J Internet Serv Appl (2012) 3:23–30
DOI 10.1007/s13174-011-0042-y

SI: FOME - THE FUTURE OF MIDDLEWARE

# Programming language impact on the development of distributed systems

**Debasish Ghosh · Justin Sheehy · Kresten Krab Thorup · Steve Vinoski**

Received: 2 November 2011 / Accepted: 7 November 2011 / Published online: 19 November 2011


**Abstract** Programming languages have long impacted the development of distributed systems. While much middleware and distributed systems code continues to be developed today using mainstream languages such as Java and C++, several forces have recently combined to drive a renewed interest in other programming languages. The result of these forces has been an increase in the use of programming languages such as Erlang, Scala, Haskell, and Clojure that allow programming at a higher level of abstraction affording better modularity, enhanced speed of development, and added power of reasoning about systems being developed. Such languages can also be used to develop embedded domain specific languages that can expressively and succinctly model issues inherent in distributed systems including concurrency, parallelism, and fault tolerance. In this paper, we first present a history of programming languages and distributed systems, and then explore several alternative languages along with modern systems built using them. We focus on language and application features, how problems of distribution are addressed, concurrency issues, code brevity, extensibility, and maintenance concerns. Finally, we speculate about the possible influences today's alternative programming languages could have on the future of middleware and distributed systems development.

**Keywords** Distributed systems · Middleware · Erlang · Scala · RPC · Java · C++ · Interoperability · Portability · Programming languages · Functional programming

D. Ghosh
Anshin Software Pvt Ltd., Infinity Building, Tower-II, 5th Floor, Plot A-3, Block - GP, Salt Lake, Kolkata 700 091, India
e-mail: dghosh@acm.org

J. Sheehy · S. Vinoski (✉)
Basho Technologies, 485 Massachusetts Ave., Cambridge, MA 02139, USA
e-mail: vinoski@ieee.org

J. Sheehy
e-mail: justin@basho.com

K.K. Thorup
Trifork, Margrethepladsen 4, 8000 Århus C, Denmark
e-mail: krab@trifork.com

## 1 Introduction

Distributed systems and programming languages have long influenced each other. The act of implementing a distributed system using a particular language reveals strengths and weaknesses of that language as they pertain to distributed systems. Likewise, the features and capabilities of many distributed systems have, for better or worse, been influenced by idioms of different programming languages.

In this paper, we examine a variety of distributed systems and middleware developments from the past four decades to examine how programming languages influenced those systems. We then detail the features and capabilities of two highly useful nonmainstream programming languages along with analyzing the benefits of using them for distributed system projects. Finally, we surmise what the future of distributed systems and middleware might be like with respect to programming languages.

## 2 Procedure call influences

An early document revealing the effects programming languages have had on distributed systems was published in

1975: RFC 707 [1]. It addressed the need for distributed application protocols, first by calling attention to the fact that networked resources of that time supported command languages intended for human rather than application interaction. The RFC then raised the concern of whether developers could reasonably write networked applications, given the difficulty of developing and using interprocess communication (IPC) facilities.

To address these concerns, the RFC author, James E. White, proposed the "Procedure Call Model" (PCM) to help developers build networked applications. He reasoned that developers were already familiar with writing applications that invoked libraries of procedures, so the PCM would make calls to networked applications look just like normal procedure calls.

Following on from White's PCM was Nelson's Remote Procedure Call (RPC) [2]. Unfortunately, today the term "RPC" has been incorrectly genericized to refer to the sending of any remote message, but originally it dealt specifically with the transparent invocation of a procedure on a remote system. Transparency was key; within the text of the calling application's implementation language, the call to the remote procedure was intended to look just like the invocation of a regular local procedure.

RPC has had significant influence on distributed systems research and development, but in the 1980s it was not the only influential force.

