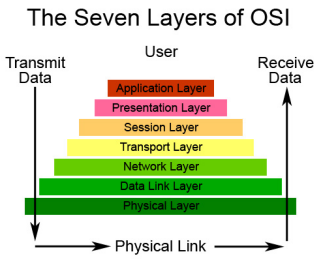
COS 461
Recitation 7

Remote Procedure Calls

Let's Look at Layers Again



Common Networked Application Pattern

- APP1 sends message to APP2, expecting Reply
- Message has a *static* part and a *dynamic* part
- Similar to a function call!
  - *Static* = function name
  - *Dynamic* = function args

Remote Procedure Calls (RPCs)

- Not a particular "protocol", rather a class of application protocols
- Common Elements:
  - Procedure names known a priori
  - Arguments are fixed length, usually typed
  - Often: Arguments supplied *as plain code objects*
  - Protocols need to define:
    - Message Format
    - How to translate from code to message format

Remote Procedure Calls (RPCs)

- Are RPCs just like normal procedure calls?
  - No!
  - Calls traverse network: many possible problems / exceptions
- Can't libraries abstract away the networking?
  - NO!!

Trying to Abstract the Network

- Trying to mask failures is a Bad Thing ™
- Example: network timeout
  - Do you retransmit automatically?
- Potential Solutions?

Using *Nonces*

- Nonce : unique-ish number
- Receiver can tell if a message is repeated
- What about responses to the client?
- Can we guarantee the following?
  - *If a RPC is processed by the server, the client will receive a successful response.*

Let's Look at Real RPC Protocols

- Message Formats:
  - XML and JSON
- Protocols / Libraries
  - Java RMI
  - Google Protobufs

Common Message Formats

- XML and JSON most common "general formats"
  - These are "string" formats
  - (typically UTF-8 or even ASCII)
- XML is horrible

```
<Message type="terribleRPCformat" version="1">
    <procedure name="foo">
        <argument number="1" value="bar">
    </procedure>
</Message>
```

- Compare to just saying "foo(bar)"
  - Message is longer, harder to parse, etc.

JSON is a bit better

- JSON has lists, values and "dictionaries"
- Looks like:

```
{"type": "sillyRPCFormat",
 "procedure": "Foo",
 "arguments": ["bar"]
}
```

- Still kind of a silly format
  - That's what you get for string-based "object" formats, though.

Java RMI

- Biggest Issue for Java Library:
  - Allowing *objects* to be used in procedure calls
- Java Serializable
  - POJOs in, Bytes out
  - MAGIC?!

Java Serialization is not, as it turns out, Magic.

- Marking class "Serializable" indicates that it is "okay to serialize"
- Library inspects the object:
  - For every field, attempt to serialize()
  - Primitives, such as int and char, have hardcoded serialization functions
  - Write an "identifier" for the Object's type.
    - Includes Object's fully-qualified name, and a *version*

Default Java Serialization is Expensive

- The algorithm is not *theoretically* expensive
- However, crawling object reference graphs is expensive in practice.
- This requires lots of indirect memory fetches, which are not necessarily known by the library
- *E.g.,* Object A may have an Object[] array. This array can store arbitrary types!
  - *How much space would you need to allocate?*
  - *What kind of Objects do you expect to need to serialize?*

Google Protobufs

- Programmers define the contents of the message
  - Specify exactly what the *output* of the serialization will be
  - Allows for arrays – but these arrays must be of single types
- Programmers must also define exactly how objects are translated
  - There are automatic tools to help with this
- With the definition, the library optimizes the output, packs it into a condensed binary format