

DOI:10.1145/1810891.1810910

**The same component isolation that made it effective for large distributed telecom systems makes it effective for multicore CPUs and networked applications.**

BY JOE ARMSTRONG

# Erlang

ERLANG IS A CONCURRENT programming language designed for programming fault-tolerant distributed systems at Ericsson and has been (since 2000) freely available subject to an open-source license. More recently, we've seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multicore computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multicore computers.

Erlang<sup>2</sup> is used for programming fault-tolerant, distributed, real-time applications. What differentiates it from most other languages is that it's a concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only limited memory

overhead; programmers can create large numbers of Erlang processes yet ignore any preconceived ideas they might have about limiting the number of processes in their solutions.

All Erlang processes are isolated from one another and in principle are "thread safe." When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes. In supporting strong isolation between processes and polymorphism, Erlang could be viewed as extremely object-oriented though without the usual mechanisms associated with traditional OO languages.

Erlang has no mutexes, and processes cannot share memory.<sup>a</sup> Even within a process, data is immutable. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state.<sup>b</sup> Moreover, instead of classes, methods, and inheritance, Erlang has modules that contain functions, as well as higher-order functions. It also includes processes, sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

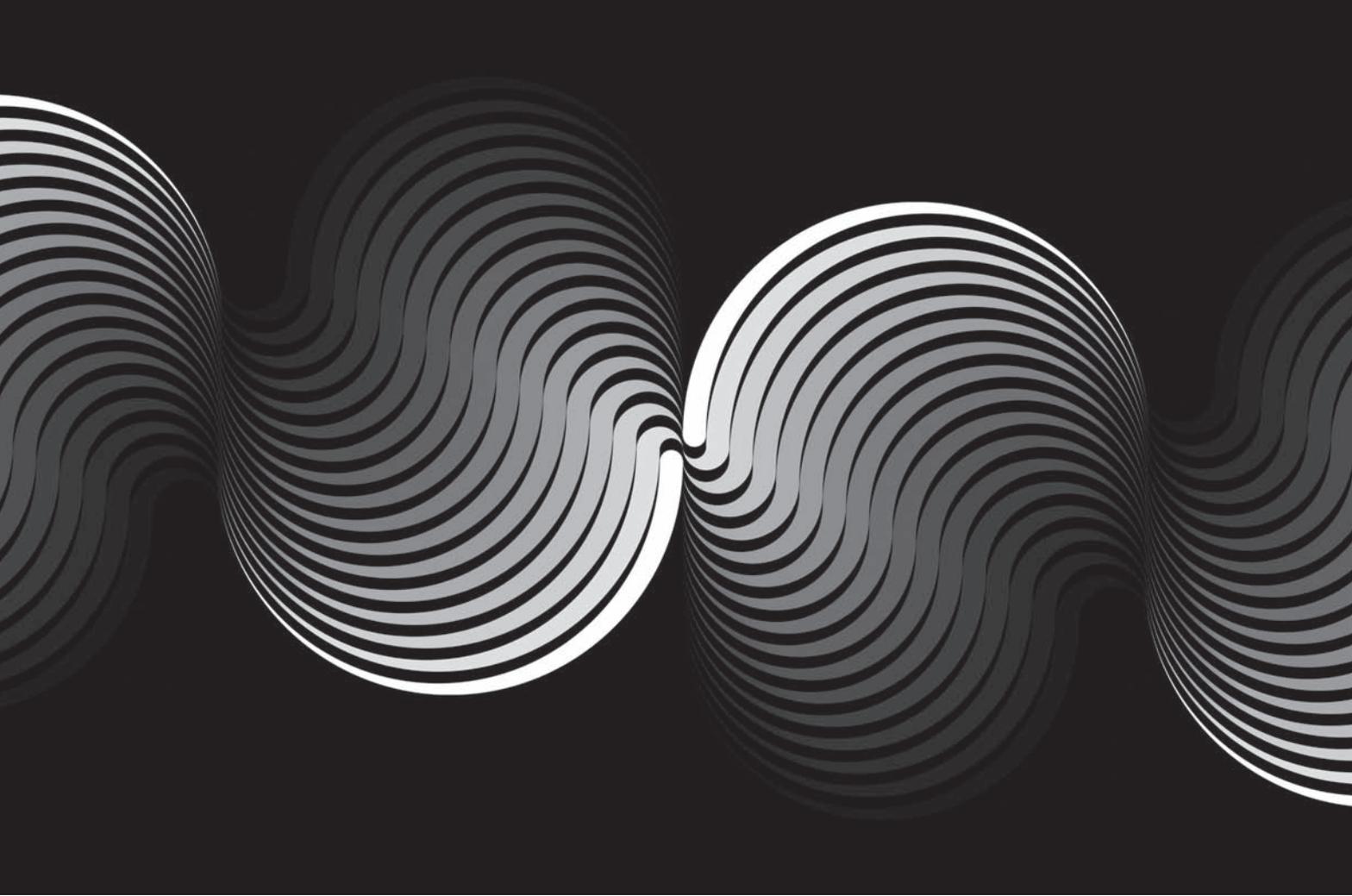
Here, I outline the key design criteria behind the language, showing how they are reflected in the language itself, as well as in programming language technology used since 1985.

