From Middleware Implementor to Middleware User

(There and Back Again)

Steve Vinoski
Member of Technical Staff
Verivue, Inc.
Westford, MA USA

Middleware 2009

4 December 2009

10 Years of Middleware!

"...the 10th International Middleware Conference will be **the premier event for middleware research and technology** in 2009."

- 1998: Lake District, UK
- 2000: Palisades, NY
- 2001: Heidelberg
- 2003: Rio de Janeiro
- 2004: Toronto
- 2005: Grenoble
- 2006: Melbourne
- 2007: Long Beach, CA
- 2008: Leuven, Belgium
- 2009: Urbana-Champaign, IL

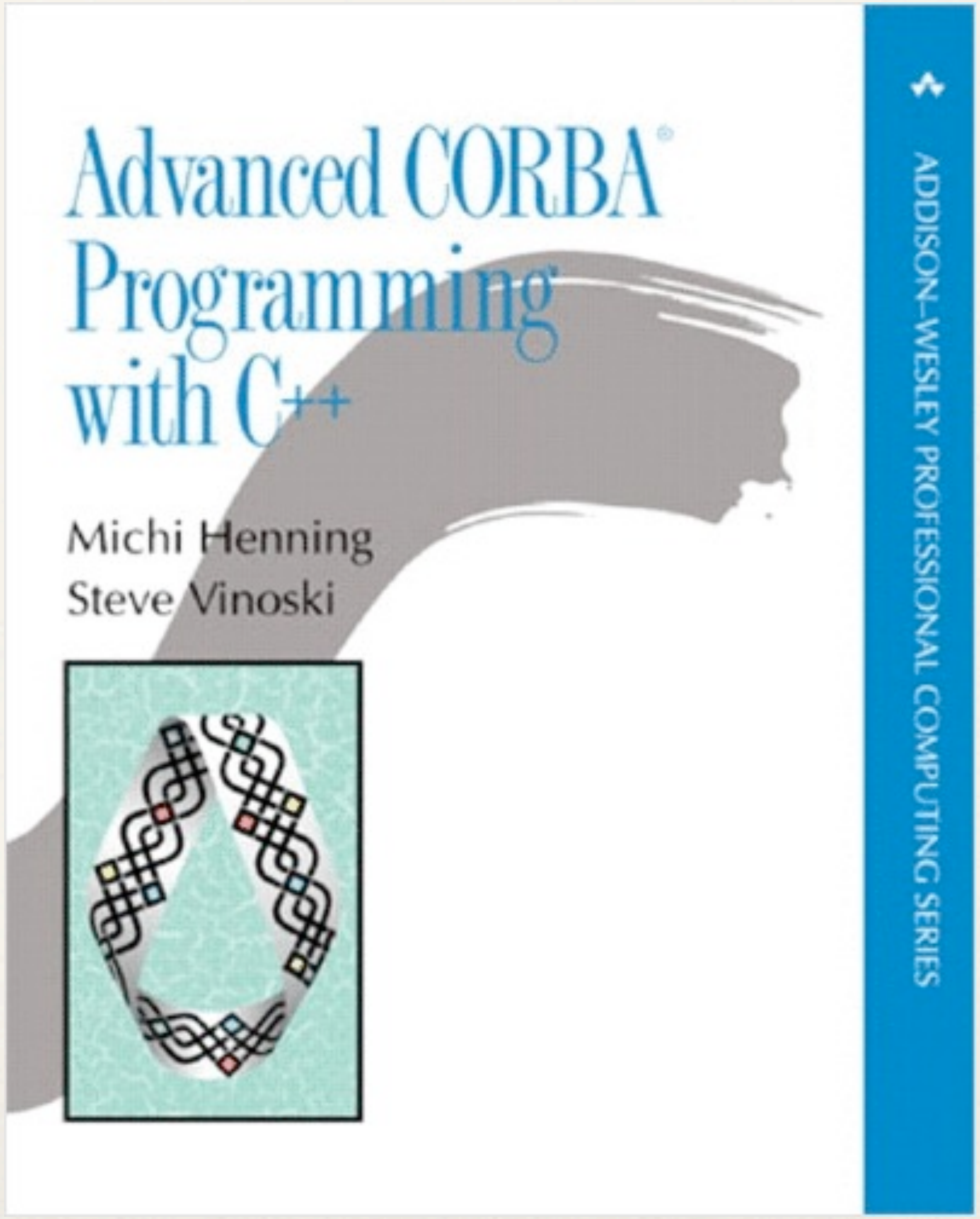
Why A Middleware Conference?

