RPC and its Offspring:
Convenient, Yet Fundamentally Flawed

Steve Vinoski
Verivue, Inc.
Westford, MA USA

QCon London 2009

Some RPC Observations from programming.reddit.com

- From last week, some pre-talk commentary from someone posting a link to the abstract for this talk:
  - *"...you're using a rigorous and precise definition of 'RPC', whereas, loosely speaking,* ***any message sent online could be considered as a Remote Call to a Procedure****.*

    Sometimes rigor and precision can be excessive, but I think your definition is justified because it represents exactly how people used it for many years."
- The definition of RPC seems to be widely misunderstood
  - "Remote Procedure Call" means something very specific, which we'll cover in detail later
  - For now, let's be clear: just "any message sent online" cannot be considered to be RPC

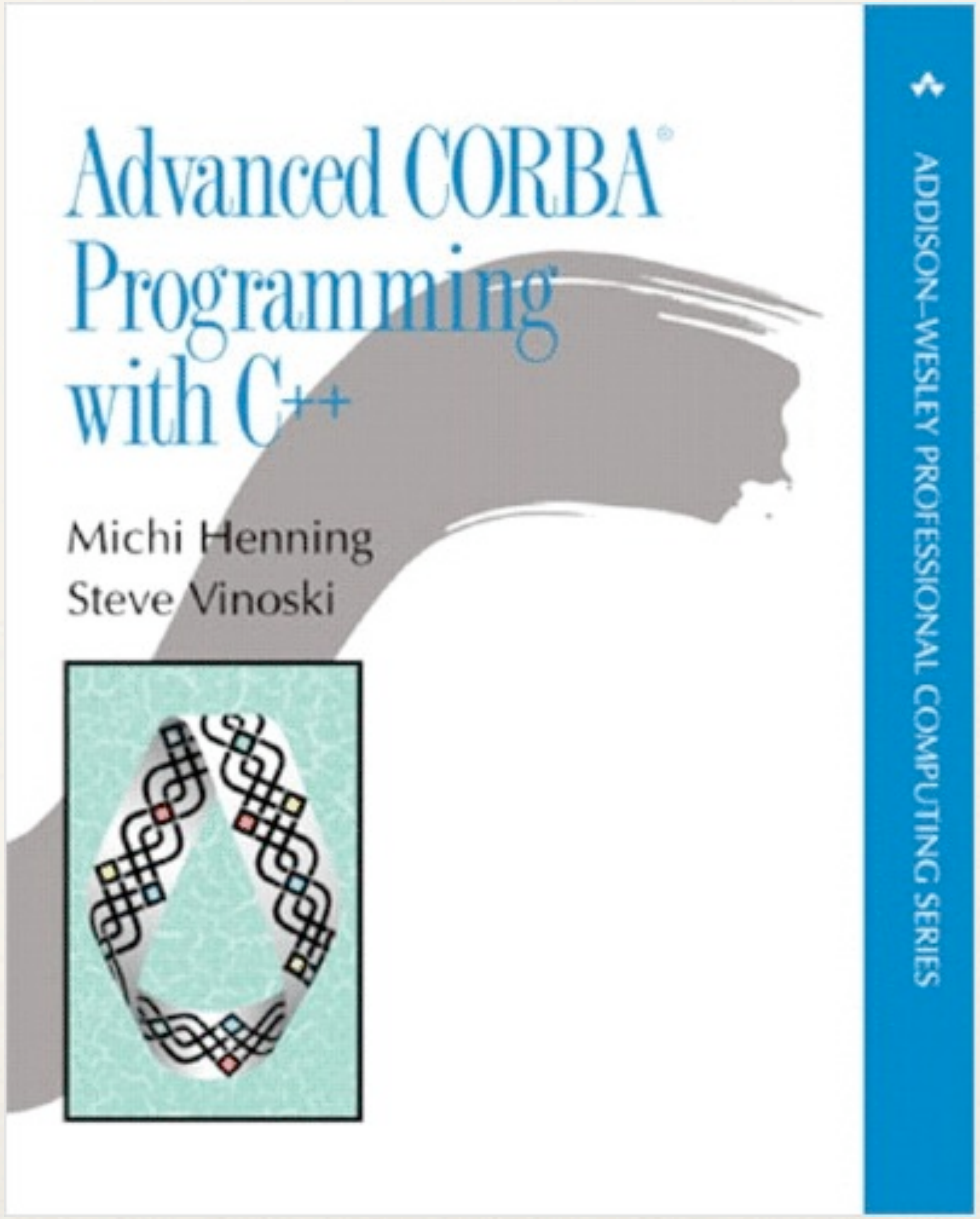
More programming.reddit.com Observations

