IPC Design Space → GIPC

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)

Design Space

	Xinu	Pipes	Sockets	NIO	GIPC
Location	Intra-Proc	Intra-Host	Distributed	Distributed	Distributed
Reliable, In-Order?	Yes	Yes	Yes/No	Yes/No	
Access?	Input	Free	Bound./Inp	Bound./Inp	
Send block times	No	System Buf	System buf	SB/No	
Explicit receive?	Yes	Yes	Yes	Yes	
Rcv blck times?	Sync/No	Part Sync	Sync	Sync/No	
Select?	No	No	No	Yes	
Language?	No	No	No	No	No
Naming?					
Reply?					
Reply blck times?	N/A	N/A	N/A	N/A	
Msg Unit	word	Byte	Byte	Byte	
Buf size?	1 word				

How can we improve distributed IPC without language support?

Goals

Customizable Implementation

Unlike RMI (should) run on Android

Informed by the IPC Design Space

How can we “improve” distributed IPC without language support?

Access

	Xinu	Pipes	Sockets	NIO	GIPC
Location	Intra-Proc	Intra-Host	Distributed	Distributed	Distributed
Reliable, In-Order?	Yes	Yes	Yes/No	Yes/No	
Access?	Input	Free	Bound./Inp	Bound./Inp	Inp/Session/ Broadcast
Send block times					
Explicit receive?					
Rcv blck times?					
Select?					
Language?					
Naming?					
Reply?	No	No	No	No	
Reply blck times?	N/A	N/A	N/A	N/A	
Msg Unit					
Buf size?	1 word				

No input port abstraction allowing reliable data communication

Must be simulated by multiple sockets and channels

No broadcast and session port

GIPC: Input, Broadcast, Sesssion Port, Simplex and Duplex, Cannot derive duplex ports from input port

Two Dimensional Access

```
public enum PortKind {
  CLIENT_INPUT_PORT,
  SERVER_INPUT_PORT,
  SESSION_PORT,
  MULTI_SERVER_PORT
}
```

```
public enum PortAccessKind {
  SIMPLEX,
  DUPLEX,
  GROUP
}
```

```
ByteBuffer message = …;
clientInputPort.send(message);
```

```
ByteBuffer message = …;
serverInputPort.send(message);
```

Any port in which the queue is kept with the receiver is called an input port

Message Kind and Unit

	Xinu	Pipes	Sockets	NIO	GIPC
Location					
Reliable, In-Order?					
Access?					
Send block times	No	System Buf	System buf	SB/No	
Explicit receive?	Yes	Yes	Yes	Yes	
Rcv blck times?					
Select?					
Language?					
Naming?					
Reply?					
Reply blck times?	N/A	N/A	N/A	N/A	
Msg Unit	word	Byte	Byte or Objects	Byte or Object	Buf, Object, RPC
Buf size?	1 word				

Message unit?

Byte or Objects, and RPC hides both

ByteBuffer unit: no fragmentation on receive, unit sent == unit received

GIPC: Bytebuffer, Object and RPC orthogonal to other features

Complete Port Description

```
public enum PortKind {
  CLIENT_INPUT_PORT,
  SERVER_INPUT_PORT,
  SESSION_PORT,
  MULTI_SERVER_PORT
}
```

```
public enum PortAccessKind {
  SIMPLEX,
  DUPLEX,
  GROUP
}
```

```
public enum PortMessageKind {
  BUFFER,
  OBJECT,
  RPC
}
```

```
public interface PortDescription {
  PortKind getPortKind();
  void setPortKind(PortKind aPortKind);
  PortAccessKind getPortAccessKind();
  setPortAccessKind(PortAccessKind aPortAccessKind);
  PortMessageKind getPortMessageKind();
  setPortMessageKind(PortMessageKind aPortMessageKind);
  …
}
```

Monolithic Port class/interface?

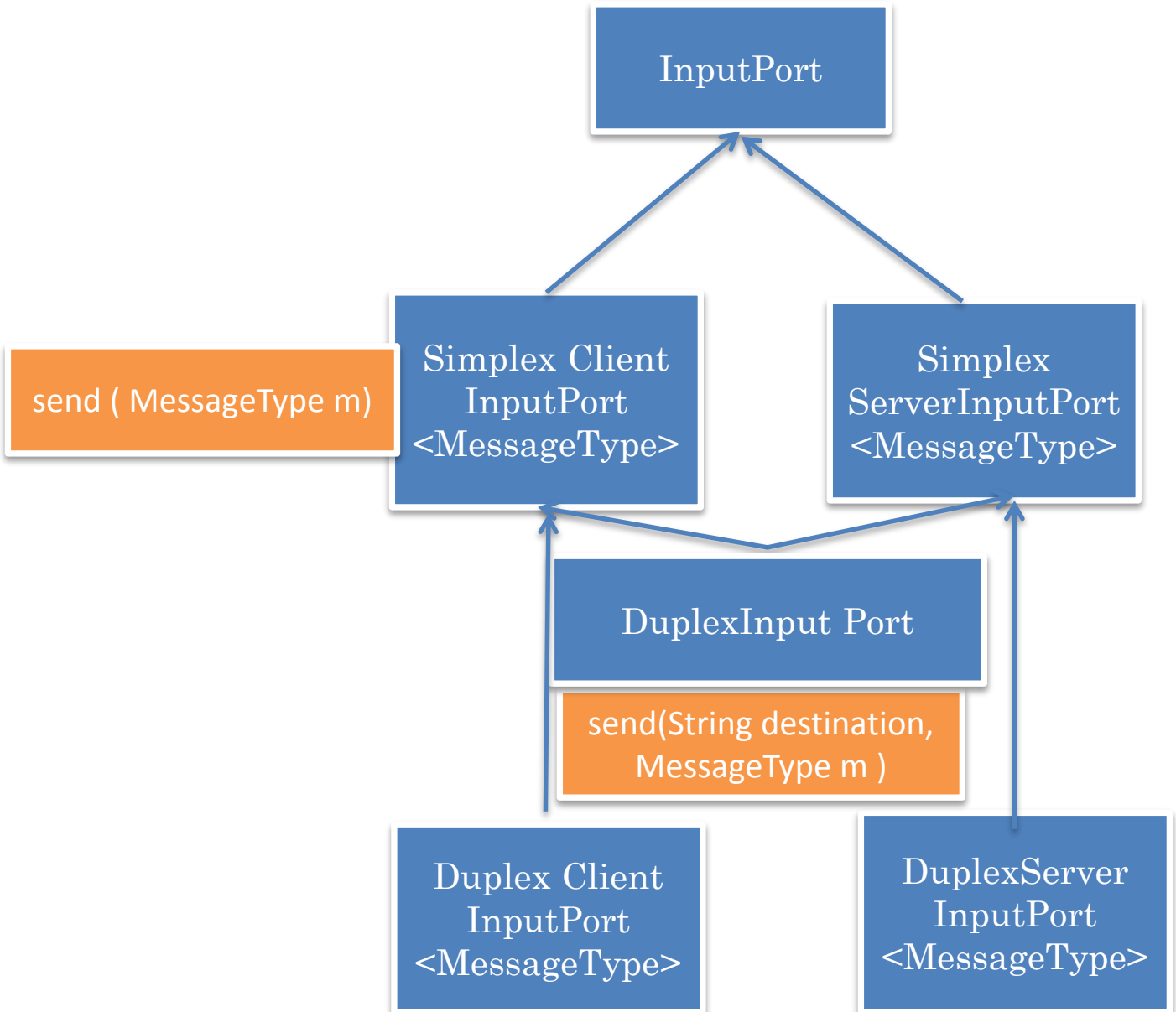
Port Types?

Each combination of parameters is a different port type

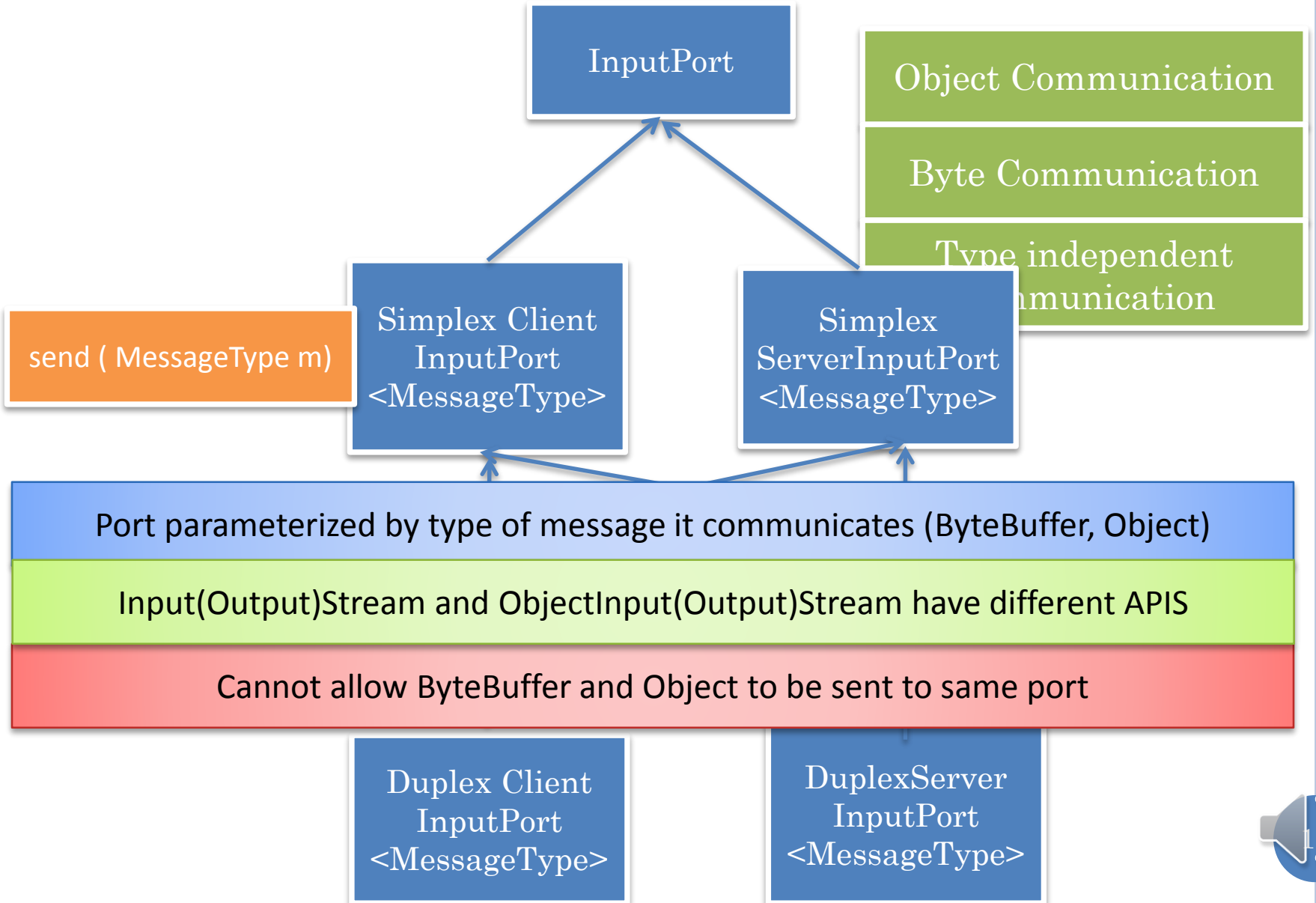
Some types subsume others, so IS-A relationship preserved

Abstract types to capture similarities

Some IS-A Relationships



Generic Interface



Send Buffer vs. Object

```
protected void processInput(String anInput) {
    ByteBuffer message =  ByteBuffer.wrap(anInput.getBytes());
    simplexBufferClientPort.send(message);
}
```

```
protected void processInput(String anInput) {
    simplexObjectClientPort.send(anInput);
}
```

Integration with RPC?

