Erlang

for Concurrent Programming

Designed for concurrency
from the ground up,
the Erlang language can be
a valuable tool to help
solve concurrent problems.

Jim Larson, Google

Erlang is a language developed to let mere mortals write, test, deploy, and debug fault-tolerant concurrent software.[1] Developed at the Swedish telecom company Ericsson in the late 1980s, it started as a platform for developing soft realtime software for managing phone switches.[2] It has since been open-sourced and ported to several common platforms, finding a natural fit not only in distributed Internet server applications, but also in graphical user interfaces and ordinary batch applications.

Erlang's minimal set of concurrency primitives, together with its rich and well-used libraries, give guidance to anyone trying to design a concurrent program. Erlang provides an effective platform for concurrent programming for the following reasons:

- The language, the standard libraries (Open Telecom Platform, or OTP), and the tools have been designed from the ground up for supporting concurrency.
- There are only a few concurrency primitives, so it's easy to reason about the behavior of programs (though there are limits to how easy this can ever be).
- The implementation makes the simple primitives fast and scalable, and makes effective use of modern multi-core hardware, eliminating the need for more complex mechanisms.
- The execution model eliminates some classes of errors from unsynchronized access to shared state—or at least makes these errors more noticeable.

Erlang
for Concurrent Programming

- The model of concurrency is natural to think about and requires no mathematical sophistication.
- The environment makes failures detectable and recoverable, making it possible to deploy a less-than-perfect system in the field that can nonetheless maintain high availability.
- The concurrency model maps naturally to distributed deployments.

This article introduces the Erlang language and shows how it can be used in practice to implement concurrent programs correctly and quickly.

SEQUENTIAL ERLANG

Erlang is built from a small number of sequential programming types and concepts, and an even smaller number of concurrent programming types and concepts. Those who want a full introduction can find several excellent tutorials on the Web,[3] but the following examples (required by functional programming union regulations) should convey the essentials.

As shown in figure 1A, every file of Erlang code is a module. Declarations within the file name the module (which must match the filename) and declare which functions can be called from other modules. Comments run from the percent sign (%) to the end of the line.

Factorial is implemented by two functions. Both are named factorial, but they have different numbers of arguments; hence, they are distinct. The definition of factorial/2 (the two-argument version) is split into two clauses, separated by a semicolon. When factorial/2 is called, the actual parameters are tested against the patterns in each clause head in turn

to find the first match, then the body (after the arrow) is evaluated. The value of the final expression in the body is the return value of the call; no explicit return statement is needed. Erlang is dynamically typed, so a call to factorial("pancake") will compile but will raise a runtime exception when it fails to match any clause. Tail-calls are optimized, so this code will run in constant space.

Lists are enclosed in square brackets (see figure 1B). A

FIGURE 1

A

example1.erl

```
-module(example1).
-export([factorial/1, qsort/1, member/2, foldl/3, sum/1]).

% Compute the factorial of a positive integer.
factorial(N) when is_integer(N), N > 0 -> factorial(N, 1).

% A helper function which maintains an accumulator.
factorial(1, Acc) -> Acc;
factorial(N, Acc) when N > 1 -> factorial(N - 1, N * Acc).
```

B
```
% Return a sorted copy of a list.
qsort([]) -> [];
qsort([Pivot | Xs]) ->
    qsort([X || X <- Xs, X < Pivot])
      ++ [Pivot]
      ++ qsort([X || X <- Xs, X >= Pivot]).
```

C
```
% Is X an element of a binary search tree?
member(_, empty) -> false;
member(X, {_, X, _}) -> true;
member(X, {Left, Y, _}) when X < Y -> member(X, Left);
member(X, {_, _, Right}) -> member(X, Right).
```

D
```
% "Fold" a function across elements of a list, seeding
% with an initial value.
% e.g. foldl(F, A0, [A, B, C]) = F(C, F(B, F(A, A0)))
foldl(_, Acc, []) -> Acc;
foldl(F, Acc, [X | Xs]) ->
    NewAcc = F(X, Acc),
    foldl(F, NewAcc, Xs).

% Give the sum of a list of numbers.
sum(Numbers) -> foldl(fun(N, Total) -> N + Total end, 0, Numbers).
```

single vertical bar separates the first element from the rest of the list. If a list is used in a clause head pattern, it will match list values, separating them into their components. A list with a double vertical bar is a "list comprehension," constructing a list through generator and filter expressions. A double-plus (++) concatenates lists.

