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**

- **Activation Record**
  - Parameters, return value, return address, and other info

- **(Un)marshalling of call**
  - Translation between call and message

- **Call results in marshalling and unmarshalling of call/return value**
  - Rather than push and pop of activation record on stack

- **Activation Record**
  - Parameters, return value, return address, and other info

- **Calling Process**
  - Call message (operation id, parameters, source)

- **Called process**
  - Return message (return value)
**DATA SET UP**

1. Server exports external description of data (RA) port
2. Statically or dynamically
3. Server starts and makes data port connectable
4. Client determines external description
5. Statically or dynamically
6. Client starts and establishes connection
## RPC Set Up

<table>
<thead>
<tr>
<th>Server exports external description of calls serviced on RA (data) port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statically or dynamically</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client finds this information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statically or dynamically</td>
</tr>
</tbody>
</table>
PROCESSING OF CALL MESSAGE AND RETURN VALUE

1. Generate call message for call using exported information
2. Process call message in caller process
3. Communicate call message on data port and handle errors
4. Process call message in callee process
5. Invoke called operation
PROCESSING OF CALL MESSAGE AND RETURN VALUE

1. Possibly create return message in callee process
2. Communicate return message to caller process
3. Process return message in callee process and handle errors
4. Return value to caller operation
AUTOMATION OF REMOTE PROCEDURE CALL

- Process/Thread
  - <method>
  - <params>
- Port (Mailbox)
  - <method>
  - <params>
- Process/Thread
  - <method>
  - <params>

App @ Calling Site
RPC System @ Calling Site

Division of labor?

App @ Callee 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
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

Transparent Remote Procedure Call

Opaque Lower Level Layers

Lower level layers API is hidden by the RMI Layer

Though it allows socket to be passed to RMI layer
GIPC Layer Visibility

- Transparent Remote Procedure Call
- Aware Remote Procedure Call
- Typed Data Communication
- Byte Communication

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

Designed for customization (for teaching)
<table>
<thead>
<tr>
<th><strong>GIPC Layer Visibility</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow remote assignment and remote procedure call on same port</td>
</tr>
<tr>
<td>Notion of connected client and server port visible</td>
</tr>
</tbody>
</table>
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

- Transparent Remote Procedure Call
- Aware Remote Procedure Call
- Typed Data Communication
- Byte Communication
- Java NIO (Non Blocking IO)
Replaceable Lowest Layer