- Prior to the creation of the Middleware Conference, there was no clear forum for the topic. Previously, middleware papers were typically published at
  - programming language conferences, or
  - conferences focusing on specific distributed systems techniques, e.g. objects
  - other middleware “conferences” were marketing- or vendor-focused and so lacked the submission evaluation rigor necessary for quality control
- The 10 Middleware conferences have successfully provided a venue for:
  - the publication and presentation of high-quality middleware R&D
  - the dissemination and intermixing of ideas from multiple middleware camps

Back When the Middleware Conference Started...

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

John Maynard Keynes

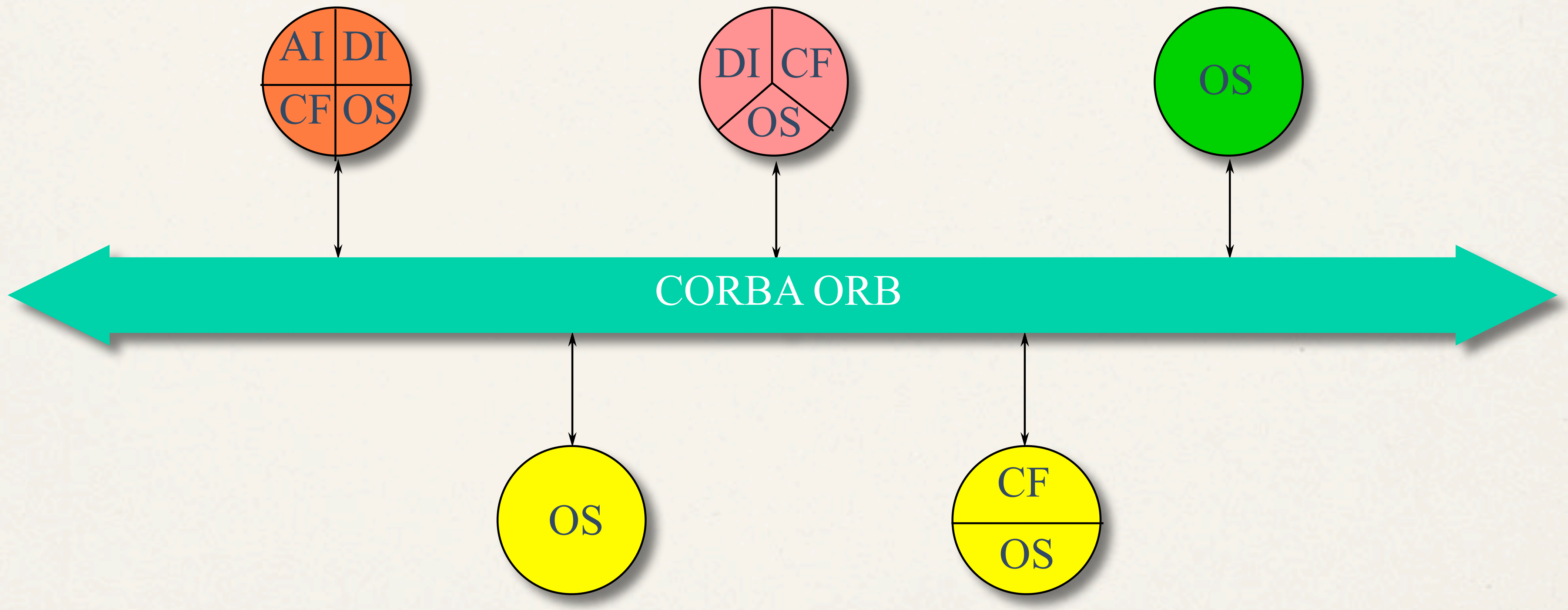

Advanced CORBA® Programming with C++
Michi Henning
Steve Vinoski
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

What Changed?