- By that time, transaction monitors, originally invented and deployed in the 1960s when American Airlines and IBM jointly created the SABRE airline reservation system, had established the value of middleware, code that was neither application nor operating system (OS), but instead sat in between providing clearly useful, and sometimes business-critical, computing capabilities.
- Computer systems were evolving from mainframes to minicomputers to workstations to personal computers. These newer systems, particularly workstations, required network connectivity, and so networking technologies such as token ring systems and Ethernet were also evolving.
- Meanwhile, new programming languages were being invented even as old ones were still maturing and advancing. For example, in the 1980s Lisp, Pascal, C++, Smalltalk, Eiffel, Objective-C, Perl, and Erlang were all either invented or were receiving notable attention.
- Around the beginning of the 1980s, the structured programming methodology was already popular and becoming more so, but interest in object-oriented programming (OOP) was beginning to rise. As the 1980s progressed, researchers and developers realized that objects were a convenient unit for parceling out functionality across a distributed application.

Together these forces had significant influence on 1980s distributed systems research, resulting in important developments in the areas of operating systems, programming languages, and distributed applications. A number of academic systems developed during this period were complete top-to-bottom systems—OS, programming language, distribution middleware, and application programming interface (API), all rolled together—primarily because all of these levels were still the object of intense research and development. There were several notable research systems of the day, including:

- Argus [3], a distributed programming language and system designed to help with reliability issues such as network partitions and node crashes.
- Eden [4], a full object-oriented distributed operating system that made use of RPC.
- Emerald [5], a distributed RPC-based object language providing local/remote transparency and addressing new and difficult problems of the time such as object mobility.

Systems like these tended to rely on RPC, focus on uniformity such as local/remote transparency and strong/static typing across the whole system, and use their own closed distribution protocols hidden within each system as part of its underlying RPC mechanisms.

As business computing networks grew during the 1980s, vendors knew they needed to convert distributed systems research into practical applications. But their customers wanted to use standard programming languages of the time, such as Pascal and C. This pressure led vendors to incorporate distributed systems research into their own stacks, but do so by making distribution features available through standard programming languages. For example, Apollo Computer's Network Computing System (NCS) [6], an RPC system featuring a declarative interface definition language (IDL), and Sun Microsystem's Open Network Computing (ONC) system [7], were both commercial distributed computing systems heavily influenced by 1980s distributed systems research, yet each worked with C. Note that each system focused on RPC; by this time, White's and Nelson's early RPC ideas and work had come to fruition, becoming the accepted norm for distributed application invocation.

## 3 Distributed objects

One of the problems with 1980s distributed systems was that they were closed silos. Portability for applications between these systems was nonexistent, as was system interoperability. 1980s customers and users started to express interest in standardized APIs and protocols to allow for application portability and interoperability.

As the 1990s approached, the growing popularity of OOP led some to see distributed objects as an answer to

the portability problem. In 1989, the Object Management Group (OMG) formed to standardize an object model for distributed applications.

### 3.1 CORBA

In 1991, the OMG published the Common Object Request Broker Architecture (CORBA) specification [8], one of the most successful middleware standards ever created. At the center of CORBA was the concept of the *object request broker* (ORB), an abstract entity for controlling and managing the invocation of distributed object methods. CORBA defined the ORB interface, the OMG IDL, a set of methods common to all CORBA objects, and a number of interfaces local to the ORB itself.

The goals of CORBA included application portability, system interoperability, and the integration of disparate systems. The CORBA approach to application portability was to map the features of OMG IDL and the ORB interfaces to programming languages. The initial CORBA specification included a mapping to the C language, but few used it since C is not an object language. Instead, users wanted C++.

CORBA and C++ evolved together, and C++ had crucial impact on CORBA. Numerous ORB implementations were written in C++, and most initial CORBA users wrote their applications in C++. Through much of the 1990s, a number of the authors of various C++ ORB implementations cooperated within the OMG to evolve the CORBA standard. Their knowledge and experience with C++ helped guide CORBA through its formative years. This meant, though, that the CORBA standard would continue to be true to its RPC roots, even though it was well understood by that time that the RPC approach was fundamentally flawed [9].

### 3.2 Objective-C

Objective-C, another C-derived OOP language, is a dynamic language adding Smalltalk-inspired features on top of C. It also provides an incremental type system, class/interface separation, dynamic code loading, and other features allowing rapid application development. Objective-C was developed in the early 1980s [10], and popularized in the NeXTSTEP operating system [11]. NeXT was acquired by Apple in 1996; Objective-C and NeXTSTEP, now iOS, are still in widespread use today.