## Shared Nothing

The Erlang story began in mid-1985 when I was a new employee at the Ericsson Computer Science Lab in Stock-

a The shared memory is hidden from the programmer. Practically all application programmers never use primitives that manipulate shared memory; the primitives are intended for writing special system processes and not normally exposed to the programmer.

b This is not strictly true; processes can mutate local data, though such mutation is discouraged and rarely necessary.



holm charged with “doing something about how we write software.” Ericsson had a long tradition of building highly reliable fault-tolerant systems (telephone exchanges) specified to have at most four minutes of downtime per year and system software that could be upgraded without stopping the system.

How would we do it? The question was answered in the mid-1970s and has been the same ever since. The system would have to be constructed from physically isolated components communicating through well-defined “pure” protocols. The word “pure” has special significance, meaning that after a message passes there should be no dangling pointers or data references to data structure residing on other machines.

Fault-tolerance is achieved like this: If a machine crashes, the failure is detected by another machine in the network. The machine (or machines) detecting the failure must have sufficient data to take over from the machine that crashed and continue with the application. Users should not notice the failure.

This technique was used by Jim Gray<sup>10</sup> in the design of the fault-tolerant Tandem computer. The Tandem hardware architecture was similar to the software architecture used to build Erlang applications. Using failure detection plus replication to make reliable systems has a long history.<sup>11</sup>

Now assume we have a single machine, and the probability that it will fail during some time period is  $10^{-3}$ . If we have two identical isolated machines, then the probability they both will fail in the same time period is  $10^{-6}$ , with three machines  $10^{-9}$ , and so on. Component isolation is the key to building reliable systems. Individual components might fail, but the probability that all components will fail at the same time can be made arbitrarily small by having a sufficiently large number of replicated components.

This approach works for hardware, but what about for software? If 10 copies of some software run on 10 different isolated machines, won't they all fail for the same reason if they all have the same software and are trying to solve the same problem? Of course they will, but in the systems we build, this is not

a problem. Imagine a system in which 10,000 transactions are in progress simultaneously, including telephone calls, Web sessions, database queries, anything. Each transaction could be running the same software, but each instance of the software will also have some private state. An individual process crashing due to a software error is not problematic, provided all other processes in the system (where no errors have occurred) are not affected by the crash.

Building fault-tolerant software requires the same trick used to build fault-tolerant hardware. We arrange for one process to observe the behavior

## » key insights

- **Message-passing systems scale easily, are surprisingly efficient, and can be made fault tolerant through replication over several isolated machines.**
- **Non-defensive programming and Erlang's “let it crash” style of programming lead to clear, compact code.**
- **Upgrading systems without taking them out of service has been practiced in the telecom world for years; Erlang makes it relatively easy.**

of another process. The observing process must be able to detect failures in the observed process and take over in the event of an error.

We also forbid dangling pointers and shared data structures between processes. The entire system is constructed so the observed processes and their observers need not even be on the same machine. For example, in distributed Erlang, processes can be scattered over physically separated nodes and behave semantically as if they were on the same node. The only difference is in the pragmatics of the system; the latency of an operation performed on a local process and a process located on some physically separated node are very different.

This property of Erlang processes means programs can be developed on a single node and deployed on a cluster without major changes to most of the software in the system.

