The Computer **Nonsense** Guide

Or the influence of chaos on reason! 27.03.2018

Contents

Abstract

My aim is: to teach you to pass from a piece of disguised nonsense to something that is patent nonsense.

- Ludwig Wittgenstein

The Computer Nonsense Guide describe both the languages and the nonsense operating system.

The software environment and operating-system-like parts contain many things which are still in a state of flux. This work confines itself primarily to the stabler parts of the system, and does not address the window system, user interface and application programming interface at all.

This guide is product of the efforts of many people too numerous to list here and of the unique environment of the Nonsense Worlds, Inc. Artificial Intelligence Laboratory.

We are an open-source multidisciplinary research laboratory that conducts work on distributed systems, artificial intelligence and high-performance computing.

Our Mission: driven by technical nonsense we focus on multi-dimensional research providing tools inside a simple unix-like workspace for play, work and science that build predictive models of the environment through observation and action.

Our Goal: provide a distributed AI toolkit and workspace environment for machines of all ages!

We build on top of Debian plus all computer nonsense tools like additional semi-autonomous assistant, custom tiling window interface and heavy focus on LFE.

We make a stable workspace that anyone can use today, at nonsense these things work together into one unified environment with native support for Python 3, LuaLang and the BEAM ecosystem.

Prerequisites

It is assumed that the reader has done some programming and is familiar with concepts such as data types and programming language syntax.

Core ideas

In essence our model of computation is compatible with the Actor model theory we build up on the next five core ideas.

- Lightweight, isolated, millions of processes are possible on one machine.
- Asynchronous message passing is necessary for non-blocking systems.
- Selective receive mechanisms allow you to ignore messages which are uninteresting now.
- Message passing is a form of function calling or function calling a form of message-passing.
- Objets are a form of functions or functions a form of objects.

Objects and functions

An object is really a function that has no name and that gets its argument a message and then look at that message and decide what to do next.

Getting started

Please make sure that your system have the latest releases of Erlang, LuaJIT (with luarocks) and Singularity 3 or newer version installed.

Installation

Then run this command:

luarocks install cube-cli

For help using cube-cli, including a list of commands, run:

$ cube-cli --help

Congratulations, you are jacked up and good to go!

Hello Luerl

Luerl is an implementation of standard Lua 5.2 written in Erlang/OTP.

Lua is a powerful, efficient, lightweight, embeddable scripting language common in games, eSports, IoT devices, system infrastructure, machine learning and scientific computing research.

It supports procedural, object-oriented, functional, data-driven, reactive, organizational programming and data description.

Being an extension language, Lua has no notion of a "main" program: it works as a library embedded in a host. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and call Erlang functions by Lua code.

Luerl is a library, written in clean Erlang/OTP. For more information, check out the get started tutorial. You may want to browse the examples and learn from the luerl_demo source code.

The goal

A proper implementation of the Lua language

- It should look and behave the same as Lua
- It should include the standard libraries
- It MUST interface well with Erlang

The result

Luerl implements all of Lua 5.2 except goto, _ENV and coroutines.

- Easy for Erlang to call
- Easy for Lua to call Erlang
- Erlang concurrency model and error handling

Through the use of the BEAM ecosystem, Luerl can be augmented to cope with a wide range of different domains, creating a customized language sharing a syntactical framework.

The Luerl API

Lua is an embeddable language implemented as a library that offers a clear API for applications inside a register-based virtual machine.

This ability to be used as a library to extend an application is what makes Lua an extension language.

At the same time, a program that uses Lua can register new functions in the Luerl environment; such functions are implemented in Erlang (or another language) and can add facilities that cannot be written directly in Lua. This is what makes any Lua implementation an extensible language.

These two views of Lua (as extension language and as extensible language) correspond to two kinds of interaction between Erlang and Lua. In the first kind, Erlang has the control and Lua is the library. The Erlang code in this kind of interaction is what we call application code.

In the second kind, Lua has the control and Erlang is the library. Here, the Erlang code is called library code. Both application code and library code use the same API to communicate with Lua, the so called Luerl API.