- Earlier this decade I started to question the fundamentals of CORBA and its descendants
  - Partly due to some internal integration projects I worked on for my previous employer
  - Partly because of encountering other approaches that opened my eyes to different, better ways
- I left the middleware industry in early 2007 for something different, which I'll talk about later
- But first I want to cover some of my thinking that led to the change

Idealized Enterprise Architecture

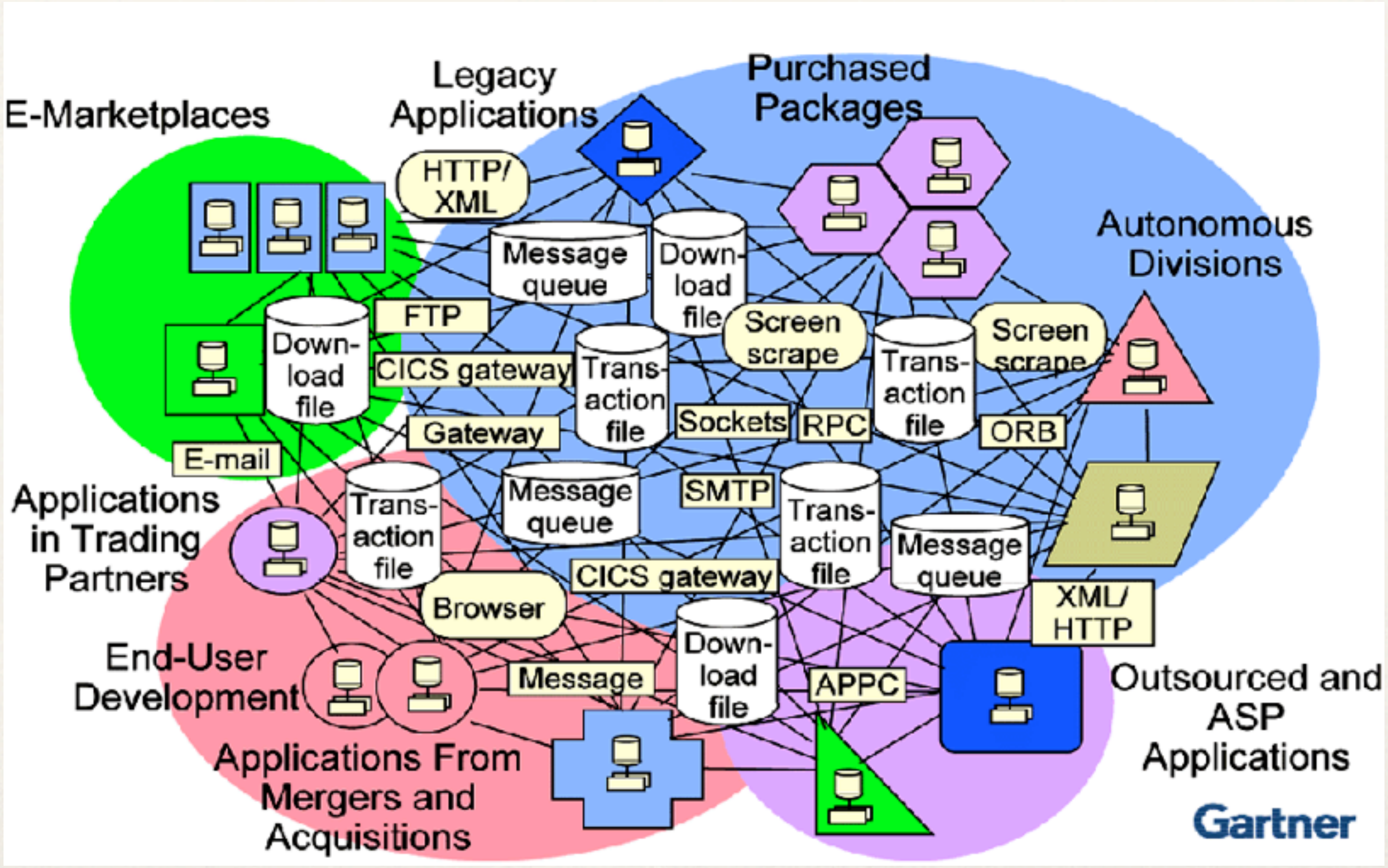
Example: Object Management Architecture (OMA)
from the Object Management Group (OMG)



