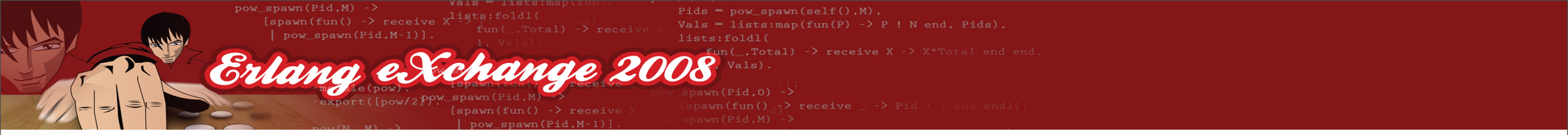


Enterprise Integration
Displacing the Status Quo

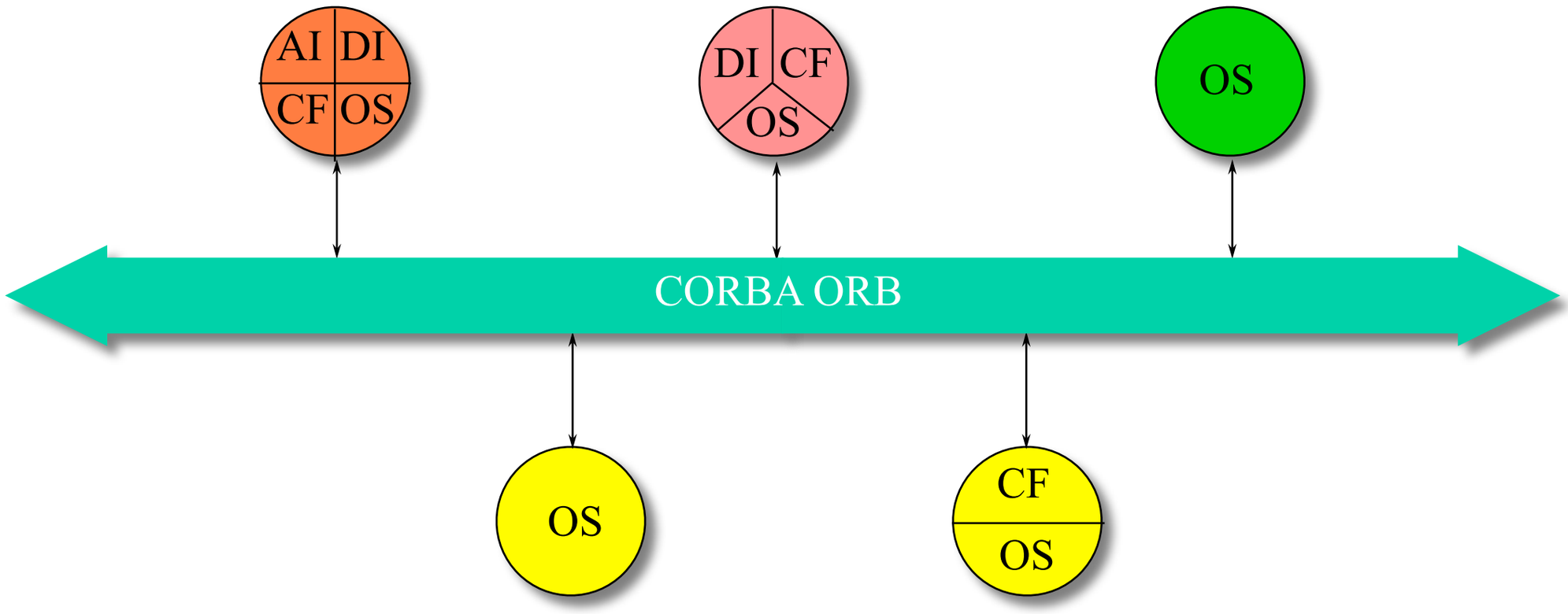
Steve Vinoski

Member of Technical Staff
Verivue
Westford, MA USA
vinoski@ieee.org

Idealized Enterprise Architecture

Example: Object Management Architecture (OMA)

