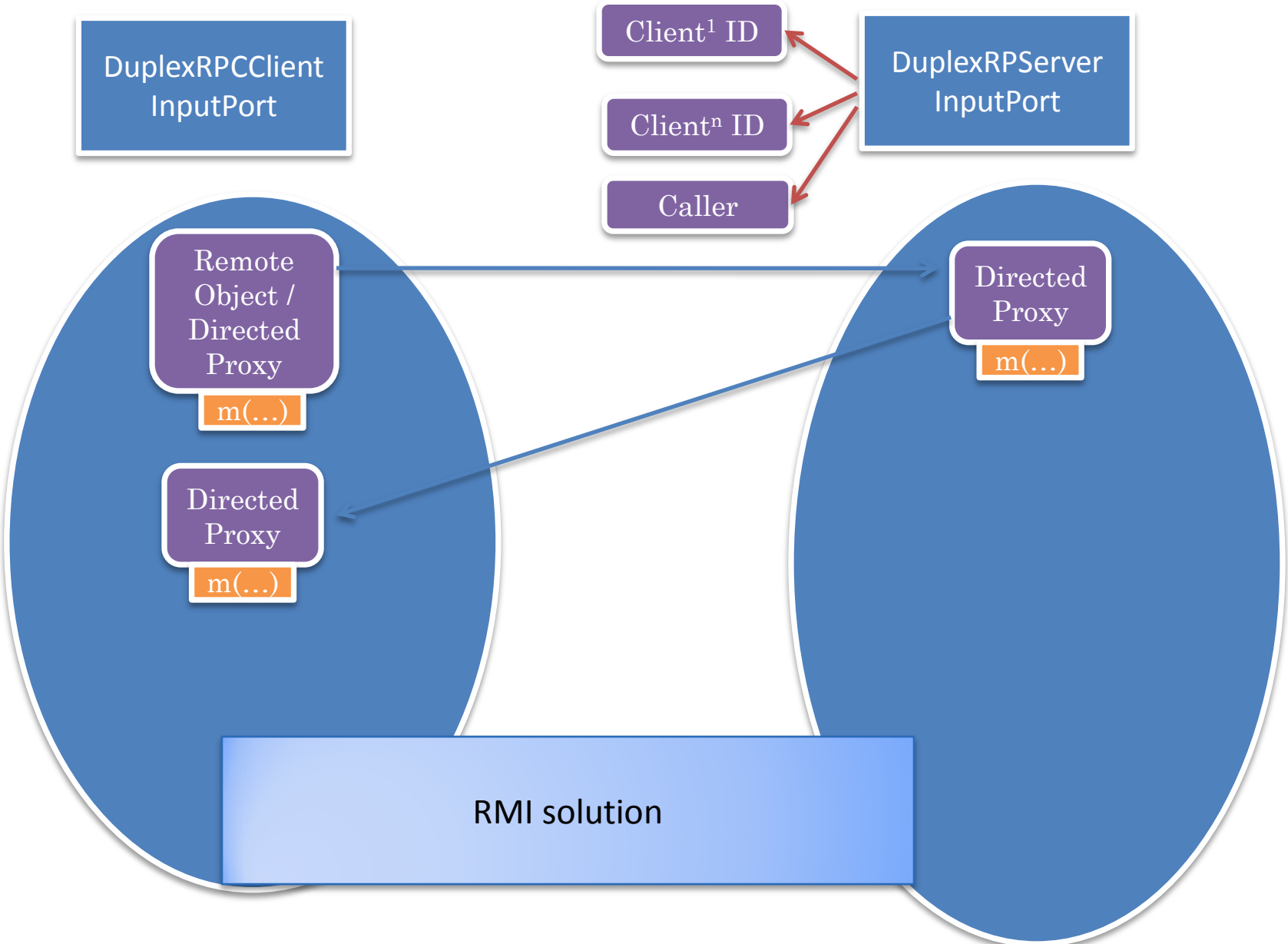
Communication and Types of Proxies



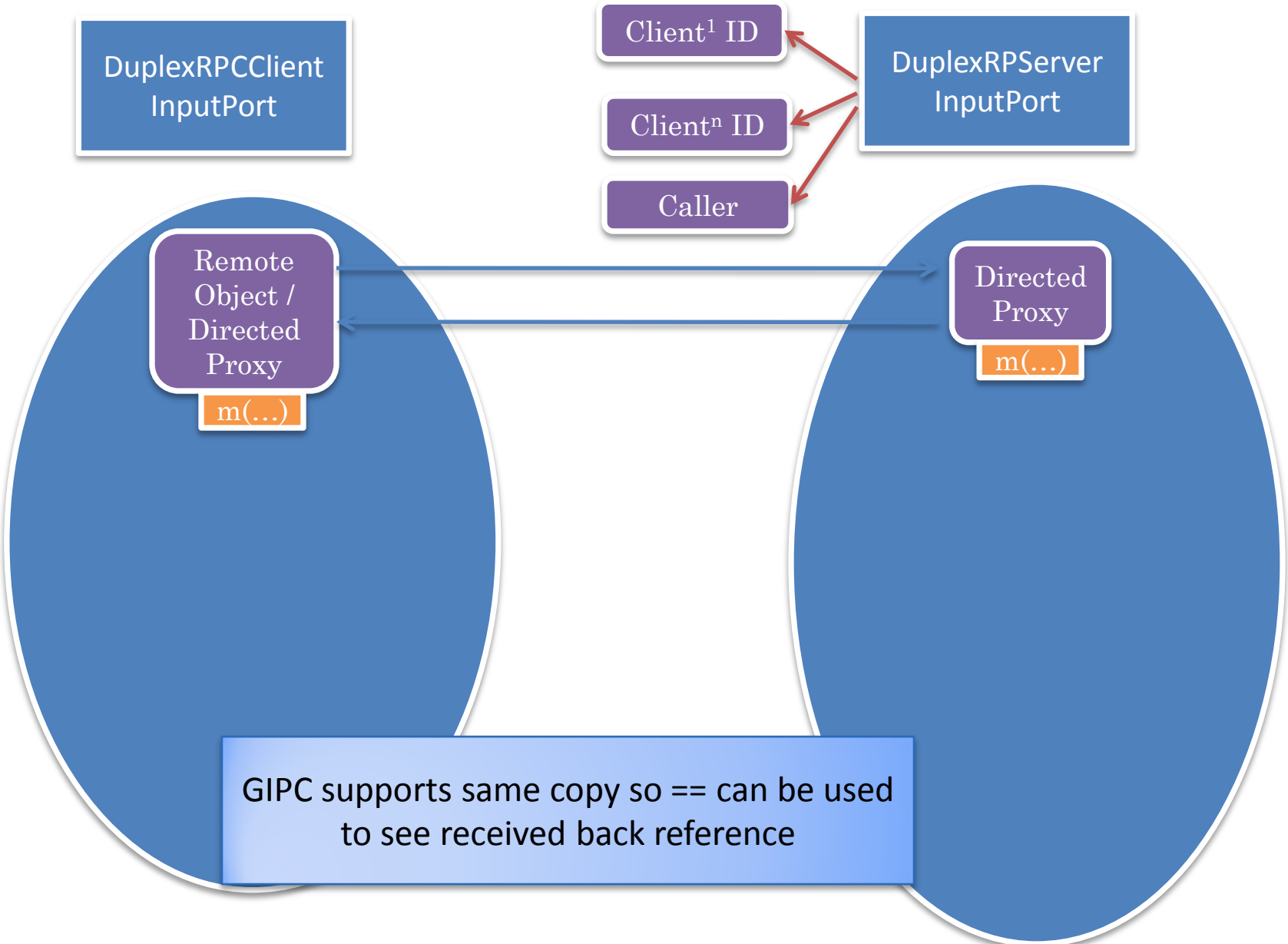
Communication and Types of Proxies



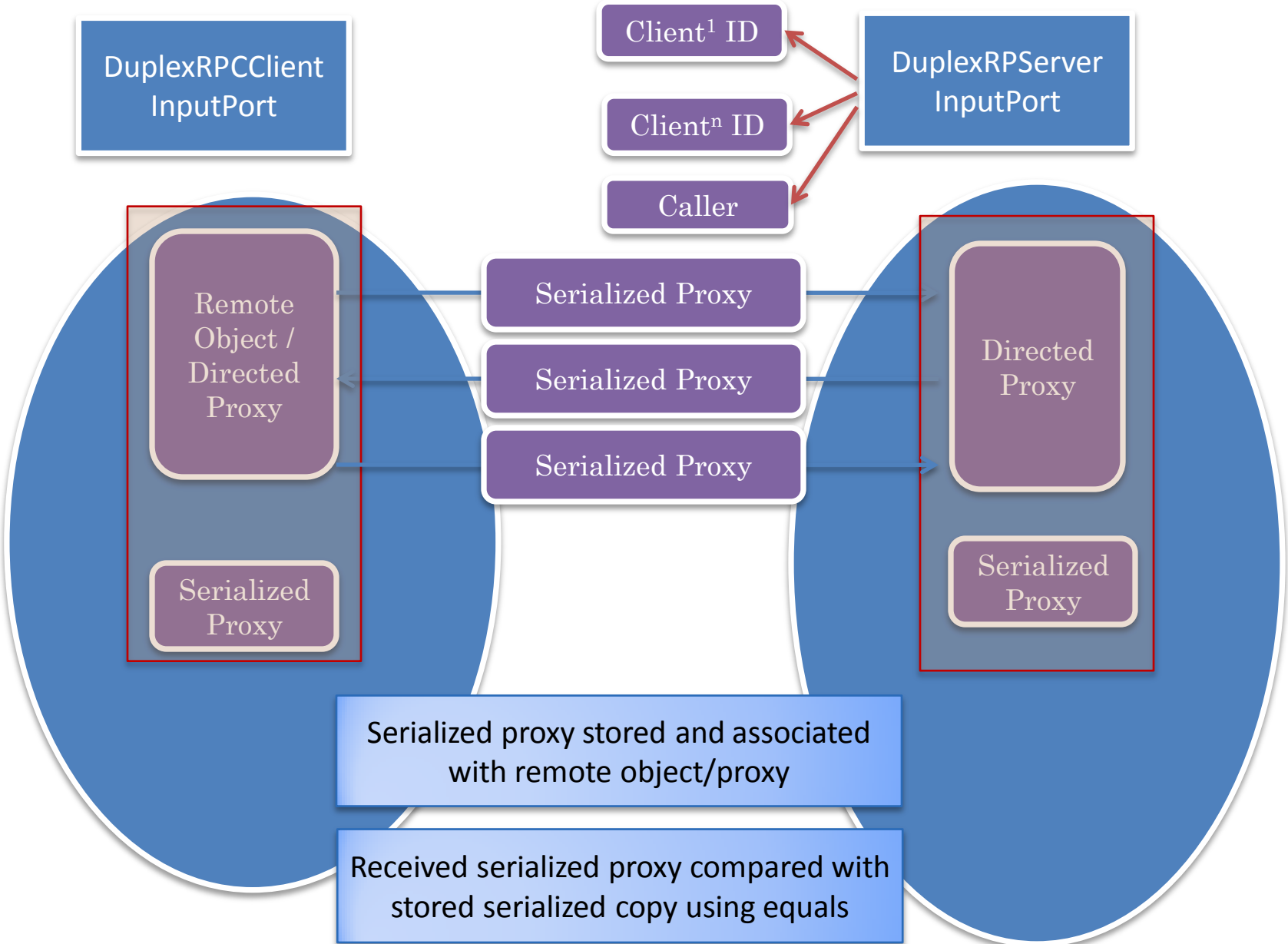
Proxies to Same Object: New Proxy?



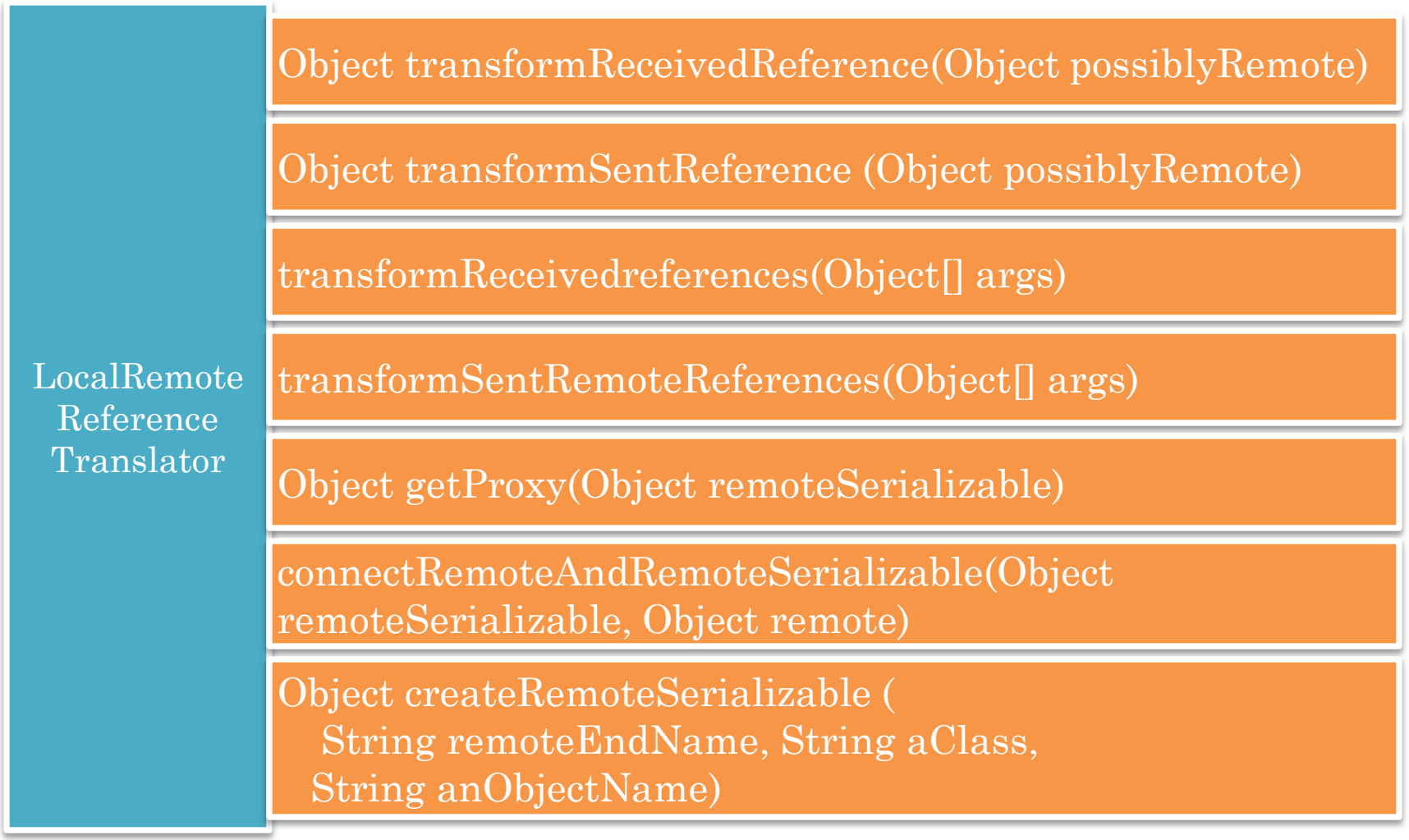
Proxies to Same Object: Same Proxy?