Can send messages and invoke methods on the same port

Remote Procedure Call: Example

```
rpcServerPort.register(COUNTER1, new ACounter());
```

```
RPCProxyGenerator rpcProxyGenerator = rpcClientPort.getRPCProxyGenerator();
counter11Proxy = (Counter) rpcClientPort.
                              generateRPCProxy(ACounter.class, COUNTER1);
```

Registration and Lookup

SimplexRPCServer InputPort

register(String aName, Object aRemoteObject)

DuplexRPCInputPort

RPCProxyGenerator getRPCProxyGenerator()

Object generateRPCProxy (Class aType, String aName)

ADuplexRPCProxy Generator

Object generateRPCProxy (String aRemoteEnd, Class aType, String aName)

Callee registers method relative to port, and caller invokes methods relative to it

Data transfer can also be done with respect on the same port

Naming

	Xinu	Pipes	Sockets	NIO	GIPC
Location	Int				ributed
Reliable, In-Order?					
Access?	I				
Send block times					
Explicit receive?					
Rcv blck times?	Sy				
Select?					
Language?	No	No	No	No	
Naming?	pid	filedescp	Host, Port#	Host, Port#	Host, Id
Reply?	No	No	No	No	
Reply blck times?	N/A	N/A	N/A	N/A	
Msg Unit	word	Byte	Byte	Byte	
Buf size?	1 word				

Socket and NIO use port within machine

GIPC: Use more general concept of String Id as unique physical address

Client and server have additional logical names unique within the scope of input port

Logical names with each message to address destination as we have multiple parties

Naming

```
SimplexServerInputPort<ByteBuffer> aServerInputPort =
      BufferSimplexInputPortSelector.createServerSimplexInputPort(
                SERVER_PORT, SERVER_NAME);
```

```
SimplexClientInputPort<ByteBuffer> clientInputPort =
      BufferSimplexInputPortSelector.createClientSimplexInputPort(
                SERVER_HOST, SERVER_PORT, SERVER_NAME, clientName);
```

```
System.out.println (inputPort.getLocalName())
```

Receive?

	Xinu	Pipes	Sockets	NIO	GIPC
Location					
Reliable, In-Order?					
Access?					
Send block times	No	System Buf	System buf	SB/No	No
Explicit receive?	Yes	Yes	Yes	Yes	No
Rcv blck times?	Sync/No	Sync	Sync	No	N/A
Select?	No	No	No	No	No
Language?					
Naming?					
Reply?					
Reply blck times?					
Msg Unit	word	Byte	Byte	Byte	Buf
Buf size?	1 word				

Blocking receive requires thread programming and runtime overhead if multiple ports

Programming select is tedious

GIPC: No explicit receive, as in RPC

Explicit (receive) can be built on top using monitors

Underlying implementation may create >= 1 thread

Two Dimensional Access (Review)

```
public enum PortKind {
  CLIENT_INPUT_PORT,
  SERVER_INPUT_PORT,
  SESSION_PORT,
  MULTI_SERVER_PORT
}
```

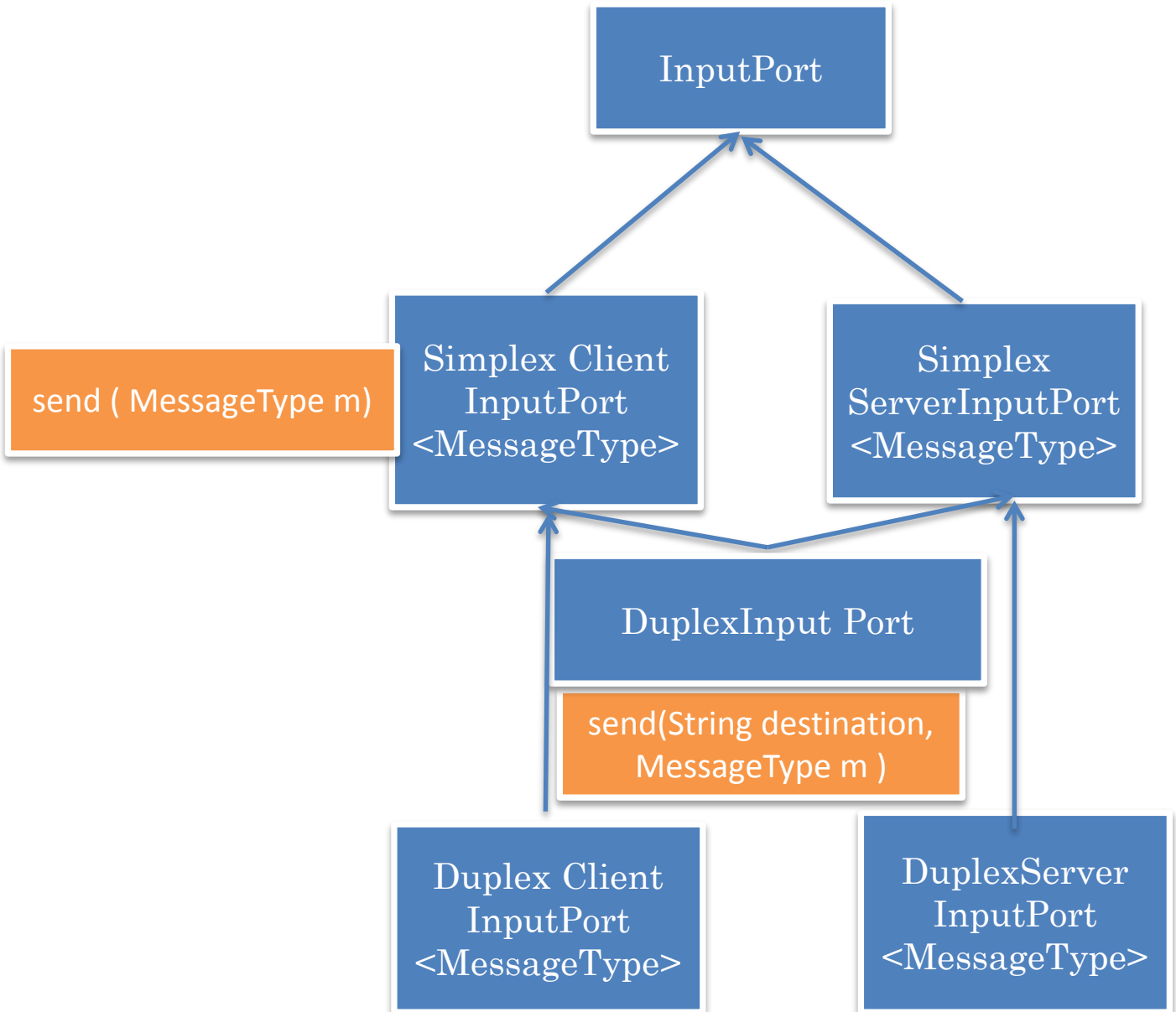
```
public enum PortAccessKind {
  SIMPLEX,
  DUPLEX,
  GROUP
}
```

```
ByteBuffer message = …;
clientInputPort.send(message);
```

```
ByteBuffer message = …;
serverInputPort.send(message);
```

Any port in which the queue is kept with the receiver is called an input port

Some IS-A Relationships (Review)



Registration and Lookup (Review)

SimplexRPCServer InputPort

register(String aName, Object aRemoteObject)

DuplexRPCInputPort

RPCProxyGenerator getRPCProxyGenerator()

Object generateRPCProxy (Class aType, String aName)

ADuplexRPCProxy Generator

Object generateRPCProxy (String aRemoteEnd, Class aType, String aName)

Callee registers method relative to port, and caller invokes methods relative to it

Data transfer can also be done with respect on the same port

Naming (Review)

```
SimplexServerInputPort<ByteBuffer> aServerInputPort =
      BufferSimplexInputPortSelector.createServerSimplexInputPort(
                SERVER_PORT, SERVER_NAME);
```

```
SimplexClientInputPort<ByteBuffer> clientInputPort =
      BufferSimplexInputPortSelector.createClientSimplexInputPort(
                SERVER_HOST, SERVER_PORT, SERVER_NAME, clientName);
```

```
System.out.println (inputPort.getLocalName())
```

Receive? (review)

	Xinu	Pipes	Sockets	NIO	GIPC
Location					
Reliable, In-Order?					
Access?					
Send block times	No	System Buf	System buf	SB/No	No
Explicit receive?	Yes	Yes	Yes	Yes	No
Rcv blck times?	Sync/No	Sync	Sync	No	N/A
Select?	No	No	No	No	No
Language?					
Naming?					
Reply?					
Reply blck times?					
Msg Unit					
Buf size?					