Modules, Object Oriented programming and iterators need no extra features in the Lua API. They are all done with standard mechanisms for tables and first-class functions with lexical scope.

Exception handling and code load go the opposite way: primitives in the API are exported to Lua from the base system C, JIT, BEAM.

Lua implementations are based on the idea of closures, a closure represents the code of a function plus the environment where the function was defined.

Like with tables, Lua itself uses functions for several important constructs in the language.

The use of constructors based on functions helps to make the API simple and general.

There are no coroutines in Luerl it may seems counter intuitive coming from a more common Lua background.

In this ecosystem you always want to use processes instead, the BEAM Virtual Machine it's build for handling independent isolated processes that are very small and almost free at creation time and context switching. The main difference between processes and coroutines is that, in a multiprocessor machine a OTP release on the BEAM Virtual Machine runs several processes concurrently in parallel.

Coroutines, on the other hand, runs only one at the time on a single core and this running coroutine only suspends its execution when it explicitly requests to be suspended.

Hello LFE

Good news, bad news and how to WIN BIG ™.

LFE tried Lisp 1 but it didn't really work, Lisp 2 fits the BEAM better so LFE is Lisp 2+, or rather Lisp 3?

LFE is a proper Lisp based on the features and limitations of the BEAM Virtual Machine, attuned to vanilla Erlang and OTP it coexists seamlessly with the rest of the ecosystem.

The bad new is that almost everything that what we have been using is WORNG! and yet we still don't know all of the smelly parts yes if you are a business the JVM ecosystem make sense, sure if you build websites Ruby and Elixir are one of the obvious choices, same apply to classical mathematics, no one can denied the respect that The Black Mesa Research deserves but even with it's limited parallelism implemented by a communicating sequential processes language, full of monads or types have yet it classical boundaries, the ceiling is to low and it shows.

We were not that into Lisp until reading some tweets from certain no-horn vikings what is shameful so forget me if we resume a little bit, Lisp 1.5 din't have anonymous functions with lexical scope 'closures' for short, Scheme is Lisp 2, the goal of Scheme was to implement a Lisp following the actor model computation but they discover closures instead got hyped with them and forget about the actor-model goal.

Erlang from Stockholm, Sweden since the 90's a team of 5 giants, Bjarne Däcker, Jane Walerud, Mike Williams, Joe Armstrong and Robert Virding open-source a language that implement this academic model of universal computation without even know about it, just pure engineering and a great problem to solve.

It's a language out of a language out of Sweden that can be used to build web scale, asynchronous, non-blocking, event driven, message passing, NoSQL, reliable, highly available, high performance, real time, clusterable, bad ass, rock star, get the girls, get the boys, impress your mom, impress your cat, be the hero of your dog, AI applications.

It's Lisp, you can blast it in the face with a shotgun and it keeps on coming.

The goal

An efficient implementation of a "proper" Lisp on the BEAM with seamless integration for the Erlang/OTP ecosystem.

The result

A New Skin for the Old Ceremony where the thickness of the skin affects how efficiently the new language can be implemented and how seamlessly it can interact.

Why Lisp?

A lot has changed since 1958, even for Lisp it new has even more to offer.

- It's a programmable programming language
- As such, it's excellent language for exploratory programming.
- Due to it's venerable age, there is an enormous corpus of code to draw from.

History

The original idea was to produce a compiler, but in the 50's this was considered a major undertaking, and McCarthy and his team needed some experimenting in order to get good conventions for subroutine linking, stack handling and erasure.

They started by hand-compiling various functions into assembly language and writing subroutines to provide a LISP environment.

They decided on garbage collection in which storage is abandoned until the free storage list is exhausted, the storage accessible from program variables and the stack is marked, so the unmarked storage is made into a new free storage list.

At the time was also decided to use SAVE and UNSAVE routines that use a single contiguous public stack array to save the values of variables and subroutine return addresses in the implementation of recursive subroutines.

Another decision was to give up the prefix and tag parts of the message, this left us with a single type an 15 bit address, so that the language didn't require declarations.

These simplifications made Lisp into a way of describing computable functions much neater than the Turing machines or the general recursive definitions used in recursive function theory.