- Transparent Remote Procedure Call
- Aware Remote Procedure Call
- Typed Data Communication
- Byte Communication
- Java Sockets
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

```java
public class ARelayingCollaborativeRMIEchoerUpperCaser extends ACollaborativeRMIEchoerUpperCaser implements RelayingCollaborativeRMIEchoerUpperCaser {
    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);
    }
}
```
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 Byte Buffer 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:

```java
(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

- **Caller**
- **RMI Registry**
- **Callee**

Registry idea inconsistent with (peer to peer) groups
**GIPC Required Architecture**

- **Caller**
- **Callee**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Proxy can be created before callee object</td>
<td></td>
</tr>
<tr>
<td>Unlike Java RMI, target object name may be invalid, as can the arguments</td>
<td></td>
</tr>
<tr>
<td>Supports (peer to peer) sessions with late binding of members</td>
<td></td>
</tr>
<tr>
<td>Server is optional and can be built using basic primitives</td>
<td></td>
</tr>
<tr>
<td>Consistent with teaching goal</td>
<td></td>
</tr>
</tbody>
</table>
JAVA RMI: Serialization → Marshalling

ObjectOutput Stream

extends

RMI Marshaller
GIPC: Serialization → Marshalling

Serializer → ObjectInputStream

HAS-A

RMI Marshaller
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

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

Client Object

Server Proxy

m(p₁, ..., pᴺ)

Server Object

m(p₁, ..., pᴺ)
PORT-BASED RMI ARCHITECTURE

Client Object

Server Proxy

m(p₁, … pᴺ)

Client Port

Server Port

Server Object

m(p₁, … pᴺ)

???
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

Caller -> Callee
LOCAL REGISTRATION PORT CALLS: CALLEE PORT INTERFACE

RPCRegistry

void register(String aName, Object aServerObject)

void register (Class aType, Object aRemoteObject)

register(Object aServerObject)

Object getServer(String aName)

Can define alternative to Remote

Well known mapping from type to object Name

Well known mapping from object’s class to object Name
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);
    }
}
INTERFACE EXTENSIONS

Simplex Client InputPort <Object>

SimplexRPC ClientInputPort

SimplexRPC ServertInputPort

RPCRegistry

Simplex Server InputPort <Object>

DuplexRPC ClientInputPort

DuplexRPC ServerInputPort
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 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

Calling Process

Call message (operation id, parameters, source)
Non RPC Message
Return message (return value)
Non RPC Message

Called process
METHOD RETURN VALUE

RPCReturnValue Object returnValue
CALL MESSAGE

Call <ObjectIDType>

- transient Method method
- ObjectIDType targetObject
- Object[] args

extends <String>

SerializableCall

- SerializedMethod serializedMethod

SerializeMethod

- transient Method method
- String methodName
- String className
- String[] parameterTypeNames
SERIALIZING A METHOD

Call<ObjectIdType> marshallCall(
    ObjectIdType aTargetObject, Method aMethod, Object[] anArgsList)

transient Method serializableMethod

ObjectIdType targetObject

Object[] args

Marshaller<ObjectIdType>
**PORT vs. GIPC-RMI**

- **Client Object**
- **Server Proxy**
- **m(p₁, ... pᴺ)**
- **Server Port**
- **Client Port**
- **Server Object**
- **m(p₁, ... pᴺ)**

**Synchronization?**
PROCESSING OF CALL MESSAGE AND RETURN VALUE

1. Generate call message for call using exported information
2. Process call message in caller process
3. Communicate call message on data port and handle errors
4. Process call message in callee process
5. Invoke called operation
PROCESSING OF CALL MESSAGE AND RETURN VALUE

- Possibly create return message in callee process
- Communicate return message to caller process
- Process return message in callee process and handle errors
- Return value to caller operation
OBJECT FORWARDERS AND SERIALIZERS

Sender <Object> (Object Port)

Send Forwarder <Object, Object>

Serializer <Object, ByteBuffer>

Sender<ByteBuffer> (Buffer Skeleton)

Receive Notifier<Object>

ReceiveForwarder <Object, Object>

Deserializer <ByteBuffer, Object>

Receive Listener <ByteBuffer> (Buffer Skeleton)
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)

Sender <Object> (RPC Port)

Call Message and Return Message SendTrapper <Object, Object>

Sender <Object> (Object Port)

Receive Notifier <Object>

Call Message and Return Message ReceiveTrapper <Object, Object>

Receive Listener <Object> (RPC Port)

Coordinating Shared Data Structures

Send trapper may need to block sender to implement synchronous operation
**OBJECT FORWARDERS AND SERIALIZERS (REVIEW)**

- **Sender <Object> (RPC Port)**
  - Call Message and Return Message SendTrapper <Object, Object>
  - **Receive Listener <Object> (RPC Port)**
  - SendTrapper and ReceiveTrapper at Caller Share an Instance of Synchronizing SentCallCompleter

- **Receive Notifier <Object>**
  - Call Message and Return Message ReceiveTrapper <Object, Object>
  - Receive Listener <Object> (RPC Port)

**Coordination between Caller and Called Site?**
**Duplex Sent Call Completer (Public and Non Public Methods)**

**public Object**

$returnValueOfRemoteMethodCall(String aRemoteEndPoint, Object aCall);

**Object**

$returnValueOfRemoteFunctionCall(String aRemoteEndPoint, Object aCall);

$returnValueOfRemoteProcedureCall(String aRemoteEndPoint, Object aCall);

**processReturnValue(String aSource, Object aMessage);**

$returnValueOfRemoteMethodCall() called, after call message sent: receive return value and block caller if necessary

- Blocks for return value
- Non blocking, simply returns
- Unblocks
Function Call Synchronization and Matched Client/Server Objects

- **Call**
  - ReceiveTrapper
    - receive()
  - SendTrapper
    - send()
  - RPCReturnValue Queue (BoundedBuffer)
    - put()
    - get()
  - ReceiveTrapper
    - receive()
  - process Return Value()

- **Return Val**
  - Call
  - Return Val
  - ReceiveTrapper
    - receive()
  - SendTrapper
    - send()
  - RPCReturnValue Queue (BoundedBuffer)
    - put()
    - get()

- **handle Function Return()**
- **message Received()**

- **Remote function call is blocking**
- **RemoteObject**
  - T m(...)

- **marshall Call**
  - Proxy
  - send()

- **SentCall Completer**
  - returnValueOfRemoteFunctionCall()
**Procedure Call Synchronization and Matched Client/Server Objects**

1. **Received Call Invoker**
   - `Received()`
   - `Received Received()`
   - `handle Procedure Return()`

2. **Remote procedure call is not blocking**
   - `Tm(...)`

3. **RemoteObject**
   - `RemoteObject`

4. **Proxy**
   - `Proxy`
   - `marshall Call`
   - `Tm(...)`

5. **Marshaller**
   - `Marshaller`

6. **SendTrapper**
   - `send()`
   - `SendTrapper`

7. **SentCall Completer**
   - `returnValueOfRemoteProcedureCall()`
   - `SentCall Completer`
SYNCHRONOUS CALLBACK IN REMOTE CALL

Remote function call is blocking

RPCMessage Receiver

messageReceived()

m¹(…)

RemoteObject

RPCReturnValue Queue (BoundedBuffer)

put()

get()

Proxy

T m²(…)

Call

Return Val

NIO Selection Thread

Selector

select()
**GIPC Asynchronous Invoker (Factory Selected)**

- **RPC Message Receiver**
  - `messageReceived()`

- **Selector**
  - `select()`

- **NIO Selection Thread**
  - **Call**
  - **Return Val**

- **RPC ReturnValue Queue (BoundedBuffer)**
  - `put()`
  - `get()`

- **Remote Object**
  - `m^1(...)`

- **Remote function call is blocking**

- **GIPC Asynchronous Method Invoker**
  - Factory determines if asynchronous method invoker is created

- **Id in Call and Return Value?**

- **How many return value queues and queue size?**
GLOBAL RECEIVED VALUE INSTEAD OF BUFFER

Object `returnValueOfRemoteFunctionCall(String aSource, Call aCall)`

- Received Return Value
- `wait()`
- Clear and return received value

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

Object `processReturnValue(String aSource, Object aMessage)`

- Store return value
- `wait()`
**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 \(\leftarrow\) 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

- **Function `processReturnValue(String aSource, Object aMessage)`**
  - Concurrent calls can be made to different callees
  - Local bounded buffer \(\leftarrow\) lookup(aSource)
  - Put return value in local bounded buffer
METHOD RETURN VALUE

RPCReturnValue Object returnValue

No Id sent to match with call
Proxy?

Client Object

Server Proxy

m(p₁, ..., pᴺ)

Server Object

m(p₁, ..., pᴺ)
**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

Proxy should have no direct awareness of marshaller or sent call completer

SentCall Completer

returnValueofRemoteProcedureCall

SendTrapper

send()

T m(…)

marshall Call

Marshaller

Proxy
Minimum Proxy State

Stub object keeps forwarding information (remote server and remote object)
Port vs. GIPC-RMI

How much work should the proxy do?

Interface between proxy and data port assuming minimal code?

Separation of concerns – do only forwarding.
NAMING RPC PORT

Simplex Client InputPort
<Object>

NamingRPC

RPCRegistry

Simplex Server InputPort
<Object>

SimplexRPC ClientInputPort

SimplexRPC ServerInputPort

DuplexRPC ClientInputPort

DuplexRPC ServerInputPort
VARIABLY AWARE PORT CALLS

Object call(String aRemoteEnd, String anObjectName, Method aMethod, Object[] args)

Object call(String aRemoteEnd, Class aType, Method aMethod, Object[] args)

Object call(String aRemoteEnd, Method aMethod, Object[] args)

Transparency in Application → Awareness in System

Well known mapping from interface to object Name

Well known mapping from method’s declaring class to object Name
AWARE/TRANSIENT LAYERS

- Transparent Proxy-based Call
- Aware Procedure Call
- Object (Copy) Communication
- Byte Communication
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;
        }
    }
}
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?

Client Object

Server Proxy

m(p₁, … pᴺ)

Client Port

Server Port

Server Object

m(p₁, … pᴺ)

Given by system to programmer

Created by programmer

How?
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