Blocking receive requires thread programming and runtime overhead if multiple ports

Programming select is tedious

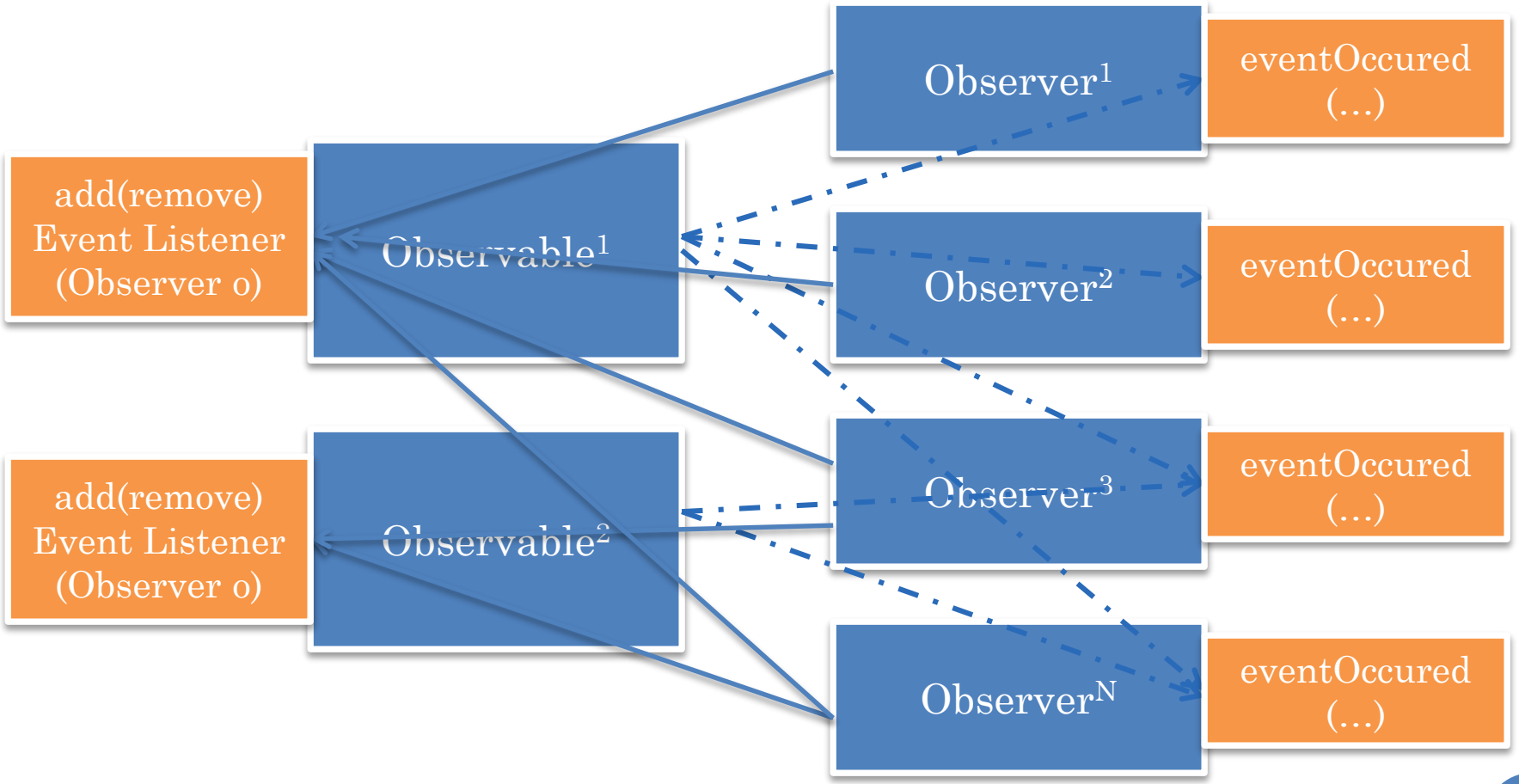
GIPC: No explicit receive, as in RPC

Use observer pattern for completion of send, receive, and all other operations : reactive programming

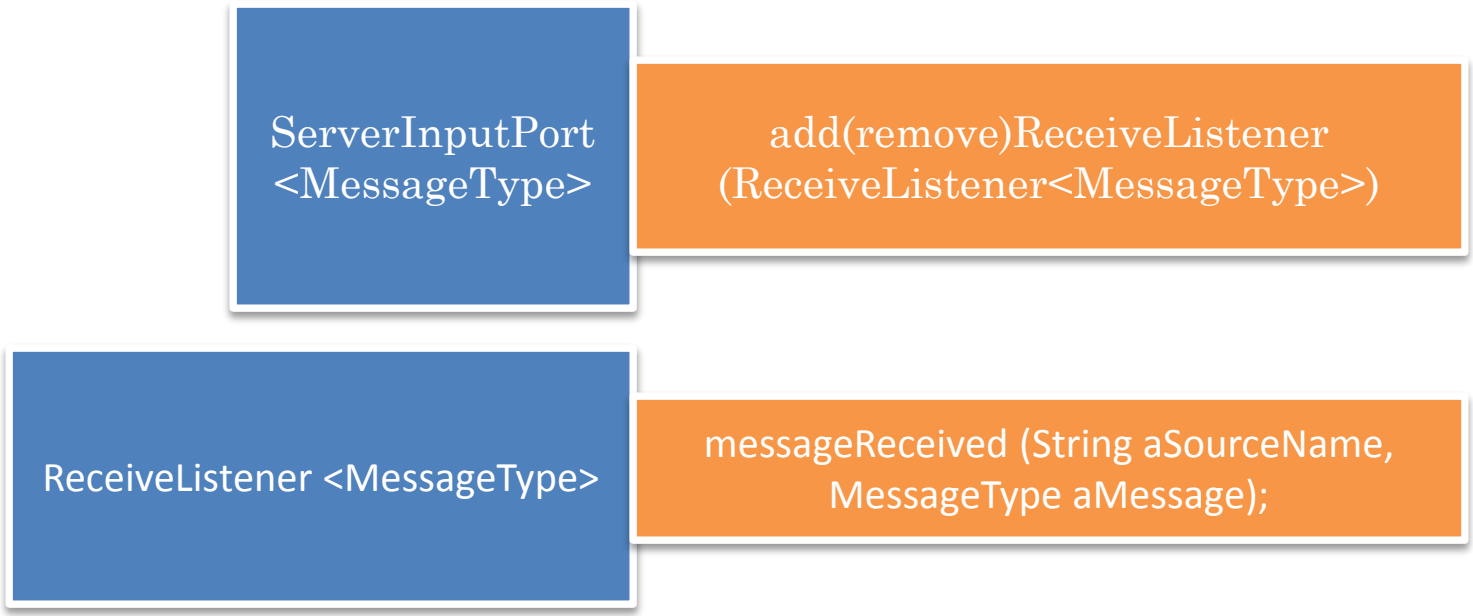
Explicit (receive) can be built on top using monitors

Underlying implementation may create >= 1 thread

Observer Pattern



Receive Observable and Observer



Simplex input port may be bound to different message types

InputPort, so aSourceName indicates logical name of sender. Can be used to "simulate "socket-like duplex ports

Receive notification when complete typed sent message received

Echoing Receive Listener

```
aServerInputPort.addReceiveListener(
        new AnEchoingReceiveListener(aServerInputPort);
}
```

```
public class AnEchoingReceiveListener
                implements ReceiveListener<ByteBuffer> {
  InputPort inputPort;
  public AnEchoingReceiveListener (InputPort anInputPort) {
     inputPort = anInputPort;
  }
  public void messageReceived(String aRemoteEnd,
                              ByteBuffer aMessage) {
     System.out.println(
     inputPort.getLocalName() + "<--" + aRemoteEnd + ":" +
     Misc.toString(aMessage));
  }
}
```

Message Reuse and Blocking Semantics

	Xinu	Pipes	Sockets	NIO	GIPC
Location	Intra-Proc	Intra-Host	Distributed	Distributed	Distributed
Reliable, In-Order?	Yes	Yes	Yes/No	Yes/No	Yes/No
Access?	Input	Free	Bound./Inp	Bound./Inp	
Send block times	No	System Buf	System buf	SB/No	No
Explicit receive?					
Rcv blck times?					
Select?					
Language?	No	No	No	No	
Naming?	pid	filedescp	Host, Port#	Host, Port#	
Reply?					
Reply blck times?					
Msg Unit	word	Byte	Byte of Objects	Byte of Object	
Buf size?	1 word				

Reuse and blocking semantics

Currently: Either block for system buffer or use complex select

GIPC: Have the IPC layer use observer pattern to notify when buffer written by non blocking send is reusable

Send Observer and Observable

Send notification when byte buffer allocated for message completely sent

Expected to be used for buffer management, so is not generic

Tracing Send Listener

```
aClientInputPort.addSendListener(
        new ATracingSendListener(aClientInputPort));
```

```
public class ATracingSendListener
                implements ByteBufferSendListener {
  InputPort inputPort;
  public ATracingSendListener(InputPort inputPort) {
        inputPort = anInputPort;
  }
  public void messageSent(String aRemoteEnd,
                        ByteBuffer aMessage, int aSentId) {
        System.out.println(clientInputPort.getLocalName() +
           "-->"  + aRemoteEnd + ":(" + aSentId + ")" + aMessage);
  }
}
```

Listeners are port dependent

Explicit Connection?

```
SimplexServerInputPort<ByteBuffer> aServerInputPort =
      BufferSimplexInputPortSelector.createServerSimplexInputPort(
                  SERVER_PORT, SERVER_NAME);
```

```
SimplexClientInputPort<ByteBuffer> clientInputPort =
      BufferSimplexInputPortSelector.createClientSimplexInputPort(
                  SERVER_HOST, SERVER_PORT, SERVER_NAME, clientName);
```