In light of such considerations, we concluded (in 1986) that in order to program fault-tolerant applications Erlang would need four key properties<sup>4,9</sup>;

- ▶ Isolated processes;
- ▶ Pure message passing between processes;
- ▶ The ability to detect errors in remote processes; and
- ▶ A method for determining what error caused a process to crash.

We did not want to use shared memories, mutexes, or semaphores, so our only method of process synchronization was through message passing—viewed by most programmers at the time as a crazy method for designing systems. The principal objection was efficiency; copying messages between processes (instead of using shared memory) was considered horrendously inefficient. The counterargument was that shared memory was something preventing fault-tolerance. I have always believed that systems should be made to work correctly before they are made fast. Fault-tolerance in the presence of both hardware and software errors must be addressed ahead of efficiency.

Fast-forward almost 25 years from 1986 to see that networked applications are extremely common and multicore computers are everywhere. As the number of cores increases so does the need for isolation throughout the

system. Small isolated computations are easily allocated to a pool of cores. Shared memory translates to cache-misses in multicore computers. If a process running on one core of a multicore computer wants to access data in the cache of a physically distant core, pipeline stalls will occur, and the entire operation will take much longer than if the memory had been available locally.

Erlang today is well-placed for programming multicore CPUs. Faced with a multicore CPU, most programmers turn legacy code into a parallel program. Erlang programmers face an entirely different problem. They already have a parallel program, but it might have some sequential bottlenecks, so their job is to find the bottlenecks.

Here, I explore the language, along with some of the more interesting applications that have been written in it. Though Erlang started in the telecom world, it has escaped to wider pastures, rather like Unix and C.

### Erlang View of the World

The Erlang view of the world is that everything is a process that lacks shared memory and influences one another only by exchanging asynchronous messages. This view is broadly similar to the actors model proposed by Gul Agha.<sup>1</sup> Each process has a mailbox to which messages can be sent. Messages are retrieved from the mailbox with a `receive` statement or pattern-matching construction that removes messages matching a particular pattern in the mailbox and can also be used to selectively remove messages from the mailbox.

Hardware in Erlang is interfaced through processes. A process that controls hardware has two interfaces: one toward the Erlang system, where it behaves as a regular Erlang process, the other toward the hardware controlled through a port providing an I/O channel to the outside world. All communication with the outside takes place through ports.

Foreign-language software (not in Erlang) cannot be linked to the Erlang kernel but must be run in a separate operating system process that executes outside the Erlang runtime system and is interfaced through a port. Security is the reason for not linking foreign-language code into the kernel; we do not

want errors in external code crashing the Erlang runtime system.

### Erlang View of Errors

Erlang differs from most other programming languages in the way it handles errors. An Erlang system typically consists of large numbers of lightweight processes. It is of no particular consequence if any one of them dies. The recommended way of programming is to let failing processes crash and other processes detect the crashes and fix them.

Erlang has a safe type system. Data structures are dynamically typed, and it is impossible to create corrupt data structures. Extensive user checking of data structures is unnecessary, since the worst that can happen is an individual process might crash if it performs an illegal operation. The important thing to note is that the crash of one process does not affect any other unlinked process in the system.

However, being type-safe does not solve all programmer problems; for example, exception handlers must still be written to correct type errors, and sets of observing processes must be created to correct errors caused when processes crash due to type errors. Some of these errors could have been caught by static type checking, but adding complete static type checking to Erlang would change the flavor of the language and make upgrading dynamic code and other things virtually impossible.

In a single-threaded application one has only one chance to correct an error, so the consequence of not correcting an error is that an entire application might fail, thus single-threaded applications and languages take great care to fix errors locally. With thousands of processes at one's disposal one is less concerned about the failure of individual processes than about detection and correction of errors. The system is divided into worker processes that perform computations and supervisor processes that check that the worker processes are behaving correctly.

Erlang has an internal mechanism, or “link,” that provides a form of inter-process error detection and performs as an error-propagation channel. If process A is linked to process B and

process A dies, then an error signal will be sent to process B, and vice versa.

The ability to monitor a process provides a clue for building reliable systems. The idea is to try to solve the problem, but if the processes in the solution cannot do the job, the system tries to solve a simpler problem. “Cannot do the job” means detecting the failure of a process; the system detects such failures and tries to solve a simpler problem.