```java
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

**UnicastRemoteObject**

```java
static exportObject(Remote object, int port)
```

**Proxy** (Counter_Stub instance)

```java
m(...)```

**Remote Object**

ADistributedInheritingCounter

```java
m(...)
```

**Proxy Class** (Counter_Stub)

```java
m(...)```

- Creates a proxy object for the remote object at the server end that can later be sent to the 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**
- Pre(compiler) generates proxy class
- Efficient

**Interpretation**
- At run time, a proxy class and object is created
- No compile-run-cycle delay
- No inconsistency as code evolves
**Runtime Abstraction?**

- **RMI system must generate application-specific proxy classes**
- **Runtime abstraction for generation?**
- **Interpretation**
  - At run time, a proxy class and object is created
Higher-Level Abstraction for Proxy

```java
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();
    }
}
```

Reflection and dynamic class generation and loading?

Higher-level abstraction for proxy generation and loading?
**Higher-Level Abstraction for Proxy**

```java
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();
    }
}
```
public int getValue() {
    try {
        Method method = Counter.class.getMethod("getValue");
        Object[] 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};
        ...
    } catch (Exception e) {
        e.printStackTrace();
    }
}
**Proxy and Invocation Handler**

---

**Proxy**

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

**Invocation Handler**

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

---

**Why class loader?**

Stores (unnamed) class object for this and later references

Internal name derived from interfaces so duplicate classes not created
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)`

- **Invocation Handler**
  - `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

- **Why class loader?**
- Stores (unnamed) class object for this and later references

- Internal name derived from interfaces so duplicate classes not created
Counter counter = (Counter)
    Proxy.newProxyInstance(
        Counter.class.getClassLoader(),
        new Class[] {Counter.class},
        invocationHandler );
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.printStackTrace();
        return null;
    }
}
protected Object call(String aDestination, String aName, Method aMethod, Object[] args) {
    Object retVal = rpcInputPort.call(aDestination, aName, aMethod, args);
    return retVal;
}
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 Registry

Caller

Callee

Caller

Callee
RMI Binding

Remote lookup(String name)

Registry

rebind(String name, Remote obj)

exportObject(Object, PortNumber)

UnicastRemoteObject

How to create proxy at caller directly?

Caller

Proxy
m(...)

Has-a

Port Ref.
Obj ID

Callee

Remote Object
m(...)

Has-a

Obj ID

Proxy
m(...)

Has-a

Port Ref.
Obj ID

RMI Registry

Port Ref.
Obj ID
**DIRECT SERVER PROXY**

- **SimplexRPC**
  - Has-a **Callee ID**
  - String `getRemoteEndPoint();`

- **SimplexRPServer**
  - InputPort
  - `register(String aName, Object aRemoteObject)`

- **Directed RPC Proxy**
  - Has-a **Port Ref.**
  - RemoteEnd
  - **Obj ID**

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

- **DirectedRPCProxy Generator**

- **Callee**
  - Remote Object
  - **Obj ID**
  - `m(...)`

- **Directed RPC Call**
  - Has-a **Obj ID**
**Directed Client Proxy**

- **DuplexRPCClient InputPort**
  - `register(String aName, Object aRemoteObject)`

- **DuplexRPServer InputPort**
  - `Client^1 ID`
  - `Client^n ID`

- **Remote Object**
  - `m(...)`

- **Obj ID**

- **DirectedRPCProxy Generator**

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

- **Port Ref**
  - `Remote End`
  - `Obj ID`
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;
        }
    }
}
**Reply Proxy**

DuplexRPCClient
- InputPort
- register(String aName, Object aRemoteObject)

Remote Object
- m(…)

Callee

Obj ID

ReplyRPCProxy
- Generator

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

DuplexRPServer
- InputPort

Client¹ ID
- Has-a
- Caller

Clientⁿ ID

Has-a Reply Proxy
- Port Ref
- Obj ID

Has-a Caller
- m(…)