Special Call: can add listeners before activity

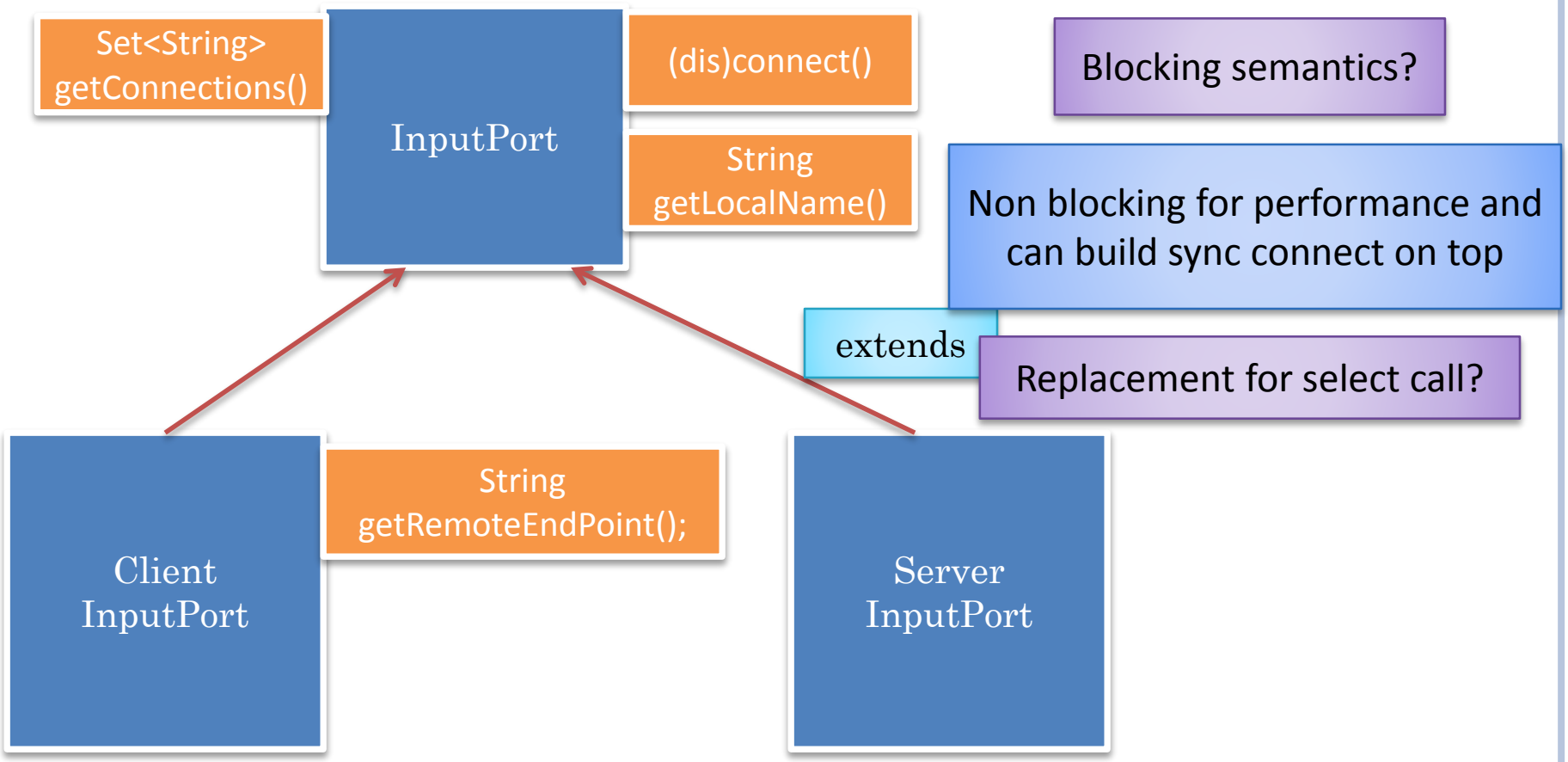
```
SimplexServerInputPort<ByteBuffer> aServerInputPort =
      BufferSimplexInputPortSelector.createServerSimplexInputPort(
            SERVER_PORT, SERVER_NAME);
aServerInputPort.addSendListener(
      new AnEchoingReceiveListener(aServerInputPort);

aServerInputPort.connect();
```

```
SimplexClientInputPort<ByteBuffer> clientInputPort =
      BufferSimplexInputPortSelector.createClientSimplexInputPort(
            SERVER_HOST, SERVER_PORT, SERVER_NAME, clientName);
aClientInputPort.addSendListener(
      new ATracingSendListener(aClientInputPort));
clientInputPort.connect();
```

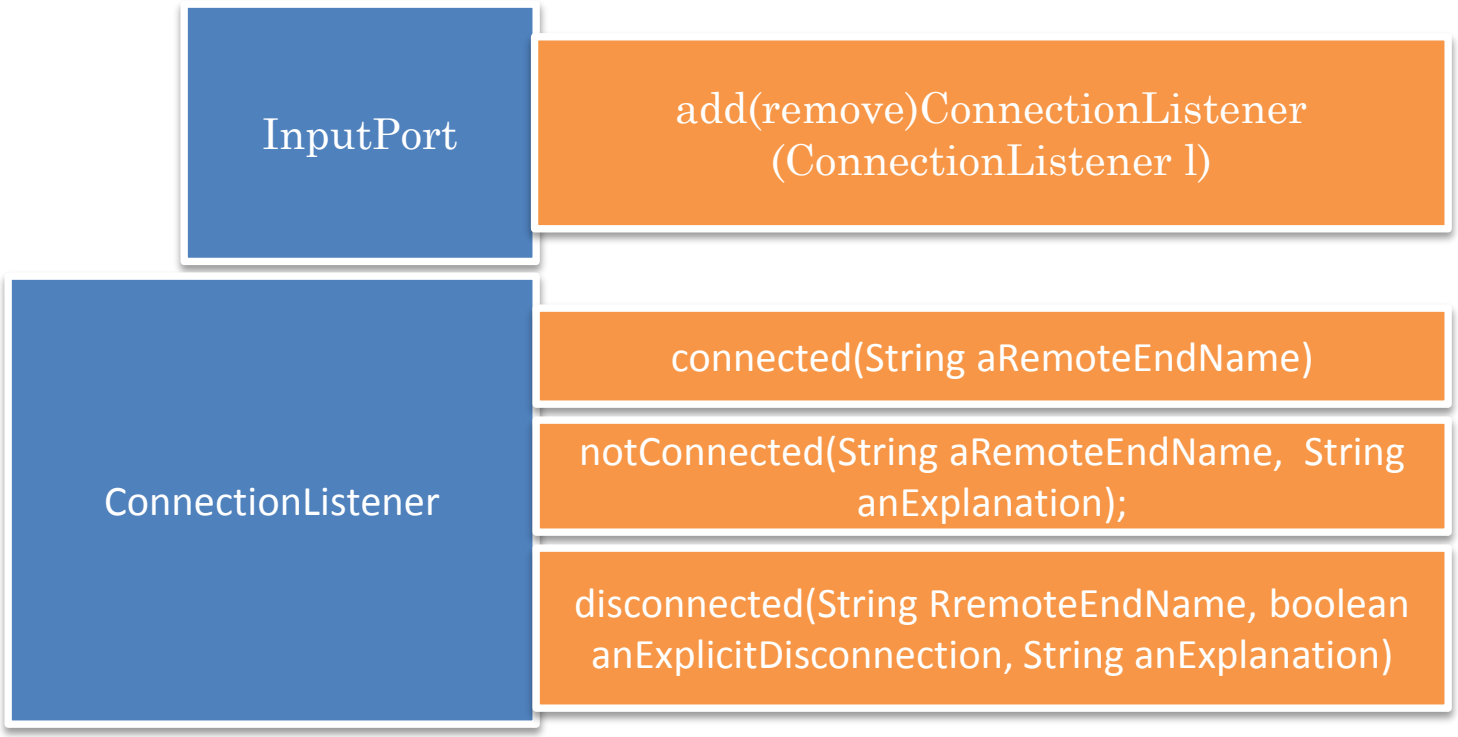
A la Thread start() and Frame setVisible()

Connection Interfaces



Both client and server have logical names used for linking (discussed earlier)

Connection Observer and Observable



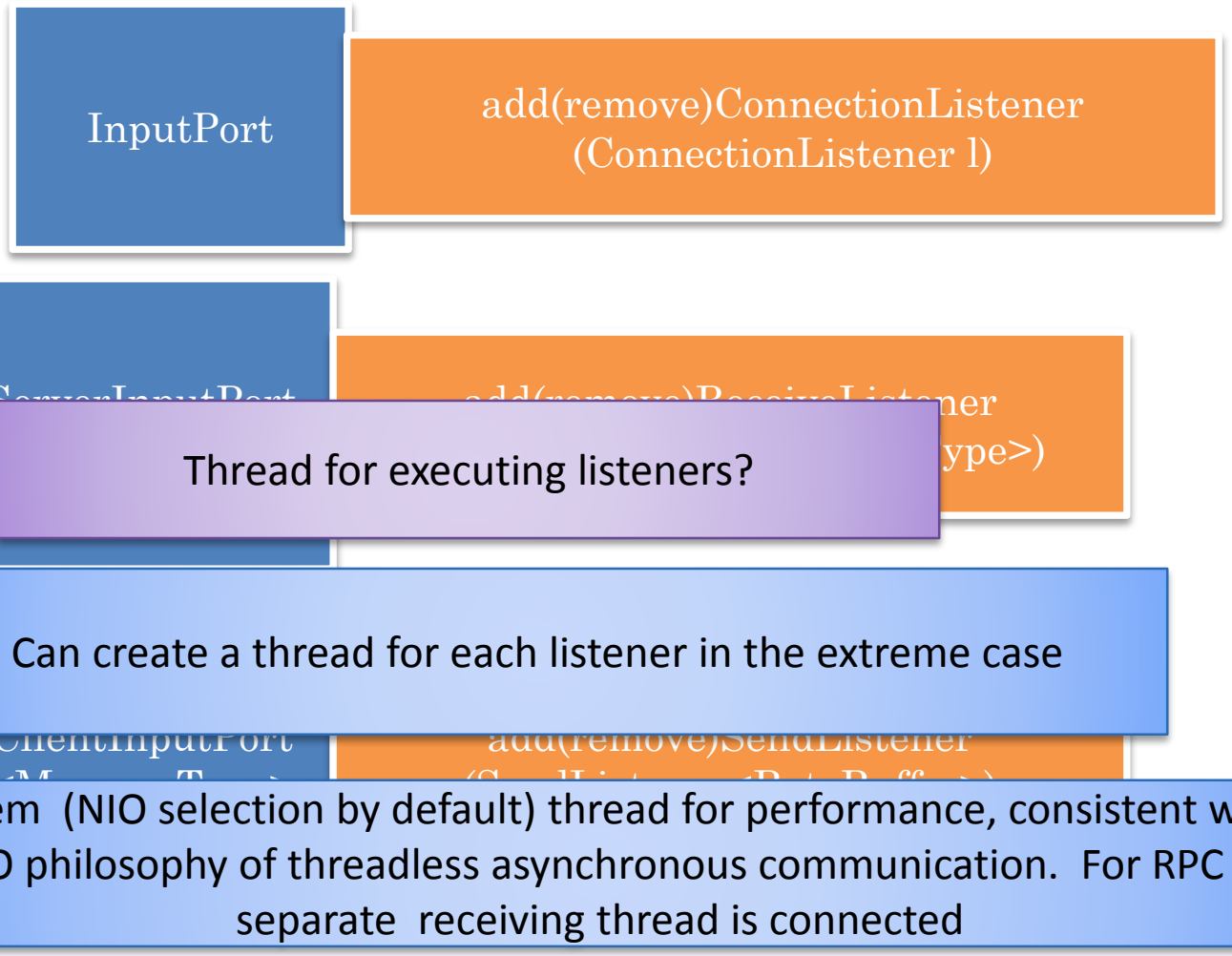
Same interface for server and client

Remote end name is logical name of other party, server connect called only after client connect message with its name has been received