AI = Application Interfaces
CF = Common Facilities

DI = Domain Interfaces
OS = Object Services

Enterprise Integration Reality



Why the Difference?

- Integration is both inevitable and inevitably difficult
  - all it requires is achieving agreement between what's being integrated — simple, right? :-)
  - too many integration approaches impose too many assumptions, requirements, or overhead
  - the agreement has to be as simple as possible but no simpler
- It's interesting to examine computing history to see how certain forces pushed some middleware approaches toward fundamentally flawed assumptions, requirements, and trade-offs

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, as opposed to human-to-application protocols like telnet:
  - *"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 programmatically 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."*

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."*

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, top to bottom
  - 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 whatever 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

Converting Research To Practice

- Vendors ultimately 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, the birth of middleware was underway:
  - 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
  - formation of the Object Management Group (OMG)

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 (already) 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

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

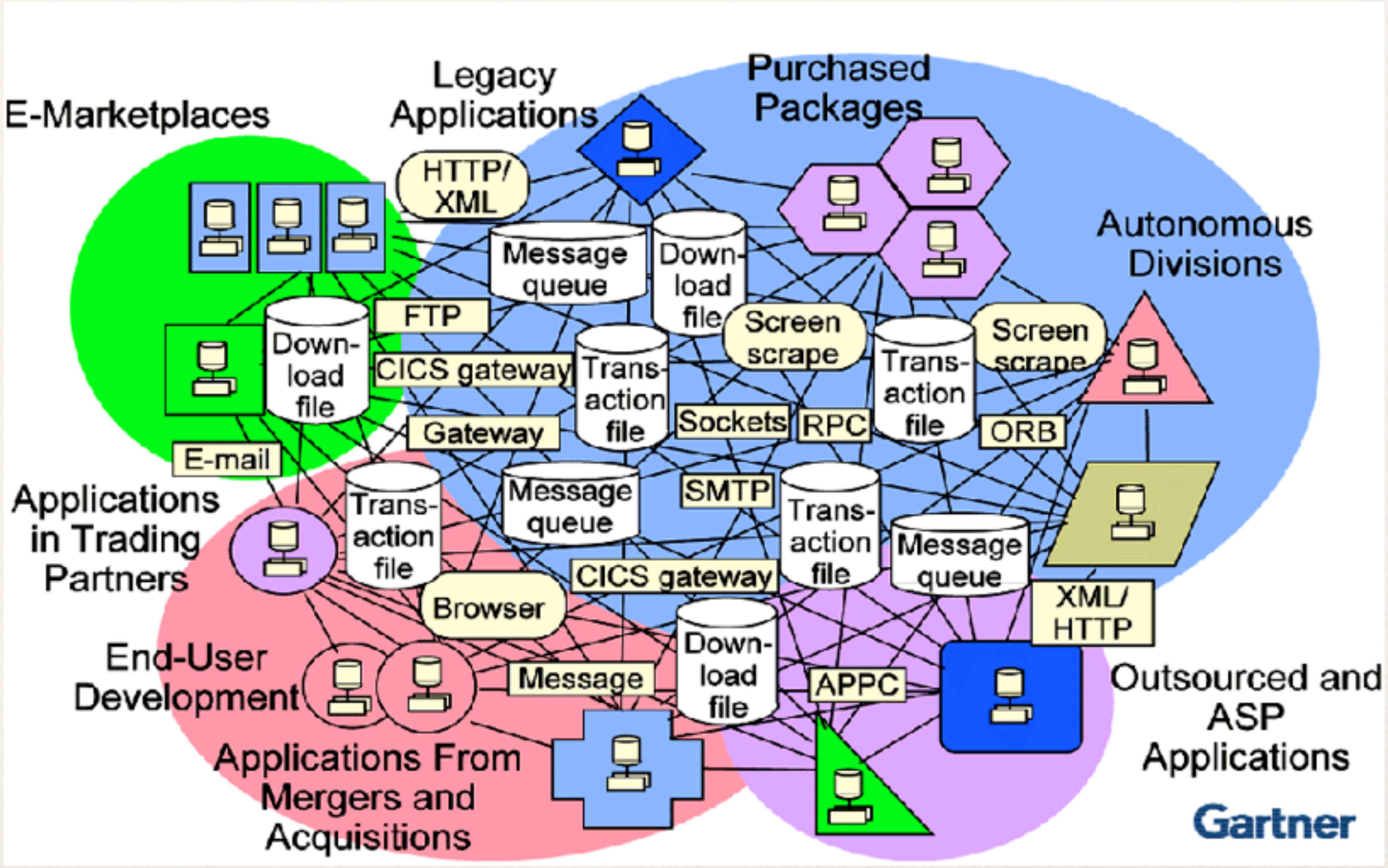
Properties and Constraints

- 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

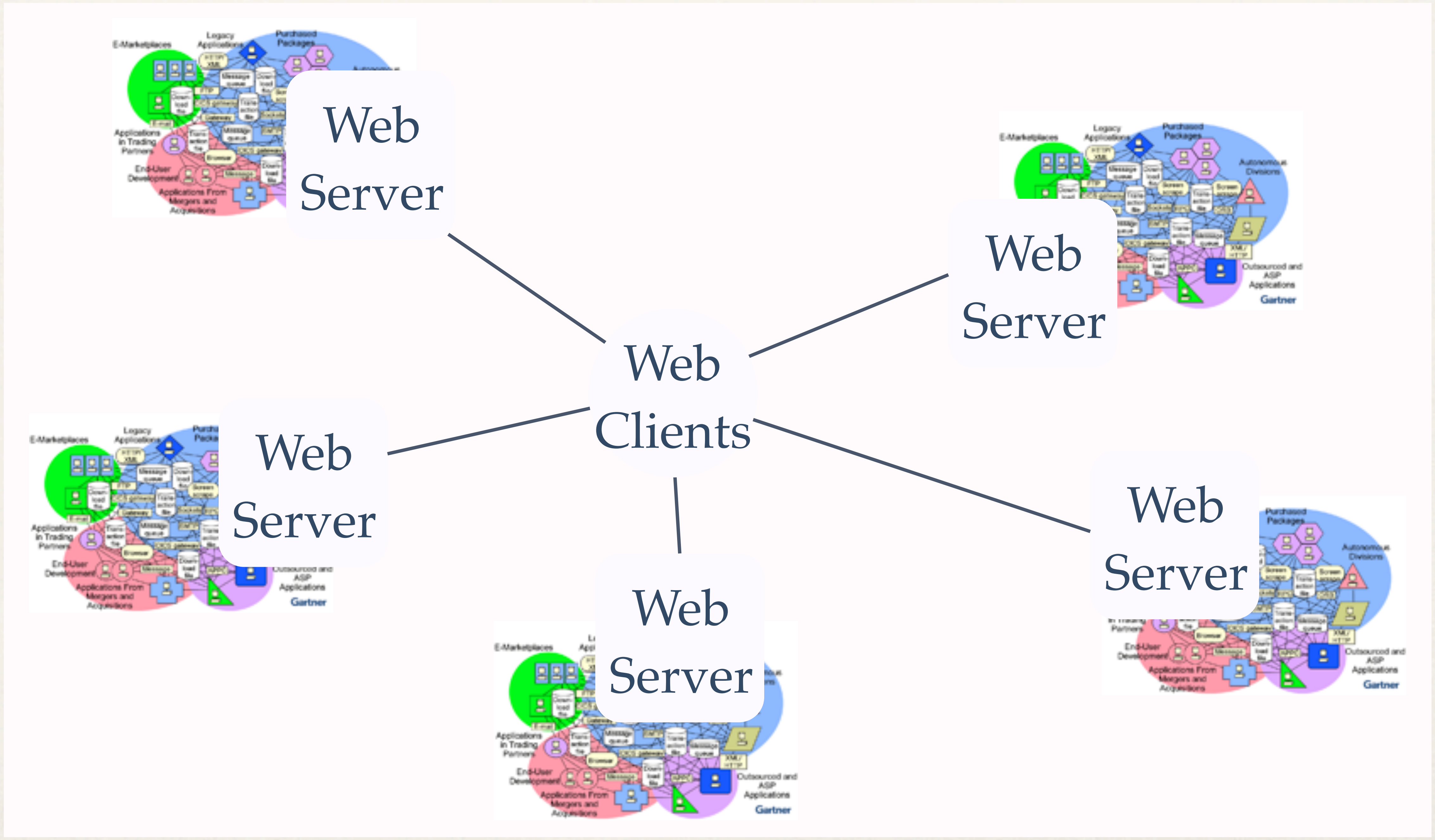
Contrast with RPC Systems