Client¹ ID

Clientⁿ ID
CLIENT DIRECTED SERVER GENERATOR

Client InputPort

Set<String> getConnections()

(dis)connect()

String getLocalName()

extends

Server InputPort

String getRemoteEndPoint();

DirectedRPCProxy Generator

Object generateRPCProxy(SimplexRPC aPort, Class aType)

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

DirectedRPCProxy Generator

OutputPort

Client InputPort

String getRemoteEndPoint();

Call destination

extends

Server InputPort

String getLocalName()
**Server Reply Undirected Server Generator**

- **InputPort**
  - Set<String> getConnections()
  - (dis)connect()
  - String getLocalName()

- **Client InputPort**
  - String getRemoteEndPoint();

- **Server InputPort**

- **ReplyRPCProxyGenerator**
  - Object generateRPCProxy(DuplexRPC aPort, Class aType)
  - Object generateRPCProxy(DuplexRPC aPort, Class aType, String aName)

*Last caller is destination*
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[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;
}
**Others and All Proxy**

- **Callee**
  - DuplexRPCClient
  - InputPort
  - register(String aName, Object aRemoteObject)
  - Remote Object
  - Obj ID
  - m(…)

- **Caller**
  - GroupRPServer
  - InputPort
  - Client^1 ID
  - Client^n ID
  - Caller

- **GroupRPCProxy Generator**
  - Object generateOthers(All)RPCProxy(DuplexRPC aPort, Class aType, String aName)
  - Others (All) Proxy
  - Obj ID
  - m(…)

- **Has-a**
  - Port Ref
  - Obj ID
Server Reply Undirected Server Generator

- Set<String> getConnections()
- (dis)connect()
- String getLocalName()
- String getRemoteEndPoint();
- extends

- Client InputPort
- Server InputPort

Group of clients are destination of call

GroupRPCProxy Generator

Object generateOthersRPCProxy(GroupRPC aPort, Class aType)
Object generateAllRPCProxy(GroupxRPC aPort, Class aType, String aName)
RETURN VALUE IN GROUP CALLS

Server Proxy
- void m(p¹, ... pᴺ)
- Object m(p¹, ... pᴺ)
- Integer m(p¹, ... pᴺ)

Server Object
- void m(p¹, ... pᴺ)
- Object m(p¹, ... pᴺ)
- Integer m(p¹, ... pᴺ)

GIPC returns an array of return values

Otherwise can be cast to Object[]

Exception if return value is not an Object

(Object[]) m(p¹, ... pᴺ)
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 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.
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

- Transparent Remote Procedure Call
- Aware Remote Procedure Call
- Typed Data Communication
- Byte Communication

- Group Transparent Remote Procedure Call
- Group Aware Remote Procedure Call
- Group Typed Data Communication
- Group Byte Communication

- Multicast and marshal
  - Marshaling done multiple times
- Marshal and multicast
  - Marshaling done once
Multicast and Serialize

Multicast and serialize

(Expensive!) Serialization done multiple times

Serialize and multicast

(Expensive!) Serialization done once
Relayer Method Calls

Relayer Client \(\xrightarrow{\text{insert (5, 'c')}}\) Relay Server

Relay Server \(\xrightarrow{\text{insert (5, 'c')}}\) Relay Client

Relay Client \(\xrightarrow{\text{insert (5, 'c')}}\) Relayer Client

Relayer Client \(\xrightarrow{\text{insert (5, 'c')}}\) Relayer Client

Multicast and marshal

Relayer must know about each forwarded call
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?

- **RMI**
  - Exported? (Remote)
    - Proxy
  - Serializable?
    - Not Serializable Exception
      - Proxies generated by callee process through export
      - Only Remote instances can be exported

- **GIPC**
  - Remote?
    - Proxy
  - Serializable?
    - Proxy
      - Proxies generated by caller process end, so no export
      - Proxies generated for arbitrary objects
**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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object transformReceivedReference(Object possiblyRemote)</td>
<td></td>
</tr>
<tr>
<td>Object transformSentReference (Object possiblyRemote)</td>
<td></td>
</tr>
<tr>
<td>void transformReceivedReferences(Object[] args)</td>
<td></td>
</tr>
<tr>
<td>void transformSentRemoteReferences(Object[] args)</td>
<td></td>
</tr>
<tr>
<td>Object getProxy(Object remoteSerializable)</td>
<td></td>
</tr>
<tr>
<td>connectRemoteAndRemoteSerializable(Object remoteSerializable, Object remote)</td>
<td>Connects a remote proxy with a local proxy for remoteSerializable</td>
</tr>
<tr>
<td>Object createRemoteSerializable ( String remoteEndName, String aClass, String anObjectName)</td>
<td>Creates a remote Serializable object with specified properties</td>
</tr>
<tr>
<td>RemoteSerializable getRemoteSerializable( Object getProxy)</td>
<td></td>
</tr>
</tbody>
</table>

Delegation analogue of Java ReplaceObject overriding
**Object Forwarders and Serializers**

- **Sender <Object> (RPC Port)**
  - Call Message and Return Message SendTrapper <Object, Object>

- **LocalRemote Reference Translator**
  - Synchronizing Shared Data Structures

- **Receive Notifier <Object>**
  - Call Message and Return Message ReceiveTrapper <Object, Object>
  - Receive Listener <Object> (RPC Port)