One layer of the system usually performs the application logic, and an error-trapping layer monitors the application and restores it to a safe state if an error occurs. This application structure is formalized in the Erlang Open Telecom Platform (OTP) system using so-called supervision trees providing a precise description of what is to happen if a computation fails. OTP applications organize problems into tree-structured groups of processes, letting the higher nodes in the tree monitor and correct errors occurring in the tree’s lower nodes.

### Erlang Programs

Erlang was first implemented in Prolog<sup>6</sup> in 1986, and thus many of the syntactic conventions used in Erlang come from Prolog; Erlang’s syntactic conventions include:

**Variables.** When variables, or single-assignments (written starting with an uppercase letter like `Day` and `File`), acquire a value, that value cannot be changed; variables acquire values in successful pattern-matching operations;

**Atoms.** Used to represent constants, they are similar to enumerated types in Java and C and written starting with a lower-case letter; for example, `monday`, `orange`, and `cat` are atoms;

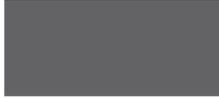
**Tuples.** Like structs in C and used for storing fixed numbers of items, tuples are written in curly brackets; for example, `{Var, monday, 12}` is a tuple containing a variable atom and an integer; and

**Lists.** Used for storing variable numbers of items, lists are written enclosed in square brackets; for example, `[a,X,b,Y]` is a list containing two atoms and two variables.

Erlang’s syntax is designed to make it easy to express parallel computations. Here, I jump in the deep end



**The recommended way of programming is to let failing processes crash and other processes detect the crashes and fix them.**



of Erlang program development with a code fragment that creates a counter process. Many of the examples are from my 2007 book *Programming in Erlang*<sup>2</sup> and contain all the gruesome details one would need to write Erlang code. I begin by creating a counter process:

```
Pid = spawn(fun() -> counter(0)
end),
```

`spawn(Fun)` means “create a parallel process that evaluates `Fun`,” or an Erlang function.

The function `counter(N)` in Figure 1 waits for one of two messages: If the process is sent the message `tick`, it calls `counter(N+1)`. If it is sent the message `{From, read}` it replies by sending a message `{self(), N}` to the process `From` and then calls `counter(N)`. The notation `A!B` means send the message `B` to the process `A`, `self()` is the process identity of the process running the counter function and

```
receive
  Pattern1 ->
    Actions1;
  Pattern2 ->
    Actions2;
  ...
end
```

means wait for a message. If the next message matches `Pattern1`, then execute the code `Actions1`; otherwise if the message matches `Pattern2`, then execute the code `Actions2`, and so on. If no pattern is matched, then queue the message for later and wait for the next message.

To bump the counter, some process that knows the name of the process executes the code:

```
Pid ! tick.
```

To read the counter, we evaluate:

```
Pid ! {self(), read},
receive
  {Pid, Result} ->
    Result
end
```

We send a `{self(), read}` message to the counter process, then wait for a

**Figure 1. A simple counter process.**

```
counter(N) ->
  receive
    tick ->
      counter(N+1);
  {From, read} ->
    From ! {self(), N},
    counter(N)
  end.
```

return message {Pid, Result}. Variables in Erlang are bound only once and thereafter can never be changed, so when one enters the receive statement, Pid has a value; it must have a known value, since otherwise Pid ! .. would be meaningless. We must know the identity of a process in order to send it a message.

The receive statement then waits for a message {Pid, Result} where Pid is a bound variable and Result is an unbound variable. This code fragment means “Wait for a message that is a tuple with two arguments where the first argument matches Pid and bind the value of the second argument in the tuple to the variable Result. That is, the code fragment waits for a message from the process Pid and queues any other messages that might arrive while waiting for the message.

This code fragment occurs so often it’s been given a name and made into a library function:

```
rpc(Pid, Request) ->
  Pid ! {self(), Request},
  receive
    {Pid, Response} ->
      Response
  end.
```

It is simply a remote procedure call.