The dynamic nature of Objective-C made it quite easy to extend the core frameworks with a distributed object system built on a simple proxy mechanism. The language was amended with annotations for method declarations such as `oneway`, `bycopy`, `in`, `out`, etc., all made accessible from Objective-C's runtime reflection and thus available as means to control these aspects of object-based RPC. Given this approach to implementing distributed objects, there was no need for a separate IDL.

## 4 The rise of Java

In the mid to late 1990s, Java [12] became very popular. One of the distinctive features of the language was that the Java platform came with means to move code between machines, and securely execute code in a "sandbox," allowing execution of untrusted code in a secure way. Many other systems for allowing remote execution of code were based on a "domain specific language" for specific actions (such as SQL), but Java allowed distributing code in a way that blended more easily with the rest of the programming environment.

Another prominent feature of the standard Java libraries was an implementation of distributed objects called *Remote Method Invocation* (RMI). Like Objective-C, RMI defines its own transport level protocol for remote method calls, and thus did not aim for inter-language interoperability. RMI is also based on having proxy objects represent remotely available objects, and in a fashion similar to CORBA, these proxies would be generated by a separate RMI-enabled compiler (`rmic`). Later, Java introduced dynamic RMI, in which compile-time code generation was no longer necessary due to the use of Java interfaces to describe remote objects and runtime reflection to control the underlying RPC mechanism.

### 4.1 CORBA in Java

As Java grew to become the language of choice for many enterprises, the need for further integration technologies rose. Several groups developed CORBA bindings for Java as a means to provide interoperability and integration with other software systems. Some of the mappings even worked by reverse-mapping Java to CORBA. With the advent of the Java 2 Enterprise Edition (J2EE) platform, CORBA and its Internet Inter-ORB Protocol (IIOP) became part of the requirement set, and a multitude of vendors implemented CORBA/IIOP, including a new standard for using it as a transport for Java's RMI APIs [13].

Sadly, many of these IIOP implementations were incompatible; the effort did not reach the general level of quality needed to make this interoperability story popular. Messaging based on self-describing data transports (XML, JSON, etc.) has since grown in popularity in part because of the transparency, debuggability, and increased possibilities for coping with evolving APIs.

## 5 Current systems

Today, Java continues to be the platform of choice for middleware and distributed systems development. Targeting the Java Virtual Machine (JVM) allows developers to write

portable code, which is important in heterogeneous environments. Java is largely considered easier than C++, allowing more developers to use it effectively.

One telling sign of the influence of Java is that for better or worse, today's developers use Java artifacts to define remote interfaces and data interchange formats, and then simply push a button in their integrated development environment (IDE) to automatically code-generate middleware infrastructure. Contrast this with past approaches, such as with CORBA, where the system was defined in IDL from which tools generated glue code. Rather than being "just another language," Java today is often the complete basis for the distributed system architecture, design, and implementation.

## 6 Alternative languages

Numerous developers have written working distributed systems in Java and C++, but that does not mean such languages are ideal for the task. One oft-cited problem with distributed systems and middleware defined in these languages is the monumental size of the code required to express the features of the system. As systems grow in scale and features, code size becomes critical to being able to understand, maintain, and enhance them; should the code grow too large, it cripples developers, who can no longer keep up with it. This can lead to systems that wither prematurely and must be replaced, often at immense cost due to delays and missed opportunities.

Since general-purpose language models do not work well as distributed system models, a better approach is to employ a language whose model is informed by the problems of distribution, including concurrency issues, partial failure, messaging, and the need for fault tolerance. Because such languages better fit the problem domain, developers can use them to create solutions that are much more elegant, brief, readable, maintainable, and extensible than what can be achieved with general languages.

### 6.1 Erlang

Erlang is a dynamically typed and garbage-collected language with a virtual machine designed for soft real-time behavior. Though it is sometimes placed in a category of new alternative languages, it is actually about half a decade older than Java. Erlang was originally conceived as part of an experiment in the Ericsson Computer Science Laboratory to find a better way to quickly and easily develop clear, easy-to-read programs that would also be highly robust.