Implementing Same Copy: Algorithm

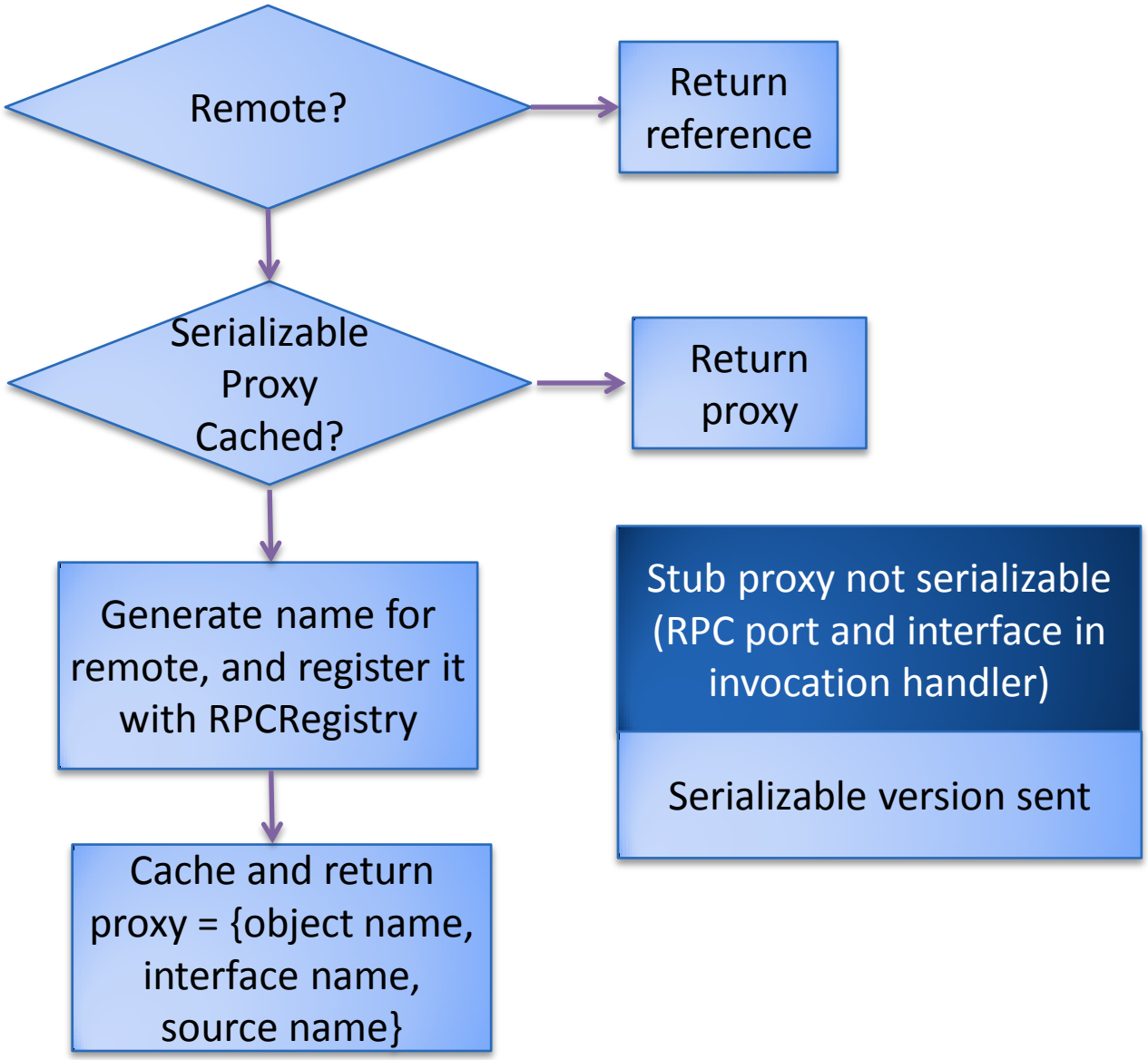


LocalRemoteReferenceTranslator

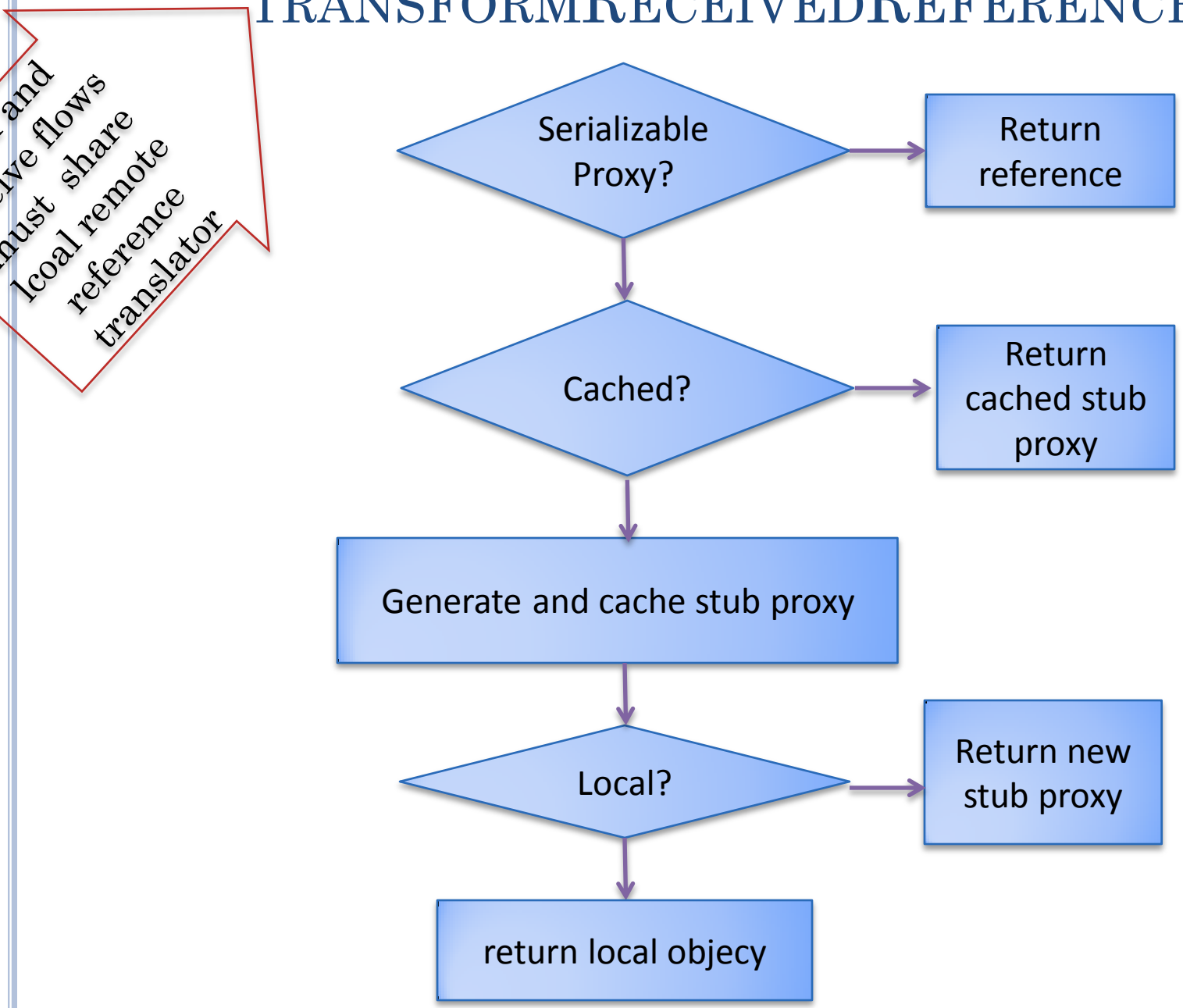


Delegation analogue of Java ReplaceObject overriding

TRANSFORMSENTREFERENCE



TRANSFORMRECEIVEDREFERENCE



Transform Sent Reference

```
public Object transformSentReference(
  Object possiblyRemote) {
  if (!(possiblyRemote instanceof Remote))
    return possiblyRemote;
  Remote remote = (Remote) possiblyRemote;
  RemoteSerializable remoteSerializable =
                        remoteToRemoteSerializable.get(remote);
  if (remoteSerializable == null) {
    String objectName = GENERATED_SUFFIX + objectId;
    objectId++;
    duplexRPCInputPort.register(objectName, remote);
    remoteSerializable = new
       ARemoteSerializable(duplexRPCInputPort.getLocalName(),
    remote.getClass().getName(), objectName);
    remoteToRemoteSerializable.put(remote, remoteSerializable);
  }
  return remoteSerializable;
}
```

Transform Received Reference

```
public Object transformReceivedReference(Object
                                    possiblyRemoteSerializable) {
  if (!(possiblyRemoteSerializable instanceof
                                      RemoteSerializable))
    return possiblyRemoteSerializable;
  RemoteSerializable remoteSerializable = (RemoteSerializable)
    possiblyRemoteSerializable;
  Object localObject =
        duplexRPCInputPort.getServer(remoteSerializable
                                              .getObjectName());
  Object proxy = getProxy(remoteSerializable);
```

Transform Received Reference

```
  try {
    if (proxy == null) {
      Class remoteInterface =
           Class.forName(remoteSerializable.getTypeName());
      proxy = (Remote) StaticRPCProxyGenerator.generateRPCProxy(
        duplexRPCInputPort, remoteSerializable.getRemoteEndName(),
        remoteInterface, remoteSerializable.getObjectName());
     remoteSerializableToProxy.put(remoteSerializable, proxy);
     // with traditional map a call back for hashcode happens
     remoteToRemoteSerializable.put(proxy, remoteSerializable);
     if (localObject != null)
       remoteToRemoteSerializable.put(localObject,
                                      remoteSerializable);
  }
  if (localObject != null)
    return localObject;
  return proxy;
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }
}
```

Duplex RPC Proxy Generator

```
public  Object generateRPCProxy(String aDestination,
                        Class aClass, String anObjectName) {
  Object remoteSerializable =
          localRemoteReferenceTranslator.
            createRemoteSerializable(aDestination, aClass.getName(),
                                     anObjectName);
  Object retVal =
      localRemoteReferenceTranslator.getProxy(remoteSerializable);
  if (retVal != null)return retVal;
    retVal = super.generateRPCProxy(aDestination, aClass,
                                     anObjectName);
    localRemoteReferenceTranslator.
      connectRemoteAndRemoteSerializable(
                              remoteSerializable, retVal);
  return retVal;
}
```

Counter

```
public interface Counter extends Remote {
  void increment(int val);
  int getValue();
}
```

ACounter

```
public class ACounter implements Counter {
  int counter;
  public int getValue() {
    return counter;
  }
  public void increment(int val) {
    counter += val;
  }
  public boolean equals(Object otherObject) {
    if (!(otherObject instanceof Counter))
      return false;
    return getValue() == ((Counter) otherObject).getValue();
  }
}
```

ComparableCounter

```
public interface ComparableCounter extends Counter{
  public ComparableCounter greater(ComparableCounter aCounter);
}
```

AComparableCounter

```
public class AComparableCounter extends ACounter
                                implements ComparableCounter {
  public ComparableCounter greater(ComparableCounter aCounter) {
    if (getValue() < aCounter.getValue()) {
      return aCounter;
    }
    else return this;
  }
}
```

Server

```
public class GIPCComparableCounterServer
             extends  omparableCounterLauncher{
  public static void main (String[] args) {
    try {
      DuplexRPCServerInputPort aServerInputPort =
      DuplexRPCInputPortSelector.createDuplexRPCServerInputPort(
                  SERVER_PORT, SERVER_NAME);
      ComparableCounter counter1 = new AComparableCounter();
      ComparableCounter counter2 = new AComparableCounter();
      aServerInputPort.register(COUNTER1, counter1);
      aServerInputPort.register(COUNTER2, counter2);
      aServerInputPort.connect();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Client

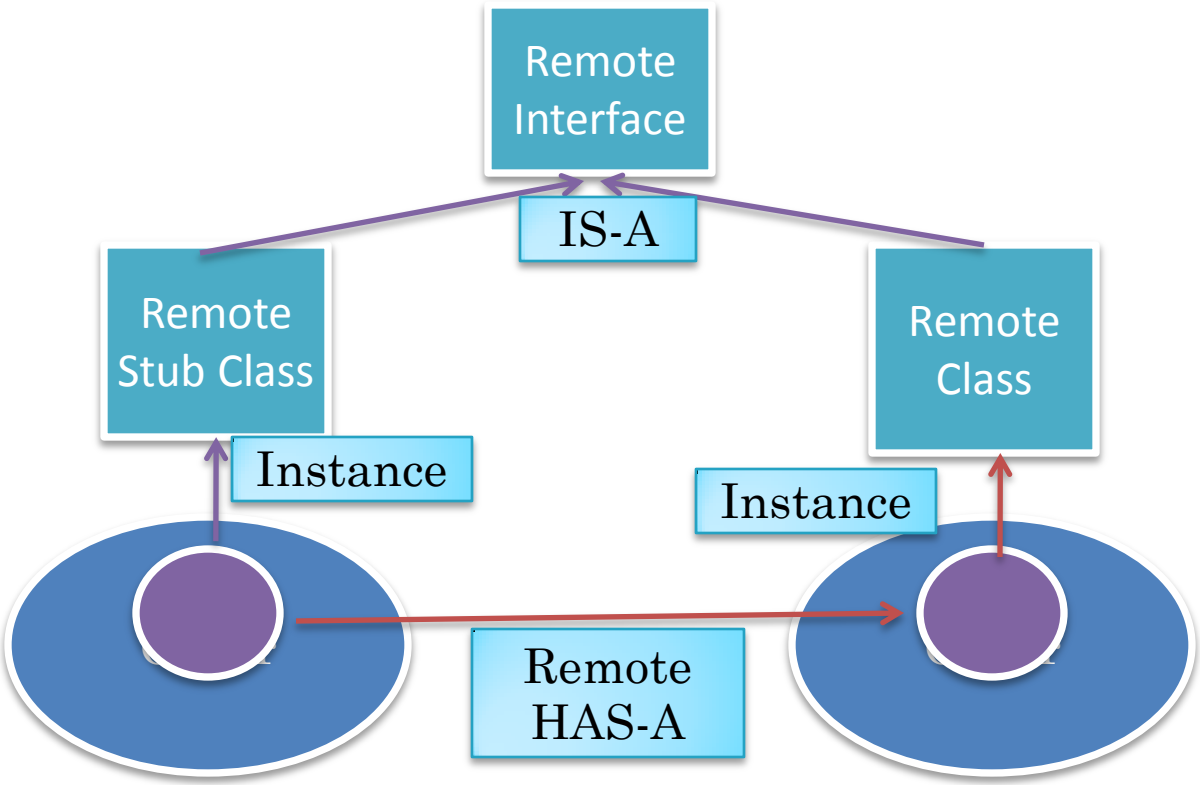
```
public static void main (String[] args) {
try {
  DuplexRPCClientInputPort aClientInputPort =
    DuplexRPCInputPortSelector.createDuplexRPCClientInputPort(
      "localhost", SERVER_PORT, SERVER_NAME, "counter client");
  RPCProxyGenerator rpcProxyGenerator =
    aClientInputPort.getRPCProxyGenerator()
```

Client

```
    ComparableCounter counter11 = (ComparableCounter)
       rpcProxyGenerator.generateRPCProxy(ComparableCounter.class,
           COUNTER1);
    ComparableCounter counter12 = (ComparableCounter)
        rpcProxyGenerator.generateRPCProxy(ComparableCounter.class,
          COUNTER1);
    ComparableCounter counter2 = (ComparableCounter)
        rpcProxyGenerator.generateRPCProxy(ComparableCounter.class,
          COUNTER2);
    aClientInputPort.connect();
    ComparableCounter greaterCounter = counter11.greater(counter11);
    System.out.println(greaterCounter == counter11);
    System.out.println(greaterCounter.equals(counter11));
    System.out.println(counter12 == counter11);
    System.out.println(counter12.equals(counter11));
    System.out.println(counter11.hashCode() ==
                       counter12.hashCode());
    System.out.println(greaterCounter.hashCode() ==
      counter11.hashCode());
    System.out.println(counter11.equals(counter2));
  } catch (Exception e) {
    e.printStackTrace();
}}}
```