Tuples (vectors) are enclosed in curly braces (see figure 1C). Tuples in patterns will extract components out of tuples that they match. Identifiers that start with an uppercase letter are variables; those that start in lowercase are *atoms* (symbolic constants such as enum values, but with no need to define a numerical representation). Boolean values are represented simply as atoms true and false. An underscore (_) in a pattern matches any value and does not create a binding. If the same fresh variable occurs several times in a pattern, the occurrences must be equal to match. Variables in Erlang are *single-assignment* (once a variable is bound to a value, that value never changes).

Not all list-processing operations can be expressed in list comprehensions. When we do need to write list-processing code directly, a common idiom is to provide one clause for handling the empty list and another for processing the first element of a non-empty list. The foldl/3 function shown in figure 1D is a common utility that chains a two-argument function across a list, seeded by an initial value. Erlang allows anonymous functions ("fun" or closures) to be defined on the fly, passed as arguments, or returned from functions.

Erlang has expressions that look like assignments but have a different semantics. The right-hand side of = is evaluated and then matched against the pattern on the left-hand side, just as when selecting a clause to match a function call. A new variable in a pattern will match against the corresponding value from the right-hand side.

CONCURRENT ERLANG
Let's introduce concurrent Erlang by translating a small example from Java:

Sequence.java
```
// A shared counter.
public class Sequence {
    private int nextVal = 0;

    // Retrieve counter and increment.
    public synchronized int getNext() {
        return nextVal++;
    }
```

```
    // Re-initialize counter to zero.
    public synchronized void reset() {
        nextVal = 0;
    }
}
```

A sequence is created as an object on the heap, potentially accessible by multiple threads. The synchronized keyword means that all threads calling the method must first take a lock on the object. Under the protection of the lock, the shared state is read and updated, returning the preincrement value. Without this synchronization, two threads could obtain the same value from getNext(), or the effects of a reset() could be ignored.

Let's start with a "raw" approach to Erlang, using the concurrency primitives directly.

sequence1.erl (raw implementation)
```
-module(sequence1).
-export([make_sequence/0, get_next/1, reset/1]).

% Create a new shared counter.
make_sequence() ->
    spawn(fun() -> sequence_loop(0) end).

sequence_loop(N) ->
    receive
        {From, get_next} ->
            From ! {self(), N},
            sequence_loop(N + 1);
        reset ->
            sequence_loop(0)
    end.

% Retrieve counter and increment.
get_next(Sequence) ->
    Sequence ! {self(), get_next},
    receive
        {Sequence, N} -> N
    end.

% Re-initialize counter to zero.
reset(Sequence) ->
    Sequence ! reset.
```

The spawn/1 primitive creates a new *process*, returning its *process identifier* (pid) to the caller. An Erlang process, like a thread, is an independently scheduled sequential activity with its own call stack, but like an operating-system process, it shares no data with other processes—pro-
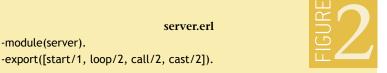
Erlang
for Concurrent Programming

The code here implements a simple client-server protocol. In a *call*, the client process sends a request message to the server process and blocks waiting for a response message. Here, the get_next/1 call request message is a two-element tuple: the client's own pid followed by the atom get_next. The client sends its own pid to let the

cesses interact only by sending messages to each other. The self/0 primitive returns the pid of the caller. A pid is used to address messages to a process. Here the pid is also the data abstraction—a sequence is just the pid of a server process that understands our sequence-specific messaging protocol.

The new process starts executing the function specified in spawn/1 and will terminate when that function returns. Long-lived processes therefore avoid premature returns, often by executing a loop function. Tail-call optimization ensures that the stack does not grow in functions such as sequence_loop/1. The state of the sequence process is carried in the argument to this eternal call.

Messages are sent with the syntax pid ! message. A message can be any Erlang value, and it is sent atomically and immutably. The message is placed in the receiving process's *mailbox,* and the sender continues to execute—it does not wait for the receiving process to retrieve the message.