- *"Yet another Anti-RPC rant. Yawn."*
- If this were just another "anti-RPC rant," I'd agree, not much point
- The point of this track is to learn from the history of software development
  - there's a great deal to be learned from studying and understanding the assumptions and circumstances surrounding RPC
- Today we'll cover issues that go well beyond pure RPC considerations

Say, Aren't You That CORBA Guy?

- Published in January 1999
- I think it was good work, but 10 years is a long time
- *"When the facts change, I change my mind. What do you do, sir?"*

John Maynard Keynes


Advanced CORBA® Programming with C++
Michi Henning
Steve Vinoski
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Early Networked Systems

- ARPANET, forerunner of the Internet, started operating in late 1969
- Early host-to-host protocols facilitated human-to-computer communications
  - Email in 1971
  - FTP and interoperable Telnet in 1973
- Interest started growing in application-to-application protocols

RFC 707: the Beginnings of RPC

- In late 1975, James E. White wrote RFC 707, "A High-Level Framework for Network-Based Resource Sharing"
- Tried to address concerns of application-to-application protocols:
    - *"Because the network access discipline imposed by each resource is a human-engineered command language, rather than a machine-oriented communication protocol, it is virtually impossible for one resource to programatically draw upon the services of others."*
- Also concerned with whether developers could reasonably write networked applications:
    - *"Because the system provides only the IPC facility as a foundation, the applications programmer is deterred from using remote resources by the amount of specialized knowledge and software that must first be acquired."*

Procedure Call Model

- RFC 707 proposed the "Procedure Call Model" to help developers build networked applications
  - developers were already familiar with calling libraries of procedures
  - *"Ideally, the goal…is to make remote resources as easy to use as local ones. Since local resources usually take the form of resident and/or library subroutines, the possibility of modeling remote commands as 'procedures' immediately suggests itself."*
  - the Procedure Call Model would make calls to networked applications look just like normal procedure calls
  - *"The procedure call model would elevate the task of creating applications protocols to that of defining procedures and their calling sequences."*

Interesting RFC 707 Quotes

- *"The Model is further confirmed by the similarity that exists between local procedures and the remote commands to which the Protocol provides access.* ***Both carry out arbitrarily complex, named operations on behalf of the requesting program (the caller); are governed by arguments supplied by the caller; and return to it results that reflect the outcome of the operation****. The procedure call model thus acknowledges that, in a network environment, programs must sometimes call subroutines in machines other than their own."*

- *"This integration of local and network programming environments can even be carried as far as modifying compilers to provide minor variants of their normal procedure-calling constructs for addressing remote procedures…"*

- The RFC also describes the basic implementation issues that implementations would need to provide to support the Procedure Call Model

RFC 707 Warnings

- The RFC also documents some potential problems with the Model
- *"Although in many ways it accurately portrays the class of network interactions with which this paper deals, the Model...may in other respects tend to mislead the applications programmer.*
  - *Local procedure calls are cheap; remote procedure calls are not.*
  - *Conventional programs usually have a single locus of control; distributed programs need not."*
- It presents a discussion of synchronous vs. asynchronous calls and how both are needed for practical systems.
- *"...the applications programmer must recognize that by no means all useful forms of network communication are effectively modeled as procedure calls."*

So What is RPC?