---

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**

- DuplexSerializableCall
- TrapperSharedState
- DuplexSentCallCompleter
- duplexSentCallCompleter
- LocalRemoteReferenceTranslator
- localRemoteReferenceTranslator
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);
}
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;
    }
}
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));
}
**Duplex Caller Flow**

- **Proxy**
  - `m()`
  - `Proxy` -> `m()`

- **Naming RPC**
  - `call()`
    - `Naming RPC` -> `call()`
    - `call()` -> `marshalledCall()`
    - `marshalledCall()` -> `send()`
      - `send()` -> `send()`

- **Marshaller**
  - `marshallCall()`
    - `marshallCall()` -> `send()`

- **LocalRemote Reference Translator**
  - `LocalRemote Reference Translator`
    - `transform ReceivedRemote References()`
    - `transform ReceivedRemote References()`

- **Duplex SentCall Completer**
  - ` DUPLEX SENTCALL COMPLETER`
    - `maybe Process Return Value()`
    - `maybe Process Return Value()`

- **ReceiveListener**
  - `message Received()`
    - `message Received()`
    - `receive()`

- **ReceiveTrapper**
  - `notifyPort Receive()`
    - `notifyPort Receive()`
    - `send()`

- **Send Trapper**
  - `send()`
    - `Send Trapper`
    - `send()`

- **RPC Return Value Queue**
  - `putReturn Value()`
    - `putReturn Value()`
    - `takeReturn Value()`
    - `takeReturn Value()`

- **Receive Listener**
  - `message Received()`
    - `message Received()`
    - `receive()`

- **Receive Trapper**
  - `notifyPort Receive()`
    - `notifyPort Receive()`
    - `send()`

- **Send Trapper**
  - `send()`
    - `send()`

- **Duplex Sent Call Completer**
  - `maybe Process Return Value()`
    - `maybe Process Return Value()`

- **Receive Listener**
  - `message Received()`
    - `message Received()`
    - `receive()`

- **Receive Trapper**
  - `notifyPort Receive()`
    - `notifyPort Receive()`
    - `send()`

- **Send Trapper**
  - `send()`
    - `send()`

- **Duplex Sent Call Completer**
  - `maybe Process Return Value()`
    - `maybe Process Return Value()`

- **Receive Listener**
  - `message Received()`
    - `message Received()`
    - `receive()`

- **Receive Trapper**
  - `notifyPort Receive()`
    - `notifyPort Receive()`
    - `send()`

- **Send Trapper**
  - `send()`
    - `send()`
**Duplex Callee Flow**

- **ReceiveListener**
  - `message Received()`

- **ReceiveTrapper**
  - `notifyPort Receive()`

- **Duplex ReceiveCall Invoker**
  - `message Received()`

- **Runnable**
  - `put()`
  - `Blocking Queue`
  - `take()`

- **Duplex ReceiveCall Invoker**
  - `message Received()`

- **Send Trapper**
  - `send()`

- **NamingSender**
  - `send()`

- **LocalRemote Reference Translator**
  - `transform SentRemote References()`

- **Object**
  - `m()`

- **Send Trapper**
  - `send()`

- **Duplex ReceiveCall Invoker**
  - `message Received()`

- **ReceiveListener**
  - `message Received()`
RMI vs. GIPC Binding Time

RMI Registry

Caller

Callee

Caller

Callee
GIPC: Sharing with Other Clients?

**Diagram:**
- DuplexRPCClient InputPort
- Client
- Remote Object
- m(...)
- DuplexRPCClient InputPort
- Duplex(Group)RPC Server InputPort
- Client\(^1\) ID
- Client\(^n\) ID
- Caller
- Proxy
- m(...)

No direct connection between clients

Could relay calls through server
Only exported objects can be communicated as parameters

Exporting an object creates a proxy and creates a GIPC-like server port if such a port has not been created already
COMMUNICATION AND TYPES OF PROXIES

- DuplexRPCClient InputPort
- GroupRP(Server InputPort

- Client¹ ID
- Clientⁿ ID
- Caller

Directed Proxy
m(…)

Directed Proxy
m(…)

Directed Proxy
m(…)

Directed Proxy
m(…)

Reply Proxy
m(…)

Others (All) Proxy
m(…)

1. Directed Proxies
2. Reply Proxies
3. Others (All) Proxies
COMMUNICATION AND TYPES OF PROXIES

SimplexRPCClient InputPort

Directed Proxy
m(...)

Client\(^1\) ID

Client\(^n\) ID

Caller

SimplexRPServer InputPort

Directed Proxy
m(...)
PROXIES TO SAME OBJECT: NEW PROXY?

DuplexRPCClient InputPort

Client¹ ID
Clientⁿ ID
Caller

DuplexRPServer InputPort

Remote Object / Directed Proxy

m(…)

Directed Proxy

Directed Proxy

m(…)

RMI solution

125
Proxies to Same Object: Same Proxy?

DuplexRPCClient InputPort

Remote Object / Directed Proxy

m(...)

DuplexRPCServer InputPort

Client^{1} ID

Client^{n} ID

Caller

Directed Proxy

m(...)
IMPLEMENTING SAME COPY: ALGORITHM

Serialized proxy stored and associated with remote object/proxy

Received serialized proxy compared with stored serialized copy using equals
LocalRemoteReferenceTranslator

Object transformReceivedReference(Object possiblyRemote)

Object transformSentReference (Object possiblyRemote)

transformReceivedReferences(Object[] args)

transformSentRemoteReferences(Object[] args)

Object getProxy(Object remoteSerializable)

connectRemoteAndRemoteSerializable(Object remoteSerializable, Object remote)

Object createRemoteSerializable (String remoteEndName, String aClass, String anObjectName)

Delegation analogue of Java ReplaceObject overriding
**TransformSentReference**

Remote?