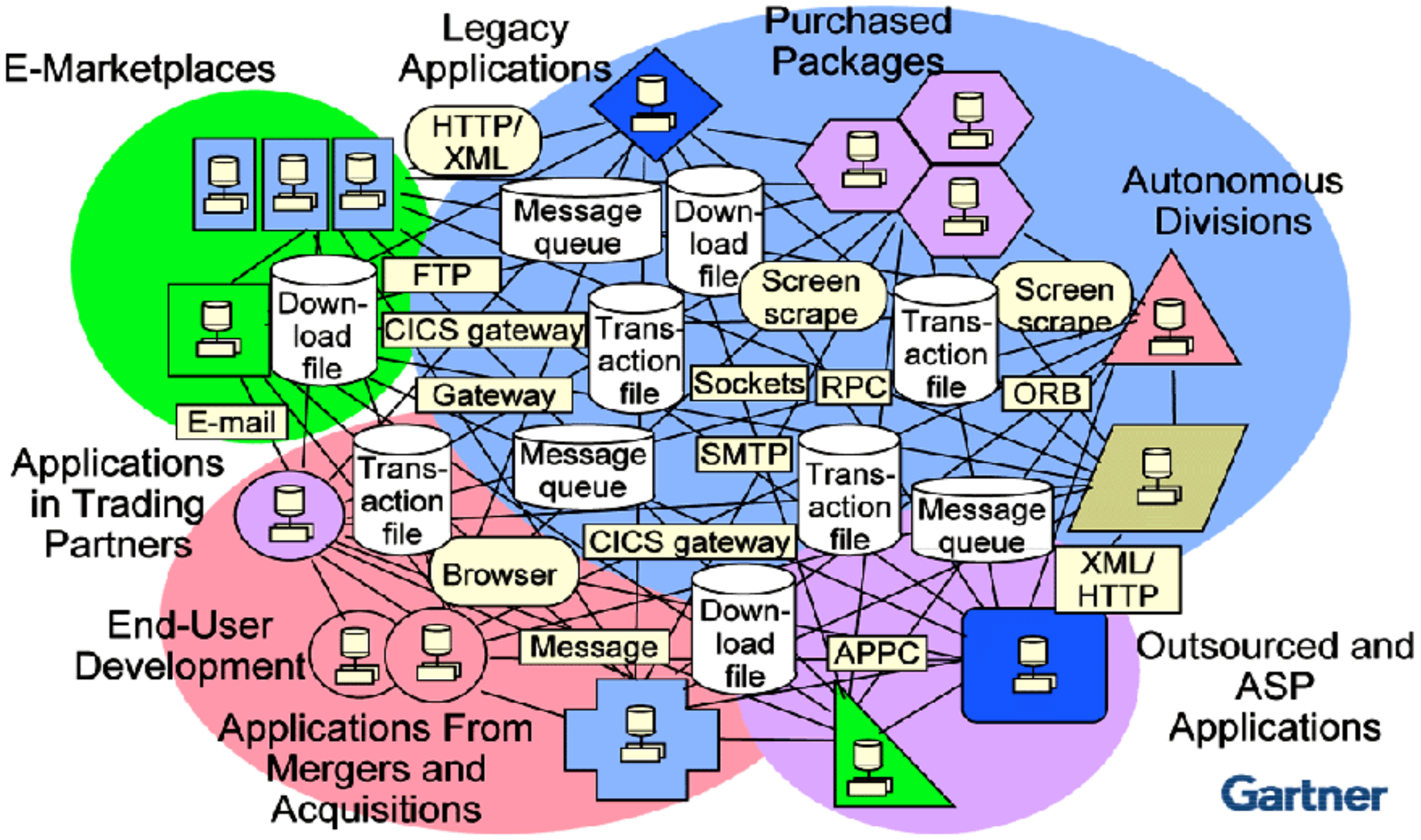
from the Object Management Group (OMG)



AI = Application Interfaces
CF = Common Facilities

DI = Domain Interfaces
OS = Object Services

Enterprise Integration Reality



Compounding the Problem

- Several compelling but basically broken distributed system ideas led some down the wrong path
- Non-technical problematic forces:
  - significant marketing investment in flawed approaches
  - popular technologies attract more research attention, regardless of flaws
- Technical problematic forces:
  - ignorance of fundamental technical issues
  - applying inappropriate abstractions and trade-offs
  - choosing developer convenience over correctness

1976: Remote Procedure Call

- RFC 707: “A High-Level Framework for Network-Based Resource Sharing”, James E. White
  - “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 elevate the task of creating applications protocols to that of defining procedures and their calling sequences.”
- The RFC includes warnings about this model being an inappropriate abstraction

