ARCHITECTURE OF AN RPC IMPLEMENTATION

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)
Interesting and Uninteresting Architectural Aspects

Interesting: Those that can be justified through accepted principles
# Architectural Principle

## Separation of Concerns

If logical component A can be used with logical component B and B’ then implement the three as separate physical components.
STRONGER ARCHITECTURAL PRINCIPLE

Arrange components in “layers”
Layering Principle

Layer^N

Layer^{N-1}

Layer^1

Layer^0

Layer^i can see the API of Layer \( i-1 \)

Maybe even lower-layers

Layer^i cannot see the API of Layer \( i+1 \)

Layer^i make upcalls in Layer \( i+1 \)

Signatures of upcalls is defined by Layer^i
Classical Layering Example: Networking

Port-based Communication (TCP, UDP)
Internetworking (IP)
Physical Layer (Ethernet, WIFI)

Relevance to RPC?
Understanding Layering: Operating Systems

- Memory Management
- File Management
- Interprocess Communication
- Process Management
Understanding Layering: Operating Systems (Unix, Windows)

- Interprocess Communication
- File Management
- Process Management
- Memory Management
- Distributed File Management

Logical function (file management) split into two physical parts to follow layering
Understanding Layering: Operating Systems (Mach)

- Distributed File Management
- Local File Management
- Interprocess Communication
- Process Management
- Memory Management

Relevance to RPC?
Understanding Layering: Compilers

- Code Generation
- Parser
- Scanner

Relevance to RPC?
Why Layering?

- Can explain/understand incrementally
- Can implement incrementally
- Can bootstrap off lower layers
- Can use lighter-weight lower level layers directly
- Can replace layers (separation of concerns)
Classical RPC (Sun RPC)

Can we do better?

Can replace one socket implementation with another
Object communication often used directly as an abstraction

Cannot replace Sockets with NIO
**GIPC RPC**

<table>
<thead>
<tr>
<th>Buffer Input Port (Server and Client)</th>
<th>Buffer Input Port (Server and Client)</th>
<th>Others Channel-Specific Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>NIO</td>
<td>Other</td>
</tr>
</tbody>
</table>

**Can choose driver**

Channel: Mechanism that delivers and receives messages on the network

Layer Visibility?
Layer Visibility

Proxy calls processed through RPC Port that offers interface similar to Sun RPC

RPC Port IS-A Object Port IS-A Buffer Port

Higher-layers expose the functionality of lower-level layers
## GIPC RPC (Review)

<table>
<thead>
<tr>
<th>Buffer Input Port (Server and Client)</th>
<th>Buffer Input Port (Server and Client)</th>
<th>Others Channel-Specific Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket Driver</td>
<td>NIO Driver</td>
<td>Other</td>
</tr>
</tbody>
</table>

- **RPC (RPC Input Port)**
- **Object Communication (Object Input Port)**
- **Can choose driver**
  - **Channel: Mechanism that delivers and receives messages on the network**
- **Layer Visibility?**
Layer Visibility (Review)

Proxy calls processed through RPC Port that offers interface similar to Sun RPC

RPC Port IS-A Object Port IS-A Buffer Port

Higher-layers expose the functionality of lower-level layers

Call message (operation id, parameters, source)

Non RPC Byte or Object Message

Return message (return value)