Serializable Proxy Cached?

Generate name for remote, and register it with RPCRegistry

Cache and return proxy = {object name, interface name, source name}

Return reference

Return proxy

Stub proxy not serializable (RPC port and interface in invocation handler)

Serializable version sent
Serializable Proxy?

Cached?

Generate and cache stub proxy

Local?

return local object

Return reference

Return cached stub proxy

Return new stub proxy
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;
}
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);
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;
}
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;
}
public interface Counter extends Remote {
    void increment(int val);
    int getValue();
}
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();
    }
}
public interface ComparableCounter extends Counter{
    public ComparableCounter greater(ComparableCounter aCounter);
}
public class AComparableCounter extends ACounter implements ComparableCounter {
    public ComparableCounter greater(ComparableCounter aCounter) {
        if (getValue() < aCounter.getValue()) {
            return aCounter;
        }
        else return this;
    }
}
public class GIPCComparableCounterServer
    extends ComparableCounterLauncher{
    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();
        }
    }
}
public static void main (String[] args) {
    try {
        DuplexRPCClientInputPort aClientInputPort =
            DuplexRPCInputPortSelector.createDuplexRPCClientInputPort(
                "localhost", SERVER_PORT, SERVER_NAME, "counter client");
        RPCProxyGenerator rpcProxyGenerator =
            aClientInputPort.getRPCProxyGenerator();
    }
}
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();
}}
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**

- **Client Object**
  - **Server Proxy**
    - $m(p^1, \ldots, p^N)$
  - **Client Port**
- **Server Object**
  - **Server Port**
    - $m(p^1, \ldots, p^N)$

**Client and Server Port Tasks**
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

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

An operation should not have to wait unless it has to

Programmer makes tradeoff between reusability and error checking

Factories, Abstract Factories, PortLauncherSupport, PortLauncher, Open Source, Layering

Client and server input ports define name space and used for connecting, sending, receiving remote object registration and proxy generation

Observer pattern for async IPC operations and only function calls block for compatibility with local procedure call syntax

Can generate proxies for all methods, and warning given if method in a Remote interface does not acknowledge RemoteException

Programmer makes tradeoff between reusability and error checking
GIPC GOALS VS. FEATURES

- No central server for object registration and proxy creation
- Only code that needs client awareness should be client aware
- Support group communication

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

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

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**

**Server**
- Start or connect to existing RMIRegistry
- Create server RPC port and connect to it, possibly defining connect listeners and threads
- Generate proxy object(s) for server object(s)
- Register server object(s) with port
- Register proxy object(s) with RMIRegistry

**Client**
- Connect to RMIRegistry
- Create client server port (after initializing port launcher support)
- Fetch proxy objects
- Generate proxy objects for server objects
- Use proxy objects to invoke methods
- Connect to port, possibly implementing connect listener and threads
- Use proxy objects to invoke methods

Automation?
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**

**Caller**
- Called site-independent operation identifier and actual parameters
- Called interface description (if type checking)
- Called site-independent instance identifier (if object-oriented)
- How to send actual parameters and receive result
- Port (Scope)

**Callee**
- Called site-independent operation identifier and actual parameters
- Mapping between site-independent operation and local operation
- Mapping between site-independent instance and local instance
- Called site-independent instance identifier (if object-oriented)
- How to receive actual parameters and send result
- Port (Scope)

*Systems differ in how much information provided by application*
**Full RPC Awareness**

- **Caller**
  - Called site-independent operation identifier and actual parameters
  - Called interface description (if type checking)
  - Called site-independent instance identifier (if object-oriented)
  - How to send actual parameters and receive result
  - Port (Scope)

- **Callee**
  - Called site-independent operation identifier and actual parameters
  - Mapping between site-independent operation and local operation
  - Mapping between site-independent instance and local instance
  - Called site-independent instance identifier (if object-oriented)
  - How to receive actual parameters and send result
  - Port (Scope)

RPC clients provide and thus are aware of all information needed.
**Full Awareness: Sun non-OO RPC**

- `registerrpc (prognum, versnum, procnum, procaddr, inproc, outproc)`
- `callrpc (host, prognum, versum, procnum, inproc, &in, outproc, &out)`
- `svc_run()`
```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0

typedef struct {
    int f1;
    float f2
} S;

extern float add();

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

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);

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

Common

XDR Routines

Callee

Call

Add Interface

Full Awareness: Sun non-OO RPC
```
## Implementing Full Awareness

### Caller

- `callrpc(host, PROG_NUM, VERS_NUM, PROC_NUM, xdr_arg, arg, xdr_result, result)`
- `clientPort ← getOrCreateAndConnect(host, PROG_NUM) if not already connected`
- `send(clientPort, VERS_NUM, PROC_NUM, arg, xdr_arg)`
- `return(receive(dataPort, xdr_result))`

### Callee

- `registerrpc(PROG_NUM, VERS_NUM, PROC_NUM, proc_adr, xdr_arg, xdr_result)`
- `register({VERS_NUM, PROC_NUM}, {proc_adr, xdr_arg, xdr_result})`
- `serverPort ← getOrCreateAndConnectServerPort(PROG_NUM)`
- `reply(execute(proc_adr, xdr_arg, xdr_result))`
- `svc_run();`
- `{vers_num, proc_num, arg_bytes} ← receive(serverPort)`
- `{proc_adr, xdr_arg, xdr_result} ← lookup({vers_num, proc_num})`
- `reply(execute(proc_adr, xdr_arg, xdr_result))`
FULL APPLICATION AWARENESS: SUN NON-O0
RPC