A process uses the receive expression to extract messages from its mailbox. It specifies a set of patterns and associated handler code and scans the mailbox looking for the first message that matches any of the patterns, blocking if no such message is found. This is the only blocking primitive in Erlang. Like the patterns in function clauses, the patterns in receive options match structures and bind new variables. If a pattern uses a variable that has already been bound to a value, then matching the pattern requires a match with that value, as in the value for Sequence in the receive expression in get_next/1.

FIGURE 2

server.erl

```
-module(server).
-export([start/1, loop/2, call/2, cast/2]).

% Client-server messaging framework.
%
% The callback module implements the following callbacks:
% init() -> InitialState
% handle_call(Params, State) -> {Reply, NewState}
% handle_cast(Params, State) -> NewState

% Return the pid of a new server with the given callback module.
start(Module) ->
    spawn(fun() -> loop(Module, Module:init()) end).


loop(Module, State) ->
    receive
        {call, {Client, Id}, Params} ->
            {Reply, NewState} = Module:handle_call(Params, State),
            Client ! {Id, Reply},
            loop(Module, NewState);
        {cast, Params} ->
            NewState = Module:handle_cast(Params, State),
            loop(Module, NewState)
    end.

% Client-side function to call the server and return its reply.
call(Server, Params) ->
    Id = make_ref(),
    Server ! {call, {self(), Id}, Params},
    receive
        {Id, Reply} -> Reply
    end.

% Like call, but no reply is returned.
cast(Server, Params) ->
    Server ! {cast, Params}.
```

server know where to send the response, and the `get_next` atom will let us differentiate this protocol operation from others. The server responds with its own two-element tuple: the server pid followed by the retrieved counter value. Including the server pid lets the client distinguish this response from other messages that might be sitting it its mailbox.

A *cast* is a request to a server that needs no response, so the protocol is just a request message. The `reset/1` cast has a request message of just a bare atom.

ABSTRACTING PROTOCOLS

Brief as it is, the Erlang implementation of sequences is much longer and less clear than the original Java version. Much of the code is not particular to sequences, however, so it should be possible to extract the message-passing machinery common to all client-server protocols into a common library.

Since we want to make the protocol independent of the specifics of sequences, we need to change it slightly. First, we distinguish client call requests from cast requests by tagging each sort of request message explicitly. Second, we strengthen the association of the request and response by tagging them with a per-call unique value. Armed with such a unique value, we use it instead of the server pid to distinguish the reply.

As shown in figure 2, the `server` module contains the same structure as the `sequence1` module with the sequence-specific pieces removed. The syntax `Module:function` calls `function` in a module specified at runtime by an atom. Unique identifiers are generated by the `make_ref/0` primitive. It returns a new *reference*, which is a value guaranteed to be distinct from all other values that could occur in the program.

The server side of sequences is now boiled down to three one-line functions, as shown in figure 3. Moreover, they are purely sequential, functional, and *deterministic* without message passing. This makes writing, analyzing, testing, and debugging much easier, so some sample unit tests are thrown in.

STANDARD BEHAVIOURS

Erlang's abstraction of a protocol pattern is called a *behaviour*. (We use the Commonwealth spelling, as that's what is used in Erlang's source-code annotations.) A behaviour consists of a library that implements a common pattern of communication, plus the expected signatures of the callback functions. An instance of a behaviour needs some interface code wrapping the calls to the library plus the implementation callbacks, all largely free of message passing.

Such segregation of code improves robustness. When the callback functions avoid message-passing primitives, they become deterministic and frequently exhibit simple static types. By contrast, the behaviour library code is nondeterministic and challenges static type analysis. The behaviours are usually well tested and part of the standard library, however, leaving the application programmer the easier task of just coding the callbacks.

Callbacks have a purely functional interface. Information about any triggering message and current behaviour state are given as arguments, and outgoing messages

FIGURE 3

sequence2.erl (callback implementation)

```erlang
-module(sequence2).
-export([make_sequence/0, get_next/1, reset/1]).
-export([init/0, handle_call/2, handle_cast/2]).
-export([test/0]).

% API
make_sequence()       -> server:start(sequence2).
get_next(Sequence)    -> server:call(Sequence, get_next).
reset(Sequence)       -> server:cast(Sequence, reset).

% Server callbacks
init()                -> 0.
handle_call(get_next, N) -> {N, N + 1}.
handle_cast(reset, _)    -> 0.

% Unit test: Return 'ok' or throw an exception.
test() ->
    0 = init(),
    {6, 7} = handle_call(get_next, 6),
    0 = handle_cast(reset, 101),
    ok.
```

Erlang
for Concurrent Programming

and a new state are given in the return value. The process's "eternally looping function" is implemented in the library. This allows for simple unit testing of the callback functions.