Erlang is sometimes referred to as a "concurrent functional language." It inherits fundamental qualities from a computer science heritage of concurrency (a process and communication model informed by Modula and Ada) and functional/logic programming (as in languages like Haskell, ML, Miranda, and Prolog), and fuses those concepts in the interest of rapid development of highly available and reliable systems.

Central to Erlang's concurrency model is the idea of processes as a language primitive. Processes in Erlang systems run concurrently in separate memory, communicating with each other by message passing. Processes are useful for a wealth of applications, including gateways to databases, handlers for protocol stacks, and managing the logging of trace messages from other processes. A process communicates with others in the same fashion regardless of whether those other processes are in the same VM, in another VM on the same host, or in a VM on another host across a network. A single VM can simultaneously run millions of processes, and so it is common for applications to use a process for each fine-grained subtask that might possibly be concurrent with other work. This architecture also allows Erlang applications to easily take full advantage of multicore systems.

Erlang variables are single assignment; once bound to values, they cannot be updated. This immutability, while inspired by functional and logic programming models, takes on new value in a concurrency-focused language by allowing for easier reasoning about distributed behavior.

The language makes extensive use of other functional programming (FP) idioms as well. One example is pattern matching, as shown in the example below[1]:

```
-module(factorial).
-export([fac/1]).
fac(0) -> 1;
fac(N) when N > 0 ->
   Prev = fac(N-1),
   N*Prev.
```

Note that calling the `fac` function with a negative number will result in a runtime error, as no clauses match. Not handling this case is an example of nondefensive programming, a practice encouraged in Erlang. Instead of enumerating and catching the various exceptions that might occur, Erlang programmers adopt a "let it crash" philosophy [14]. Essential to that philosophy is Erlang's supervision model, in which some processes in a system are supervisors with the sole role of starting and monitoring other processes. If a process throws a runtime error, it simply dies. Its supervisor will see the error that killed it, and may use a predetermined strategy to decide if and how to restart it.

Another example of the FP idioms to which Erlang has given new power in a concurrent context is the combination of pattern matching with message primitives to create *selective receive*, shown in this simple example:

[1] Our factorial function is intentionally not tail-recursive to keep it as simple as possible.

```
receive
   {ok, N} ->
      N+1;
   {error, _} ->
      0
end
```

This attempts to receive and process the next incoming message to this process that is a 2-tuple beginning with an atom, specifically either `ok` or `error`. If no matching message is in the mailbox, the expression will wait until such a message arrives. The order of clauses matters, so an `ok` message will be processed in preference to an `error` message. The pattern can bind variables, so the `N` in the expression will be bound to the second term in that message. This technique allows for a more declarative and understandable style of programming concurrent and communicating systems.

These examples demonstrate that Erlang has been quite successful at something unusual: taking language design elements often thought of as "academic" and using them in a very practical industrial fashion. One last such example would be proper tail calls, sometimes mistakenly described as "tail call optimization." In some contexts this language feature is seen as an FP trick that is not really needed by programmers, but Erlang's use of tail calls shows their real usefulness. Erlang provides powerful control flow abstractions such as networked finite state machines that simply need their per-state and per-event behavior to be dropped in by a given application. The modular control flow that this allows is made possible by a tail-call-driven structure for programming the state machines.

One of Erlang's most unusual features, but one that is a core element of building highly-available systems, is *hot code loading*, the ability to upgrade a running application without pausing it. Most of an Erlang system's control flow is implemented via tail calls; a process running an infinite tail call loop can simply call the new version of the hot-loaded code upon reaching the next loop iteration.

What might be most surprising to mainstream language developers is that Erlang provides all these benefits to distributed systems builders while remaining a very small and simple language with few concepts to learn. It is worth remembering that Erlang was not designed just for highly-robust concurrent systems, but also for rapid development of such systems, and "beautiful" code was an explicit goal.