Erlang has no built-in mechanism for doing remote procedure calls, but one can easily program a remote procedure call using the built-in primitives send and receive. Why are built-in primitives important? Because we can roll our own interprocess communication mechanisms. If we want to do two remote procedure calls in parallel, it could be done like this:

```
Pid1 ! {self(), Request1},
Pid2 ! {self(), Request2},
receive
  {Pid1, Response1} ->
    Response1
end,
receive
  {Pid2, Response2} ->
    Response2
end,
...
```

This code is non-blocking since receive automatically queues any out-of-order messages sent to the processes. If Pid1 replies first, then the first receive clause is triggered, and execution steps to the second receive statement and waits for the second process to reply. If Pid2 replies first, then the message is queued; once Pid1 replies, the first receive statement is satisfied and the program steps to the second receive statement, but the message will have been saved and queued, so the second receive statement is triggered immediately. The net result is that on completion of the code fragment both messages will have been received irrespective of the order in which they were sent. The time spent waiting is the longer of the response times from the two processes.

Note, too, if the program had exposed only a composite remote proce-

dure call function, such programming would be more difficult, since two intermediate processes would have been spawned, where each performed a remote procedure call, and the results would then have to be combined. With three or more processes, coordinating the actions of the parallel processes would be difficult to program, were it not for the queuing mechanism built into the Erlang receive statement.

**Detecting errors.** Recall that in order to build reliable systems one must be able to remotely detect errors.<sup>c</sup> Figure 2 defines a function that can detect an error in a remote process and perform an action on detecting the error, and on\_exit(Pid, F) creates a process that monitors the process Pid. If the monitored process dies with reason Why, the newly created process evaluates the function F(Why), and process\_flag(trap\_exit, true) turns the current process into a “system process” that can trap exit signals. The statement link(Pid) sets up a “link” to the process Pid. A link is an error-propagation channel, and link(Pid) means “if the process Pid dies, send me an exit signal.” An exit signal is an out-of-band message sent when a process dies. Processes normally die when they receive out-of-band exit signals, but because the process evaluated process\_flag(trap\_exits, true), it became a system process, and thereafter the exit signal can be received as a message containing a {‘EXIT’, Pid, Why} tuple.

The function on\_exit is the workhorse needed to build fault-tolerant code. Using on\_exit allows one to build a hierarchical tree of processes. Some processes do the work, and other processes monitor the processes that do the work and fix things up if the worker processes die.

Recall that the Erlang philosophy is “Let it crash”; in fact, processes that cannot perform the task they were told to do should crash immediately. Another process will correct the error. This is exactly the opposite of defensive programming but leads to a clean

**Figure 2. A process monitor.**

```
on_exit(Pid, F) ->
  spawn(fun() -> monitor(Pid, F) end).

monitor(Pid, F) ->
  process_flag(trap_exit, true),
  link(Pid),
  receive
    {‘EXIT’, Pid, Why} ->
      F(Why)
  end.
```

<sup>c</sup> Local error detection is no good; the local machine might have crashed and cannot perform error recovery, so the error must be detected on a remote machine unaffected by the crash.

separation of interest between code that does the job and code that cleans up an error when it occurs. Erlang does not provide an `on_exit` function, but it is easy to program one using the Erlang's built-in primitives.

**Dynamic code upgrade.** One thing users and developers alike want to do is run their systems forever. Assuming things change, they will also want to change the code in a running system, but how? Imagine a simple server written as follows:

```
loop(State, F) ->
  receive
    {From, Request} ->
      {Response, State1}
      =F(Request, State),
      From ! {self(), Response},
      loop(State1, F)
  end.
```

This server is a simple extension to the counter process in Figure 1. The server process has state `State` and a processing function `F`. We could create a process that evaluates this loop like this:

```
F1 = fun(N, State) -> {N*N,
State+1} end,
Pid = spawn(fun() -> loop(0,
F1) end,
```

The processing function `F1` returns the square of its first argument and keeps a running total of the number of requests to the server.

The code in the server cannot be changed, but a small addition can be made to allow for dynamic code upgrade by adding a `{newFunction, F1}` pattern to the receive statement:

```
loop(State, F) ->
  receive
    {newFunction, F1} ->
      loop
        (State, F1);
    {From, Request} ->
      {Response, State1} =
        F(Request, State),
      From ! {self(), Response},
      loop(State1, F)
  end.
```

Now a new processing function can be sent to the server without interrupting it; for example, we could write:

```
F1 = fun(N, State) -> {N*N,
State+1} end,
Pid = spawn(fun() -> loop(0,
F1) end),
...
... some time later
...
F2 = fun(N, State) -> {N*N*N,
State+1} end,
Pid ! {newFunction, F2},
...
```

This new function dynamically upgrades the code in the server.