```
GIPCComparableCounterClient [Ja
true
true
true
true
true
true
```

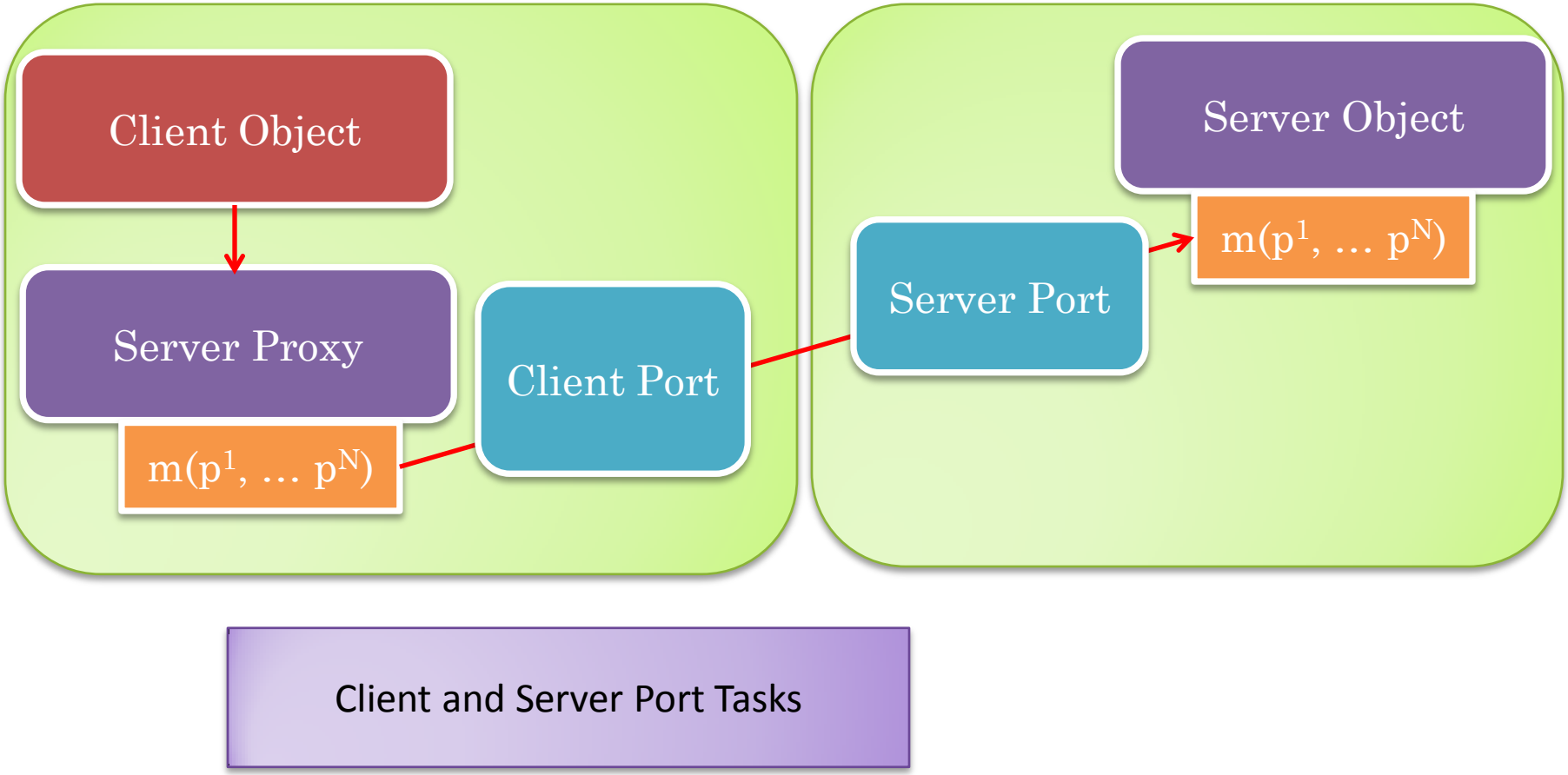
Stubs and Garbage Collection



Remote HAS-A relationships must be used in garbage collection

RPC system can communicate message when remote references are created/garbage collected

Port vs. GIPC-RMI



GIPC Features

- API of lower level layers (except byte communication) is visible
- Only remote function calls have to wait
- Proxies can be generated for all methods of an object
- Server method can determine the host and client who made the call
- Special call made by client to pass its name to the RPC system
- GIPC has group communication for each communication abstraction
- Server is optional and can be built using basic primitives

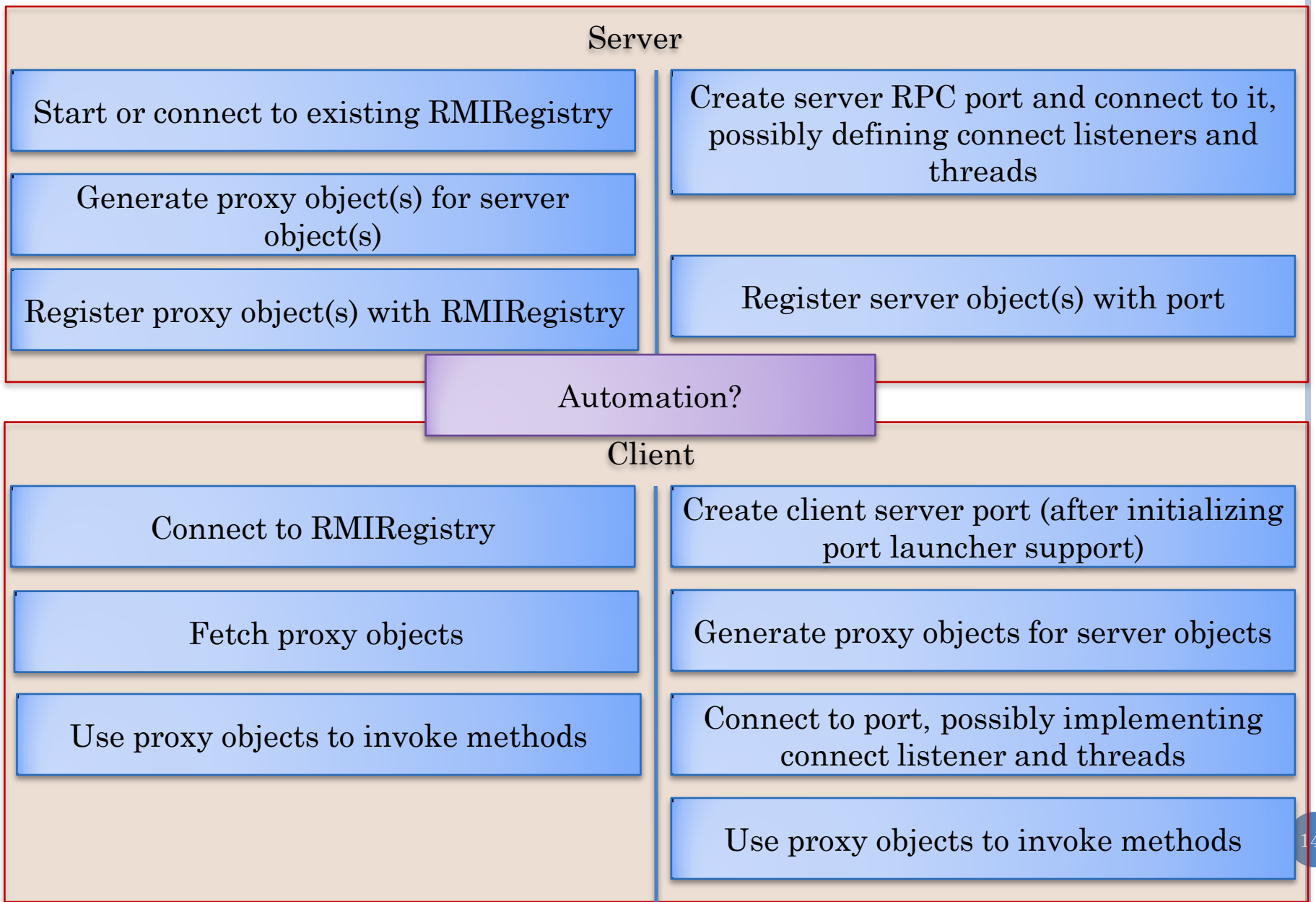
GIPC Goals vs. Features

Extensibility	Factories, Abstract Factories, PortLauncherSupport, PortLauncher, Open Source, Layering
API of lower level layers (except byte communication) is visible when using RPC	Client and server input ports define name space and used for connecting, sending, receiving remote object registration and proxy generation
An operation should not have to wait unless it has to	Observer pattern for async IPC operations and only function calls block for compatibility with local procedure call syntax
Programmer makes tradeoff between reusability and error checking	Can generate proxies for all methods, and warning given if method in a Remote interface does not acknowledge RemoteException

GIPC Goals vs. Features

No central server for object registration and proxy creation	Late binding of proxy to remote object. Proxy created based on class and name of remote object which is passed at call time to remote port end point which looks it up
Only code that needs client awareness should be client aware	Can make a return call to current caller without being aware of it because of late binding
	At client port creation time, client name passed, communicated to server, which can make server port call to get name client making current call
Support group communication	Late binding to remote object allows the same call by a server to be invoked on different sets of clients

RMI vs. GIPC-RPC Steps



Port Creation in RMI?

- When a process first exports an object on a specific port, an underlying server communication channel is created to receive and send messages on that channel to that process and associates the object with that channel.
- The underlying communication channel need not support an explicit connection.
- Proxy calls to remote objects result in messages being sent to the communication channels associated with the remote objects.
- Thus the port creation and connection calls in GIPC are replaced with an export call that takes only the port number of the underlying channel

RPC Awareness in Application: Two Extremes

RPC clients and servers handle all aspects of RPC and are completely aware of all of the steps listed below

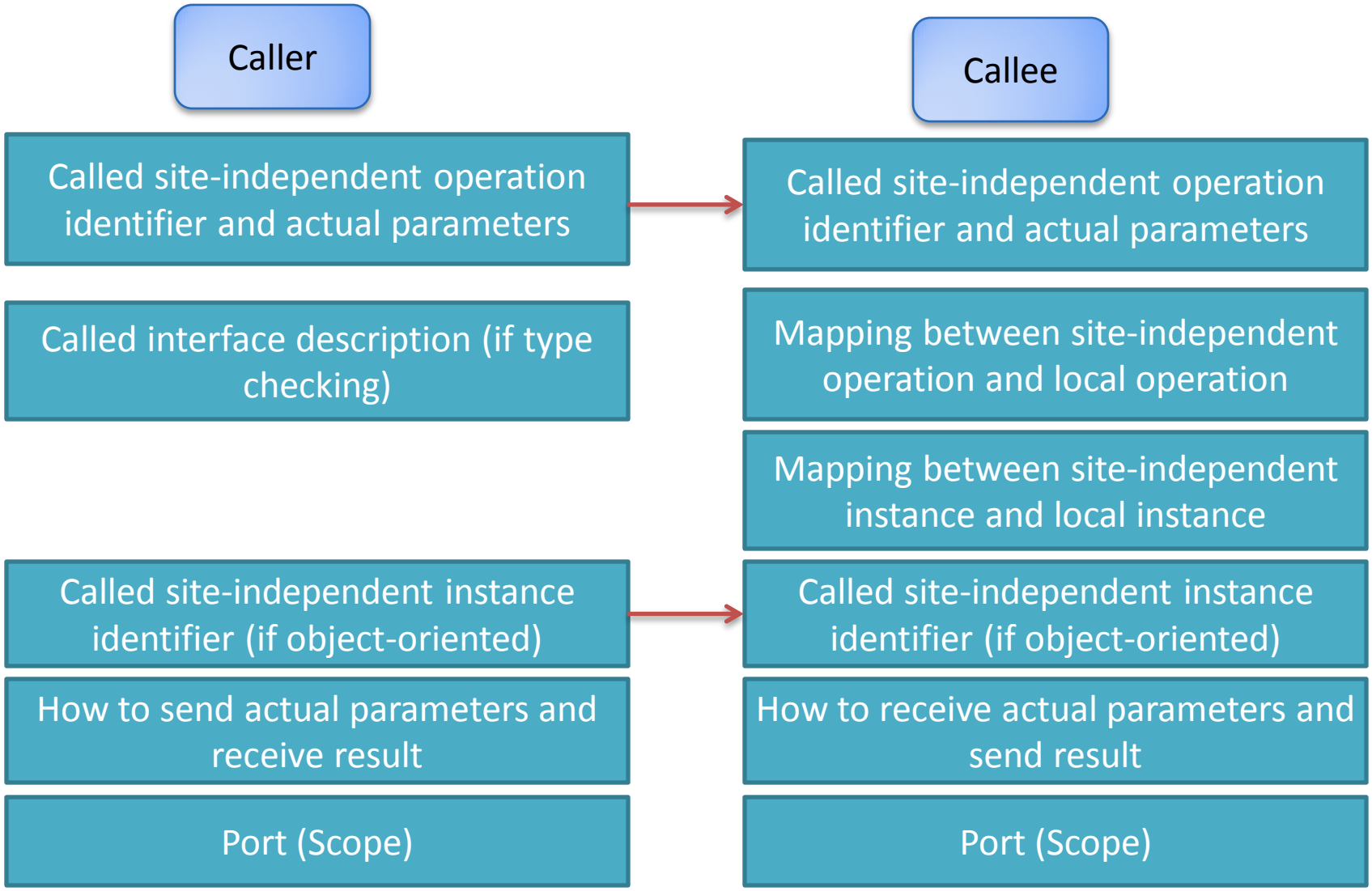
RPC clients and servers are completely unaware of distribution details

Java RMI falls close to complete transparency

GIPC RMI supports a wider spectrum that falls closer and further

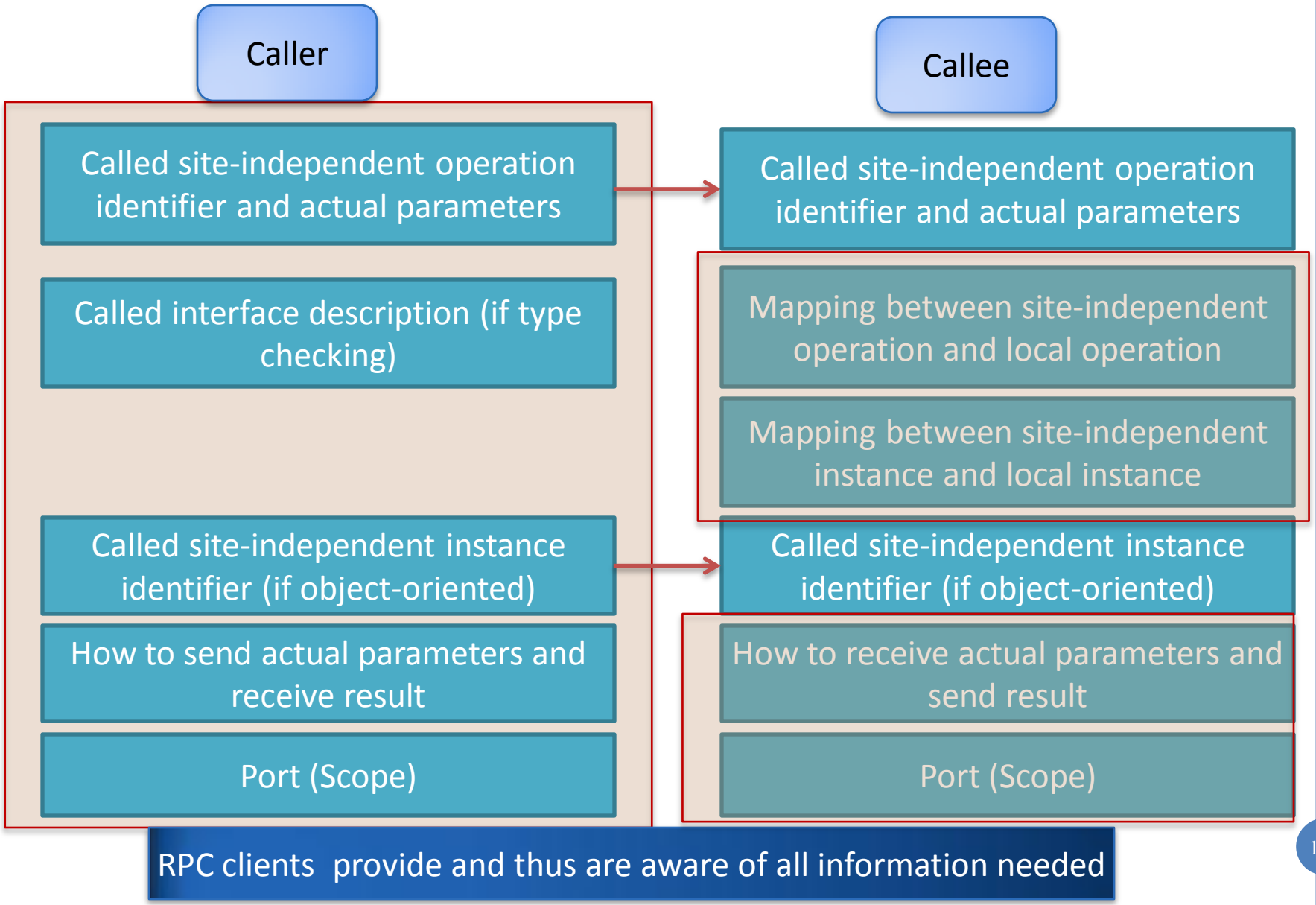
Can one have complete transparency?

Information Needed by RPC System

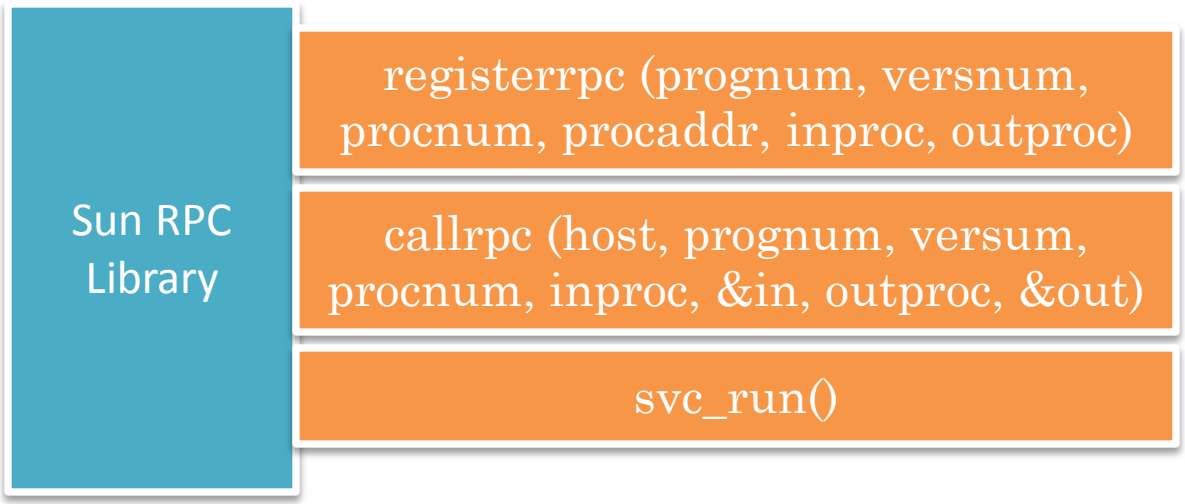


Systems differ in how much information provided by application

Full RPC Awareness



Full Awareness: Sun non-OO RPC



Full Awareness: Sun non-OO RPC

Add Interface

```
typedef struct {
   int f1;
   float f2
} S;
extern float add ();
```

```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0
```

Common

XDR Routines

```
xdr_s (xdrsp, arg)
XDRS *xdrsp;
S *arg;
{
    xdr_int (xdrsp, &arg->f1);
    xdr_float (xdrsp, &arg->f2);
}
```

Caller

```
S *s;
float *result
s->f1 = 3;
s-f2 = 4.12;
callrpc (host, PRO    UM, VERS_NUM,
    _NUM, xdr_s, s, xdr_float, result);
```

Port (Scope)

Single parameter

Site independent operation id

Parameter serializer

Result deserializer

Result location

Callee