- The Wikipedia definition is reasonable:
  - *"Remote procedure call (RPC) is an Inter-process communication technology that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer would write essentially the same code whether the subroutine is local to the executing program, or remote."*
- I'd stress one key aspect given here, and add a few others:
  - same code whether local or remote — remote calls look like local calls
  - client invokes the remote procedure directly by name within the text of its program (identical to local coupling)
  - remote procedure executes directly on behalf of the client, not as some internal side effect of the call's execution within the server

Next Stop: the 1980s

- Systems were evolving: mainframes to minicomputers to engineering workstations to personal computers
  - these systems required connectivity, so networking technologies like Ethernet and token ring systems were keeping pace
- Methodologies were evolving: structured programming (SP) to object-oriented programming (OOP)
- New programming languages were being invented and older ones were still getting a lot of attention: Lisp, Pascal, C, Smalltalk, C++, Eiffel, Objective-C, Perl, Erlang, many many others
- Lots of research on distributed operating systems, distributed programming languages, and distributed application systems

1980s Distributed Systems Examples

- BSD socket API: the now-ubiquitous network programming API
- Argus: language/system designed to help with reliability issues like network partitions and node crashes
- Xerox Cedar project: source of the seminal Birrell/Nelson paper "**Implementing Remote Procedure Calls**," which covered details for implementing RPC
- Eden: full object-oriented distributed operating system using RPC
- Emerald: distributed RPC-based object language, local/remote transparency, object mobility
- ANSAware: very complete RPC-based system for portable distributed applications, including services such as a Trader

Languages for Distribution

- Most research efforts in this period focused on whole programming languages and runtimes, in some cases even whole systems consisting of unified programming language, compiler, and operating system
- RPC was consistently viewed as a key abstraction in these systems
- Significant focus on uniformity: local/remote transparency, location transparency, and strong/static typing across the system
- Specialized, closed protocols were the norm
    - in fact protocols were rarely the focus of these research efforts, publications almost never mentioned them
    - the protocol was viewed as part of the RPC “black box,” hidden between client and server RPC stubs

Meanwhile, in Industry

- 1980s industrial systems were also whole systems
  - vendors provided the entire stack, from libraries, languages, and compilers to operating system and down to the hardware and the network
  - network interoperability very limited
- Users used what the vendors gave them
  - freely available easily attainable alternative sources simply didn't exist
- Software crisis was already well underway
  - Fred Brooks's "Mythical Man Month" published in 1975
  - Industry focused on SP and then OOP as the search for an answer continued

Research vs. Practice

- As customer networks increased in size, customers needed distributed applications support, and vendors knew they had to convert the distributed systems research into practice
  - but they couldn't adopt the whole research stacks without throwing away their own stacks
- Porting distributed language compilers and runtimes to vendor systems was non-trivial
  - only the vendors themselves had the knowledge and information required to do this
  - attaining reasonable performance meant compilers had to generate assembly or machine code
  - systems requiring virtual machines or runtime interpreters (i.e., functional programming languages) were simply too slow

Using Standard Languages

- Industry customers wanted to use "standard" languages like C, FORTRAN, Pascal so they could
  - hire developers who knew the languages
  - avoid having to rewrite code due to languages or vendors disappearing
  - get the best possible performance from vendor compilers
  - use "professional grade" methodologies like SP and OOP
- Vendors benefited from compiler research on code generation for standard languages, still a difficult craft at the time
- Side effect: we developed an affinity for imperative languages that unfortunately still exists even today

Converting Research to Practice

- Vendors had little choice but to
  - incorporate distributed systems research into their own stacks
  - but do so by making distributed programming features available for "normal" programming languages, without changing those languages
- By the end of the 1980s, we had for example:
  - Apollo's Network Computing System (NCS): RPC system with a declarative interface definition language (IDL), the start of DCE
  - Sun's Open Network Computing (ONC) RPC
  - DEC and IBM RPC projects that later fed into DCE and CORBA

Internet Influence

- ARPANET converted to TCP/IP at the beginning of 1983
- Internet services such as email and file transfer continued to improve and gain popularity through the 1980s
- Industry started adopting TCP/IP in the latter half of the 80s
- In general, standards were becoming more important
  - customers were tired of vendor lock-in
  - heterogeneous networks were starting to become more commonplace as networks continued to grow in size
  - Ethernet was taking over, and the days of proprietary networks were numbered
- In 1989, Tim Berners-Lee starts creating the World Wide Web