1980s: Languages and Distribution

- Systems evolving: mainframes to minicomputers to workstations to personal computers
  - decentralization leads to increased use of networking
- Structured Programming yields to Object Orientation
- Programming language development: e.g., C++, Eiffel, Objective C, Erlang, Lisp machines
- 1984: Birrell/Nelson paper explains how to implement RPC
- Distributed languages, e.g. Argus, Emerald
  - unite general-purpose programming with distribution

1990s: Distributed Objects

- OOP was viewed as a huge step forward for general software development
  - objects were the answer to everything
- Natural extension of RPC was to treat distributed services as distributed objects
  - objects could even encapsulate the network!
  - general-purpose language objects + distribution
- Major distributed object development efforts began
  - OMG CORBA, IBM DSOM, Microsoft COM
  - *very* significant corporate investment

1990s: Dist Objects, Java

- Distributed object wars: CORBA vs. COM
  - C++ primary language of both systems
  - continued heavy corporate investment
- Java comes along
  - a better C++
  - easier distributed objects — just put Java everywhere, it'll all work great!
- Tremendous investment in marketing Java

Late 1990s: J2EE

- Java application servers encapsulate and abstract CORBA's capabilities
  - hide CORBA's complexity behind the simplicity of Java
  - hide relational data behind object-relational mapping (ORM)
- Enterprise already bought into CORBA and relational DB, so J2EE is a no-brainer
  - J2EE uses CORBA's IIOP and distribution machinery underneath, interoperate with existing CORBA
- Begins the push toward questionable "plain ol' Java objects" (POJOs) as distributed objects

1999-2008: SOAP/WS-*, More RPC

- 1999: “Simple Object Access Protocol” appears
  - distributed objects ala CORBA/COM but with XML/HTTP
  - later renamed to just SOAP (not an acronym)
- 2002: W3C starts Web Services (WS-*) standards
  - many hundreds of pages of specifications, often just “CORBA with angle brackets”
  - often competing specifications from competing vendors
- 2006-2008: they fade as quickly as they started
  - SOAP and WS-* start to fade away, leaving a void
  - leads to “new” RPC: e.g., Service Component Architecture, Facebook Thrift, Cisco Etch

What's Wrong With This Picture?

- CORBA, J2EE, SOAP, WS-*, etc. are object RPC
- Flawed assumptions:
  - synchronous blocking local invocation model works for distributed computing communication
  - local objects can be distributed objects
  - language matters most, distribution is just an afterthought
  - interface definition languages = good language independence
  - developer convenience more important than system correctness
- RPC evolution continues even today, despite warnings
  - RFC 707 warned us from the beginning
  - Waldo et al. "A Note on Distributed Computing", 1994

Investment Prolongs the Pain

What initially look like convenient abstractions (also known as “silver bullets”)...

...leads to significant vendor investment in developing products based on the abstractions, which...

...leads to heavy investment in marketing the resulting products, which...

...often leads to investment in industry consortia and in standardization by both vendor and customer, which...

...leads the customer to push hard on the vendor to make significant product enhancements, which...