Add Interface

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

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

Common

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

XDR Routines

```c
Callee

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

```c
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);
```

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

How changed for full transparency?
**Full Application Awareness: Sun Non-OO RPC (Review)**

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

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

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);

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

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

Callee

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

Caller

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

How changed for full transparency?

Callee

float add (s)
S *s;
{
    return (s->f1 + s->f2);
}
```
FULL APPLICATION TRANSPARENCY: CEDAR RPC

Common

Add Interface

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

Caller

result = add(3, 4.12);

Callee

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

How to implement fully transparent RPC?

Assume fully aware rpc is given
**Full Application Awareness: Sun Non-OO RPC**

### Add Interface

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

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

### Caller

```c
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);
```

### Callee

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

### XDR Routines

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

### Common

```c
registerrpc (PROG_NUM, VERS_NUM, ADD_NUM,
    add, xdr_s, xdr_float);
svc_run();
```
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0

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);
    }
}

typedef struct {
    int f1;
    float f2
} S;

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();

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

**Callee Skeleton**
- Created for each callable operation
- Receives message (using lower-level mechanism) and unmarshals parameters
- Calls implementation, marshals and sends result

#### Code Examples

**add**
```c
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);
  }
}
```

**add_skel**
```c
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

Syntax transparency

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

Application transparency $\rightarrow$ syntax transparency

Syntax transparency $\rightarrow$ application transparency

Syntax transparency $\rightarrow$ client stubs

Syntax transparency $\rightarrow$ server skeleton
#define PROG_NUM 1
#define VERS_NUM 0
#define ADD_NUM 0

typedef struct {
    int f1;
    float f2
} S;

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);
    }
}

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**

- Maintains own registry of operation network identifier and local address
- Receive remote operation network identifier and arguments
- Call operation based on network identifier and registry
- Java added universal skeleton support on top of operation specific skeleton (or replaced it)
**CONVENTIONAL VS. OBJECT-ORIENTED: FULL APPLICATION AWARENESS**

**Conventional**

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

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

**Object Oriented**

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

```c
registerrpc (PROG_NUM, VERS_NUM, OBJECT_NUM,
PROC_NUM, proc_adr, xdr_arg, xdr_result )
```
**Conventional vs. Object-Oriented: Full Application Transparency**

**Conventional**

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

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

**Object-Oriented**

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

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

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

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

- Need stub/proxy instances in addition to stub/proxy server methods
- How to transparently bind multiple server instances to multiple client handles?
Stub and Skeleton Objects

**Caller Stub**
- Created for a target remote object and some interface of it
- Instance of a stub-class that implements the interface
- Stub-class has stub operation for each operation in the interface
- Syntax transparency → client stubs

**Callee Skeleton**
- Created for a target remote object and some interface of it
- Instance of a skeleton-class
- Skeleton-class has skeleton operation for each operation in the interface
- Syntax transparency (server skeleton)
Semantics vs. Syntax Transparency

**Semantic transparency**

Degree to which remote call behaves like a local call

- Semantics transparency $\rightarrow$ syntactic transparency
- Syntax transparency $\rightarrow$ 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time-out</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Exception, error code</strong></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Caller waits for callee to finish: Synchronous call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming easier if waiting is necessary, allows migration of non-distributed programs</td>
</tr>
<tr>
<td>Unnecessary waiting if not necessary</td>
</tr>
<tr>
<td>Callbacks → deadlocks if special threads not created</td>
</tr>
<tr>
<td>Requires duplex port underneath</td>
</tr>
</tbody>
</table>
## Implicit Calls

<table>
<thead>
<tr>
<th>No action taken to enable callee execution: implicit receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming easier if no synchronization needed, allows porting of non distributed programs</td>
</tr>
<tr>
<td>Must rely on monitors to synchronize, no Ada like guards or select</td>
</tr>
</tbody>
</table>
**Full Awareness: Sun non-OO RPC**

Sun RPC Library

- `registerrpc (prognum, versnum, procnum, procaddr, inproc, outproc)`
- `callrpc (host, prognum, versum, procnum, inproc, &in, outproc, &out)`
- `svc_run()`

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

Diagram:
- Client Object
  - Server Proxy
    - $m(p_1, \ldots p_N)$
- Server Object
  - $m(p_1, \ldots p_N)$
  - Server Port
  - Client Port
VARIABLY AWARE PORT CALLS

Object call(String aRemoteEnd, String anObjectName, Method aMethod, Object[] args)

Object call(String aRemoteEnd, Class aType, Method aMethod, Object[] args)

Object call(String aRemoteEnd, Method aMethod, Object[] args)

Transparency in Application → Awareness in System
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.marshalCall(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
public void send(String aDestination, Object aMessage) {
    Tracer.info(this, "Forwarding call " + aMessage + " to " + aDestination);
    destination.send(aDestination, aMessage);
}

public Object returnValue(String aDestination, Object aMessage) {
    Tracer.info(this, "Waiting for return value of call " + aMessage);
    return simplexSentCall Completer.returnValueOfRemoteMethodCall(null, (Call) aMessage);
}
**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**

- **SimplexSentCall Completer**
  - `Object returnValueOfRemoteMethodCall(String aRemoteEndPoint, Call aCall)`

- **AnAbstract SimplexSentCall Completer**
  - `protected abstract Object returnValueOfRemoteProcedureCall(String aRemoteEndPoint, Call aCall)`
  - `protected abstract Object returnValueOfRemoteFunctionCall(String aRemoteEndPoint, Call aCall)`
 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
public void messageReceived(String aSource, Object aMessage) {
    getReceiveTrapper().notifyPortReceive(aSource, aMessage);
}

Delegates to receive trapper, does little processing
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);
    }
}
Received Call Invoker

messageReceived(String aSource, Object aMessage)

abstract handleFunctionReturn(String aSender, Object aRetVal)

abstract handleProcedureReturn(String aSender)

Translates received call message into invocation of method in server object

Can change procedure and function return without understanding communication flow
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();
    }
}
protected void handleFunctionReturn(String sender, Object retVal) {
    Tracer.error("Ignoring return val of called method:" + retVal);
}

protected void handleProcedureReturn(String sender) {
    return;
}
PROCEDURE CALL SYNCHRONIZATION

ReceivedCall Invoker

messageReceived()

handle ProcedureReturn()

Remote function call is blocking

RemoteObject

T m(...) -> Proxy

SentCall Completer

returnValueOfRemoteProcedureCall()
SYNCHRONOUS CALLBACK IN CONNECT LISTENER

Client connect call is non blocking

Any blocking call (such as read input) creates an issue
**Breaking Deadlock with Application Thread**

- **InputPort**
  - (dis)connect()
  - add(remove)ConnectionListener (ConnectionListener l)

- **ConnectionListener**
  - connected(String aRemoteEndName)

- **Selector**
  - select()

- **NIO Selection Thread**

- **RPCReturnValue Queue** (BoundedBuffer)
  - put()
  - get()

- **Connector**
  - Connect to server RPC port

- **Proxy**
  - T m^2(…)

---

Client connect call is non blocking

Any blocking call (such as read input) creates an issue
**Duplex Shared State**

DuplexSerializableCall
TrapperSharedState

DuplexSentCallCompleter
duplexSentCallCompleter

LocalRemoteReferenceTranslator
localRemoteReferenceTranslator
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.dualSentCallCompleter
        .returnValueOfRemoteMethodCall(duplexRPCInputPort
            .getLastSender(), call);
}
public void notifyPortReceive(String aSource, Object aMessage) {
    Tracer.info("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);
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**

- **AnAbstract SimplexSentCallCompleter**
  - protected abstract Object `returnValueOfRemoteProcedureCall(String aSource, Call aCall)`
  - protected abstract Object `returnValueOfRemoteFunctionCall(String aSource, Call aCall)`

- **AnAbstractDuplexSentCallCompleter**
  - abstract protected Object `processReturnValue(String aSource, Object aMessage)`

- **Non blocking, simply returns**
- **Blocks for return value**
- **Unblocks**
Specialized Bounded Buffer: AnRPCReturnValueQueue