**Adding transactions.** Adding transactions is easy. In a transaction, either state is modified if it works or there is no change to the state if the transaction fails. To implement this, we add a `try-catch-end` block to the inner part of the receive statement:

```
loop(State, F) ->
  receive
    {newFunction, F1} ->
      loop(State, F1);
    {From, Request} ->
      try F(Request, State) of
        {Response, State1} ->
          From ! {self(), Response},
          loop(State1, F)
      catch
        _:Why ->
          exit(From, crash)
          loop(State, F)
      end
  end.
```

The evaluation of `F(Request, State)` is wrapped in a `try-catch-end` block. If the evaluated function raises an error, then the process evaluates the statement `exit(Pid, crash)`, which sends an exit signal to the process that caused the exception; thereafter, `loop(State, F)` is called, or recurs with the original value of the state.

The sequential part of Erlang is a functional language that does not allow the mutation of state. Because state cannot be mutated, an Erlang function can always revert to a previous state of the computation by accessing the original variable that referred to the state.

Finally, note the effect of tail-recursion. All server loops in the example code finish with tail calls. Once a tail call is made there is no going back; tail calls do not create additional stack

frames, since there is nowhere to return to, and a new stack frame is not required. Having made a tail call, all local variables in the current context can be garbage-collected, allowing tail-recursive loops to run indefinitely without consuming stack space.

### Open Telecoms Platform

OTP is a large set of libraries written mostly in Erlang bundled together with the Erlang distribution. OTP can be viewed as an application middleware package that simplifies writing large Erlang applications. Recall I mentioned language primitives that could be used to build simple functions that encapsulate errors, showing how to build a simple function `on_exit` that could be used to evaluate a specific function if an error occurred in some other process.

Functions like `on_exit`, while useful and good to include in books on programming languages, are not the stuff from which large systems are built. If a software component in a large enterprise system fails, the error report must be kept forever and the system restarted. If a code upgrade fails, the entire system must be automatically rolled back to a previous state in a controlled manner.

Organizations employing large teams of programmers cannot let individual programmers invent their own error-handling mechanisms and ways of dynamically upgrading code. The OTP libraries are thus an attempt to formalize a large body of design knowledge into workable libraries that provide a standardized way of performing the most common tasks needed to build a reliable system.

OTP is the third total rewrite of a system of libraries in Erlang designed for building telecom systems.<sup>3,4</sup> The 2010 OTP system includes 49 subsystems, each a powerful tool in its own right. Typical subsystems are `mnesia` (a real-time relational database), `megaco` (an H.248 stack), and `docbuilder` (a tool to make documentation), along with sophisticated analysis-test and analysis tools.

Because a large number of Erlang programs are written in a pure functional programming style, they are able to perform sophisticated analysis and transformations. For example, the

dialyzer<sup>14</sup> is a type-checking program that performs static analysis of Erlang programs, finding type errors (if there are any) in them. The test tool Quick-Check<sup>8</sup> generates random test cases from a specification of the formal properties of a program, and the tool Wrangler<sup>13</sup> can be used to refactor Erlang programs.

### Erlang Distribution

Ever since Erlang was first released into the public domain in 2000, it has been supported by an internal product-development group within Ericsson. Following the release of Open Source Erlang (<http://www.erlang.org/>), the language spread slowly for several years but has recently seen a dramatic upturn in the number of users and applications. This growth corresponds to a similar upturn in interest in Haskell (<http://www.haskell.org/>), a strictly typed lazy polymorphic programming language.

Two other languages in the same functional language school are OCaml and F#. The simultaneous increase in interest in different forms of functional programming can be seen as evidence that functional programming has come of age and is transitioning from the academic world to industrial practice. Industrial projects and the formation of new companies using Erlang as core technology reflect the more interesting developments. Erlang can be downloaded from <http://www.erlang.org/>, including the OTP system and a large number of tools.

### Experience

In OO languages, objects are used to structure applications. In Erlang applications, processes are used for structuring, a technique I call “concurrency oriented programming,” or COP.<sup>5</sup> The idea of building systems from communicating components is not new. Tony Hoare’s Communicating Sequential Processes<sup>12</sup> described how sets of concurrent processes could be used to model applications, and programming languages like Occam<sup>15</sup> that were based on it explored the idea. Erlang is conceptually similar to Occam, though it recasts the ideas of CSP in a functional framework and uses asynchronous message passing



## Not surprising, the leading uses of Erlang outside telecom all involve communications and reliable data storage.



instead of the synchronous message passing in CSP.