Investment Prolongs the Pain

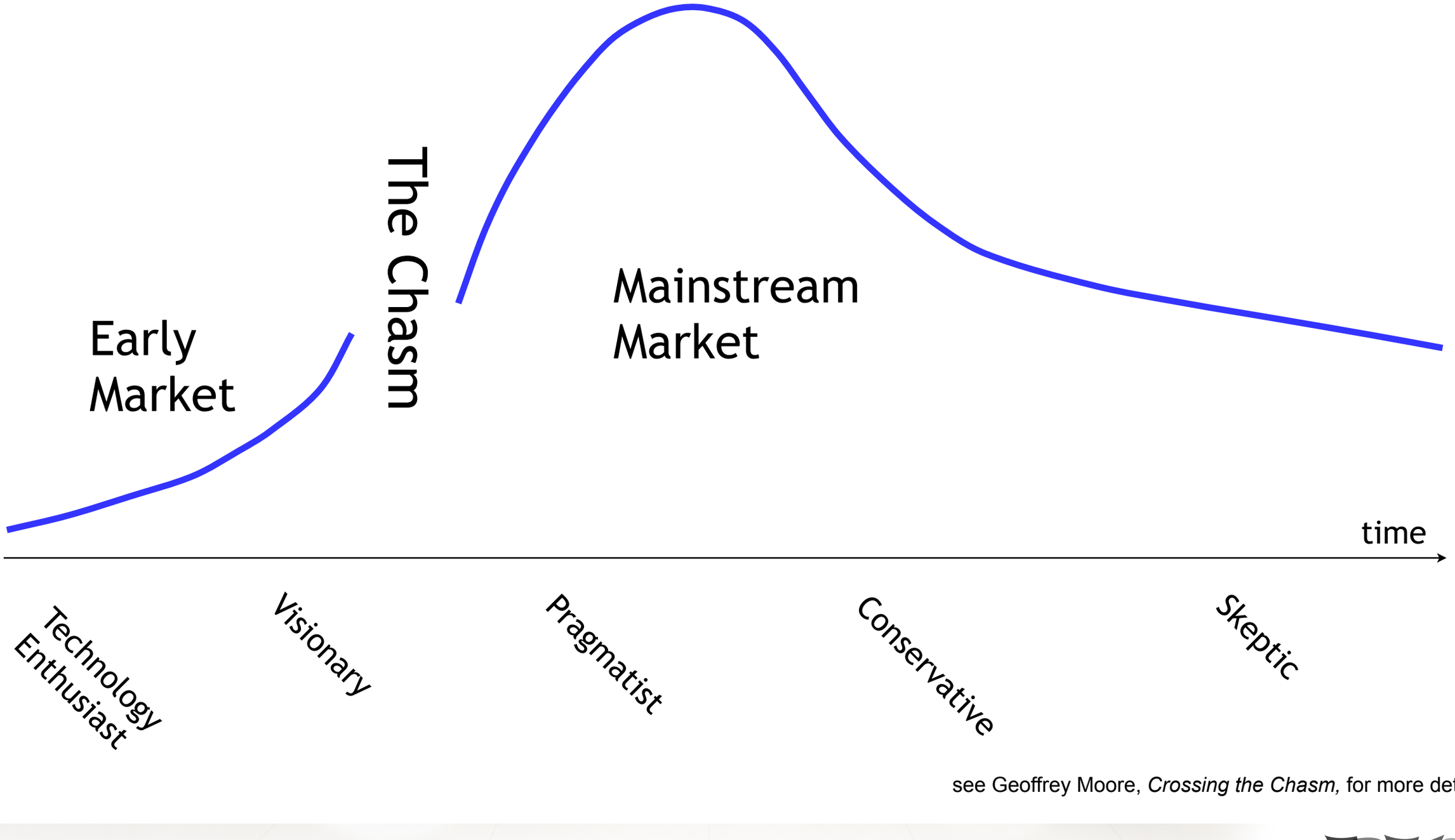
What initially look like
conv
(alse

- Which ultimately means:
- lots of money invested
- nobody wants to break the cycle
- even when the original abstractions are known to be fundamentally flawed

...le
on
pro

which...

Technology Adoption Lifecycle



see Geoffrey Moore, *Crossing the Chasm,* for more details

Disruptive vs. Sustaining Innovations

- Sustaining innovations improve the status quo, disruptive innovations displace the status quo
- Disruptive innovations initially address a low-end market overshot by incumbent technology
  - such innovations considered inferior by incumbent users
  - but “good enough” for early adopters
- Successful disruptive approach eventually moves up-market and displaces the incumbent
- See Clayton Christensen’s *The Innovator’s Dilemma* and follow-on works for the full explanation

RPC: Sustaining Innovations Only

- The entire RPC line of evolution has permitted only sustaining innovations over the years
  - every innovation has simply been an improvement of the same basic approach
  - existing customers have a lot invested in the status quo, and they don't want it to change
- Two primary areas where innovation has appeared:
  - developer convenience — languages, frameworks, products that are easier for developers to use
  - "enterprise quality" — improving systems for redundancy, failover, fault tolerance, performance, etc.

Ripe for Disruption

- The RPC approach is ultimately quite expensive
  - doesn't scale very well, due to specialized interfaces and need for same middleware at sender and receiver
  - code generation leads to brittle, hard-to-version, hard-to-upgrade systems
  - getting "enterprise qualities" right is difficult, ultimately determined by RPC infrastructure regardless of the app
  - developers write loads of code to do simple things
  - far too much accidental complexity
- Is prioritizing developer convenience at this expense really the right trade-off?