Large Erlang applications make heavy use of behaviours—direct use of the raw message-sending or receiving expressions is uncommon. In the Ericsson AXD301 telecom switch—the largest known Erlang project, with more than a million lines of code—nearly all the application code uses standard behaviours, a majority of which are the server behaviour.[4]

Erlang's OTP standard library provides three main behaviours:

Generic server (gen_server). The generic server is the most common behaviour. It abstracts the standard request-response message pattern used in client-server or remote procedure call protocols in distributed computing. It provides sophisticated functionality beyond our simple server module:
• Responses can be delayed by the server or delegated to another process.
• Calls have optional timeouts.
• The client monitors the server so that it receives immediate notification of a server failure instead of waiting for a timeout.

Generic finite state machine (gen_fsm). Many concurrent algorithms are specified in terms of a finite state machine model. The OTP library provides a convenient behaviour for this pattern. The message protocol that it obeys provides for clients to signal events to the state machine, possibly waiting for a synchronous reply. The application-specific callbacks handle

these events, receiving the current state and passing a new state as a return value.

Generic event handler (gen_event). An event manager is a process that receives events as incoming messages, then dispatches those events to an arbitrary number of event handlers, each of which has its own module of callback functions and its own private state. Handlers can be dynamically added, changed, and deleted. Event handlers run application code for events, frequently selecting a subset to take action upon and ignoring the rest. This behaviour naturally models logging, monitoring, and "pubsub" systems. The OTP library provides off-the-shelf

FIGURE 4

A
```
% Make a set of server calls in parallel and return a
% list of their corresponding results.
% Calls is a list of {Server, Params} tuples.
multicall1(Calls) ->
    Ids = [send_call(Call) || Call <- Calls],
    collect_replies(Ids).

% Send a server call request message.
send_call({Server, Params}) ->
    Id = make_ref(),
    Server ! {call, {self(), Id}, Params},
    Id.

% Collect all replies in order.
collect_replies(Ids) ->
    [receive {Id, Result} -> Result end || Id <- Ids].
```

B
```
multicall2(Calls) ->
    Parent = self(),
    Pids = [worker(Parent, Call) || Call <- Calls],
    wait_all(Pids).

worker(Parent, {Server, Params}) ->
    spawn(fun() ->      % create a worker process
        Result = server:call(Server, Params),
        Parent ! {self(), Result}
    end).

wait_all(Pids) ->
    [receive {Pid, Result} -> Result end || Pid <- Pids].
```

event handlers for spooling events to files or to a remote process or host.

The behaviour libraries provide functionality for dynamic debugging of a running program. They can be requested to display the current behaviour state, produce traces of messages received and sent, and provide statistics. Having this functionality automatically available to all applications gives Erlang programmers a profound advantage in delivering production-quality systems.

WORKER PROCESSES

Erlang applications can implement most of their functionality using long-lived processes that naturally fit a standard behaviour. Many applications, however, also need to create concurrent activities on the fly, often following a more ad-hoc protocol too unusual or trivial to be captured in the standard libraries.

Suppose we have a client that wants to make multiple server calls in parallel. One approach is to send the server protocol messages directly, shown in figure 4A. The client sends well-formed server call messages to all servers, then collects their replies. The replies may arrive in the inbox in any order, but collect_replies/1 will gather them in the order of the original list. The client may block waiting for the next reply even though other replies may be waiting. This doesn't slow things down, however, since the speed of the overall operation is determined by the slowest call.

To reimplement the protocol, we had to break the abstraction that the server behaviour offered. While this was simple for our toy example, the production-quality generic server in the Erlang standard library is far more involved. The setup for monitoring the server processes and the calculations for timeout management would make this code run on for several pages, and it would need to be rewritten if new features were added to the standard library.

Instead, we can reuse the existing behaviour code entirely by using *worker processes*—short-lived, special-purpose processes that don't execute a standard behaviour. Using worker processes, this code becomes that shown in figure 4B.