```
float add (s)
S   *s;
    ur    f1 -    );
}
```

Implementation

Port (Scope)

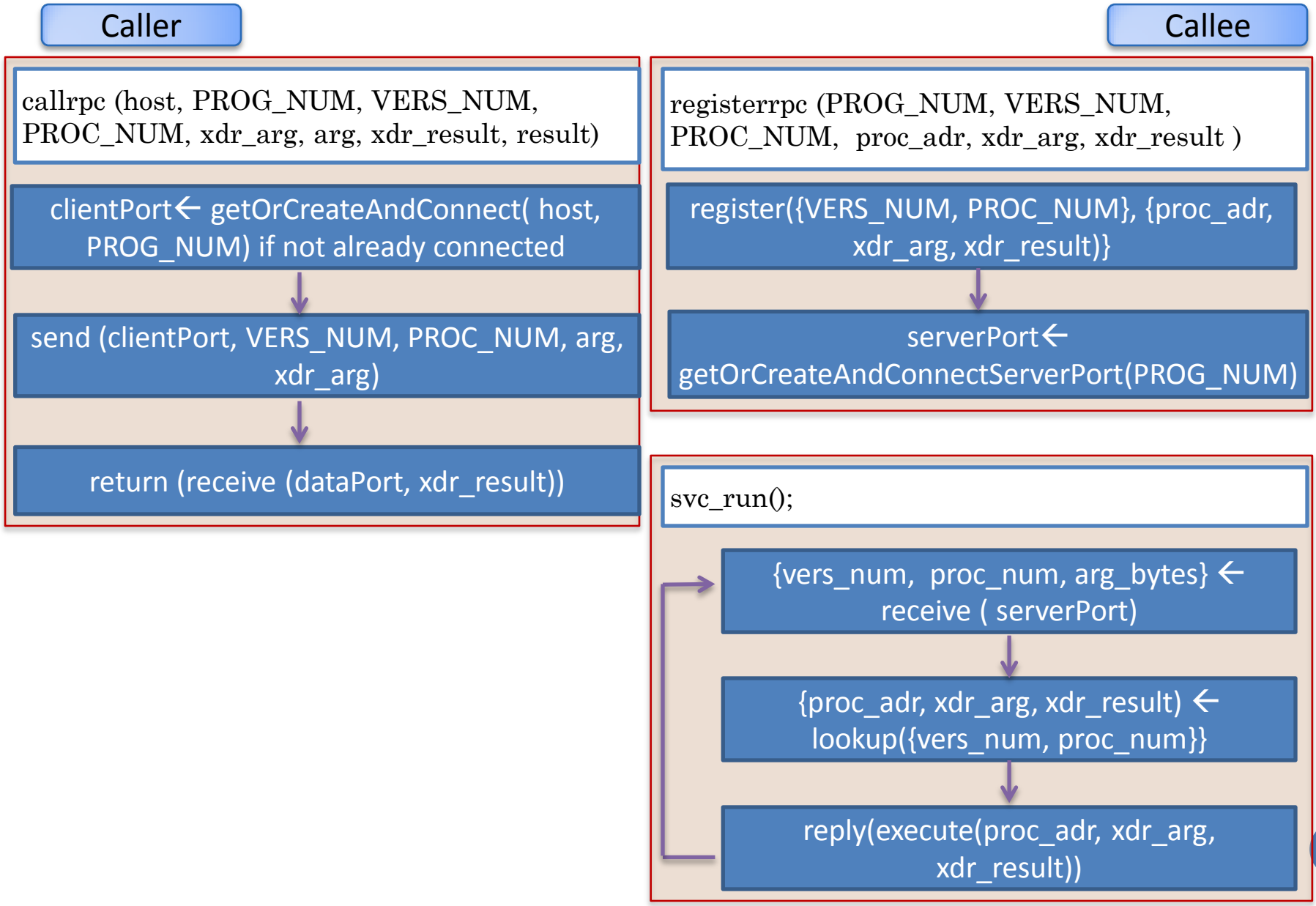
Parameter de-serializer

Result de-serializer

Site independent operation id

```
regist    PROC    M, VERS_NUM, ADD_NUM,
                add, xdr_s, xdr_float ) ;
svc_run();
```

Implementing Full Awareness



Full Application Awareness: Sun non-OO RPC

Common

Add Interface

```
typedef struct {
   int f1;
   float f2
} S;
extern float add ();
```

```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0
```

XDR Routines

```
xdr_s (xdrsp, arg)
XDRS *xdrsp;
S *arg;
{
    xdr_int (xdrsp, &arg->f1);
    xdr_float (xdrsp, &arg->f2);
}
```

Caller

```
S *s;
float *result;
s->f1 = 3;
s-f2 = 4.12;
callrpc (host, PROG_NUM, VERS_NUM,
      ADD_NUM, xdr_s, s, xdr_float, result);
```

How changed for full transparency?

Callee

```
float add (s)
S *s;
{
    return (s->f1 + s->f2);
}
```

```
registerrpc (PROG_NUM, VERS_NUM, ADD_NUM,
                add, xdr_s, xdr_float ) ;
svc_run();
```

Full Application Awareness: Sun non-OO RPC (Review)

Common

Add Interface

```
typedef struct {
   int f1;
   float f2
} S;
extern float add ();
```

```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0
```

XDR Routines

```
xdr_s (xdrsp, arg)
XDRS *xdrsp;
S *arg;
{
    xdr_int (xdrsp, &arg->f1);
    xdr_float (xdrsp, &arg->f2);
}
```

Caller

```
S *s;
float *result;
s->f1 = 3;
s-f2 = 4.12;
callrpc (host, PROG_NUM, VERS_NUM,
      ADD_NUM, xdr_s, s, xdr_float, result);
```

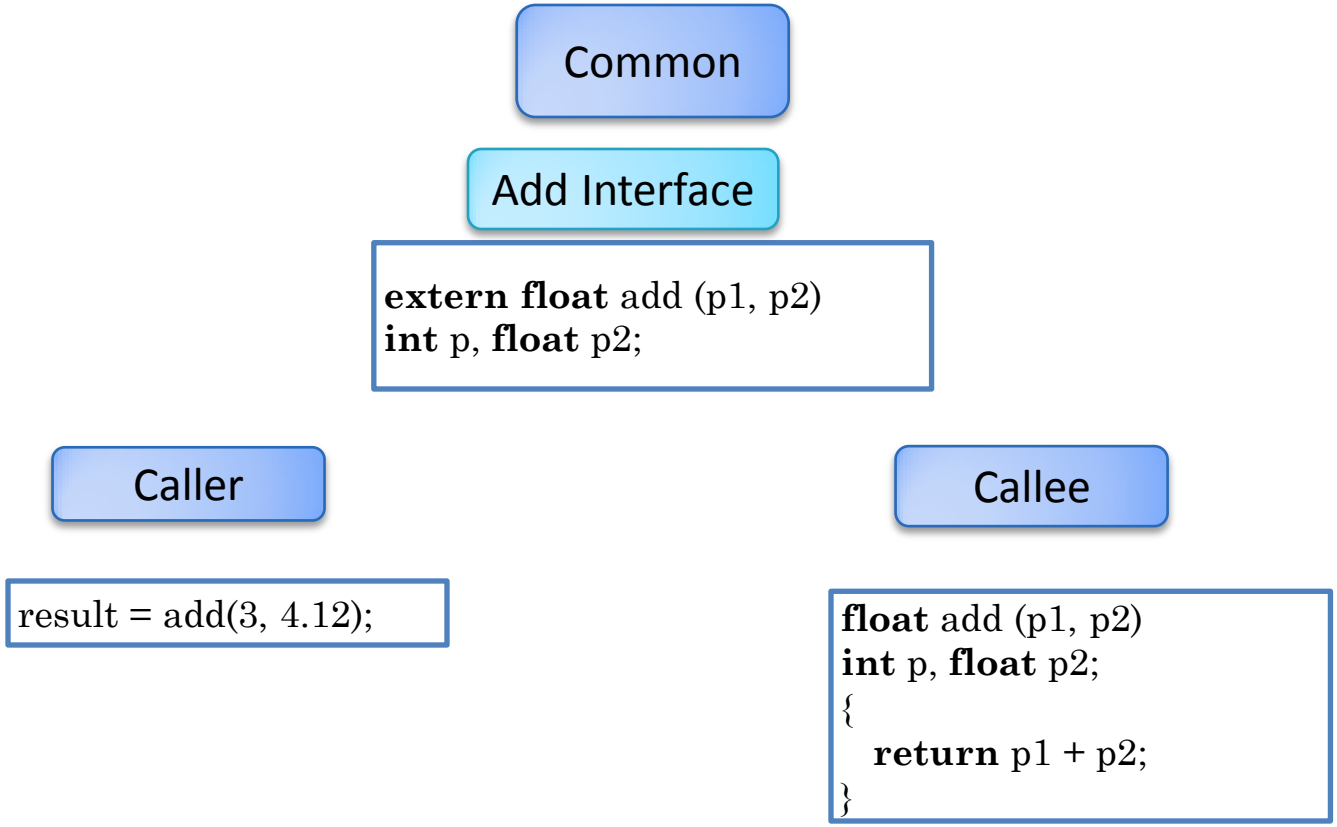
How changed for full transparency?

Callee

```
float add (s)
S *s;
{
    return (s->f1 + s->f2);
}
```

```
registerrpc (PROG_NUM, VERS_NUM, ADD_NUM,
                add, xdr_s, xdr_float ) ;
svc_run();
```

Full Application Transparency: Cedar RPC



How to implement fully transparent RPC?

Assume fully aware rpc is given

Full Application Awareness: Sun non-OO RPC

Common

Add Interface

```
typedef struct {
   int f1;
   float f2
} S;
extern float add ();
```

```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0
```

XDR Routines

```
xdr_s (xdrsp, arg)
XDRS *xdrsp;
S *arg;
{
    xdr_int (xdrsp, &arg->f1);
    xdr_float (xdrsp, &arg->f2);
}
```

Caller

```
S *s;
float *result;
s->f1 = 3;
s-f2 = 4.12;
callrpc (host, PROG_NUM, VERS_NUM,
      ADD_NUM, xdr_s, s, xdr_float, result);
```

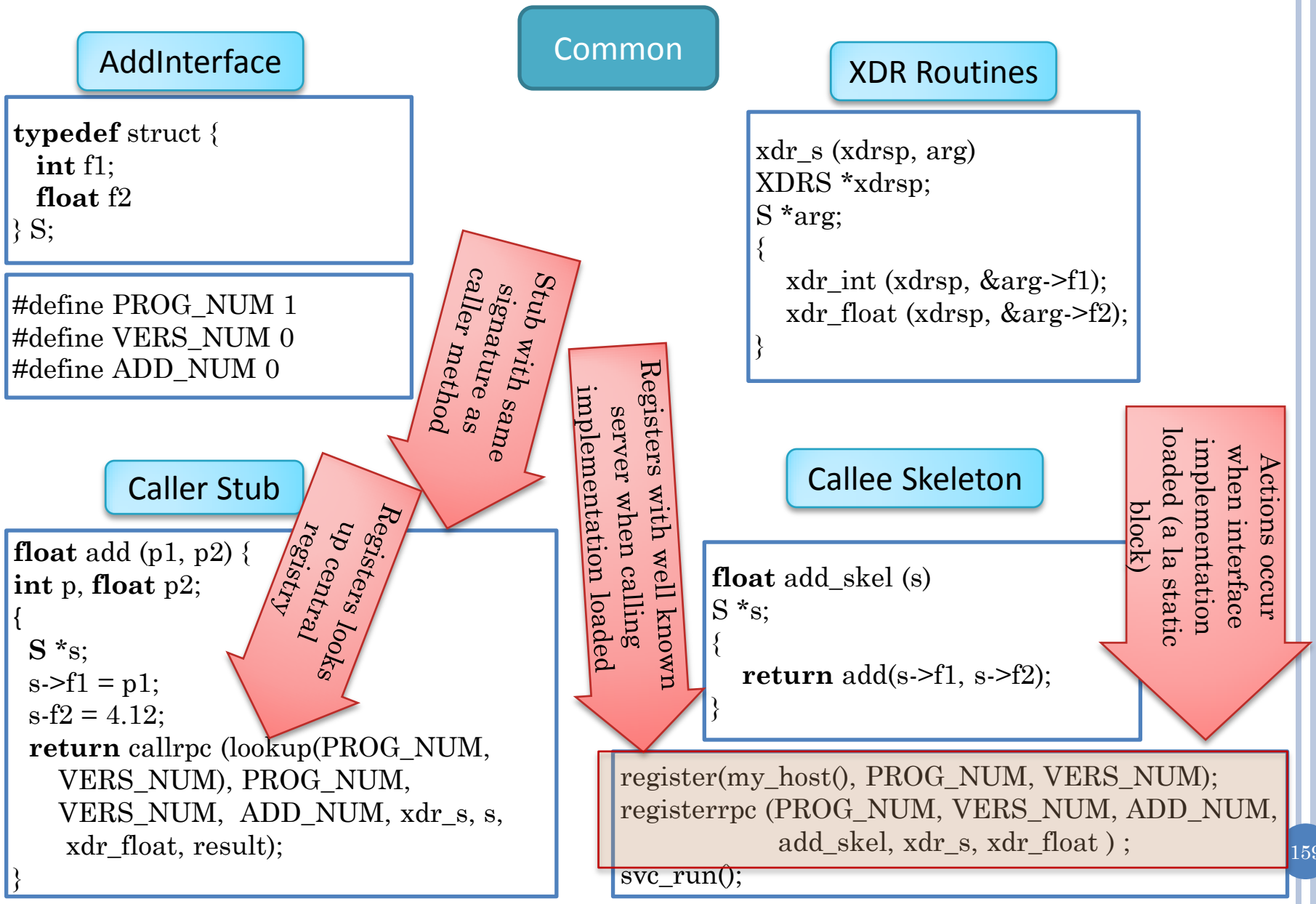
Implementation vehicle now

Callee

```
float add (s)
S *s;
{
    return (s->f1 + s->f2);
}
```

```
registerrpc (PROG_NUM, VERS_NUM, ADD_NUM,
                add, xdr_s, xdr_float ) ;
svc_run();
```

Generation based RPC



Stub and Skeleton Operations

Caller Stub

Created for each callable operation and has same signature

Marshals parameters and sends message (using lower-level mechanism)

Receives, unmarshals, and returns result

```
float add (p1, p2) {
int p, float p2;
{
  S *s;
  s->f1 = p1;
  s-f2 = 4.12;
  return callrpc (lookup(PROG_NUM,
     VERS_NUM), PROG_NUM,
     VERS_NUM,  ADD_NUM, xdr_s, s,
      xdr_float, result);
}
```

Callee Skeleton

Created for each callable operation

Receives message (using lower-level mechanism) and unmarshals parameters

Calls implementation, marshals and sends result

```
float add_skel (s)
S *s;
{
    return add(s->f1, s->f2);
}
```

Stack push/pop vs message send/receive

Application vs. Syntax Transparency

Application transparency

Degree to which calling and called application code is aware of remote interaction

Application transparency → syntax transparency

Syntax transparency

Degree to which operation declaration and call syntax is aware of remote interaction

Syntax transparency → application transparency

Syntax transparency → client stubs

Syntax transparency → server skeleton

Stub Skeleton based RPC

Common

AddInterface

```
typedef struct {
   int f1;
   float f2
} S;
```

```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0
```

XDR Routines

```
xdr_s (xdrsp, arg)
XDRS *xdrsp;
S *arg;
{
    xdr_int (xdrsp, &arg->f1);
    xdr_float (xdrsp, &arg->f2);
}
```

Can have universal dispatching skeleton

Caller Stub

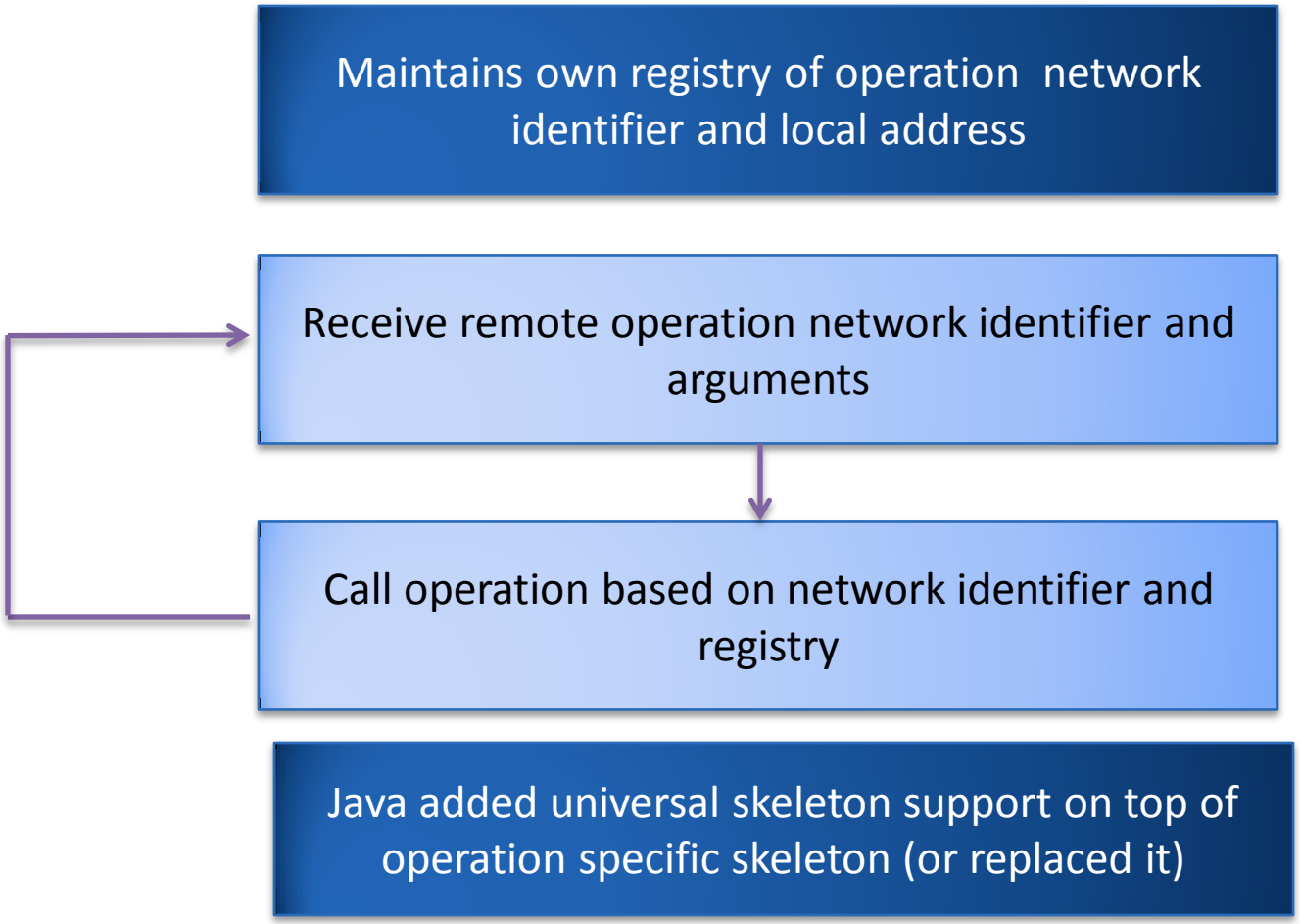
```
float add (p1, p2) {
int p, float p2;
{
  S *s;
  s->f1 = p1;
  s-f2 = 4.12;
  return callrpc (lookup(PROG_NUM,
     VERS_NUM), PROG_NUM,
     VERS_NUM,  ADD_NUM, xdr_s, s,
      xdr_float, result);
}
```

Callee Skeleton

```
float add_skel (s)
S *s;
{
    return add(s->f1, s->f2);
}
```

```
register(my_host(), PROG_NUM, VERS_NUM);
registerrpc (PROG_NUM, VERS_NUM, ADD_NUM,
                add_skel, xdr_s, xdr_float ) ;
svc_run();
```

Universal Skeleton



Conventional vs. Object-Oriented : Full Application Awareness

Conventional

callrpc (host, PROG_NUM, VERS_NUM, PROC_NUM, xdr_arg, arg, xdr_result, result)

registerrpc (PROG_NUM, VERS_NUM, PROC_NUM, proc_adr, xdr_arg, xdr_result)

Object Oriented

callrpc (host, PROG_NUM, VERS_NUM, OBJECT_NUM,PROC_NUM, xdr_arg, arg, xdr_result, result)

registerrpc (PROG_NUM, VERS_NUM, OBJECT_NUM, PROC_NUM, proc_adr, xdr_arg, xdr_result)

Conventional vs. Object-Oriented : Full Application Transparency Minimal

Conventional

```
result = add(3, 4.12);
```

```
extern float add (p1, p2)
int p, float p2;
```

```
float add (p1, p2)
int p, float p2;
{
   return p1 + p2;
}
```

Declaration Transparent

Instantiation Transparent

Need stub/proxy instances in addition to stub/proxy server methods

Object-Oriented

```
public interface {
   float add (int p1, float p2);
}
```

```
float add (int p1, float p2){
   return p1 + p2;
}
```

```
proxy = getProxy(name);
```

How to transparently bind multiple server instances to multiple client handles?