```java
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**

- **AnAbstract SimplexSentCall Completer**
  - `protected abstract Object returnValueOfRemoteProcedureCall(String aSource, Call aCall)`
  - `protected abstract Object returnValueOfRemoteFunctionCall(String aSource, Call aCall)`

- **AnAbstractDuplex SentCallCompleter**
  - `abstract protected Object processReturnValue(String aSource, Object aMessage)`

- **Non blocking, simply returns**
- **Blocks for return value**
- **Unblocks**
PRODUCER/CONSUMER CALLS IN: ADUPELXSENTCALLCOMPLETER

//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(
        (RPCReturnValueValue) message);
}
SYNCHRONOUS CALLBACK IN REMOTE CALL

Remote function call is blocking

RPCMessage Receiver

messageReceived()

m^1(...) RemoteObject

Selector

select()

NIO Selection Thread

Call

Return Val

Call

Object

notify()

wait()

Proxy

T m^2(...)
Breaking Deadlock with GIPC Thread

Remote function call is blocking

RPCMessage Receiver
messageReceived()
m¹(…)
RemoteObject

Selector
select()

NIO Selection Thread

RPCReturnValue Queue (BoundedBuffer)
put()
get()

GIPC Asynchronous MethodInvoker

GIPC thread used only for synchronous calls in remote call

Proxy
T m²(…)

Call
Return Val
Call
Synchronous Callback in Connect Listener

Client connect call is non-blocking

Any blocking call (such as read input) creates an issue
BREAKING DEADLOCK WITH APPLICATION THREAD

- InputPort
  - (dis)connect()
  - add(remove)ConnectionListener (ConnectionListener l)

- ConnectionListener
  - connected(String aRemoteEndName)

- Selector
  - select()

- NIO Selection Thread

- RPCReturnValue Queue
  - put()
  - get()

- Connector
  - Connect Listener
  - Connect to server RPC port

- Client connect call is non blocking
- Any blocking call (such as read input) creates an issue

Connected
Return Val
Call
**DIRECT SERVER PROXY**

- **SimplexRPC**
  - Has-a Callee ID
  - String getRemoteEndPoint();

- **SimplexRPServer**
  - InputPort
  - register(String aName, Object aRemoteObject)

- **DirectedProxy**
  - Has-a Port Ref
  - Has-a RemoteEnd
  - Has-a Obj ID
  - m(…)

- **DirectedRPCProxyGenerator**
  - Object generateRPCProxy(SimplexRPC aPort, Class aType, String aName)

- **Callee**
  - Remote Object
  - Obj ID
  - m(…)

- **Callee ID**