The fact that Turing machines constitute an awkward programming language doesn't much bother recursive function theorists, because they almost never have any reason to write particular recursive definitions since the theory concerns recursive functions in general.

Another way to show that Lisp was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing Machine.

This refers to the Lisp function eval(e,a) which computes the value of a Lisp expression e, the second argument a being a list of assignments of values to variables, a is needed to make the recursion work.

Objects

When writing a program, it is often convenient to model what the program does in term of objects, conceptual entities that can be likened to real-world things.

Choosing what objects to provide in a program is very important to the proper organization of the program.

In an object-oriented design, specifying what objects exist is the first task in designing the system.

In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows".

After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it.

More rigorously, the program defines several types of object, and it can create many instances of each type.

The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of that type.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as get and put are defined.

This type can be instantiated by evaluating this form you can create a new disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view.

However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask "Is this object a disembodied property list, as opposed to an ordinary list?".

We can say that the object keeps track of an internal state, which can be examined and altered by the operations available for that type of object.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as get and put are defined.

This type can be instantiated by evaluating this form you can create a new disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view.

However, such conceptual data types cannot be distinguished automatically be the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

We represent our conceptual object by one structure.

The LFE flavors we use for the representation has structure and refers to other Lisp objects.

We can say that the object keeps track of an internal state which can be examined and altered by the operations available for that type of object, get examines the state of a property list, and put alters it.

We have seen the essence of object-oriented programming. A conceptual object is modeled by a single Lisp object, which bundles up some state information.

For every type of object there is a set of operations that can be performed to examine or alter the state of the object.

Pura Vida

Overall, the evolution of Lisp has been, guided more by institutional rivalry, one-upmanship, and the glee born of technical cleverness characteristic of the hacker culture than by sober assessment of technical requirements.

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in LFE and used by the Computer Nonsense software system. Its purpose is to perform generic operations on objects.

Part of its implementation is simply a convention in procedural-calling style: part is a powerful language feature, called flavors, for defining abstract objects.

The early MIT Lisp Machine Lisp dialect was very similar to MacLisp. It lived up to its goal of supporting MacLisp programs with only minimal porting effort.

The most important extensions beyond MacLisp included: Flavors, an object-oriented, non-hierarchical programming environment the mythical lisp machine window system in particular, was written using Flavors Weinreb, 1981.

Lisp 1

Early thoughts about a language that eventually became Lisp started in 1956 when John McCarty attended the Dartmouth Summer Research Project on Artificial Intelligence. *Actual implementation began in the fall of 1958*.

Lisp 2

An exception to all was the Lisp 2 project, a concerted language that represented a radical departure from Lisp 1.5. *In contrast to most languages, in which the language is first designed and then implemented Lisp 2 was an implementation in search of a language*, in retrospect we can point out that was searching specifically from one out of Sweden.

The earliest known LISP 2 document is a one-page agenda for a Lisp 2 Specifications Conference held by the Artificial Intelligence Group at Standford. Section 2 of this agenda was "Proposals for Lisp 2.0", and included:

- Linear Free Storage
- Numbers and other full words
- Auxiliary Storage
- Input language, infix notation.
- Arrays
- Freer output format
- Sequence of implementation
- Comments
- Documentation and maintenance
- Hash Coding
- Sobroutine linkage
- Storage conventions
- Effect of various I/O apparatus
- Interaction with programs in other languages
- Expressions having property lists

LISP 3

Lisp Flavored Erlang (LFE) is a functional, concurrent, general-purpose programming language and Lisp dialect built on top of Core Erlang and the Erlang Virtual Machine (BEAM).

What isn't

- It isn't an implementation of Maclisp
- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp
- It isn't an implementation of Clojure

What is

- LFE is a proper Lisp based on the features and limitations of the Erlang VM (BEAM).
- LFE coexists seamlessly with vanilla Erlang/OTP and the rest of the BEAM ecosystem.
- LFE runs on the standard Erlang Virtual Machine (BEAM).