In addition to Ericsson's extensive success delivering products using the language, many other companies have also built high performance and extremely reliable concurrent systems atop Erlang and its standard libraries. A few representative examples are:

- Bluetail/Alteon/Nortel (distributed fault-tolerant email system, SSL accelerator)
- Basho Technologies (distributed fault-tolerant database system)
- Facebook (Facebook chat backend)
- Klarna (electronic payment system
- RabbitMQ (AMQP enterprise messaging)
- Motivity (SS7/ISDN protocol converter)
- Verivue (carrier-grade media delivery hardware)

### 6.2 Scala

Scala is a statically typed language that runs on the JVM. Scala has seamless interoperability with Java and allows reuse of the myriad of Java libraries. Syntactically, Scala is concise enough to build APIs with smaller surface area than Java, which is due to the Scala programming model offering a higher level of abstraction both as part of the core language and the standard library.

Scala is an object-functional language. It offers a powerful OO model along with functional features, allowing developers to use OO constructs like classes, objects and traits together with FP artifacts like closures, as well as pattern matching and algebraic data types. As a general-purpose language, Scala offers features that enable a more expressive programming model for distributed computing. The Scala standard library includes an implementation of the Actor model [15] that uses all the expressiveness that the language offers to implement asynchronous message-based computations. As shown in the rest of this section, various models of distributed computation can make use of expressive snippets of code made possible by the combination of language features and the Scala standard library.

Scala traits allow us to compose smaller abstractions incrementally to evolve larger ones, encouraging code reuse during implementation. Here is how the standard library defines the abstraction for an `Actor` as a composition of multiple traits:

```
trait Actor extends AbstractActor
  with ReplyReactor with ActorCanReply
  with InputChannel[Any]
  with Serializable { //..
```

All the components that `Actor` composes with are independent abstractions that can also be reused in other contexts. This leads to better code reuse across system implementations.

With the proliferation of multicore processors, applications generally perform better when they exhibit some degree of parallelism. Asynchronous messaging is one of the effective ways in which developers can scale out a distributed computation model. Java has a rich set of libraries built on thread-based concurrency, and Scala offers a library on top of this that implements the actor-based concurrency model of Erlang. Notable here is the expressive syntax that Scala offers through its ability to define custom control structures. In the snippet below, `actor` is a method that takes a higher-order function as an argument and implements the message processing loop:

```
actor {
  var sum = 0
  loop {
    receive {
      case Data(bytes)=>
      sum += hash(bytes)
      case GetSum(requester)=>
      requester ! sum
    }
  }
}
```

In the above snippet, `requester ! sum` is a message send much like in Erlang. Scala allows operators as methods—here `!` is the name of a method on class `Actor` (`requester` in this case). The brief version of the message send is a simple method invocation on the instance of an Actor (`requester.!(sum)`).

Here is an example that uses the functional power of Scala to compose APIs to implement pipeline-based processing of receive handlers in an actor.

```
trait GenericServer extends Actor {
  //..
  def act = loop {receive
   {genericHandler andThen
    specialHandler}
  }
  //..
}
```

In the above snippet, the receive loop of the actor runs the `genericHandler` first, and subsequently runs the `specialHandler`.

Besides all the syntactic features and library-based support of concurrency through actors, two of the best features of Scala that make it great for developing distributed applications are its support for immutability and its rich static type system. Immutable objects make concurrency easier, allowing developers to more easily reason about their code and test it. Scala's rich type system helps enforce constraints at compile time, thereby eliminating a class of errors during runtime. This makes distributed applications easier to test since there are fewer test cases to write, manage, and maintain.

Scala is emerging as a popular choice of developing distributed computing frameworks. The following section discusses one of them, Akka, in detail. Here are some other popular Scala frameworks being used in industrial applications today.

- Spark [16] is a cluster computing framework designed primarily to support parallel machine learning jobs. Spark is based on Scala's actor model implementing parallel loops over distributed data sets implementing a scatter-gather algorithm.
- Finagle [17] is a Scala library that helps implement asynchronous messaging clients and servers in Java, Scala, or any JVM language. Finagle is built on top of Netty [18] and provides protocol-independent abstractions for developing distributed components like streaming, pipelining, and request-response.