Processes in COP systems are isolated, responding only to messages and resulting in systems that are easy to understand, program, and maintain. Several fairly large systems written in Erlang enforce this idea. Several major product developments are based on Erlang, the largest being the AXD301 an asynchronous transfer mode (ATM) switch developed by Ericsson. Outside Ericsson, Erlang is being used by a large number of start-ups and is the principle technology of several new companies in Stockholm.

**AXD301.** The AXD301 switch<sup>9</sup> has scalable capacity ranging from 10Gbit/sec to 160Gbit/sec and modular architecture and was written in distributed Erlang. Built by a large programming team, it has more than 1.6 million lines of Erlang code, showing that COP as a structuring method and Erlang as a programming language scale to large systems. One reason it scales so well is the architecture. At one level of abstraction, it can be viewed as a system of components that communicate through pure message passing. The lack of shared state and division of the system into well-isolated communicating components make it easy to understand the system’s overall architecture and isolate problems within the system.

When a message is sent into a component, we expect a certain response, or message, from it. If this does not happen, the error lies within the component. Opening it could reveal the same internal structure found on the outside, just a set of communicating components. “Opening a component” can be performed repeatedly until a misbehaving Erlang process is found. There is no magic. Making reliable systems from isolated components leads to systems that are easy to understand and manageable in both small- and large-scale projects.

**Instant messaging.** One problematic area in Internet applications where Erlang has found notable success is implementing instant-messaging systems. An IM system looks at first approximation very much like a telephone exchange. IM and telephone exchanges must both handle very large numbers of simultaneous transac-

tions, each involving communication with a number of simultaneously open channels. The work involved in parsing and processing the data on any one channel is small, but handling many thousands of simultaneous channels is a technical challenge.

Erlang's usefulness in IM is demonstrated by three projects:

*MochiWeb* (<http://code.google.com/p/mochiweb>). Designed for building lightweight HTTP servers developed by MochiMedia for high-throughput, low-latency analytics, and ad servers, this Erlang library helps power Facebook chat among more than 70 million users;

*Ejabberd* (<http://www.ejabberd.im>). Written by Alexey Shchepin, this Erlang implementation of the XMPP protocol is the most widely used open source XMPP server; and

*RabbitMQ* (<http://www.rabbitmq.com>). This Erlang implementation of the Advanced Message Queuing Protocol standard provides reliable asynchronous message passing at Internet scale.

**Schema-free databases.** In traditional databases, data is stored in rectangular tables, where the items in a table are instances of simple types (such as integers and strings). Such storage is not particularly convenient for storing an associative array or arbitrary tree-like structure. Examples of the former are JavaScript JSON data structures (called hashes in Perl and Ruby and maps in C++ and Java) and of the latter XML parse trees. These objects are difficult to store in a regular tabular structure. Erlang has for a long time had its own database, called *mnesia*, that includes table storage but allows any item in a table cell to also be an arbitrary Erlang data structure.

Databases implemented in Erlang are particularly well-suited for such storage, especially when they interface with some form of communicating agent. Three notable databases are implemented in Erlang:

*CouchDB* (<http://incubator.apache.org/couchdb/>). Written by Damien Katz, "Apache CouchDB is a distributed, fault-tolerant, schema-free document-oriented database accessible via a RESTful HTTP/JSON API." It provides robust, incremental replication with bidirectional conflict detection and resolution, queryable and index-

able through a table-oriented view engine, with JavaScript acting as the default view-definition language;

*Amazon SimpleDB* (<http://aws.amazon.com/simpledb/>). This Web service runs queries on structured data in real time; and *Scalaris*.<sup>16</sup> This scalable, transactional, distributed key-value store has a peer-to-peer architecture for supporting reliable transactions with ACID properties.

CouchDB and Scalaris are open source projects; SimpleDB is a closed-source commercial service.