Going Up-Market

- Customers demand a steady stream of sustaining innovations from an incumbent technology
- Vendors evolve their systems, i.e. move up-market, to satisfy the most demanding customers
  - that's where the money is
- Customers with simple requirements get left behind
  - the system has evolved way beyond their needs
  - they don't want to pay for what they don't use
- When an incumbent technology moves up-market it leaves a void that disruptive technologies can fill

Representational State Transfer (REST)

- Architectural style of the web, intended for large-scale hypermedia systems
  - makes network effects important, rather than languages and developer convenience
  - puts distributed systems problems like latency and partial failure directly front and center
  - specifies clear trade-offs and constraints that help address those problems
- Contrast with Service-Oriented Architecture (SOA), which specifies no constraints at all
- HTTP is the best known RESTful application protocol, others are possible

RESTful Design with HTTP

- Name your resources with URIs
  - URIs are cheap, use plenty of them
- For each resource, decide:
  - what each HTTP method does and what status codes it returns under what circumstances
  - what media types (MIME types) are supported
  - how each representation of the resource guides the client through its application state (HATEOAS, Hypermedia As The Engine Of Application State)
  - how to handle conditional GET (for caching purposes)

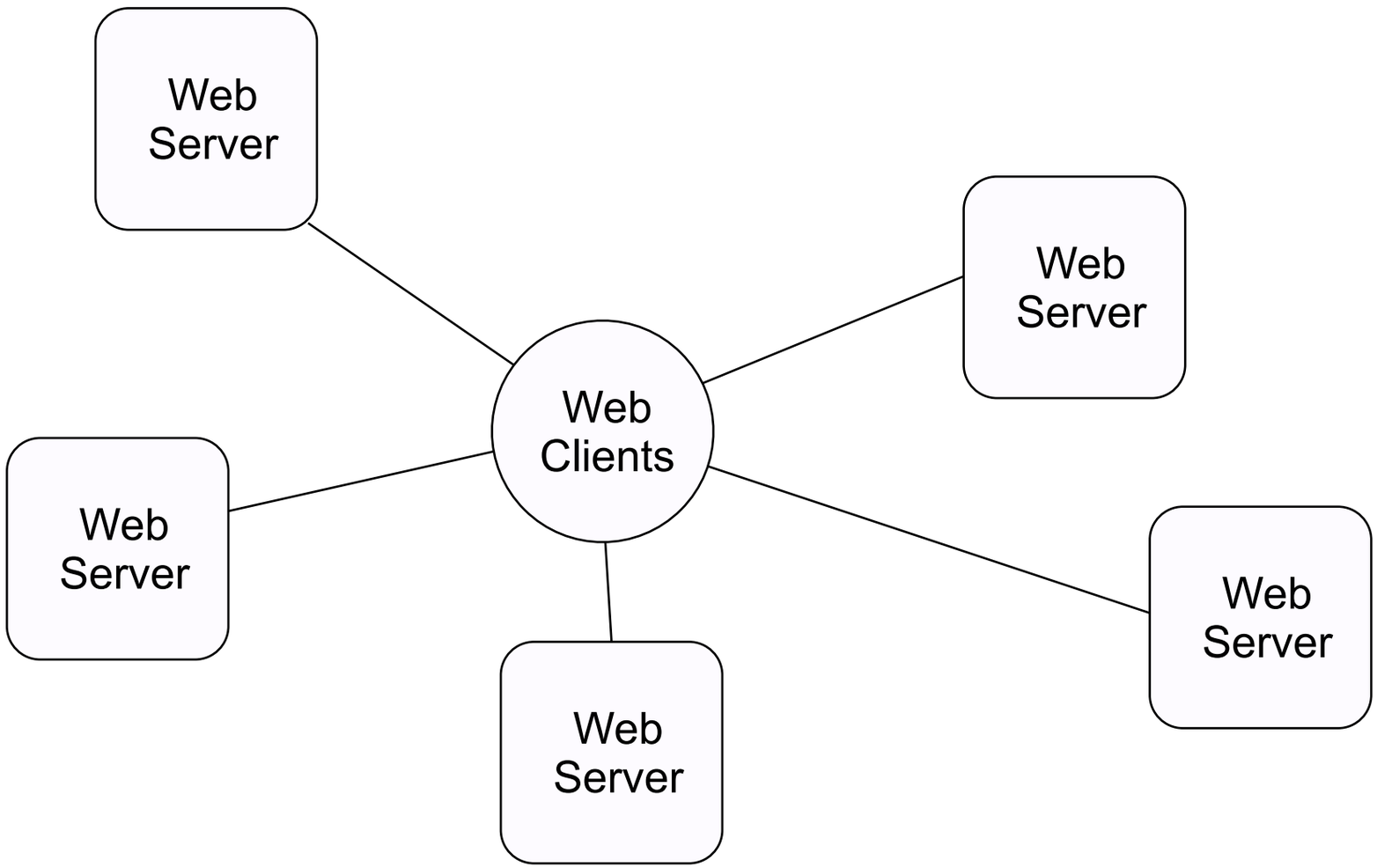
HTTP: RESTful Uniform Interface

HTTP Method	Purpose	Idempotent?
GET	Retrieve resource state representation	Yes (no side effects)
PUT	Provide resource state representation	Yes
POST	Create or extend a resource	No
DELETE	Delete a resource	Yes