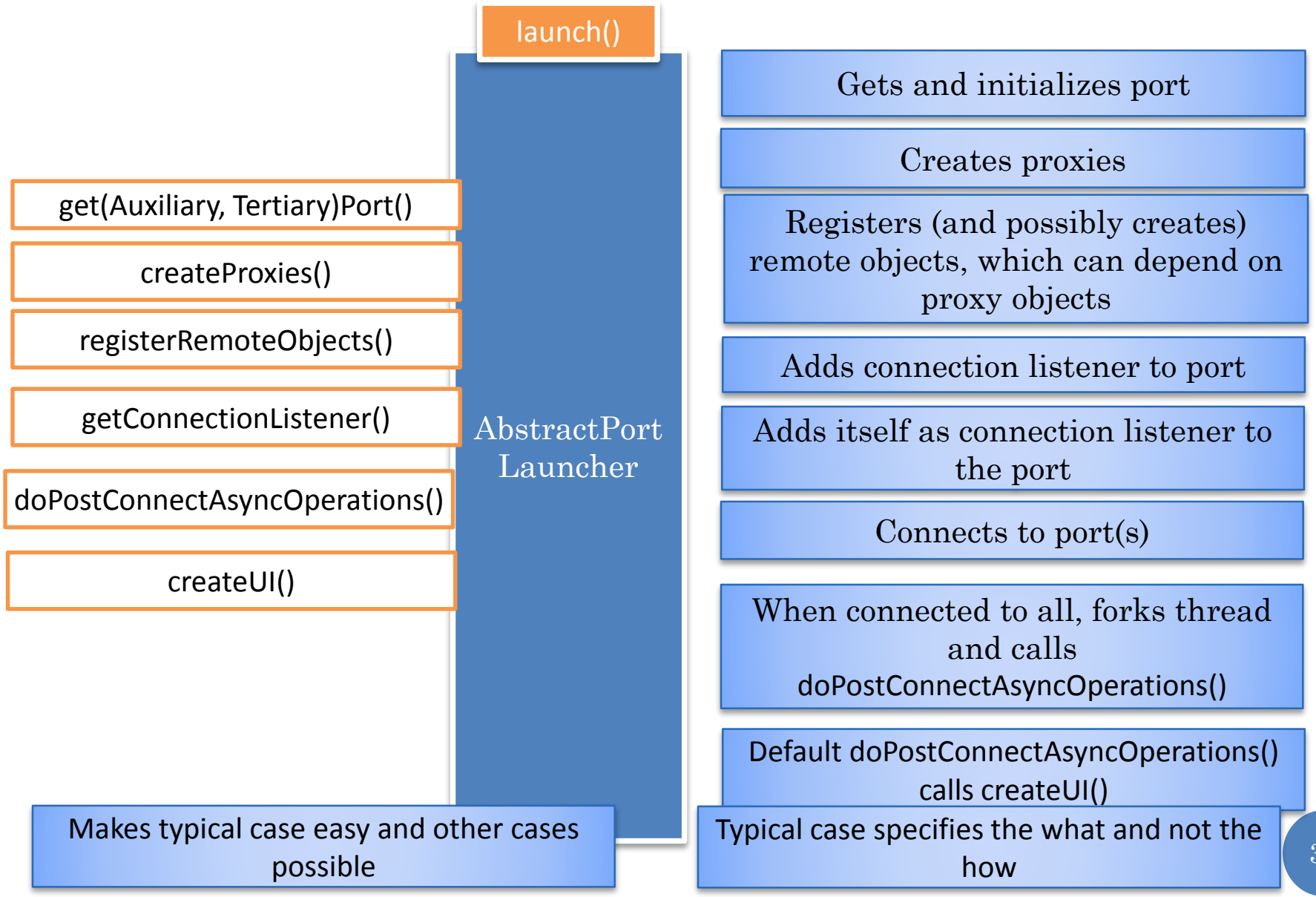
Explanation based on exception's getMessage

port may be disconnected explicitly or implicitly because of failure

Listeners Galore



Abstracting Away The Launch Details



Abstract Duplex RPC Port

```
public class AnAbstractDuplexRPCClientPortLauncher extends
                                                AnAbstractPortLauncher   {
public AnAbstractDuplexRPCClientPortLauncher(String aClientName,
       String aServerHost, String aServerId, String aServerName) {
  super(aClientName, aServerHost, aServerId, aServerName);
}
protected PortAccessKind getPortAccessKind() {
  return PortAccessKind.DUPLEX;
}
protected PortKind getPortKind() {
  return PortKind.CLIENT_INPUT_PORT;
}
protected PortMessageKind getPortMessageKind() {
  return PortMessageKind.RPC;
}
```

Specifying the kind of port, not the exact port classes

Client Launcher

```
public class ADuplexCounterClientLauncher extends
AnAbstractDuplexRPCClientPortLauncher    {
Counter counter11Proxy, counter12Proxy, counter2Proxy;
public ADuplexCounterClientLauncher(String aClientName, String aServerHost,
                                    String aServerId, String aServerName) {
  super(aClientName, aServerHost, aServerId, aServerName);
}
protected  void createProxies() {
    counter11Proxy = (Counter) createProxy(
        DuplexCounterServerLauncher.REGISTERED_COUNTER_CLASS,
        DuplexCounterServerLauncher.COUNTER1);
     ...
}
protected void doPostConnectsAsyncOperation
  System.out.println(counter11Proxy == coun
   …
}
public static void main (String[] args) {
    (new ADuplexCounterClientLauncher(CLIENT_NAME, "localhost",
       DuplexCounterServerLauncher.COUNTER_SERVER_ID,
       DuplexCounterServerLauncher.COUNTER_SERVER_NAME )).launch();
}
```

Specifying the what of proxies not which generator class to use

Abstract Launcher Code

```
protected void doPostConnectsAsyncOperations() {
   createUI(mainPort);
}

protected void createUI(InputPort anInputPort){

}
```

doPostConnectsAsyncOperations in abstract class simply calls createUI()

Alternative Application Port Launcher

```
public class ADuplexCounterClientLauncher extends
AnAbstractDuplexRPCClientPortLauncher    {
Counter counter11Proxy, counter12Proxy, counter2Proxy;
public ADuplexCounterClientLauncher(String aClientName, String aServerHost,
                                    String aServerId, String aServerName) {
  super(aClientName, aServerHost, aServerId, aServerName);
}
protected  void createProxies() {
    counter11Proxy = (Counter) createProxy(
        DuplexCounterServerLauncher.REGISTERED_COUNTER_CLASS,
        DuplexCounterServerLauncher.COUNTER1);
     ...
}
protected void createUI(InputPort anInputPort){
  System.out.println(counter11Proxy == counter12Proxy);
   …
}
public st
    (new
        Du
        Du                                                    );
}
```

doPostConnectsAsyncOperations in abstract class simply calls createUI()

Most examples are interactive and use createUI

Reply

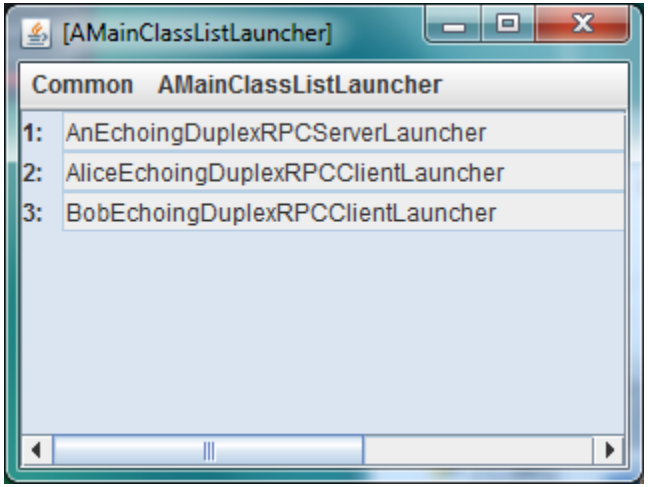
	Xinu	Pipes	Sockets	NIO	GIPC
Location	Intra-Proc	Intra-Host	Distributed	Distributed	Distributed
Reliable, In-Order?	Yes	Yes	Yes/No	Yes/No	
Access?	Input	Free	Bound./Inp	Bound./Inp	
Send block times					
Explicit receive?					
Rcv blck times?	Sy				
Select?					
Language?					
Naming?	pid	filedescp	Host, Port#	Host, Port#	
Reply?	No	No	No	No	Yes
Reply blck times?	N/A	N/A	N/A	N/A	
Msg Unit	word	Byte	Byte	Byte	
Buf size?	1 word				

Reply convenient when duplex input port

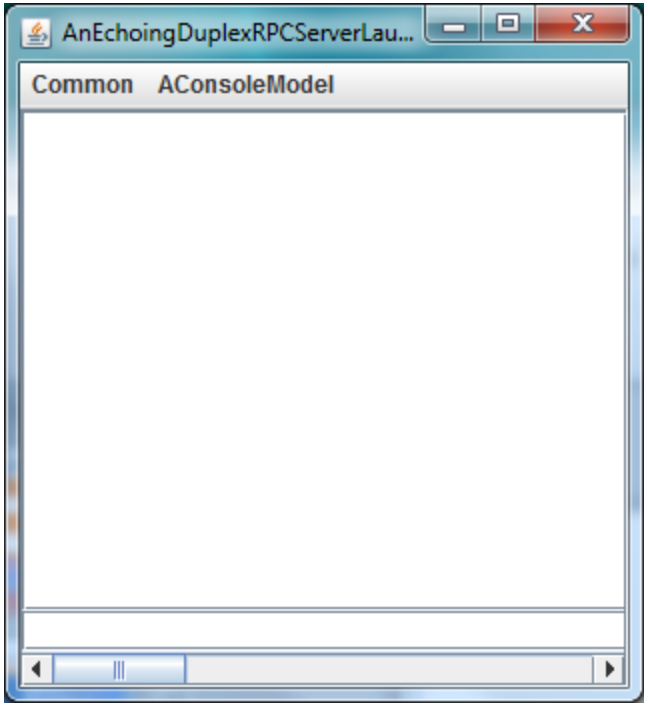
Destination implicit based on last sender