```
result = proxy.add(3, 4.12);
```

```
object = new Adder();
```

Call Transparent

Minimal awareness: registration and getProxy (local or remote calls)

```
register(object, name);
```

Stub and Skeleton Objects

Caller Stub	Callee Skeleton
Created for a target remote object and some interface of it	Created for a target remote object and some interface of it
Instance of a stub-class that implements the interface	Instance of a skeleton-class
Stub-class has stub operation for each operation in the interface	Skeleton-class has skeleton operation for each operation in the interface

Syntax transparency → client stubs	Syntax transparency → server skeleton

Semantics vs. Syntax Transparency

Semantic transparency

Degree to which remote call behaves like a local call

Semantics transparency → syntactic transparency

Syntax transparency → semantic transparency

Local Semantics

Location at which call is made is not determined by call

Callee cannot perform caller-specific actions

The call is made exactly once (if the callee crashes so does the caller)

Whether an actual parameter is copied is determined by whether it is a call-by-value or call-by-reference

Caller and caller can share memory if language has call-by-reference or pointers

Caller waits for callee to finish: Synchronous call

No action taken to enable callee execution: implicit receive

Local Semantics (Review)

Location at which call is made is not determined by call

Callee cannot perform caller-specific actions

The call is made exactly once (if the callee crashes so does the caller)

Whether an actual parameter is copied is determined by whether it is a call-by-value or call-by-reference

Caller and caller can share memory if language has call-by-reference or pointers

Caller waits for callee to finish: Synchronous call

No action taken to enable callee execution: implicit receive

Destination Binding

Location at which call is made is not determined by call

Implied by syntax transparency

Can port existing code

Earlier binding

Source Transparency

Callee cannot perform caller-specific actions

Implied by syntax transparency

Unless RPC system provides calls to determine caller

Can port existing code

In many applications, want caller-specific actions

Relaying, session manager

Authentication; if caller provides info, can cheat

Number of Invocations

Exactly once semantics

Complicated, costly orphan algorithms needed to implement it (Bruce Nelson's Thesis)

At most once semantics

No duplicate calls as long as no destination failure

Reliability overhead

At least once/Zero or more

Can have duplicate or no calls so "at least once" misnomer

Works only for idempotent calls

Stateless File Servers

Failure

At least once and at-most once semantics require failure communication/recovery
Time-out
Exception, error code

Parameter Transparency and Shared Memory

Whether an actual parameter is copied is determined by whether it is a call-by-value or call-by-reference

In call by reference caller can share data conveniently (without caller registering a handle & passing name, and callee looking up)

Sharing memory requires messages to caller site

In O-O languages, many reads/updates can be made by one access

Parameter transparency in Java?

In Java, serializing does not provide semantic transparency

Remote does

Synchronous Calls

Caller waits for callee to finish: Synchronous call

Programming easier if waiting is necessary, allows migration of non distributed programs

Unnecessary waiting if not necessary

Callbacks → deadlocks if special threads not created

Requires duplex port underneath

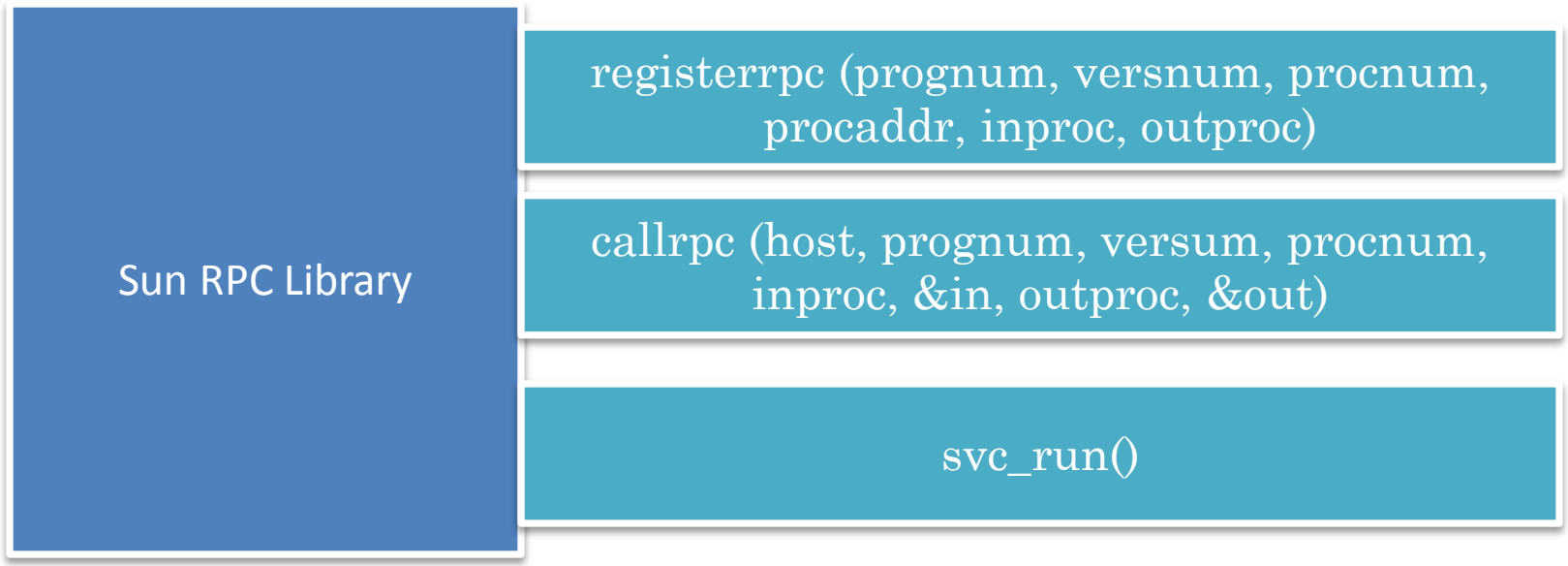
Implicit Calls

No action taken to enable callee execution: implicit receive

Programming easier if no synchronization needed, allows porting of non distributed programs

Must rely on monitors to synchronize, no Ada like guards or select

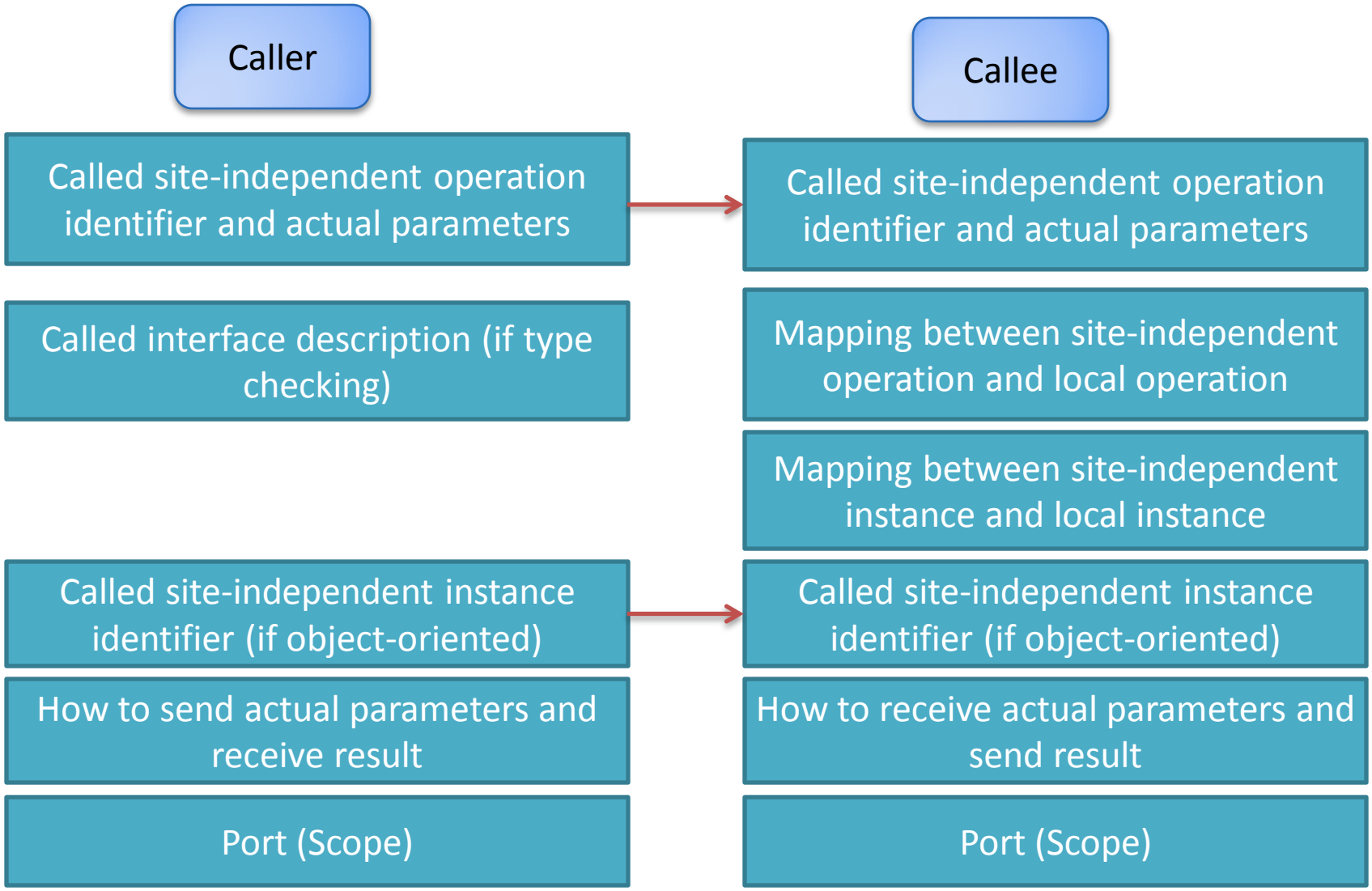
Full Awareness: Sun non-OO RPC



Operation invocation is not a proxy call but an API call such as send on some library

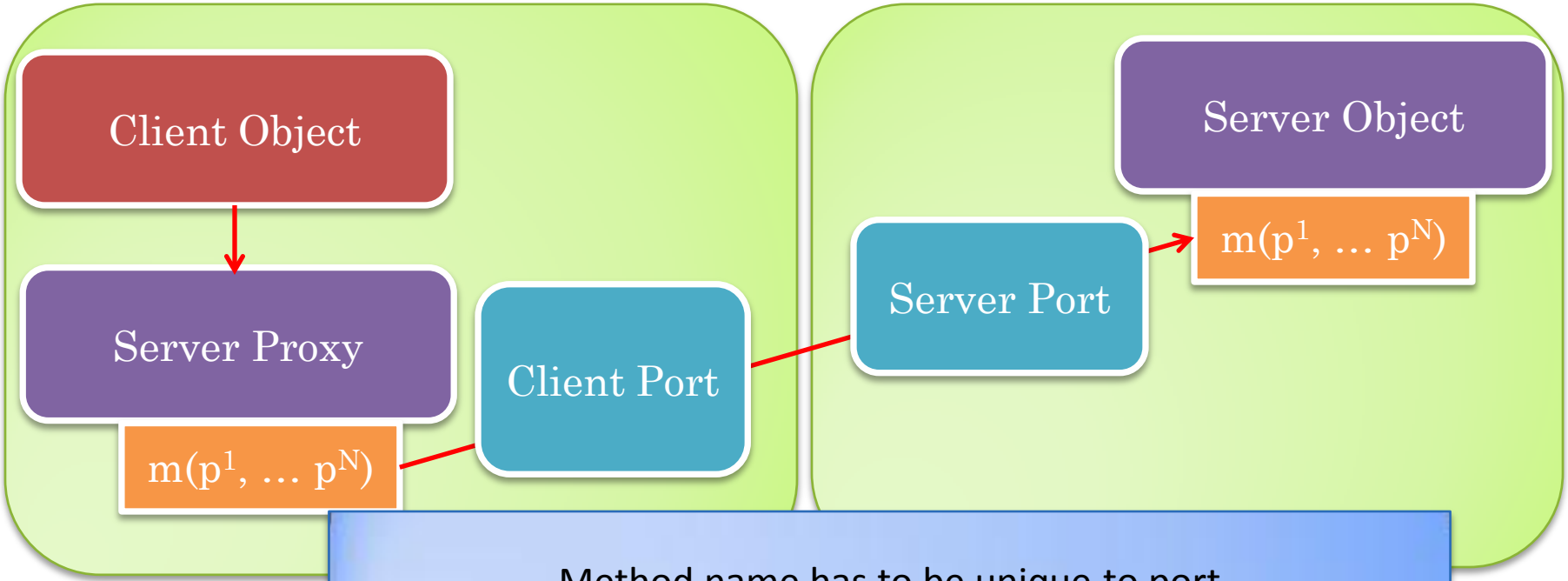
Extra Slides

Information Needed by RPC System



Systems differ in how much information provided by application – how aware it is of RPC

Port vs. GIPC-RMI

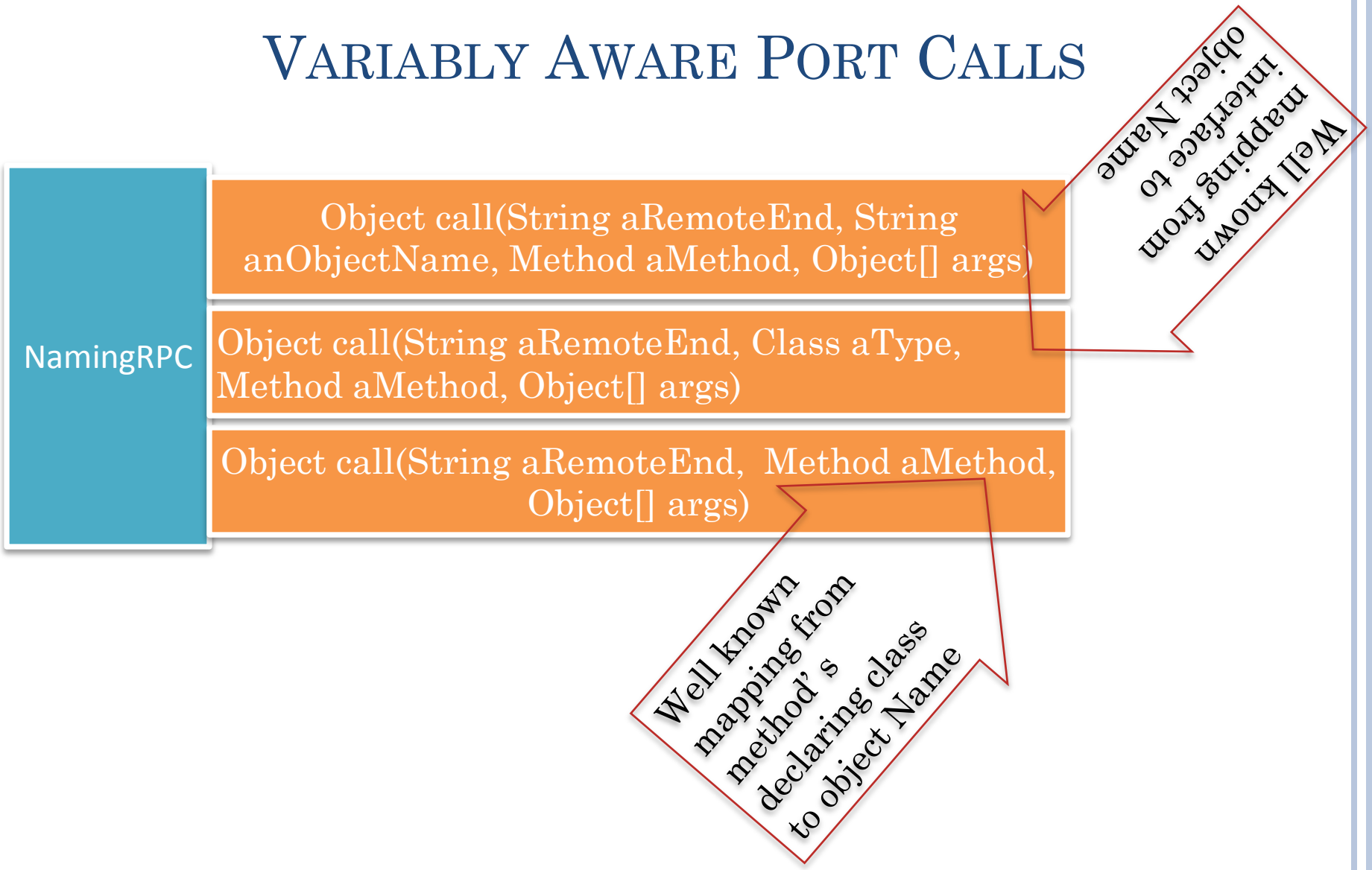


Method name has to be unique to port

Allows remote assignment and remote procedure call on same port

Distinguishing between regular and RPC messages?

Variably Aware Port Calls



Transparency in Application → Awareness in System

Call Initiating Port Code