The 90s: Distributed Objects

- By the early 90s OOP was *the* way to develop software
  - if it wasn't OOP, it was viewed with disdain
  - C++ was quickly gaining popularity because it was *efficient* OOP
- 1980s distributed objects research was quickly heading towards 1990s distributed objects in production
- Companies were running their own distributed objects projects
- But as pointed out earlier, customers demanded standards
  - RPC: Distributed Computing Environment (DCE)
  - Objects: Common Object Request Broker Architecture (CORBA)

CORBA

- First CORBA spec published in July 1991
- Comprised contributions from a number of vendors
  - married static distributed object approaches (HP, Sun, IBM) with dynamic approaches (DEC, others)
- Viable implementations started appearing in 1993-1994
- Very significant corporate investment in CORBA projects, both from vendors and from customers, through the 90s
- Based squarely on 1980s distributed objects research
  - it was all RPC-oriented and language-oriented

CORBA Language Mappings

- A primary goal for CORBA was to make its facilities available to applications in a “language natural” way
- CORBA 1.0 and 1.1 included a C language mapping
- It took 3 years to develop a C++ mapping (trust me, I was there)
  - with one false start due to vendor standardization politics, the whole effort almost completely broke down as a result
  - C++ is a multi-paradigm language, so there are multiple valid ways to use it, and different vendors liked different approaches
  - ended up with a compromise that many disliked
- Enormous investment in the programming language focus, and it was never questioned whether that was even the right focus

A Note on Distributed Computing

- Brilliant 1994 paper by Waldo, Wyant, Wollrath, and Kendall
- Pointed out that distributed objects could not be treated as local objects due to:
  - latency differences
  - differences between local access models and distributed access models (i.e,, trying to make distributed object access follow normal access patterns for local objects)
  - partial failure issues
  - concurrency issues, specifically that distributed systems are inherently concurrent
- Provides amazingly lucid and detailed explanations for all these issues and more
- See also the "Fallacies of Distributed Computing"

But Nothing Changed

- CORBA continued down the same path, as did Microsoft DCOM
- Then Java/J2EE jumped on the CORBA bandwagon, recasting the CORBA approach and CORBA services to be “native Java”
- Since 1999, just more of the same
  - 1999: “Simple Object Access Protocol” appears
    - distributed objects ala CORBA/DCOM but with XML/HTTP
  - 2002: W3C starts Web Services (WS-*) standards
    - hundreds of pages of specs, just “CORBA with angle brackets”
    - often competing specifications from competing vendors

Choosing a Path

- Distributed systems and programming language research and development efforts have taken us down many paths, some good, some problematic
- But sometimes certain forces can keep flawed approaches alive for too long:
  - significant corporate investment
  - popular technologies tend to attract more research attention, regardless of flaws
  - ignorance of fundamental technical issues
  - applying inappropriate abstractions and trade-offs
  - choosing convenience in spite of the flaws

Protocol Development: Two Paths

- From "A Note on Distributed Computing":
  - *"Communications protocol development has tended to follow two paths. One path has emphasized **integration with the current language model**. The other path has emphasized **solving the problems inherent in distributed computing**. Both are necessary, and successful advances in distributed computing synthesize elements from both camps."*
- So far we've discussed a number of developments from the RPC path
  - they're clearly a result of emphasizing "integration with the current language model"
- Let's look at a couple of examples from the other path: those emphasizing "solving the problems inherent in distributed computing"

Example 1: Representational State Transfer (REST)

- Roy Fielding defined REST in his excellent Ph.D. thesis, "Architectural Styles and the Design of Network-based Software Architectures"
- REST is the architectural style of the web, intended for large-scale hypermedia systems
  - makes network effects, not languages, the critical issues
  - puts distributed systems problems like latency and partial failure directly front and center
  - specifies clear trade-offs and constraints that help address those problems
- HTTP is the best known RESTful application protocol, others are possible