Does not make sense for simplex port

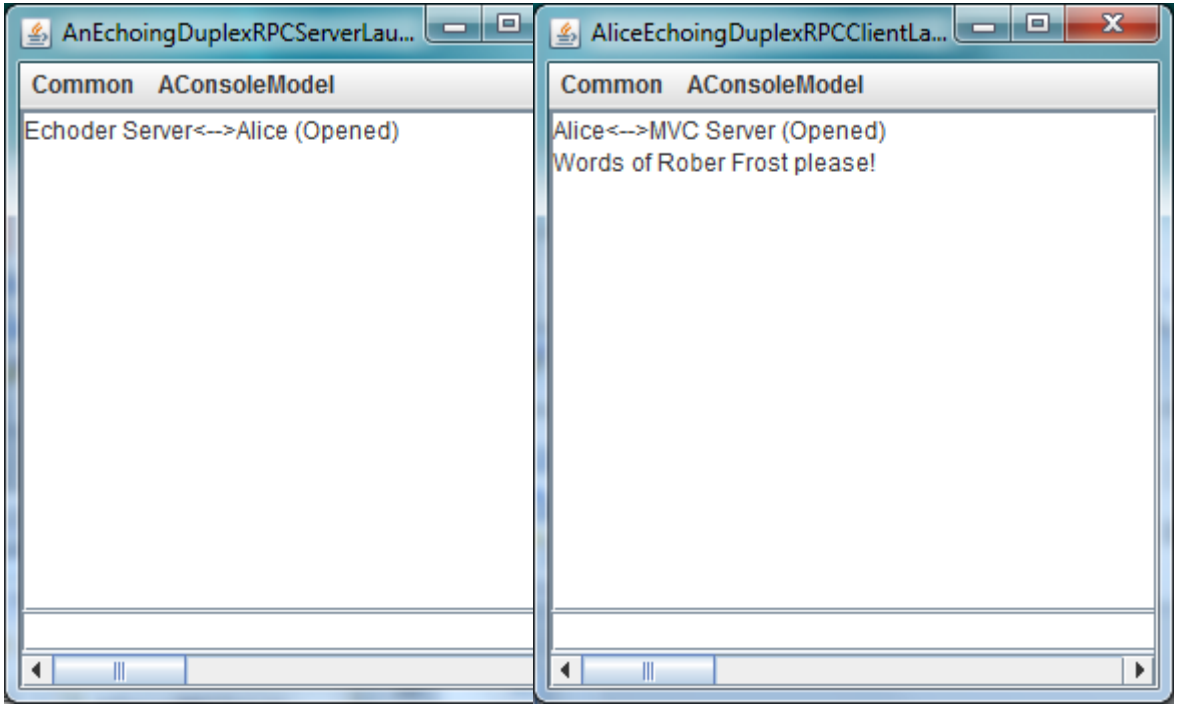
Full Example



Unconnected Echo Server

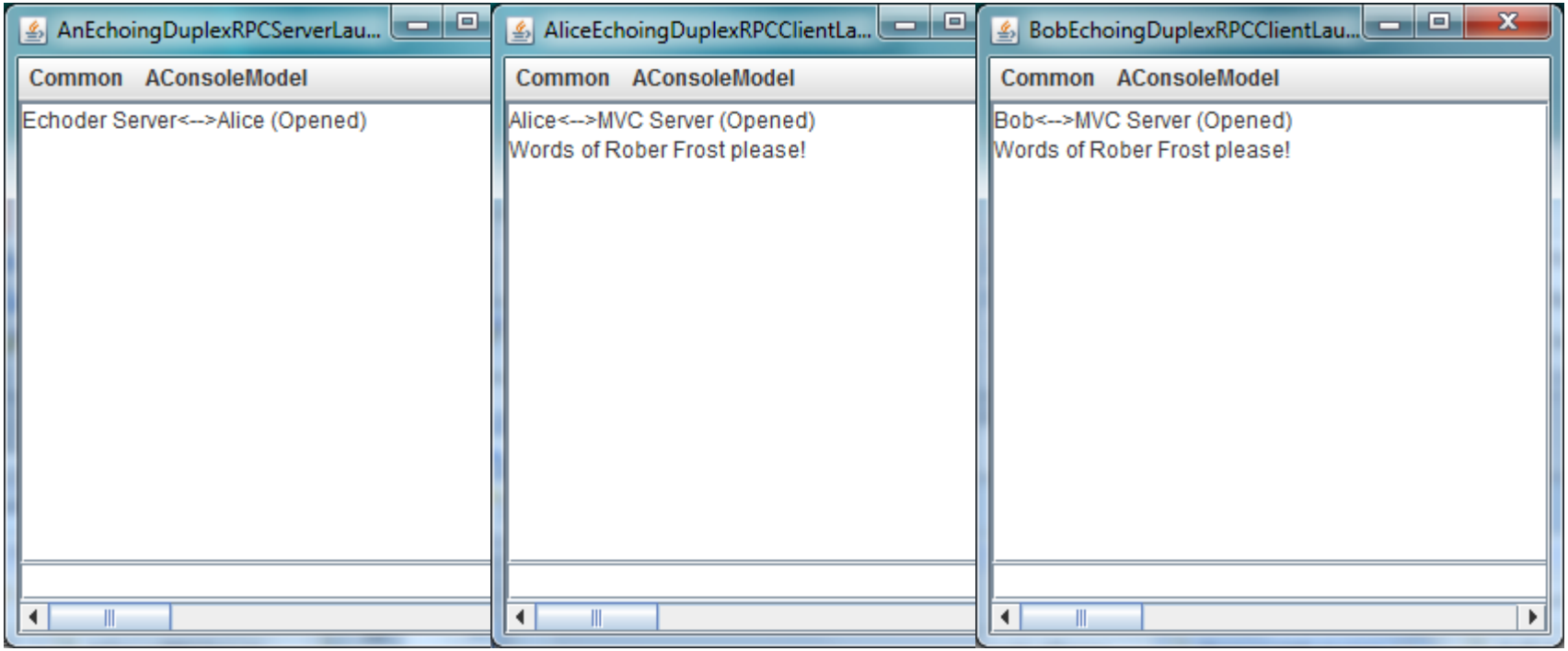


Singly-Connected Echo Server

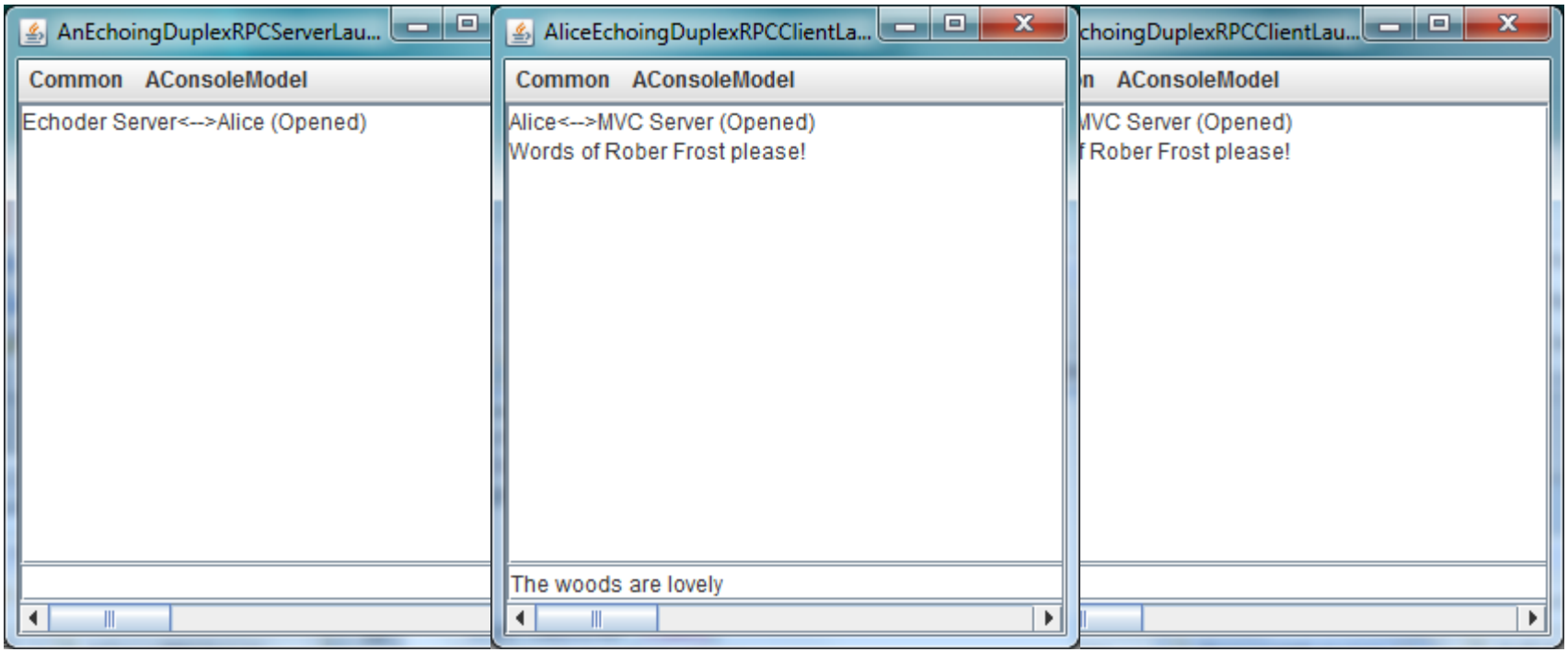


Name of the server is not actually used to make connection, so client and server have different views of what the server name is