#### *6.2.1 Akka—distributed computing on the JVM*

Akka [19] is a platform developed in Scala for building scalable event-driven fault-tolerant systems on the JVM. Akka comes as a collection of loosely coupled components primarily based on the actor model. Akka actors are implemented on top of *dispatchers*, which allow users to configure parameters for optimal performance and scalability. Dispatchers manage thread pools backed by a blocking queue and implement lightweight event-driven threads that can scale up to millions, even on commodity hardware. Both actors and dispatchers can be configured declaratively, thanks to the expressive syntax of Scala.

Akka offers many ways to build and start an actor. Scala's rich typesystem, powerful object syntax and composability of higher-order functions make actor definitions and lifecycle management expressive yet succinct, as shown here:

```
// actor definition
class MyActor extends Actor {
  def receive = { //..
  }
}

// actor instantiation
val actor = actorOf[MyActor]
actor.start()

// more concise - instantiate & start
val actor = actorOf[MyActor].start()

// actors with non-default constructors
val a = actorOf(new MyActor(..)).start()

// implicit actor based execution
spawn {
  // do stuff
}
```

Akka actors interact using asynchronous message passing. Scala's infix operator notation, type inference capabilities and pattern matching on algebraic data types help build nice abstractions (like Finite State Machines) based on message processing. Here are a few ways to do asynchronous message passing with Akka actors:

```
// fire and forget
actor ! "hello world"
```

```
// get a future
val future = actor ? "hello world"

// send and receive eventually
(actor ? msg).as[String] match {
  case Some(answer) => ...
  case None         => ...
}
```

Akka offers a fault-tolerant platform in line with Erlang's "let it crash" philosophy and implements supervisor hierarchies (like the Erlang Open Telecom Platform (OTP) framework [20]). Supervisors are responsible for starting, stopping, and monitoring child actors linked to them using the `link()` method. Supervisors can be configured declaratively by providing pluggable restart strategies and lifecycles for the supervised actors.

Callback-oriented event-based programming is not very intuitive because of nonlinear control flow in the programming model. But this is not the case for delimited continuations [21], a powerful control flow abstraction that lets developers program in a direct style even with inversion of control. Scala supports delimited continuation as a compiler plugin, and Akka uses this to implement expressive APIs for dataflow concurrency (as in Oz [22]). Dataflow programming is a popular paradigm and Akka makes it available on the JVM using Scala as the implementation language.

In addition to the features already discussed, Akka also has a number of other components that make concurrent and distributed computing simpler than what a less powerful implementation language could offer. Some of these are remote actors that can be transparently distributed across multiple JVMs, agent-based computing similar to Clojure [23], and an integrated implementation of Software Transactional Memory [24]. Upcoming Akka 2.0 will have many other features like transparent and adaptive load-balancing, cluster rebalancing, and centralized configuration management, all implemented in Scala.

The very fact that Akka has been able to implement complex distributed computing abstractions is ample testimony to the power and expressiveness that Scala offers. And the very fact that Akka has a Java API for all of its functionalities illustrates the seamless interoperability that Scala has with Java.

## 7 Future developments

At the current time, we are witnessing an explosion of programming language interest and development after years of drought. During the language drought the communities of mainstream languages, especially Java, grew extremely well. But recent years have seen a backlash against the complexities of Java and C++, with visionary developers pushing to see what they can gain from languages like Scala, Clojure, and Erlang.

JVM-based alternative languages have an advantage when it comes to displacing Java. Scala and Clojure already have this advantage, and both allow integration with existing Java libraries and frameworks. To this end, author Thorup has developed Erjang [25], an implementation of Erlang that runs on the JVM, allowing integration with existing Java systems while also being able to run regular Erlang bytecode. This integration-based approach allows for the preservation of investment in existing code even as it is enhanced or converted to alternative languages over time as maintenance requirements and cost permit.

Technical superiority appears to belong to these alternative languages and the frameworks sitting above them, plus they seem to have an advantage in important areas such as code readability, maintainability, and extensibility. We are even starting to see *little languages* being developed on top of the core language that speaks the vocabulary of distributed programming in a much more explicit way; Akka is one such example of what is essentially a Domain Specific Language [26] of distributed computing.