**Sweet spot.** Taking in the six projects described here reveals a pattern of communication with complex data structures being passed over the network. The number of clients wanting simultaneous access to the system is potentially huge, with hundreds of thousands to millions of users. The data stores must therefore be reliable and the data protocols extensible. Not surprising, this is the Erlang "sweet spot" for supporting system development. Erlang was developed for building high-performance telecom switches, with hundreds of thousands of users accessing the system simultaneously. Data structures are complex, and the system must be able to store data in a reliable manner, recovering from local failures and scaling clusters to manage varying demand. Erlang was designed to do all these things, with the intended applications domain of carrier-class telecoms systems. Also not surprising, the leading uses of Erlang outside telecom all involve communications and reliable data storage. In an abstract sense, what these projects do is serialize data terms into a transportable format (marshalling and unmarshalling), transport the data over the network, and store the data in some kind of persistent storage medium.

Beyond the sweet spot, several applications that have nothing to do with fault tolerance have also gained popularity; for example, *Wings* (<http://www.wings3d.com>), a 3D graphics modeling program written by Björn Gustavsson, and *Nitrogen* (<http://nitrogenproject.com/>), a Web-development framework written by Rusty Klophaus, show that Erlang is useful as a general-purpose programming language.

## Acknowledgments

I thank Ericsson Telecom for its contribution to the development of Erlang over the years. 

## References

1. Agha, G. *Actors: A model of concurrent computation in distributed systems*. In *MIT Series in Artificial Intelligence*. MIT Press, Cambridge, MA, 1986.
2. Armstrong, J. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC, 2007.
3. Armstrong, J. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages* (Dan Diego, CA, June 9–10). ACM Press, New York, 2007.
4. Armstrong, J. *Making Reliable Distributed Systems in the Presence of Errors*. Ph.D. Thesis, Royal Institute of Technology, Stockholm, 2003.
5. Armstrong, J. Concurrency-oriented programming in Erlang. Invited Talk at the Lightweight Languages Workshop (Cambridge MA, Nov. 9, 2002).
6. Armstrong, J.L., Viriding, S.R., and Williams, M.C. Use of Prolog for developing a new programming language. In *Proceedings of the First Conference on the Practical Application of Prolog* (London, Apr. 1–3). Association for Logic Programming, 1992.
7. Blau, S. and Rooth, J. Axd 301: A new-generation ATM switching system. *Ericsson Review* 1 (1998).
8. Claessen, K. and Hughes, J. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, 2000, 268–279.
9. Däcker, B. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Licentiate Thesis. Royal Institute of Technology. Stockholm, 2000.
10. Gray, J. *Why Do Computers Stop and What Can Be Done About It?* Tech. Rep. 85.7. Tandem Computers, Inc., 1985.
11. Guerraoui, R. and Schiper, A. Fault tolerance by replication in distributed systems. In *Proceedings of the Conference on Reliable Software Technologies*. Springer Verlag, 1996, 38–57.
12. Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, 1985.
13. Li, H., Thompson, S., Orosz, G., and Toth, M. Refactoring with Wrangler: Data and process refactorings and integration with Eclipse. In *Proceedings of the Seventh ACM SIGPLAN Erlang Workshop* (Victoria, BC, Sept. 27). ACM Press, New York, 2008, 61–72.
14. Lindahl, T. and Sagonas, K. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Proceedings of the Second Asian Symposium* (Taipei, Taiwan, Nov. 4–6). Springer, 2004, 91–106.
15. *Occam Programming Manual*. Prentice Hall, Upper Saddle River, NJ, 1984.
16. Schütt, T., Schintke, F., and Reinefeld, A. Scalaris: Reliable transactional p2p key/value store. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Erlang* (Victoria, BC, Sept. 27). ACM Press, New York, 2008, 41–48.
17. Wiger, U., Ask, G., and Boertz, K. World-class product certification using Erlang. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Erlang* (Pittsburgh, PA). ACM Press, New York, 2002, 24–33.
18. Wiger, U. Fourfold increase in productivity and quality: Industrial-strength functional programming in telecom-class products. In *Proceedings of the Workshop on Formal Design of Safety Critical Embedded Systems* (Münich, Mar. 21–23, 2001).

**Joe Armstrong** ([joe.armstrong@ericsson.com](mailto:joe.armstrong@ericsson.com)) is an expert in software architectures and programming languages in Business Unit Networks at Ericsson, Stockholm, Sweden.