```
public Object call(String destination, String objectName,
              Method method, Object[] args) {
  Object call = marshallCall(objectName, method, args);
  callSendTrapper.send(destination, call);
  return callSendTrapper.returnValue(
              destination, call);
}
protected Object marshallCall(
              String objectName, Method method, Object[] args) {
  return marshaller.marshallCall(objectName, method, args);
}
public Object call(String destination, Class type,
  Method method, Object[] args) {
  return call (destination, type.getName(), method, args);
}

public Object call(String destination, Method method, Object[] args){
  return call(destination, method.getDeclaringClass(), method, args);
}
```

Very little processing in port

Simplex Send Trapper

```
public void send(String aDestination, Object aMesage) {
  Tracer.info(this, "Forwarding call " + aMesage +
              " to " + aDestination);
  destination.send(aDestination, aMesage);
}
public Object returnValue(String aDestination, Object aMessage) {
  Tracer.info(this, "Waiting for return value of call "
                  + aMessage);
  return simplexSentCallCompleter.returnValueOfRemoteMethodCall(null,
                (Call) aMessage);
}
```

Code shared by Client and Server RPC Ports

Why not do complete processing of call?

Sent Call Completer Interface

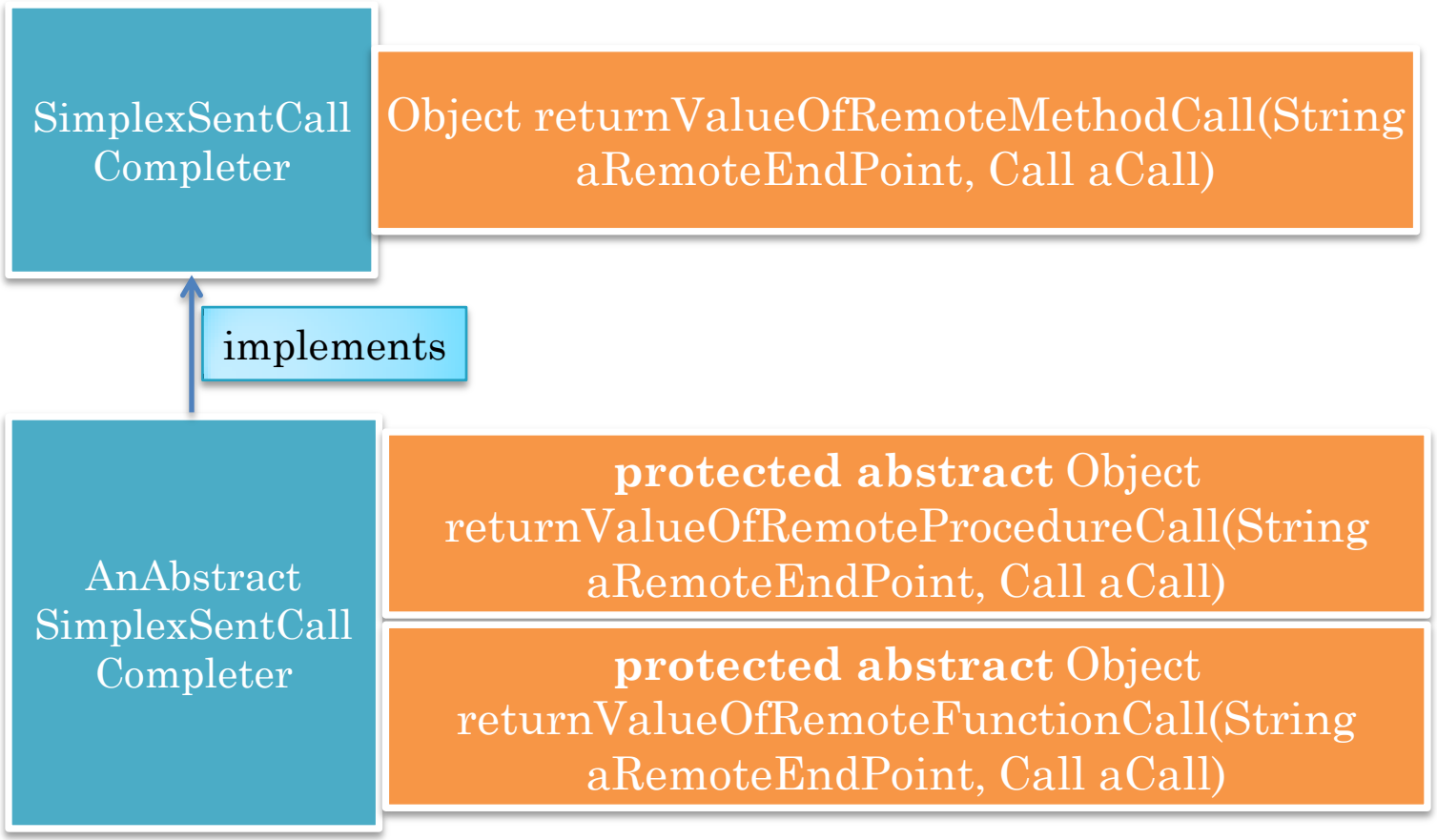
SimplexSentCall Completer

Object returnValueOfRemoteMethodCall(String aRemoteEndPoint, Call aCall)

In simplex case remote procedure and function call must be asynchronous

Not clear what should be returned by remote function call

Sent Call Completer Abstract Class



Default Concrete Sent Call Completer

```
public class ASimplexSentCallCompleter
             extends AnAbstractSimplexSentCallCompleter
             implements SimplexSentCallCompleter {
  protected Object returnValueOfRemoteProcedureCall(
               String aRemoteEndPoint, Object aMessage) {
    return null;
  }
  protected Object returnValueOfRemoteFunctionCall(
               String aRemoteEndPoint, Object aMessage) {
    Call call = (Call) aMessage;
    Tracer.error("Null returned for call on simplex port of
                  method: " + call.getMethod().getName());
    return null;
  }
}
```

Semantics of completion can be changed without understanding rest of code

Receiving Server Port

```
public void messageReceived(String aSource, Object aMessage) {
   getReceiveTrapper().notifyPortReceive(aSource, aMessage);
}
```

Delegates to receive trapper, does little processing

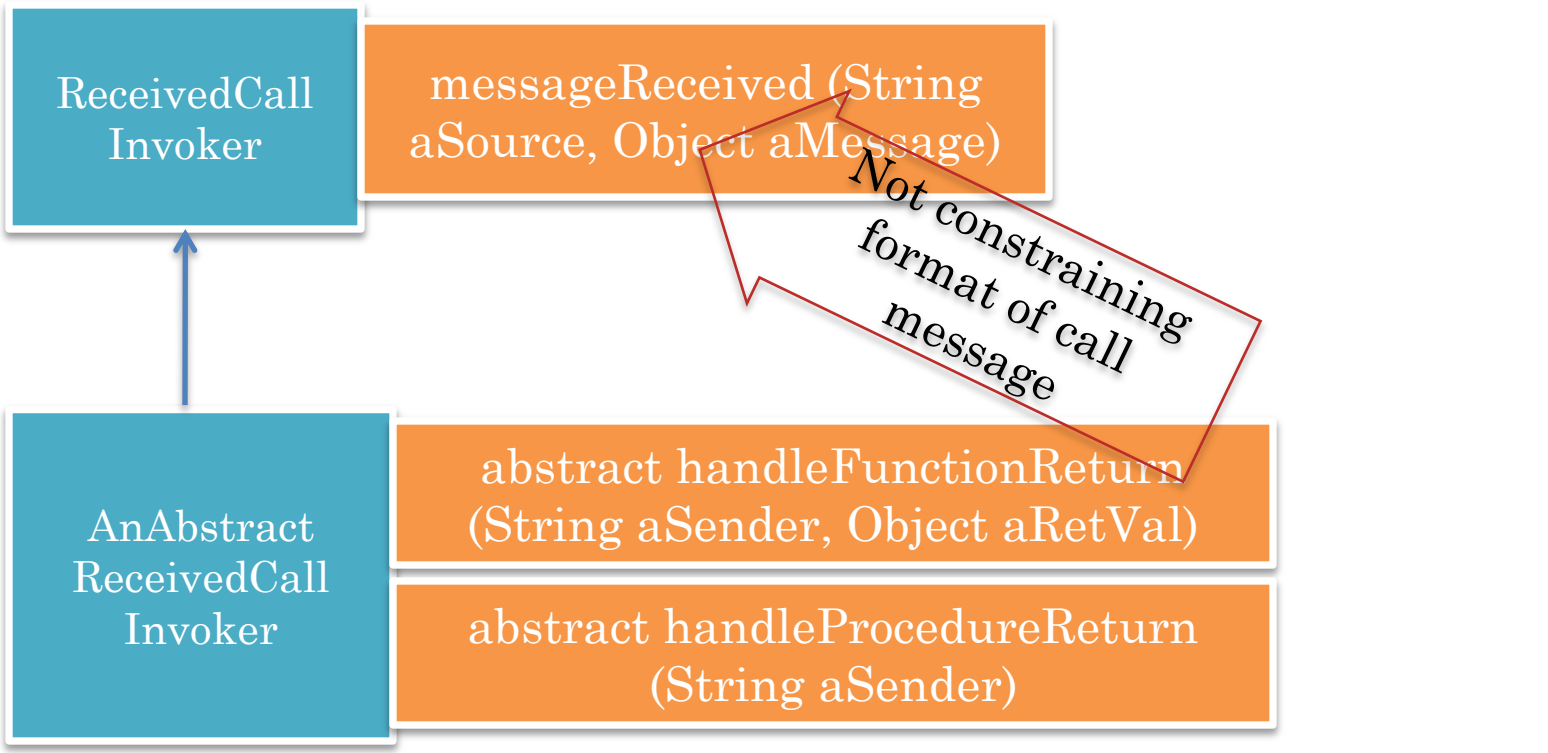
Simplex Receive Trapper

```
public void notifyPortReceive(String remoteEnd, Object message) {
  Tracer.info(this, " Processing serialized call:" +
              message + " from:" + remoteEnd);
  if (message instanceof Call) {
    receivedCallInvoker().messageReceived(
              remoteEnd,  (Call) message);
  }
  else
    destination.notifyPortReceive(remoteEnd, message);
}
```

Code shared by client and server rpc ports

Why not do complete processing in trapper?

Received Call Invoker



Translates received call message into invocation of method in server object

Can change procedure and function return without understanding communication flow

Abstract Received Call Invoker

```
public void messageReceived(String aSender, Object aMessage) {
  Call<String> aCall = (Call<String>) aMessage;
  try {
    Object targetObject =
    rpcRegistry.getServer(aCall.getTargetObject());
    if (targetObject == null) {
      throw new
        RPCOnUnregisteredObjectException(aCall.getTargetObject());
    }
    Object newVal = invokeMethod(aCall.getMethod(), targetObject,
                                 aCall.getArgs());
    if (isProcedure(aCall))
      handleProcedureReturn(aSender);
    else
      handleFunctionReturn(aSender, newVal);
  }catch (Exception e) {
    e.printStackTrace();
  }
}
```

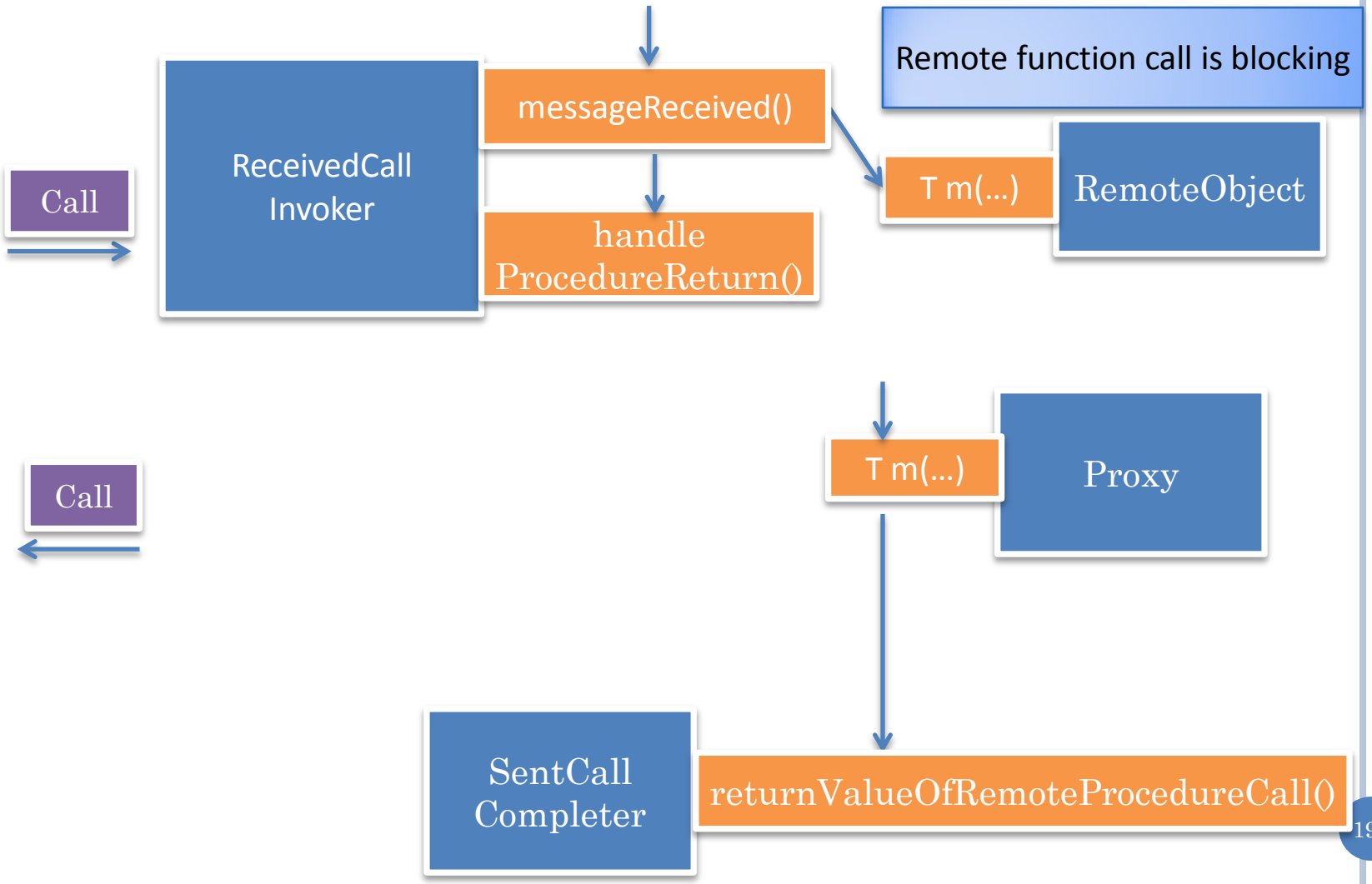
Object message argument gives flexibility in changing call message type, but requires casting

A Simplex Received Call Invoker

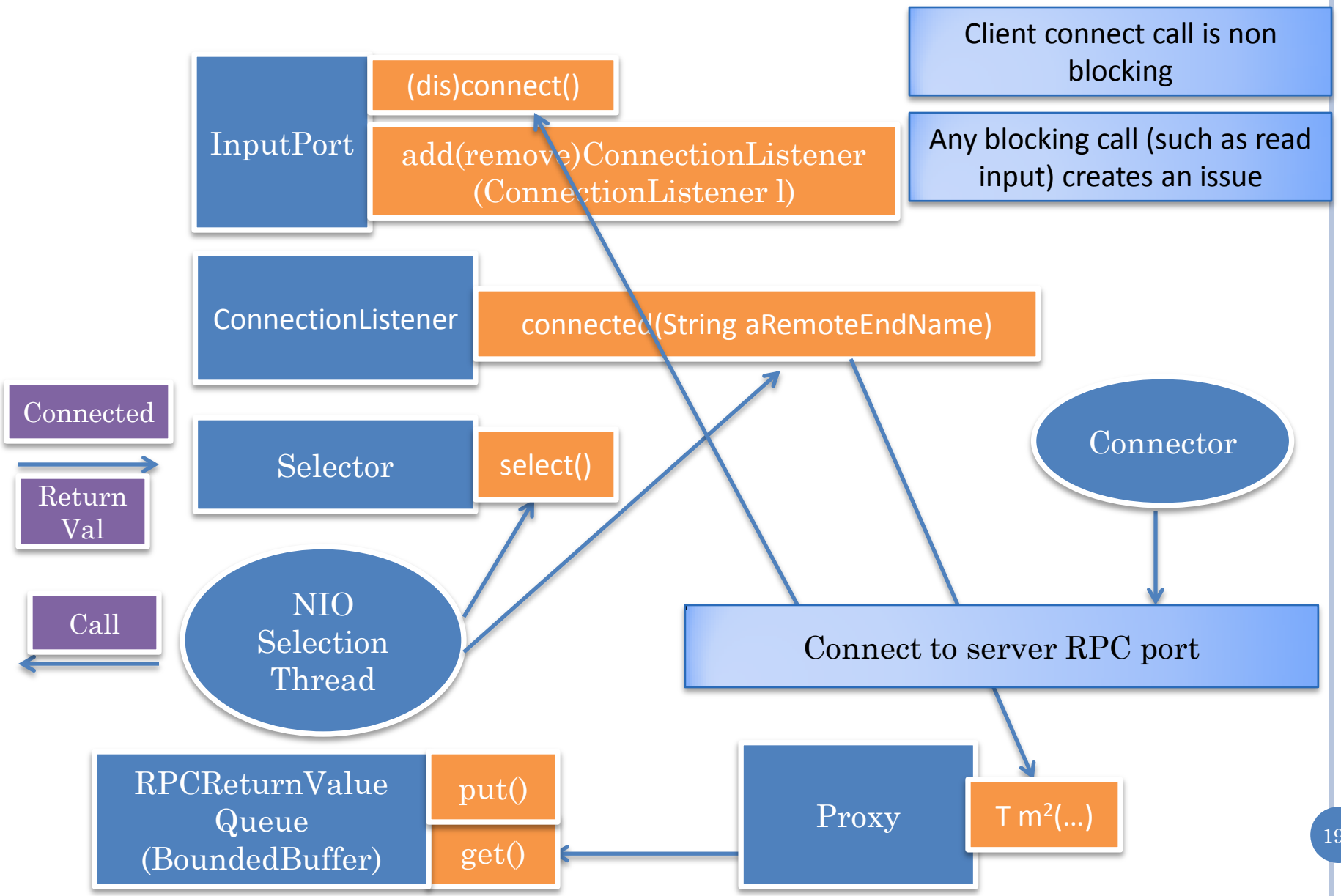
```
protected void handleFunctionReturn(String sender, Object retVal) {
  Tracer.error("Ignoring return val of called method:" + retVal);
}