Properties and Constraints

- Fielding's thesis investigates desired architectural properties for networked applications and the constraints required to induce them
- Some desired properties:
  - performance, scalability, portability, simplicity
  - visibility (monitoring, mediation)
  - modifiability (ease of changing, evolving, extending, configuring, and reusing the system)
  - reliability (handling failure and partial failure, and allowing for load balancing, failover, redundancy)
- REST's constraints: Client-Server, Statelessness, Caching, Layered System, Uniform Interface, Code-on-demand

Contrast with RPC Systems

- It’s interesting to compare Fielding’s methodical analysis of properties, constraints, and trade-offs with the typical RPC-oriented distributed system
- On the RPC side, focus is on the API
  - service interfaces
  - operations, arguments and return values
- This is a result of its focus on “language first”
- I don’t know of any RPC-oriented standards that are based on anything like the trade-off analyses by which REST was derived
- Lack of constraints also often caused by vendors wanting broad specifications that can “standardize” whatever systems they happen to have built

REST For Integration

- REST is currently infiltrating the enterprise as a better integration technology
  - true language independence
  - proven interoperability
  - reduces need for costly specialized middleware
  - can instead be implemented with free web servers, caches, etc. whose trade-offs are well known and documented on the web
  - reduced coupling across systems
- Not surprisingly, with REST's growing popularity there are some who are trying to hide REST behind language frameworks and models, ala RPC (insert forehead whack here)

Language Evolution

- *"Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more."*

- *"Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak...their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

John Backus
1977 ACM Turing Award Lecture

Example 2: Erlang

- What if you thought about the hard problems of reliable distributed systems:
  - partial failure
  - concurrent operation
  - latency
  - scalability
  - failover
  - fault tolerance
- and then designed a language to deal directly with these issues?
- Erlang is a practical language designed and built specifically to enable reliable long-running distributed systems

They Come for the Concurrency...

- What often attracts developers to Erlang is its concurrency support
  - my Macbook Pro can start and stop one million Erlang processes in 0.5 sec
  - writing concurrent programs is vastly simpler than in Java, C++, etc. due to no need to deal with error-prone concurrency primitives
- *"What if the OOP parts of other languages (Java, C++, Ruby, etc.) had the same behavior as their concurrency support? What if you were limited to only creating 500 objects total for an application because any more would make the app unstable and almost certainly crash it in hard-to-debug ways? What if these objects behaved differently on different platforms?"*

Joe Armstrong

…But They Stay for the Reliability

- Erlang's concurrency directly supports its strong reliability
- Inexpensive processes enable
  - no sharing (which greatly enhances reliability and scalability)
  - cheap recovery (if something goes wrong, let it crash, start a new one)
  - true multiprocessing (easily map processes to different cores/hosts)
- Inexpensive processes require
  - isolation, which means they communicate only via messaging
  - distribution (you need at least 2 computers for a reliable system)
  - monitoring and supervision (so one process can detect when another one fails)

Real-World Examples

- REST and Erlang are merely two examples of approaches that succeed by treating distribution as a first-class problem instead of trying to hide it
- This isn't a buzzword bandwagon — I changed my whole career so I could use these approaches
- They also happen to represent the different ways of thinking required for the next decade of large-scale distributed systems running on many-core hosts
    - do you know, or are you currently learning, a functional programming language such as Erlang or Clojure?
    - have you read Fielding's thesis, and do you understand the trade-offs REST makes and why?
    - if the answer to either is no, I strongly advise you to fix that

Summary

- RPC is a convenient but flawed accident of history
  - 1980s research focused on monoliths of programming languages, distributed applications, and operating systems
  - each computer vendor of the time owned their own full stack, from language to hardware and network, and you used what they gave you
  - imperative languages won back then simply because of their superior performance at that time
- It's almost 2010, folks — we can do WAY better
  - pull your head from the imperative language sand and learn functional programming
  - the world is many-core and highly distributed, and the old ways aren't going to keep working much longer

You are never too far down the wrong path to turn around.

—Scottish Proverb

No matter how far you have gone on the wrong road, turn back.

—Turkish Proverb

We will stay the course.

—George W. Bush