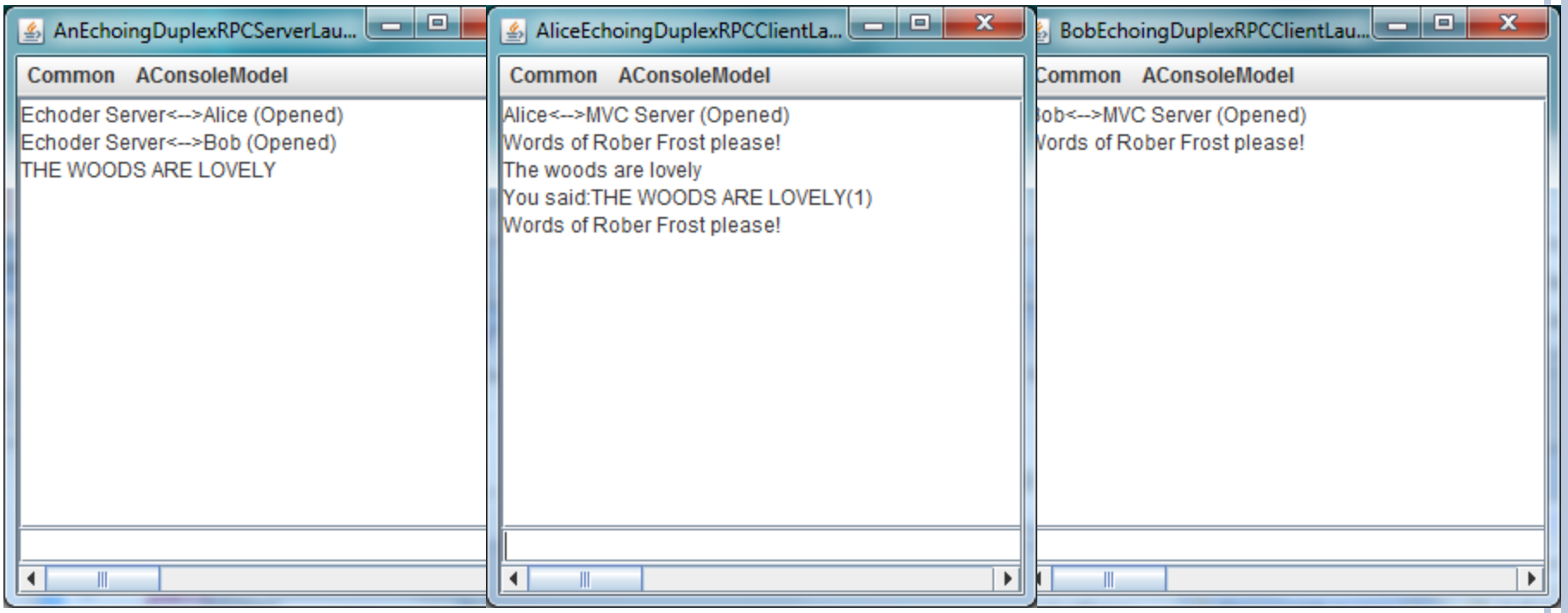
TWICE CONNECTED SERVER



Alice Enters Text

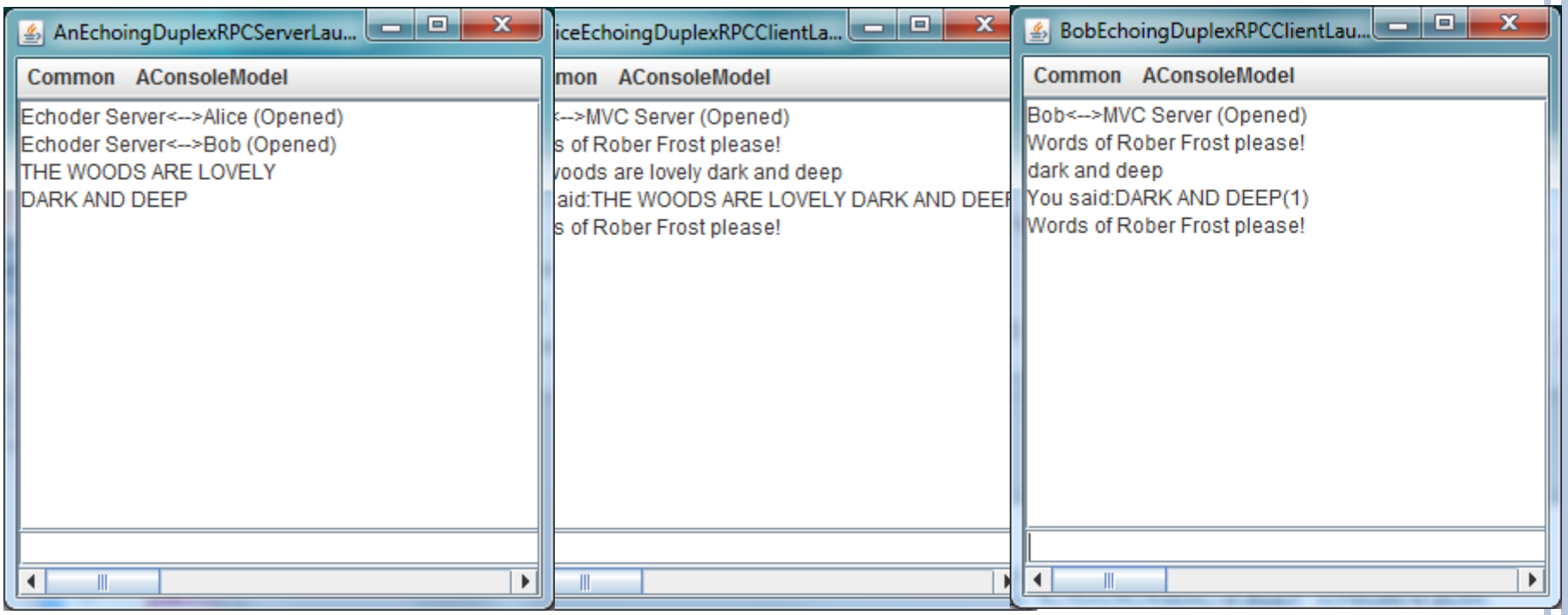


Uppercasing, Duplex Counted Echoing

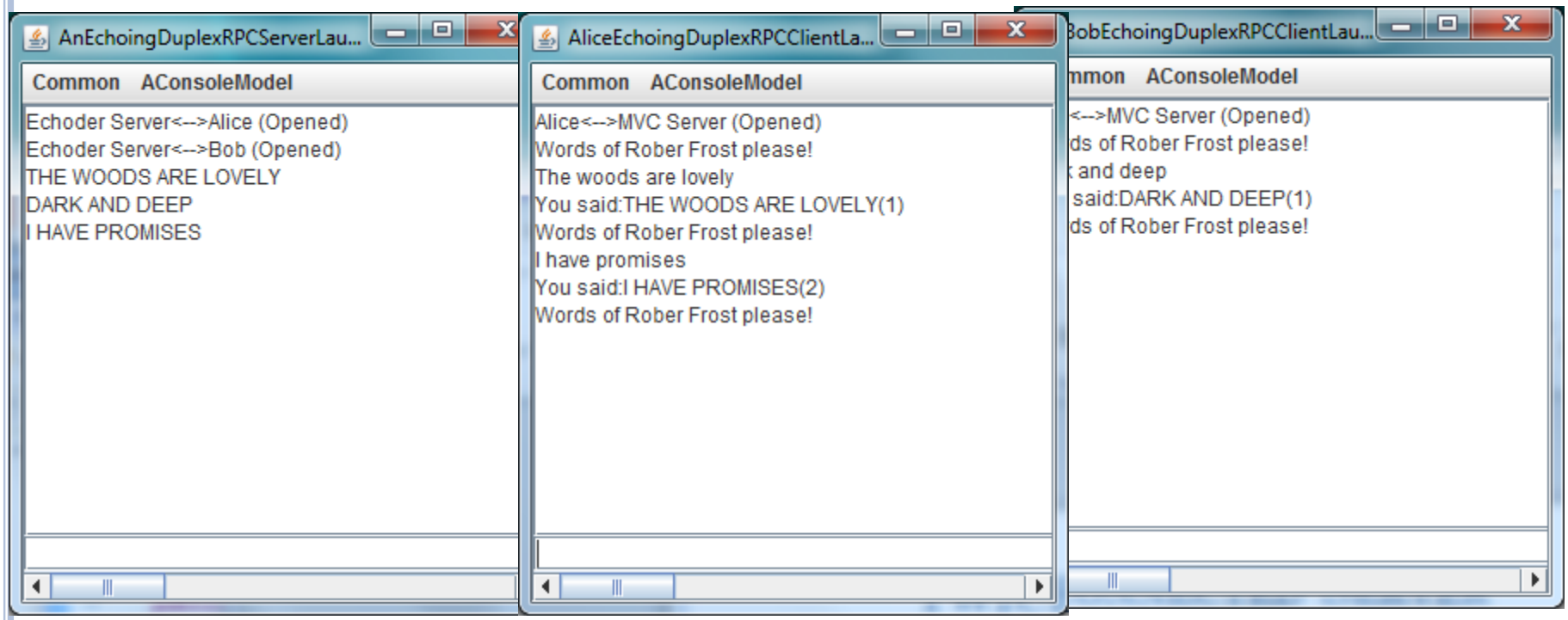


Client keeps the count

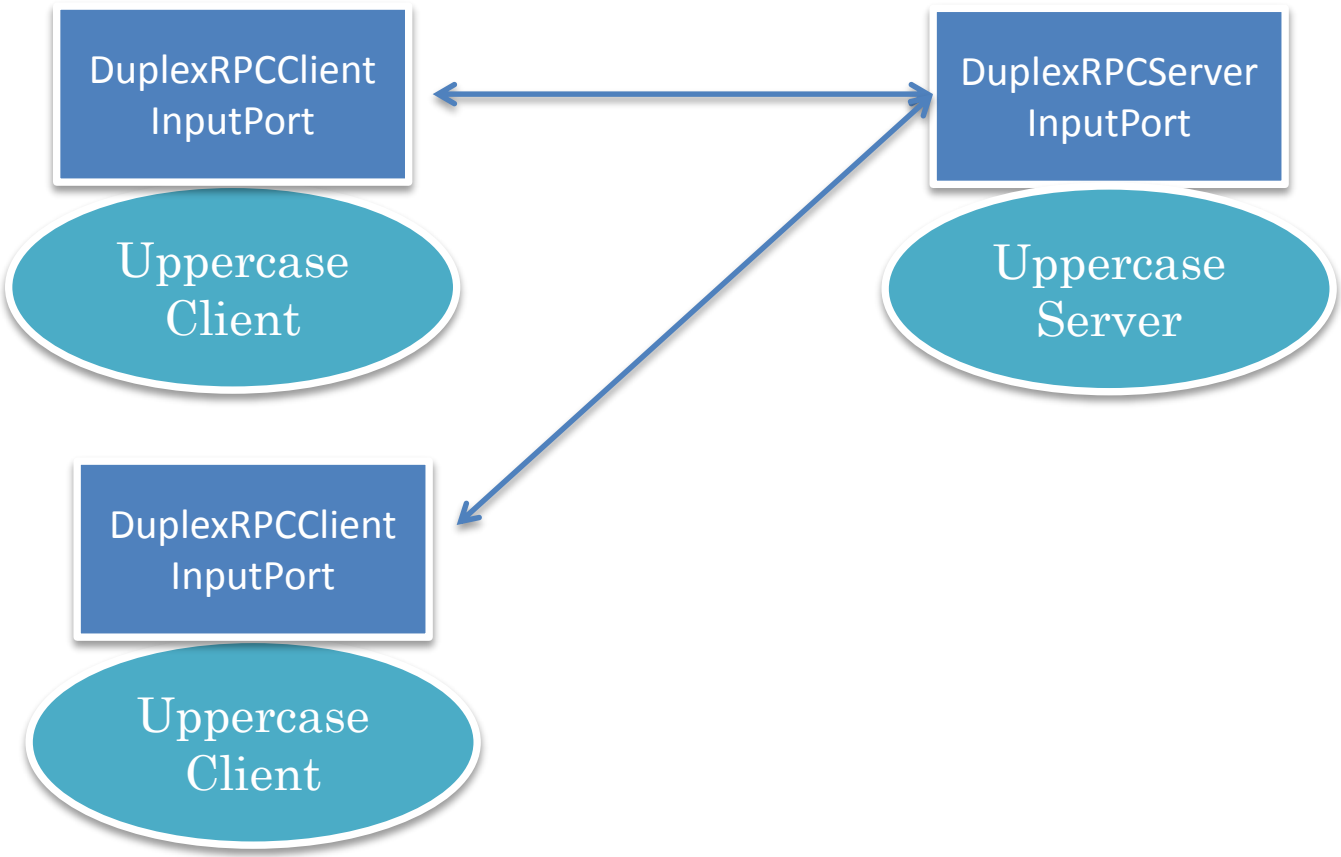
Bob Interaction



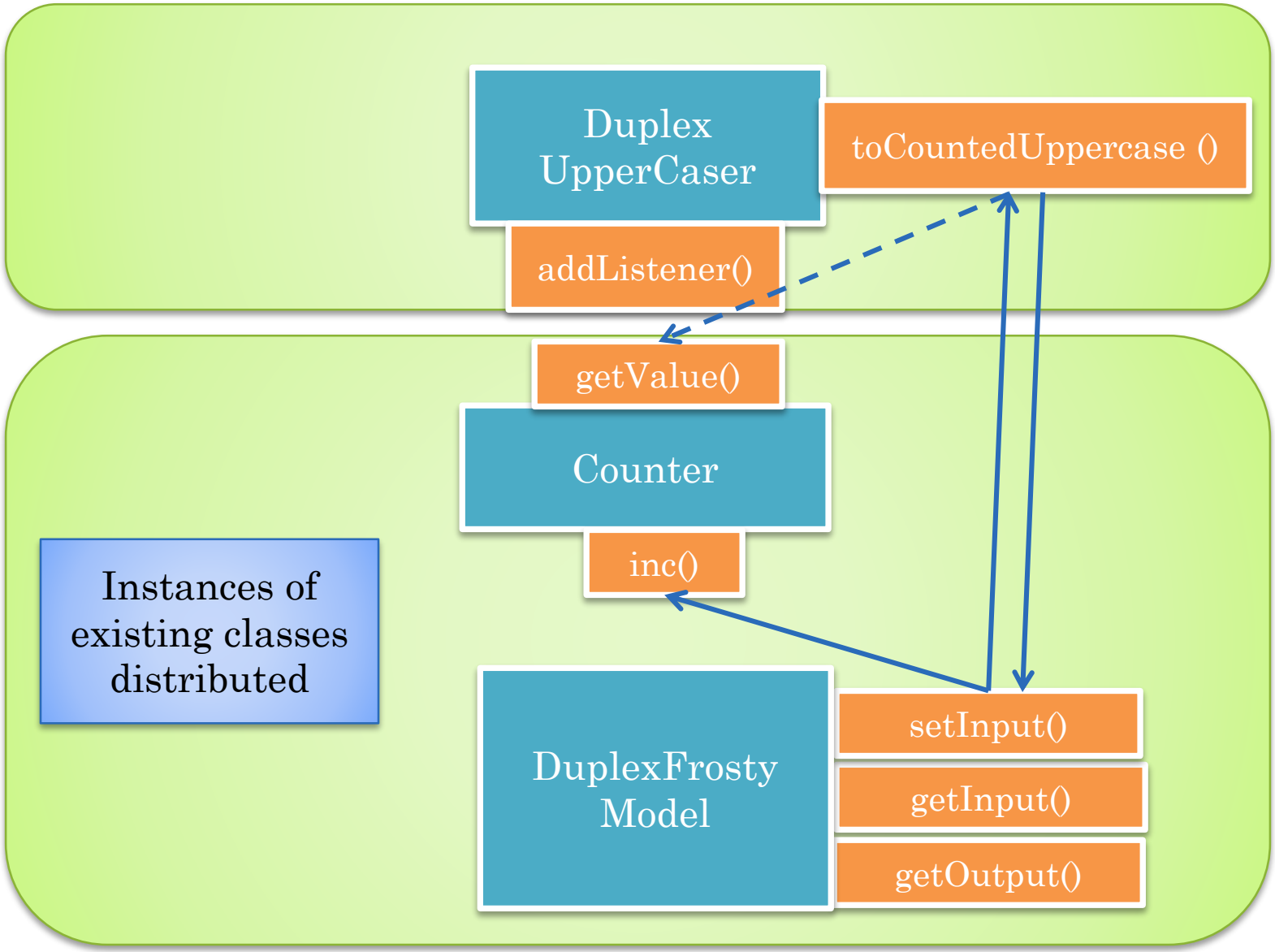
Second Alice Interaction



Port Structure



Distributed Objects



Server: Constructor/Connection/Register

```
public class AnEchoingDuplexRPCServerLauncher
    extends    AnAbstractDuplexRPCServerPortLauncher
    implements EchoingDuplexServerLauncher  {

public AnEchoingDuplexRPCServerLauncher(String aServerName,
   String aServerPort) {
   super (aServerName, aServerPort);
}
protected  ConnectionListener getConnectionListener
                  (InputPort anInputPort) {
   return new ATracingConnectionListener(anInputPort);
}
protected void registerRemoteObjects() {
  DuplexRPCServerInputPort aDuplexRPCServerInputPort =
        (DuplexRPCServerInputPort) mainPort;
  DuplexUpperCaser upperCaser = getUpperCaser();
  aDuplexRPCServerInputPort.register(upperCaser);
}
```

mainPort assigned by super constructor

Server: Callback Proxy Generation

```
protected Counter counter;
protected  void createProxies() {
    counter = (Counter) ReplyRPCProxyGenerator.generateReplyRPCProxy
    ((DuplexRPCServerInputPort) mainPort, (registeredCounterClass()));
}
protected Class registeredCounterClass() {
   return DuplexRPCClientMVCLauncher.REGISTERED_COUNTER_CLASS;
}
protected DuplexUpperCaser getUpperCaser() {
   return new ADuplexUpperCaser(counter);
}
public static void main (String[] args) {
  (new AnEchoingDuplexRPCServerLauncher(ECHOER_SERVER_NAME,
     ECHOER_SERVER_ID)).launch();
}
```

Different proxy generator used because it is callback

As in RMI a server can be passed a Remote (proxy) object from the client

Server: Launcher

```
public static void main (String[] args) {
  (new AnEchoingDuplexRPCServerLauncher(ECHOER_SERVER_NAME,
     ECHOER_SERVER_ID)).launch();
}
```

Client: Constructor/Connection/Proxies

```
public class AnEchoingDuplexRPCClientLauncher
    extends    AnAbstractDuplexRPCClientPortLauncher
    implements EchoingRPCClientLauncher  {
public AnEchoingDuplexRPCClientLauncher(String aClientName,
     String aServerHost, String aServerId, String aServerName) {
  super(aClientName, aServerHost, aServerId, aServerName);
}
protected  ConnectionListener getConnectionListener
           (InputPort anInputPort) {