But ultimately cost, not technical superiority, wins the day. The continued growth of cost-effective web, cloud, and "big data" applications, where languages like PHP, Ruby, Python, and especially JavaScript are the norm, are not only helping drive further interest in alternative languages, but they are also impacting traditional middleware development. With more enterprises deploying and using services in cloud- and web-based systems such as Amazon Elastic Compute Cloud (EC2), SalesForce.com, Yammer, Google Docs, ZenDesk, and others, integration projects, traditionally a stronghold of middleware, are shifting to the web as well, spurring an interest in RESTful web services [27] and HTTP-based integration. HTTP already allows for the true interface/implementation split that middleware has long sought, thus allowing developers to freely choose implementation languages that allow them to build and deliver their web-based services in whatever manner they deem best in terms of availability, scalability, and reliability for their customers and partners. Developers who choose languages like Scala and Erlang that help them construct their services faster and with higher reliability stand an excellent chance of beating their competition. As these web- and cloud-based approaches continue to gain favor, their lower costs will eventually attract even the most conservative enterprises, and the disruption of the traditional middleware market will be complete.

The challenge facing today's middleware researchers is to more fully explore the intersection between alternative programming languages and web- and cloud-based computing systems. Tomorrow's services will require better component abstractions, improved failover and redundancy mecha-

nisms, and more scalable and flexible consistency and availability options. Alternative programming languages like Erlang and Scala and their associated frameworks are not only rich enough for advanced research experiments in these areas, but due to their practicality can also greatly inform us as to what works and what does not.

## References


1. White JE (1975) RFC 707. http://tools.ietf.org/html/rfc707
2. Nelson BJ (1981) Remote procedure call. PhD Dissertation, Carnegie Mellon Univ, Pittsburgh, PA, USA. AAI8204168
3. Liskov B (1985) The Argus language and system. In: Paul M, Siegert HJ (eds) Distributed systems—methods and tools for specification, Lecture notes in computer science, vol 190. Springer, Berlin, pp 343–430
4. Almes G, Black A, Lazowska E, Noe J (1985) The Eden system: a technical review. IEEE Trans Softw Eng SE-11(1):43–59
5. Black A, Hutchinson N, Jul E, Levy H, Carter L (1987) Distribution and abstract types in Emerald. IEEE Trans Softw Eng SE-13(1):65–76
6. Zahn L, Dineen T, Leach P, Martin E, Mishkin N, Pato J, Wyant G (1990) Network computing architecture. Prentice-Hall, New York
7. Sun Microsystems (1988) RPC: remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc, June
8. Object Management Group (1991) The common object request broker: architecture and specification (CORBA). OMG document number 91-12-1
9. Waldo J, Wyant G, Wollrath A, Kendall S (1994) A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc
10. Cox BJ (1991) Object oriented programming: an evolutionary approach. Addison Wesley, Reading
11. NeXT Computer, Inc (1993) Object-oriented programming and the objective C language. Addison Wesley, Reading
12. Gosling J, McGilton H (1995) The Java language environment—a white paper
13. Java remote method invocation over IIOP. http://en.wikipedia.org/wiki/RMI-IIOP
14. Let it crash. http://c2.com/cgi/wiki?LetItCrash
15. Hewitt C, Bishop P, Steiger R (1973) A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd international joint conference on artificial intelligence (IJCAI'73). Morgan Kaufmann, San Francisco, pp 235–245
16. http://www.cs.berkeley.edu/~matei/spark/
17. Finagle, implementing asynchronous clients and servers. http://twitter.github.com/finagle/
18. JBoss Netty, the Java NIO client server framework. http://www.jboss.org/netty
19. Akka—simpler concurrency. http://akka.io
20. Erlang programming language. http://www.erlang.org/
21. A taste of 2.8: continuations. http://www.scala-lang.org/node/2096
22. The Mozart programming system. http://www.mozart-oz.org
23. Agents and asynchronous actions. http://clojure.org/agents
24. Software transactional memory. http://dl.acm.org/citation.cfm?id=224987
25. Erjang. https://github.com/trifork/erjang/wiki
26. Ghosh D (2010) DSLs in action. Manning, Shelter Island
27. Richardson L, Ruby S (2007) Restful web services. O'Reilly, Sebastopol