We spawn a new worker process for each call. Each makes the requested call and then replies to the parent, using its own pid as a tag. The parent then receives each reply in turn, gathering them in a list. The client-side code for a server call is reused entirely as is.

By using worker processes, libraries are free to use receive expressions as needed without worrying about blocking their caller. If the caller does not wish to block, it is always free to spawn a worker.

DANGERS OF CONCURRENCY

Though it eliminates shared state, Erlang is not immune to races. The server behaviour allows its application code to execute as a critical section accessing protected data, but it's always possible to draw the lines of this protection incorrectly.

For example, if we had implemented sequences with raw primitives to read and write the counter, we would be just as vulnerable to races as a shared-state implementation that forgot to take locks:

badsequence.erl
```
% BAD - race-prone implementation - do not use - BAD
-module(badsequence).
-export([make_sequence/0, get_next/1, reset/1]).
-export([init/0, handle_call/2, handle_cast/2]).

% API
make_sequence()    -> server:start(badsequence).
get_next(Sequence) ->
   N = read(Sequence),
   write(Sequence, N + 1), % BAD: race!
   N.
reset(Sequence)    -> write(Sequence, 0).
read(Sequence)     -> server:call(Sequence, read).
write(Sequence, N) -> server:cast(Sequence, {write, N}).

% Server callbacks
init()                 -> 0.
handle_call(read, N)       -> {N, N}.
handle_cast({write, N}, _) -> N.
```

This code is insidious as it will pass simple unit tests and can perform reliably in the field for a long time before it silently encounters an error. Both the client-side wrappers and server-side callbacks, however, look quite different from those of the correct implementation. By contrast, an incorrect shared-state program would look nearly identical to a correct one. It takes a trained eye to inspect a shared-state program and notice the missing lock requests.

All standard errors in concurrent programming have their equivalents in Erlang: races, deadlock, livelock, starvation, and so on. Even with the help Erlang provides, concurrent programming is far from easy, and the nondeterminism of concurrency means that it is always difficult to know when the last bug has been removed.

Testing helps eliminate most gross errors—to the extent that the test cases model the behaviour encountered in the field. Injecting timing jitter and allowing

Erlang
for Concurrent Programming

long burn-in times will help the coverage; the combinatorial explosion of possible event orderings in a concurrent system means that no nontrivial application can be tested for all possible cases.

When reasonable efforts at testing reach their end, the remaining bugs are usually heisenbugs,[5] which occur nondeterministically but rarely. They can be seen only when some unusual timing pattern emerges in execution. They are the bane of debugging since they are difficult to reproduce, but this curse is also a blessing in disguise. If a heisenbug is difficult to reproduce, then if you rerun the computation, you might not see the bug. This suggests that flaws in concurrent programs, while unavoidable, can have their impact lessened with an automatic retry mechanism—as long as the impact of the initial bug event can be detected and constrained.

FAILURE AND SUPERVISION

Erlang is a safe language—all runtime faults, such as division by zero, an out-of-range index, or sending a message to a process that has terminated, result in clearly defined behavior, usually an exception. Application code can install exception handlers to contain and recover from expected faults, but an uncaught exception means that the process cannot continue to run. Such a process is said to have failed.

Sometimes a process can get stuck in an infinite loop instead of failing overtly. We can guard against stuck processes with internal watchdog processes. These watchdogs make periodic calls to various corners of the running application, ideally causing a chain of events that cover all long-lived processes, and fail if they don't receive a response within a generous but finite timeout. Process failure is the uniform way of detecting errors in Erlang.

Erlang's error-handling philosophy stems from the observation that any robust cluster of hardware must consist of at least two machines, one of which can react to the failure of the other and take steps toward recovery.[6] If the recovery mechanism were on the broken machine, it would be broken, too. The recovery mechanism must be outside the range of the failure. In Erlang, the process is not only the unit of concurrency, but also the range of

failure. Since processes share no state, a fatal error in a process makes its state unavailable but won't corrupt the state of other processes.

Erlang provides two primitives for one process to notice the failure of another. Establishing *monitoring* of another process creates a one-way notification of failure, and *linking* two processes establishes mutual notification. Monitoring is used during temporary relationships, such as a client-server call, and mutual linking is used for more permanent relationships. By default, when a fault notification is delivered to a linked process, it causes the receiver to fail as well, but a process-local flag can be set to turn fault notification into an ordinary message that can be handled by a receive expression.