  return new ATracingConnectionListener(anInputPort);
}
protected Class registeredUpperCaserClass() {
  return ADuplexRPCServerMVCLauncher.REGISTERED_DUPLEX_UPPER_CASER_CLASS;
}
protected SimplexUpperCaser upperCaseProxy;
protected  void createProxies() {
    upperCaseProxy = (SimplexUpperCaser)   DirectedRPCProxyGenerator.
       generateRPCProxy((SimplexRPCClientInputPort) mainPort,
                        registeredUpperCaserClass());
}
```

Client: Register Objects

```
protected Counter counter;
protected void registerRemoteObjects() {
  DuplexRPCClientInputPort aDuplexRPCClientInputPort =
     (DuplexRPCClientInputPort) mainPort;
  counter = new ACounter();
  aDuplexRPCClientInputPort.register(counter);
}
```

Client: Build UI

```
protected void createUI(InputPort anInputPort) {
  createMVC(anInputPort);
  processConsoleInput();
}
protected FrostyConsoleInteractor frostyConsoleInteractor;
protected void createMVC(InputPort anInputPort) {
  DuplexFrostyModel clientModel = getFrostyModel();
  frostyConsoleInteractor =  new ADuplexFrostyConsoleUI();
  frostyConsoleInteractor.interact(clientModel);
}
protected DuplexFrostyModel getFrostyModel() {
  return new  ADuplexFrostyModel((DuplexUpperCaser)upperCaseProxy,
counter);
}
protected void processConsoleInput() {
  frostyConsoleInteractor.processConsoleInput();
}
}
```

Building the MVS structure using local object and proxy

Alice Launcher

```
public class AliceEchoingDuplexRPCClientLauncher  {
public static final String  USER_NAME = "Alice";
public static void main (String[] args) {
   (new AnEchoingDuplexRPCClientLauncher(USER_NAME, "localhost",
    SimplexRPCServerMVCLauncher.MVC_SERVER_ID,
    SimplexRPCServerMVCLauncher.MVC_SERVER_NAME )).launch();
}}
```

Multi-Class Launcher

```
package inputport.rpc.duplex.echoer.example;
import bus.uigen.models.MainClassLaunchingUtility;
public class DemoerOfEchoingDuplexRPCInputPort {
  public static void main(String args[]) {
    demo();
  }
  public static void demo() {
    Class[] classes = {
      AnEchoingDuplexRPCServerLauncher.class,
      AliceEchoingDuplexRPCClientLauncher.class,
      BobEchoingDuplexRPCClientLauncher.class
    };
    MainClassLaunchingUtility.interactiveLaunch(classes);
  }
}
```

Run this class and double click on each entry in sequence

Goals

Customizable Implementation

Unlike RMI runs on Android

Informed by the IPC Design Space

How can we “improve” distributed IPC without language support?