protected void handleProcedureReturn(String sender) {
  return;
}
```

Can change error message by looking only at this component

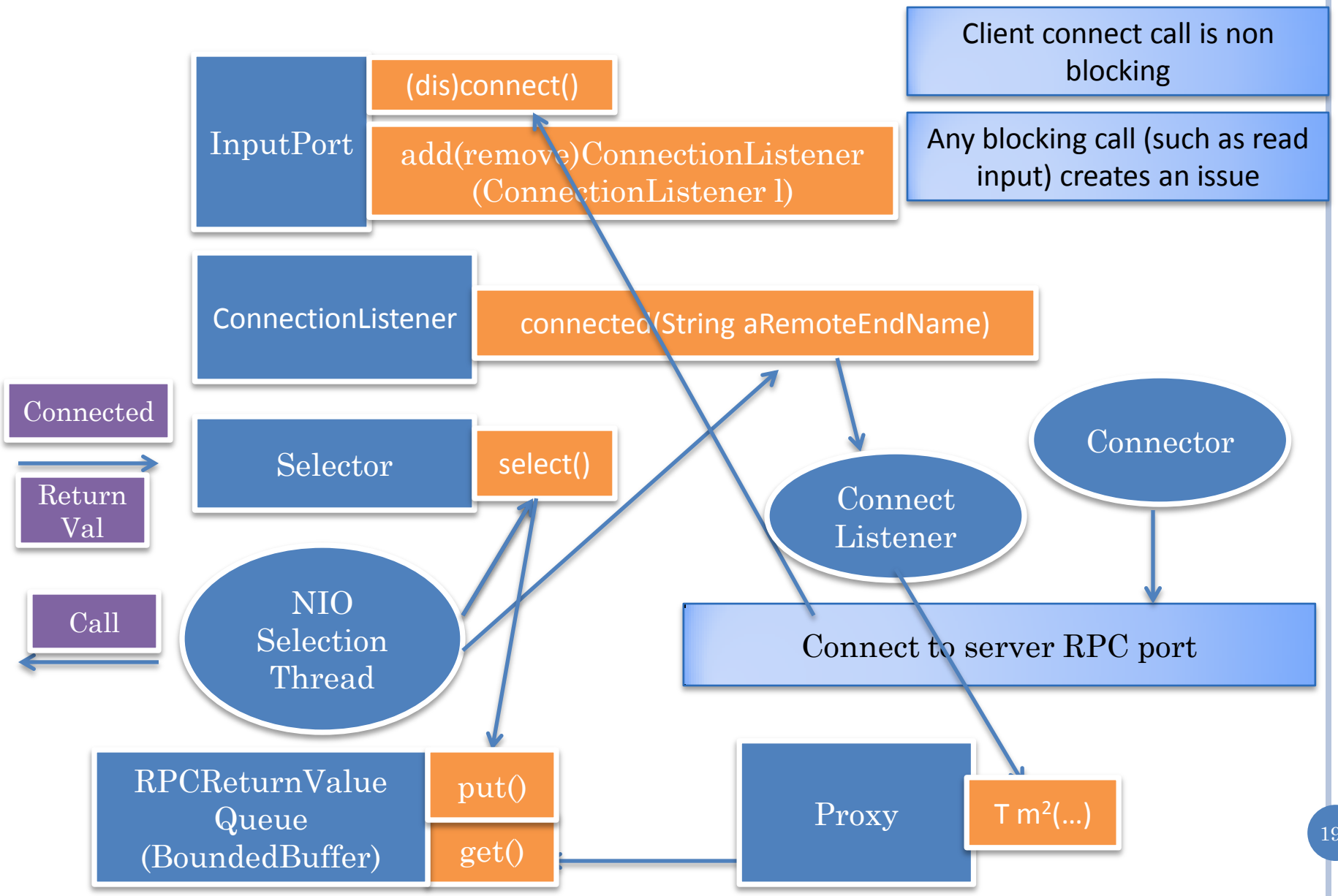
Procedure Call synchronization



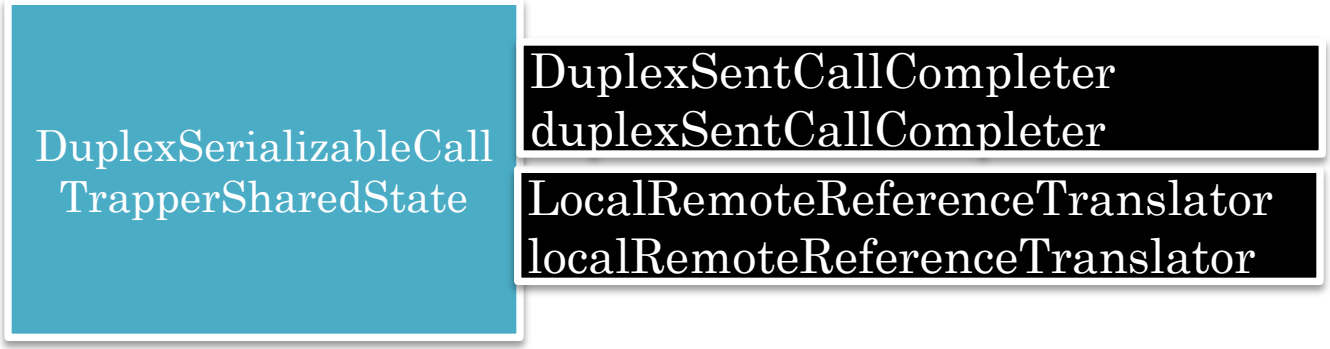
Synchronous Callback in Connect Listener



Breaking Deadlock with Application Thread

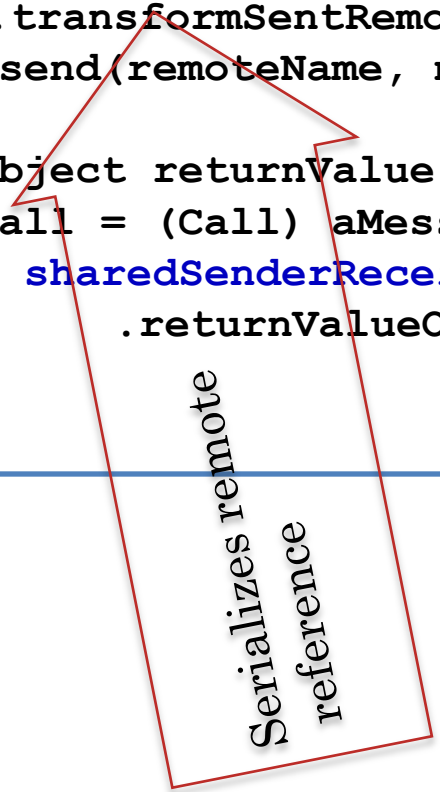


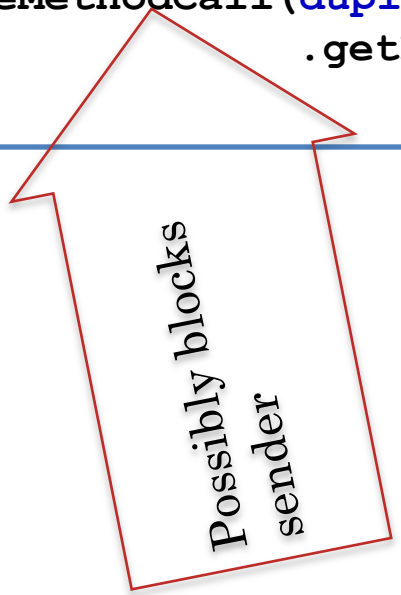
Duplex Shared State



Duplex Send Trapper

```
public Object getSharedSenderReceiverState() {
  return sharedSenderReceiverState;
}
public void send(String remoteName, Object message) {
  Call call = (Call) message;
  sharedSenderReceiverState.localRemoteReferenceTranslator
        .transformSentRemoteReferences(call.getArgs());
  super.send(remoteName, message);
}
public Object returnValue(String aDestination, Object aMessage) {
  Call call = (Call) aMessage;
  return sharedSenderReceiverState.duplexSentCallCompleter
            .returnValueOfRemoteMethodCall(duplexRPCInputPort
                                                .getLastSender(), call);
}
```





Duplex Receive Trapper

```
public void notifyPortReceive(String aSource, Object aMessage) {
  Tracer.info(this, " Processing call:" + aMessage + " from:"
    + aSource);
  DuplexSentCallCompleter returnerOfValueOfRemoteFunctionCall =
    ((DuplexSerializableCallTrapperSharedState) duplexRPCInputPort
        .getSendTrapper().getSharedSenderReceiverState()).
      duplexSentCallCompleter;
  if (!(callCompleter.maybeProcessReturnValue( aSource, aMessage)))
    super.notifyPortReceive(aSource, aMessage);
}
protected ReceivedCallInvoker createReceivedCallInvoker() {
  LocalRemoteReferenceTranslator localRemoteReferenceTranslator =
  ((DuplexSerializableCallTrapperSharedState) duplexRPCInputPort
    .getSendTrapper().getSharedSenderReceiverState()).
          localRemoteReferenceTranslator;
   return DuplexReceivedCallInvokerSelector
             .createDuplexReceivedCallInvoker(
                localRemoteReferenceTranslator,
                duplexRPCInputPort, rpcRegistry);
}
```

Used before call made

Possibly unblocks waiting sender

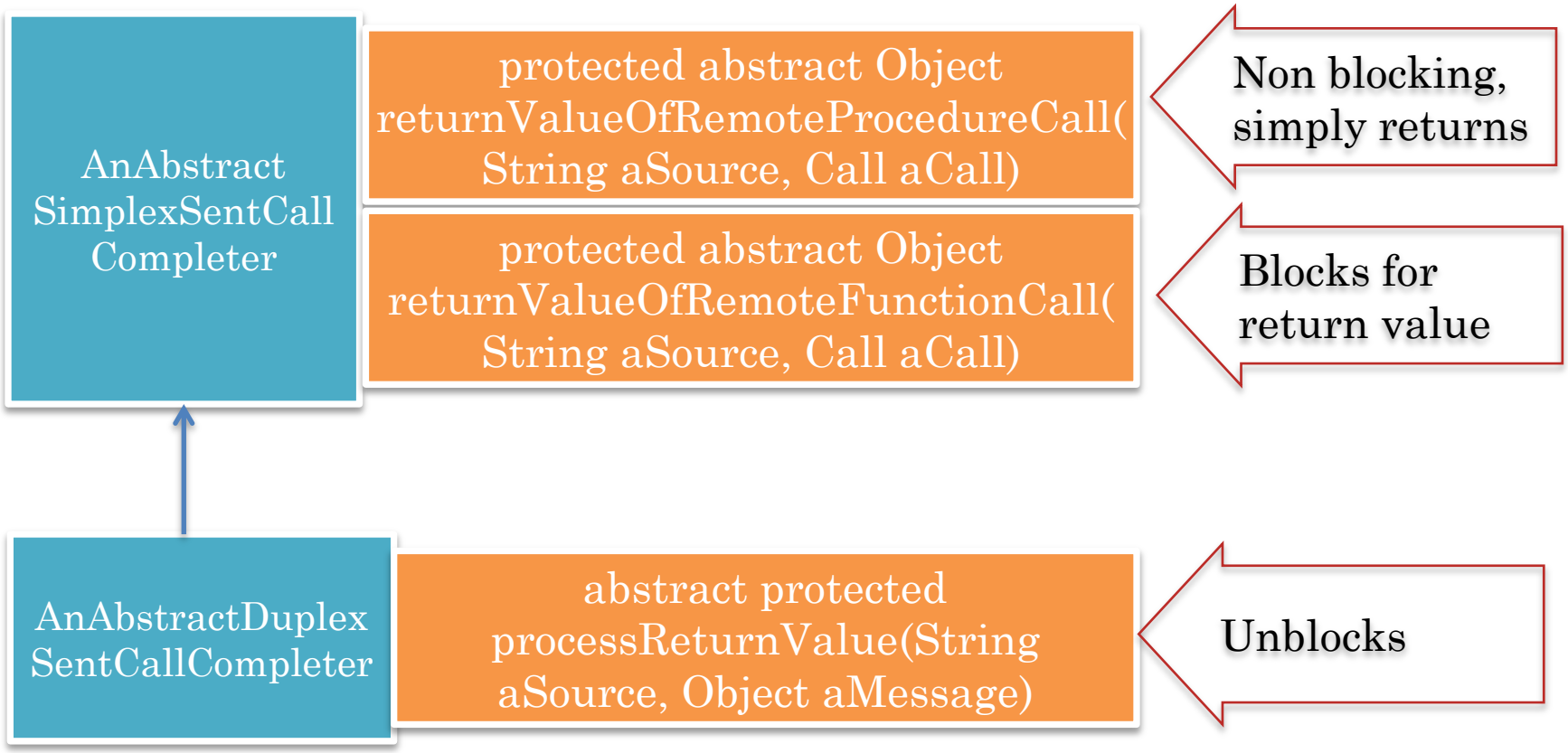
ADuplexReceivedCallInvoker

```
protected Object invokeMethod (Method method, Object targetObject,
Object[] args) {
  localRemoteReferenceTranslator.transformReceivedreferences(args);
  return super.invokeMethod(method, targetObject, args);
}
protected void handleFunctionReturn(String aSource, Object retVal) {
  Object possiblyTransformedRetVal =
      localRemoteReferenceTranslator.transformSentReference(retVal);
  replier.send (aSource,
    createRPCReturnValue(possiblyTransformedRetVal));
}
```

Procedure call handled the same as in simplex, as default procedure call is asynchronous

Can change to synchronous by overriding handleProcedureReturn

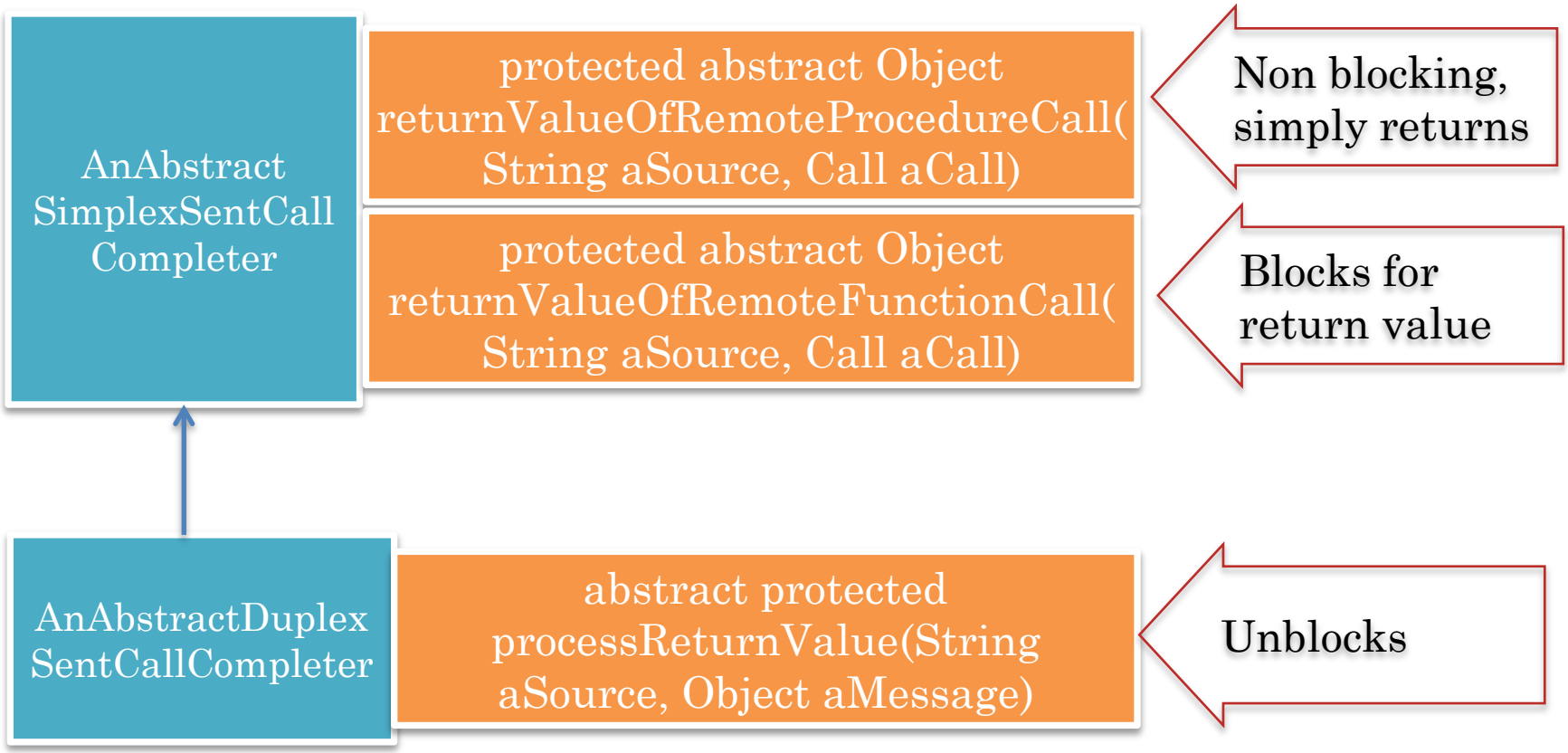
Sent Call Completer: Helper Classes



Specialized Bounded Buffer: AnRPCReturnValueQueue