In general application programming, robust server deployments include an external "nanny" that will monitor the running operating-system process and restart it if it fails. The restarted process reinitializes itself by reading its persistent state from disk and then resumes running. Any pending operations and volatile state will be lost, but assuming that the persistent state isn't irreparably corrupted, the service can resume.

The Erlang version of a nanny is the *supervisor* behaviour. A supervisor process spawns a set of child processes and links to them so it will be informed if they fail. A supervisor uses an initialization callback to specify a strategy and a list of child specifications. A child specification gives instructions on how to launch a new child. The strategy tells the supervisor what to do if one of its children dies: restart that child, restart all children, or several other possibilities. If the child died from a persistent condition rather than a bad command or a rare heisenbug, then the restarted child will just fail again. To avoid looping forever, the supervisor's strategy also gives a maximum rate of restarting. If restarts exceed this rate, the supervisor itself will fail.

Children can be normal behaviour-running processes, or they can be supervisors themselves, giving rise to a tree structure of supervision. If a restart fails to clear an error, then it will trigger a supervisor subtree failure, resulting in a restart with an even wider scope. At the root of the supervision tree, an application can choose the overall strategy, such as retrying forever, quitting, or possibly restarting the Erlang virtual machine.

Since linkage is bidirectional, a failing server will notify or fail the children under it. Ephemeral worker processes are usually spawned linked to their long-lived parent. If the parent fails, the workers automatically fail, too. This linking prevents uncollected workers from accumulating in the system. In a properly written Erlang

application, all processes are linked into the supervision tree so that a top-level supervision restart can clean up all running processes.

In this way, a concurrent Erlang application vulnerable to occasional deadlocks, starvations, or infinite loops can still work robustly in the field unattended.

IMPLEMENTATION, PERFORMANCE, AND SCALABILITY

Erlang's concurrency is built upon the simple primitives of process spawning and message passing, and its programming style is built on the assumption that these primitives have a low overhead. The number of processes must scale as well—imagine how constrained object-oriented programming would be if there could be no more than a few hundred objects in the system.

For Erlang to be portable, it cannot assume that its host operating system has fast interprocess communication and context switching or allows a truly scalable number of schedulable activities in the kernel. Therefore, the Erlang *emulator* (virtual machine) takes care of scheduling, memory management, and message passing at the user level.

An Erlang instance is a single operating-system process with multiple operating-system threads executing in it, possibly scheduled across multiple processors or cores. These threads execute a user-level scheduler to run Erlang processes. A scheduled process will run until it blocks or until its time slice runs out. Since the process is running Erlang code, the emulator can arrange for the scheduling slice to end at a time when the process context is minimal, minimizing the context switch time.

Each process has a small, dedicated memory area for its heap and stack. A two-generation copying collector reclaims storage, and the memory area may grow over time. The size starts small—a few hundred machine words—but can grow to gigabytes. The Erlang process stack is separate from the C runtime stack in the emulator and has no minimal size or required granularity. This lets processes be lightweight.

By default, the Erlang emulator interprets the intermediate code produced by the compiler. Many substantial Erlang programs can run sufficiently fast without using the native-code compiler. This is because Erlang is a high-level language and deals with large, abstract objects. When running, even the interpreter spends most of its time executing within the highly tuned runtime system written in C. For example, when copying bulk data between network sockets, interpreted Erlang performs on par with a custom C program to do the same task.[7]

The significant test of the implementation's efficiency is the practicality of the worker process idiom, as demonstrated by the multicall2 code shown earlier. Spawning worker processes would seem to be much less efficient than sending messages directly. Not only does the parent have to spawn and destroy a process, but also the worker needs extra message hops to return the results. In most programming environments, these overheads would be prohibitive, but in Erlang, the concurrency primitives (including process spawning) are lightweight enough that the overhead is usually negligible.

Not only do worker processes have negligible overhead, but they also increase efficiency in many cases. When a process exits, all of its memory can be immediately reclaimed. A short-lived process might not even need a collection cycle. Per-process heaps also eliminate global collection pauses, achieving soft realtime levels of latency. For this reason, Erlang programmers avoid reusable pools of processes and instead create new processes when needed and discard them afterward.