- 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 Integrates Systems Like This...



...Into Integrated Systems of Systems Like This



RESTful HTTP For Integration

- RESTful HTTP has been infiltrating the enterprise as an alternative integration approach
  - 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
- Don’t fear it just because it makes you think differently

Language Evolution

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.”*

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.*

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
  - live upgrades
- 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
  - Erlang makes very effective use of multiple cores due to its scheduler architecture and lightweight processes
- *"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, co-creator of Erlang

…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 two examples of approaches that succeed by treating distribution as a first-class problem rather than 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
  - functional programming languages like Erlang, Haskell, Clojure offer huge improvements for developing correct highly-concurrent systems
  - Fielding's thesis shows how understanding properties and constraints enable us to reason more effectively about the trade-offs in distributed systems

Verivue MDX 9200 Media Distribution Switch

- 20 Gb/s to 200 Gb/s of streaming & delivery capacity
- 2 to 24 TB of Flash memory storage
- Simultaneous HTTP delivery and Video On Demand (VOD) streaming
- Up to 7 Gb/s of independent ingest capacity with no impact on streaming
- Hot-swappable architecture: upgrade, add or replace modules with no downtime
- In-Service Software Upgrades and 99.999% uptime
- SNMP, NETCONF and CLI for flexible management and monitoring

Verivue MDX Middleware Aspects

- Component-based architecture running on multiple multi-core boards
  - for example, there can be up to 10 delivery modules, each multi-core and each running concurrent VOD and HTTP delivery components
- Some components are C++, some are Erlang, integrated by TCP-based point-to-point asynchronous message passing over an internal chassis network — no RPC
- Integrates with 3rd-party vendor systems, for example:
  - some HTTP-based — these are RESTful wherever possible
  - some CORBA-based, e.g. Time Warner Interactive Services Architecture (ISA)

Erlang/OTP Is Simply Amazing

- Given how much time and effort I put into building solid fault-tolerant middleware over my career, I wish I had discovered Erlang/OTP years ago
- It does everything I ever wanted my middleware to do, only better
  - message passing, queuing, failover, replication, monitoring, management, process groups, service discovery, hot code loading, in-service upgrades, pub/sub, migration, introspection, versioning, etc., etc.
- It enables tremendous developer productivity for correctly building highly-concurrent, highly-distributed, highly-reliable systems

Lessons Learned

- Middleware is everywhere. You might think you can leave it but it just finds you again, trust me :-)
    - As long as there are integration challenges to be solved, we will have middleware issues and approaches to work on
- Don’t be afraid to question the status quo. There are always better ways of doing things, you just have to find them
    - Those who don’t know history are doomed to repeat it
- For Middleware 2010 I hope to see multiple submissions based on REST, Erlang (or other functional languages), or both

Thanks