Non RPC Byte or Object Message

Higher-layers expose the functionality of lower-level layers
public class ASimpleGIPCCounterClient implements SimpleCounterClient {
protected static DistributedRMICounter counter;
protected static GIPCRegistry gipcRegistry;
public static void init(String aClientName) {
    gipcRegistry = GIPClocateRegistry.getRegistry("REGISTRY_HOST_NAME",
           "REGISTRY_PORT_NAME", aClientName);
    counter = (DistributedRMICounter)
              gipcRegistry.lookup(DistributedRMICounter.class, "COUNTER_NAME");
}
public static void doOperations() {
    try {
        counter.increment(1);
        System.out.println(counter.getValue());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public class AMultiLayerCounterClient extends ASimpleGIPCCounterClient {
    protected static DuplexRPCClientInputPort duplexRPCClientInputPort;
    protected static void setPort() {
        duplexRPCClientInputPort = gipcRegistry.getRPCClientPort();
    }
    public static void main (String[] args) {
        ....
        setPort();
        init("Client 1");
        sendByteBuffers();
        sendObjects();
        doOperations();
    }
}
Example Multi-Layer Sends

```java
protected static void sendByteBuffers() {
    ByteBuffer aByteBuffer = ByteBuffer.wrap("3").getBytes();
    duplexRPCClientInputPort.send(aByteBuffer);
}

protected static void sendObjects() {
    duplexRPCClientInputPort.send(2);
}

public static void doOperations() {
    try {
        counter.increment(1);
        System.out.println(counter.getValue());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```
Receive?

| 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 $\geq 1$ thread |
Observer Pattern

- **Observable**
  - add(remove) Event Listener (Observer o)

- **Observer**
  - Observer^1
  - Observer^2
  - Observer^3
  - Observer^N
  - eventOccurred (…)

- **Observable**
  - add(remove) Event Listener (Observer o)
Receive Observable and Observer

ServerInputPort<MessageType>

add(remove)ReceiveListener
  (ReceiveListener<MessageType>)

ReceiveListener<MessageType>

messageReceived(String aSourceName, MessageType aMessage);

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
public class AMultiLayerCounterServer extends ASimpleGIPCRegistryAndCounterServer {
    protected static DuplexRPCServerInputPort duplexRPCServerInputPort;
    protected static void setPort() {
        duplexRPCServerInputPort = gipcRegistry.getRPCServerPort();
    }
    public static void addListeners() {
        duplexRPCServerInputPort.addReceiveListener(new AMultiLayerServerReceiveListener(counter));
    }
    public static void main(String[] args) {
        init();
        setPort();
        addListeners();
    }
}
public class AMultiLayeServerReceiveListener extends ASimpleGIPCRegistryAndCounterServer implements ReceiveListener {
    protected DistributedRMICounter counter;
    public AMultiLayeServerReceiveListener(DistributedRMICounter aCounter) {
        counter = aCounter;
    }

    public void messageReceived(String aSourceName, Object aMessage) {
        try {
            if (aMessage instanceof ByteBuffer) {
                Integer anInt = Integer.parseInt(AGenericSimplexBufferServerInputPort.extractString((ByteBuffer) aMessage));
                counter.increment(anInt);
            }
            else {
                counter.increment((Integer) aMessage);
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
Separation of Concerns

- Can explain/understand incrementally
- Can implement incrementally
- Can bootstrap off lower layers
- Can use lighter-weight lower level layers directly
- Can replace layers (separation of concerns)
# Fine-Grained Componentization

<table>
<thead>
<tr>
<th>RPC (RPC Input Port)</th>
<th>Object Communication (Object Input Port)</th>
<th>Others Channel-Specific Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Input Port (Server and Client)</td>
<td>Buffer Input Port (Server and Client)</td>
<td>NIO Driver</td>
</tr>
<tr>
<td>Socket</td>
<td>NIO</td>
<td>Other</td>
</tr>
</tbody>
</table>

Components?
Pipe-Based Extensibility

Can add arbitrary long chains of application message trappers between a producer/consumer pair.
Trappers

Predefined and Programmer-defined trappers

Layer^i^ Sender (Port)

Layer^i^ Trapper

Layer^i^ Trapper

Layer^i-1^ Sender (Port)

Layer^i^ Receive Listener

Layer^i^ Receive Notifier

Layer^i-1^ Trapper

Layer^i-1^ Trapper

Layer^i-1^ Receive Listener
Proxies vs. Translators

Sender <Object (Object Port)

Send Forwarder <Object, Object>

Serializer <Object, ByteBuffer>

Sender<ByteBuffer> (Buffer Port)

Proxy

Receive Listener<Object>

Receive Notifier<Object>

ReceiveForwarder <Object, Object>

Deserializer <ByteBuffer, Object>

Receive Listener <ByteBuffer>
Shared Trappers and Synchronizations

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)

Synchronizing Shared Data Structures

Threads?

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
Threads?

- **Send pipeline:** Sender thread until NIO layer
- **Receive pipeline:** Selector thread until possibly RPC layer
ARCHITECTURE SUMMARY

- Layers are used in several fields:
  - Networks, OS, Compilers, Distributed Systems
- Regular and upcalls
  - Upcall signatures defined by layer making the call
- They have several advantages
  - Learning, implementation, bootstrapping, intermedia abstractions, customization
- Sometimes a logical function must be split to layer it properly.
- Java RMI introduces object communication layer over sockets
- GIPC introduces buffer layer over sockets, NIO
  - RPC, Object and Buffer Ports
- Allows multi-layer communication
- Implicit Receive with Distributed Send-based Observer Pattern
- Trappers
  - Send goes from high-level to low-level port through replaceable send trappers
  - Receive goes from low-level to high-level ports through replaceable receive trappers, notifiers (defined by lower-level layers), which make upcalls