Since values in Erlang are immutable, it's up to the implementation whether the message is copied when sent or whether it is sent by reference. Copying would seem to be the slower option in all situations, but sending messages by reference requires coordination of garbage collection between processes: either a shared heap space or maintenance of inter-region links. For many applications, the overhead of copying is small compared with the benefit from short collection times and fast reclamation of space from ephemeral processes. The low penalty for copying is driven by an important exception in send-by-copy: raw binary data is always sent by reference, which doesn't complicate garbage collection since the raw binary data cannot contain pointers to other structures.

The Erlang emulator can create a new Erlang process in less than a microsecond and run millions of processes simultaneously. Each process takes less than a kilobyte of space. Message passing and context switching take hundreds of nanoseconds.

Because of its performance characteristics and language and library support, Erlang is particularly good for:
• Irregular concurrency—applications that need to derive parallelism from disparate concurrent tasks
• Network servers
• Distributed systems
• Parallel databases
• GUIs and other interactive programs
• Monitoring, control, and testing tools

So when is Erlang *not* an appropriate programming language, for efficiency or other reasons? Erlang tends not to be good for:

Erlang
for Concurrent Programming

- Concurrency more appropriate to synchronized parallel execution
- Floating-point-intensive code
- Code requiring nonportable instructions
- Code requiring an aggressive compiler (Erlang entries in language benchmark shoot-outs are unimpressive—except for process spawning and message passing)
- Projects to implement libraries that must run under other execution environments, such as JVM (Java Virtual Machine) or CLR (Common Language Runtime)
- Projects that require the use of extensive libraries written in other languages

Erlang can still form part of a larger solution in combination with other languages, however. At a minimum, Erlang programs can speak text or binary protocols over standard interprocess communication mechanisms. In addition, Erlang provides a C library that other applications can link with that will allow them to send and receive Erlang messages and be monitored by an Erlang controlling program, appearing to it as just another (Erlang) process.

CONCLUSION

With the increasing importance of concurrent programming, Erlang is seeing growing interest and adoption. Indeed, Erlang is branded as a "concurrency-oriented" language. The standard Erlang distribution is under active development. Many high-quality libraries and applications are freely available for:

- Network services
- GUIs for 3D modeling
- Batch typesetting
- Telecom protocol stacks
- Electronic payment systems
- HTML and XML generation and parsing
- Database implementations and ODBC (Open Database Connectivity) bindings

Several companies offer commercial products and services implemented in Erlang for telecom, electronic payment systems, and social networking chat. Erlang-based Web servers are notable for their high performance and scalability.[8]

Concurrent programming will never be easy, but with Erlang, developers have a chance to use a language built from the ground up for the task and with incredible resilience engineered in language, runtime system, and standard libraries.

The standard Erlang implementation and its documentation, ported to Unix and Microsoft Windows platforms, is open source and available for free download from http://erlang.org. You can find a community forum at http://trapexit.org, which also mirrors several mailing lists. Q

REFERENCES

1. Erlang Web site; http://erlang.org.
2. Armstrong, J. 2003. Making reliable distributed systems in the presence of software errors. Ph.D. thesis, Swedish Institute of Computer Science; http://www.erlang.org/download/armstrong_thesis_2003.pdf.
3. Erlang course; http://www.erlang.org/course/course.html.
4. See reference 2.
5. Steele, G. L., Raymond, E. S. 1996. *The New Hacker's Dictionary*, 3rd edition, Cambridge, MA: MIT Press.
6. Armstrong, J. 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, NC: The Pragmatic Bookshelf.
7. Lystig Fritchie, S., Larson, J., Christenson, N., Jones, D., Ohman, L. 2000. Sendmail meets Erlang: Experiences using Erlang for email applications. Erlang User's Conference; http://www.erlang.se/euc/00/euc00-sendmail.ps.
8. Brown, B. 2008. Application server performance testing, includes Django, ErlyWeb, Rails, and others; http://berlinbrowndev.blogspot.com/2008/08/application-server-benchmarks-including.html.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

JIM LARSON is a software engineer at Google. He has worked with Erlang for commercial products off and on since 1999. He was the architect of the replication engine of the Amazon SimpleDB Web service at Amazon.com. He previously worked at Sendmail Inc. and the Jet Propulsion Laboratory. He has a B.A. in mathematics from Saint Olaf College, M.S. in mathematics from Claremont Graduate University, and M.S. in computer science from the University of Oregon.