```
public void putReturnValue(RPCReturnValue message) {
  try {
    returnValueQueue.put(message);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
public Object takeReturnValue() {
  try {
    RPCReturnValue message = returnValueQueue.take();
    Object possiblyRemoteRetVal = message.getReturnValue();
    Object returnValue = localRemoteReferenceTranslator
            .transformReceivedReference(possiblyRemoteRetVal);
    return returnValue;
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }
}
```

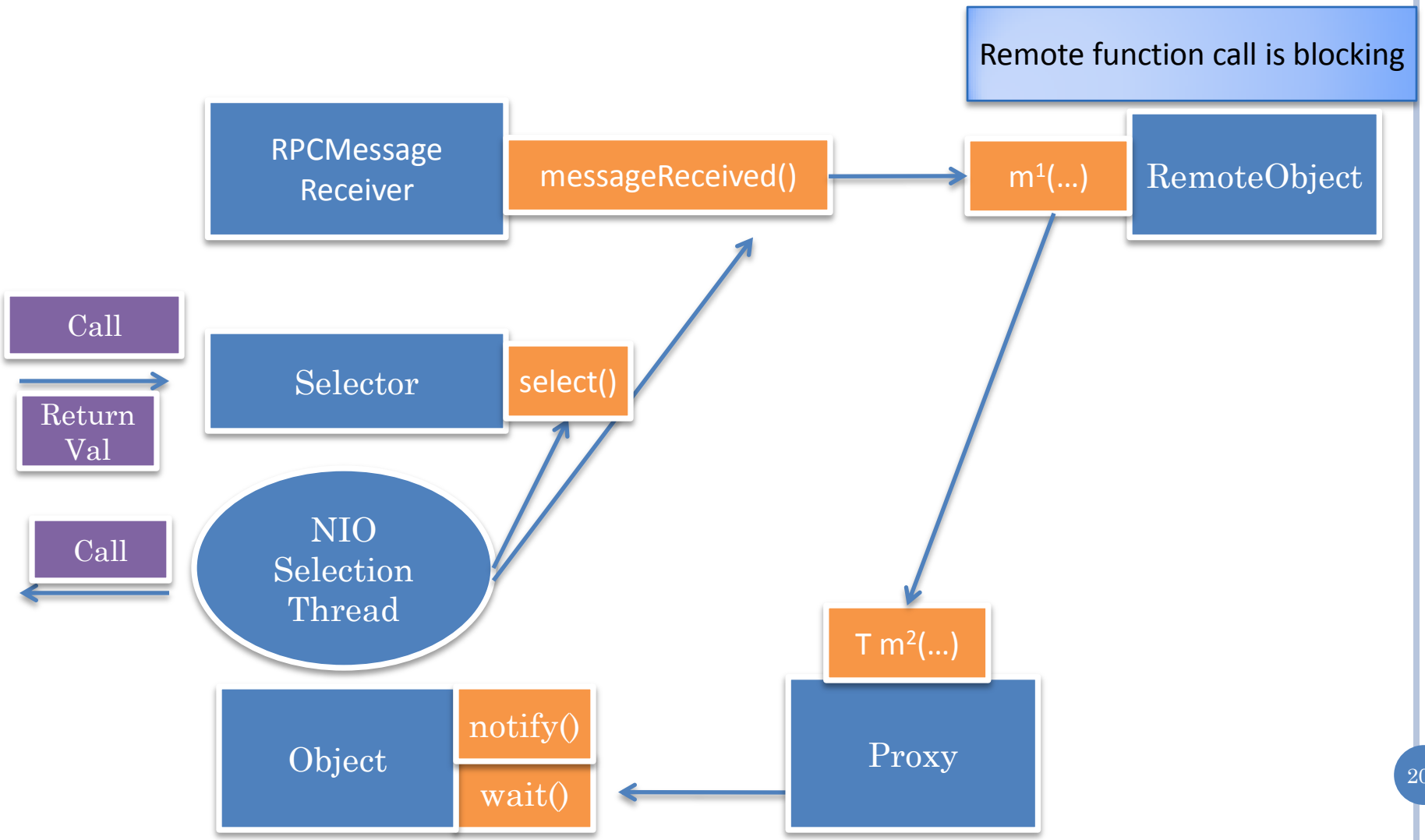
Sent Call Completer: Helper Classes



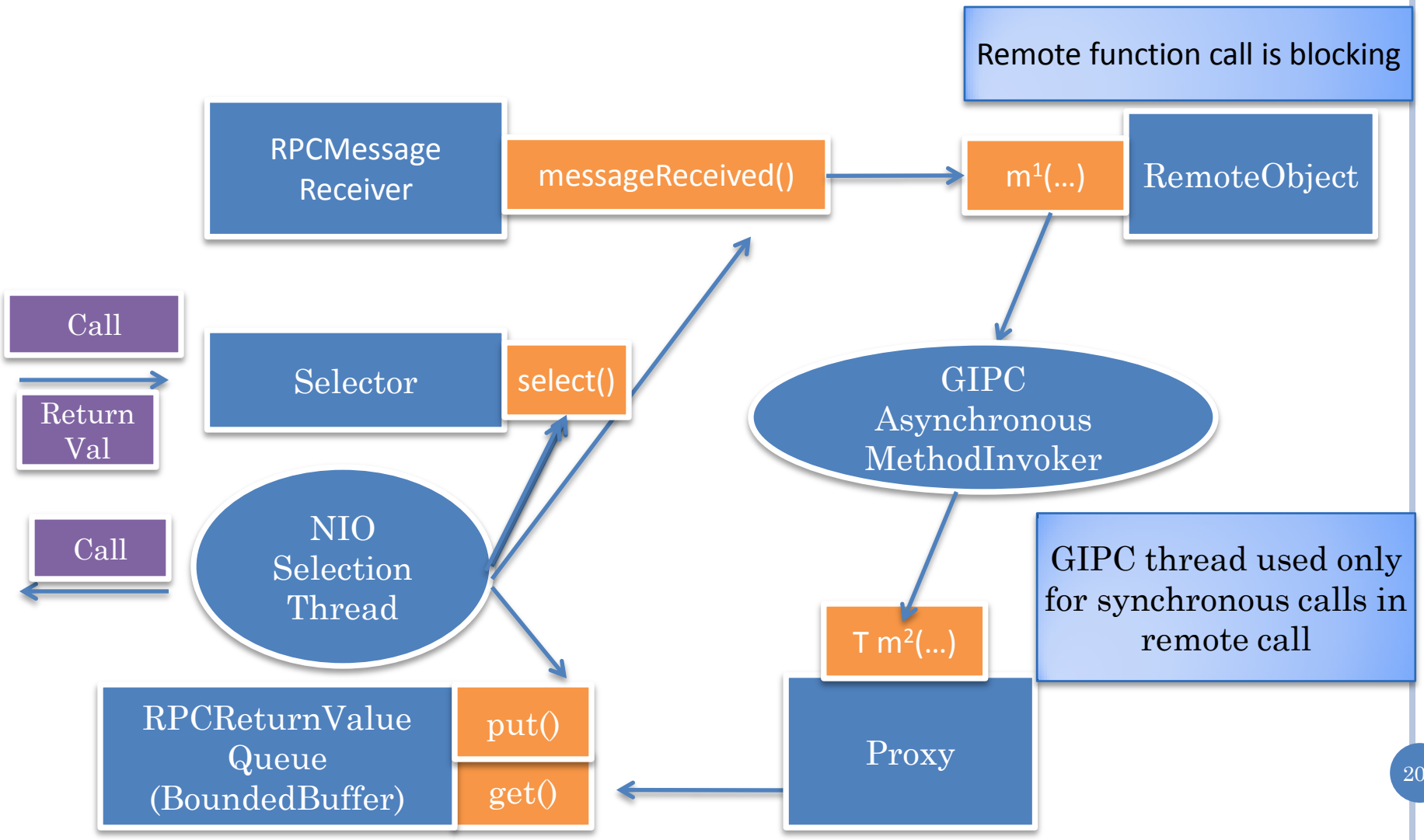
Producer/Consumer Calls in: ADuplexSentCallCompleter

```
//called by sending thread
public Object returnValueOfRemoteFunctionCall (String
aRemoteEndPoint, Object aMessage) {
  RPCReturnValueQueue rpcReturnValueReceiver =
        getRPCReturnValueReceiver(aRemoteEndPoint);
  Object returnValue =
        rpcReturnValueReceiver.takeReturnValue();
  Tracer.info(this, "took return value:" + returnValue);
  return  returnValue;
}
//called by receiving thread
protected void processReturnValue(String source, Object
message) {
  RPCReturnValueQueue rpcReturnValueReceiver =
         getRPCReturnValueReceiver(source);
  rpcReturnValueReceiver.putReturnValue(
        (RPCReturnValue) message);
}
```

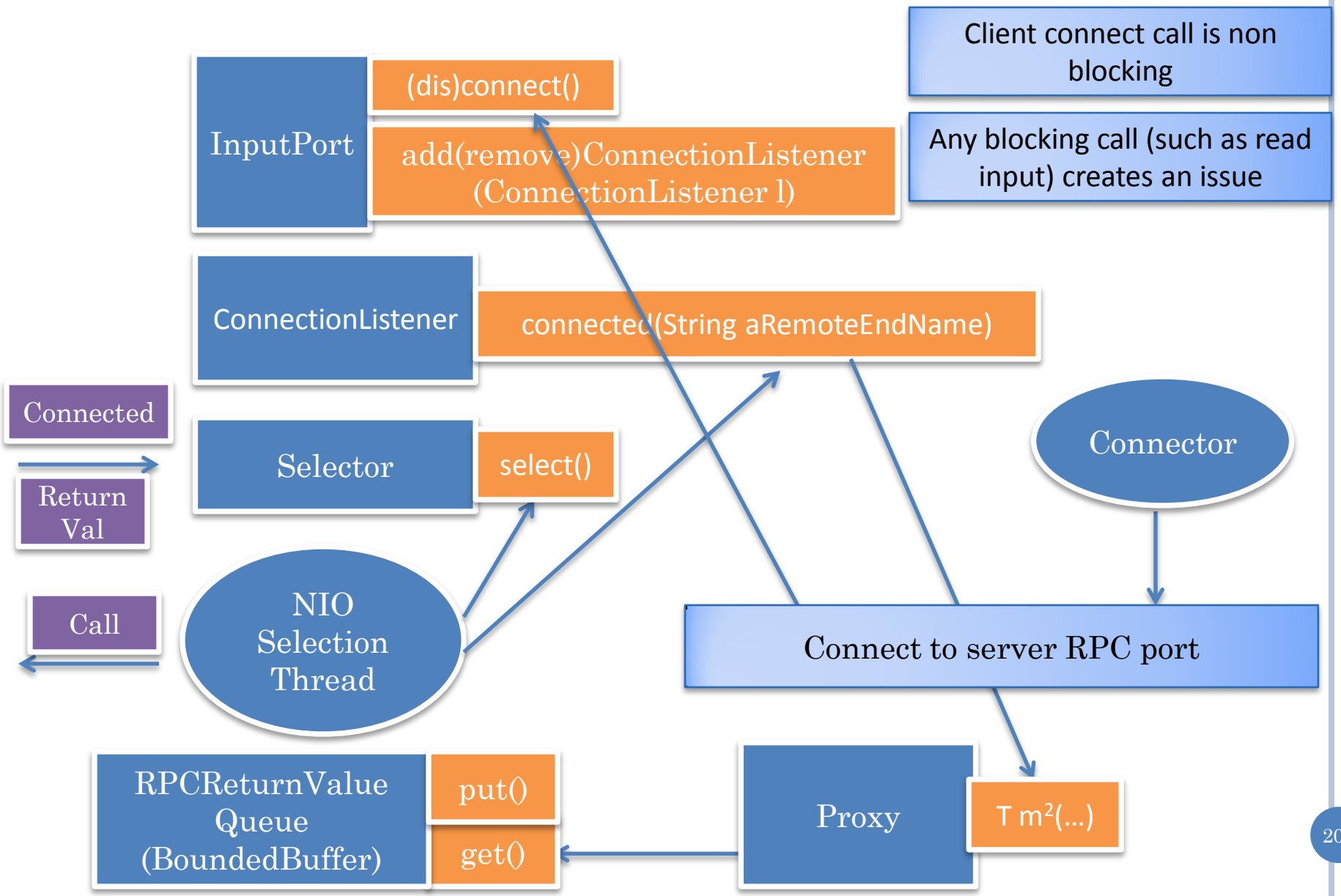
Synchronous Callback in Remote Call



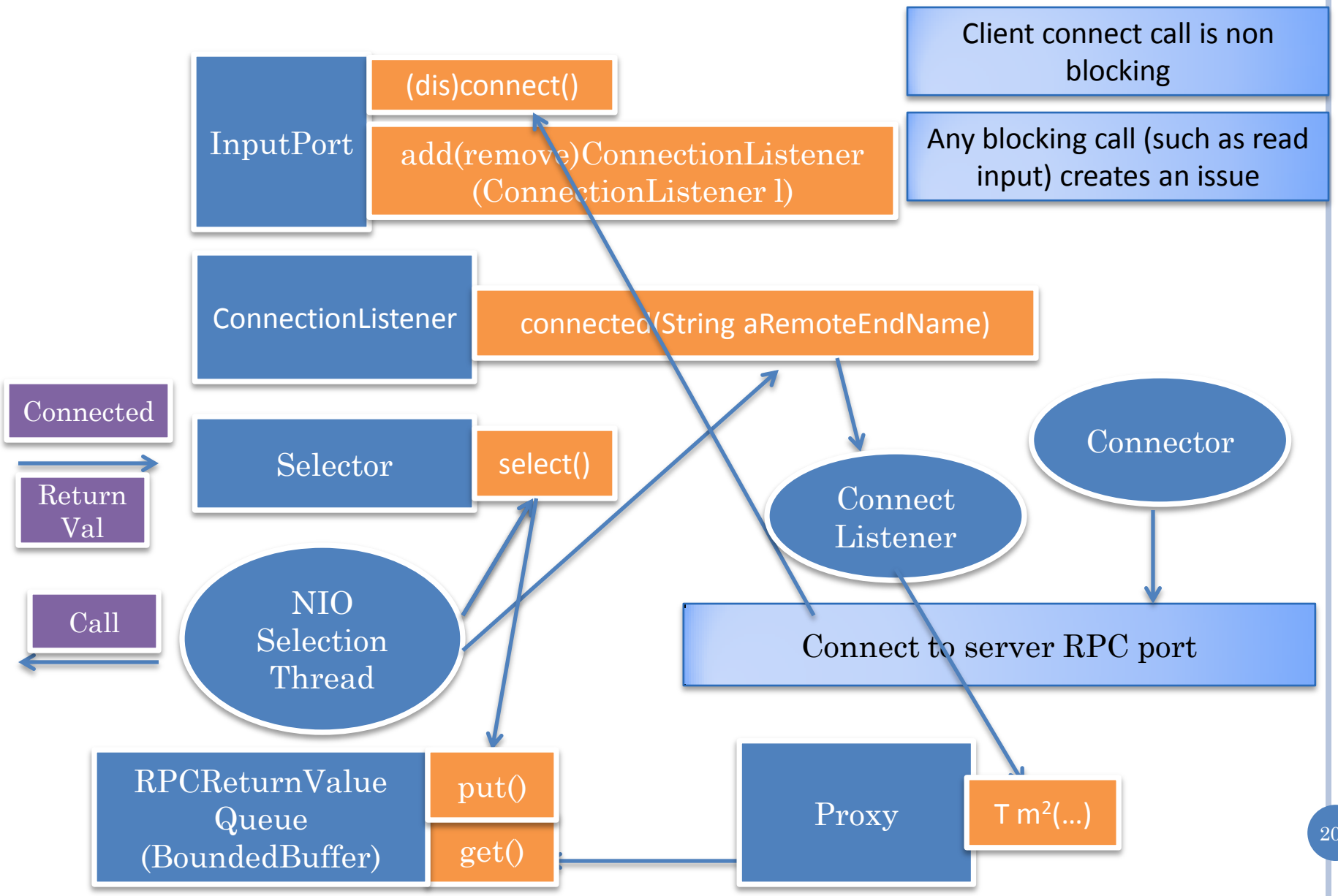
Breaking Deadlock with GIPC Thread



Synchronous Callback in Connect Listener



Breaking Deadlock with Application Thread



Direct Server Proxy