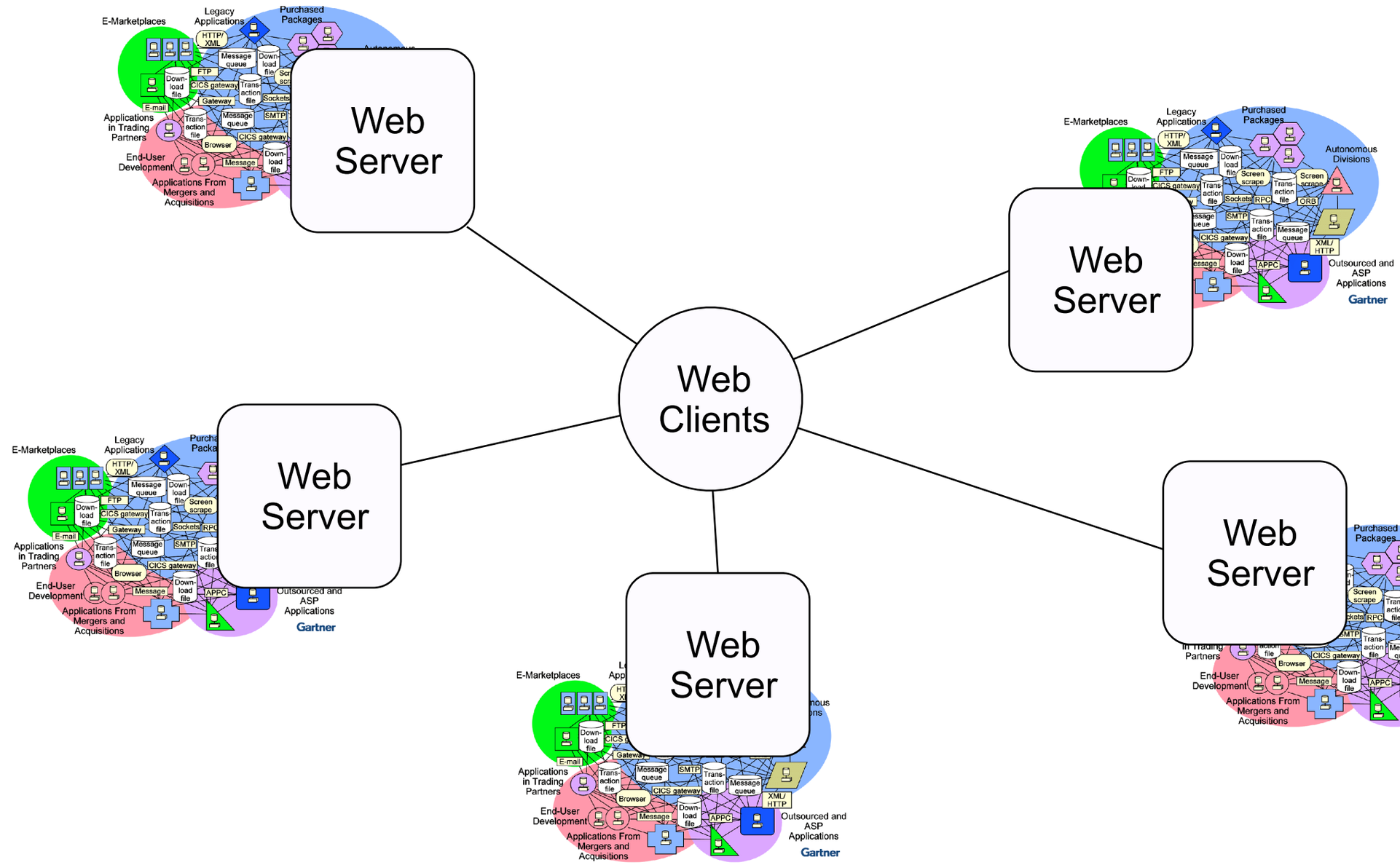
REST/HTTP Reduces Integration Costs

- True language independence
  - allows use of higher-productivity languages
  - it's helping fuel our current language renaissance
- Applicable to wide variety of integration scenarios
  - proven by the World Wide Web, where it's the incumbent integration technology
- Reduces need for costly specialized middleware
  - excellent web servers are free (e.g., Yaws)
  - different web servers can make different trade-offs for different applications, different scalability choices
- Proven interoperability

The Web: Integrated System of Systems

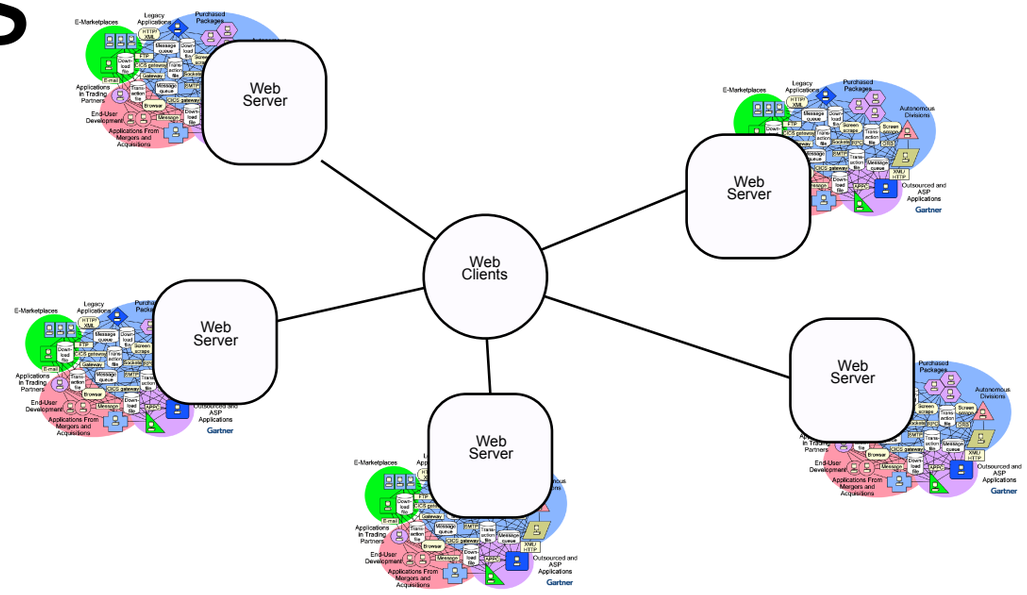


The Web: Integrated System of Systems

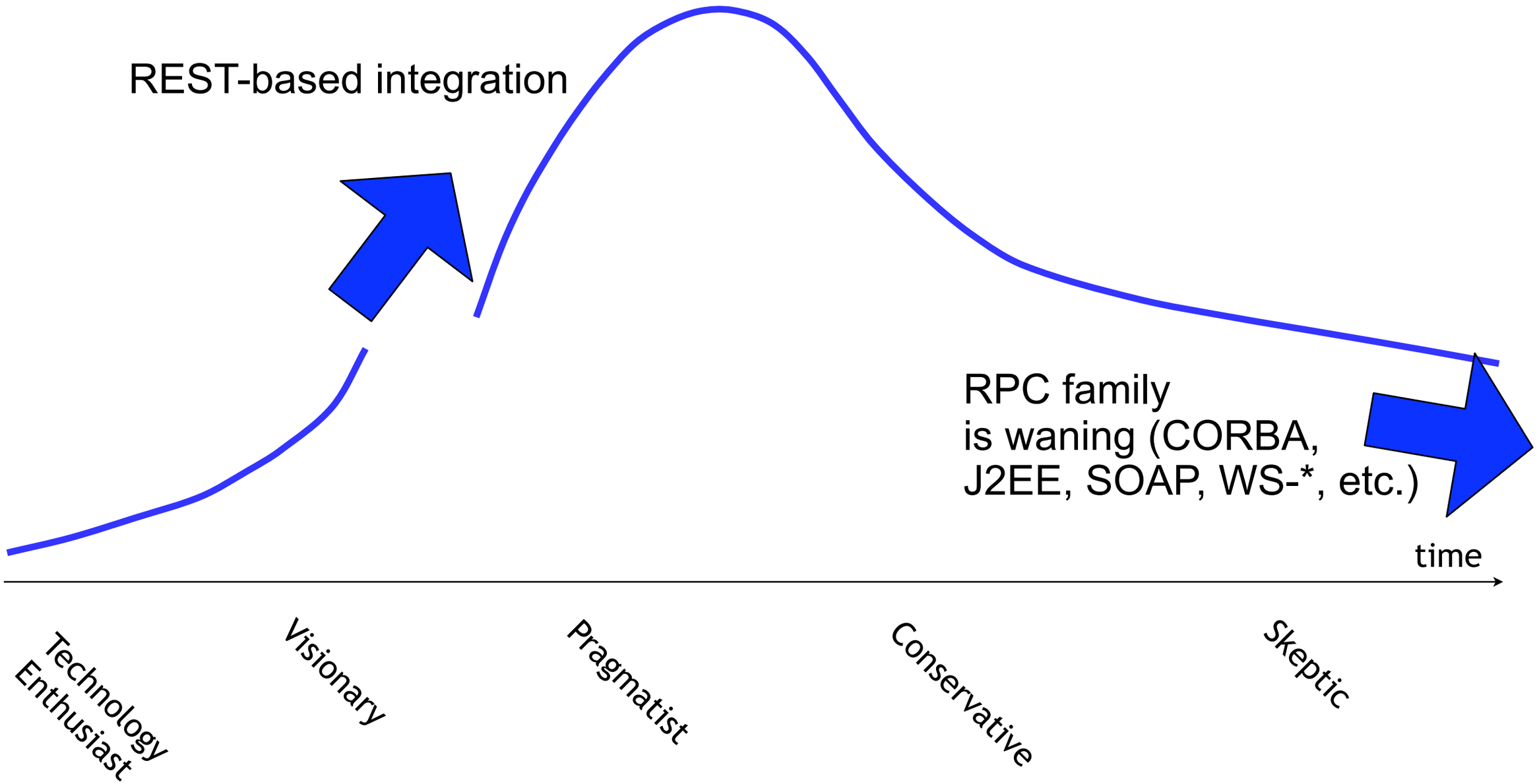


The Web: Integrated System of Systems

- Web clients can talk to RESTful web services, even though they're developed completely independently
  - integration nirvana
- Most enterprise integration projects do not require expensive "enterprisey" qualities
- REST can therefore address the enterprise integration market overshot by the up-market enterprise SOA/RPC systems



REST/HTTP: Disruptive Innovation for Enterprise Integration



Erlang is Disruptive Here Too

- Web servers and Erlang: identical design centers
  - long-running, highly concurrent, highly reliable systems
- Energy costs and global warming require finding better ways to scale than endless racks of machines
- Avoid considerable costs of reinventing high reliability, concurrency, hot upgrades, replication
  - I wasted many years on this for RPC middleware
  - Variation of Greenspun's Tenth Rule: *"Any sufficiently complicated middleware platform in another language contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Erlang/OTP."*

Summary

- RPC-based enterprise integration ripe for disruption
  - RPC is long in the tooth, and wrong to begin with
  - current programming language renaissance driving early adopters to look beyond Java, C++, C#
- REST/HTTP looks like a disruptive enterprise integration alternative
  - REST/web “good enough” for many integration needs
- Erlang/OTP disruptive too
  - proven enterprise quality, at lower development costs
  - works quite well for RESTful systems
  - gets distributed systems right

For More Information

- *Crossing the Chasm*, *Inside the Tornado*, and other books by Geoffrey A. Moore
- *The Innovator's Dilemma*, *The Innovator's Solution*, *Seeing What's Next* by Clayton Christensen
- *RESTful Web Services*, Leonard Richardson and Sam Ruby (O'Reilly)
- *rest-discuss* Yahoo! mailing list
- Numerous articles on http://steve.vinoski.net/
- "RESTful Services with Erlang and Yaws" (http://www.infoq.com/articles/vinoski-erlang-rest)