Singularity Containers

Singularity enables full control of the environment on whatever host you are on. This include distributed systems, your favorite blockchain,HPC centers, resource & scheduler managers, file systems, GPU's, IoT edge devices, docker containers and the whole computing enchilada.

Containers are used to package entire scientific workflows, software libraries, and datasets.

Did you already invest in Docker? The Singularity software can import your Docker images without having Docker installed or being a superuser.

As the user, you are in control of the extent to which your container interacts with its host. There can be seamless integration, or little to no communication at all.

- Reproducible software stacks: These must be easible verifiable via cheksum or cryptographic signature in such a manner that does not change formats. By default Singularity uses a container image file which can be checksummed, signed and easily verified.
- Mobility of compute: Singularity must be able to transfer (and store) containers in a manner that works with stadandard data mobility tools and protocols.
- Compatibility with complicated architectures: The runtime must be compatible with existing HPC, scientific, compute farm and enterprise architectures maybe running legacy vintage systems which do not support advanced namespace features.

Container instances

Singularity has support for container instances, which means services!

Images instances can be started, stopped, and listed.

Along with instances comes **Network Namespace Isolation**

Background

A Unix operating system is broken into two primary components, the kernel space, and the user space. The kernel supports the user space by interfacing with the hardware, providing core system features and creating the software compatible layers for the user space. The user space on the other hand is the environment that most people are most familiar with interfacing with. It is where applications, libraries and system services run.

Containers shift the emphasis away from the run-time environment by commoditizing the user space into swappable components. This means that the entire user space portion of a Linux operating system, including programs, custom configuration, and environment can be interchangeable at run-time.

Software developers can now build their stack onto whatever operating system base fits their needs bets, and create distributable run-time encapsulated environments and the users never have to worry about dependencies, requirements, or anything else from the user space.

Singularity provides the functionality of a virtual machine, without the heavyweight implementation and performance costs of emulation and redundancy!

Apps

What if we want a single container with three or eight different apps that each have their own runscripts and custom environments? It may be redundant to build different containers for each app with almost equivalent dependencies, based on the Standard Container Integration Format. For details on apps, see the apps documentation.

StarCraft: Brood War AI

StarCraft serve as an interesting domain for Artificial Intelligence (AI) research, since represent a well defined complex adversarial system which pose a number of interesting AI challenges in areas of planning, dealing with uncertainty, domain knowledge exploitation, task decomposition, spacial reasoning, and machine learning.

Unlike synchronous turn-based games like chess and go, StarCraft games are played in real-time, meaning the state will continue to progress even if the player takes no action, and so actions must be decided in fractions of a second, game frames can consist of issuing simultaneous actions to hundreds of units at any given time.

Core ideas

- StarCraft is about MORE MINERALS.
- Strong armies vs mobile armies.

Stages of a game

- Early, Make/defend a play & double expand if you can.
- Middle, Core armies, make/defend pressure & take a base.
- Late, Matured core army, multi-pronged tactics & take many bases.
- Final, The watcher observes, the fog collapses, an event resolves.

Meet blueberry

Read the Quick Start, join the nonsense, build your own bot and compete in the tournaments.

Iterative programming

Software development (programming) is an iterative process, **iterative** is another name for *"Intelligent Trial and Error"*.

Programing is an iterative process with a large amount of trial and error to find out

- What needs to be implemented
- Why does it need to be implemented
- How should be implemented

Erlang is ideally suited iterative development specially when your application requires concurrency.

A primitive of problem solving, characterized by repeated, varied attempts which are continued until success, or until the agent stops trying.

- If you don't know what is wanted, you have to find out by a lot of trial and error.
- If you don't know how to do it, you have to find out by a lot of trial and error.
- Trial and error is unpredictable, must humans don't like unpredictable things.
- Some of us hate being asked to predict the unpredictable.

Error handling

Bugs are a natural part of programming, if there are a lot of bugs in one part of the system, find out why. Errors will always occur! "Let it crash" model separate failure recovery code from application code.

Primitives

- Links between processes
- Exit signals sent along links when a process terminates
- Processes can trap exits

Methodology

The existence of different available strategies allows us to consider a separate superior domain of processing, a "meta-level" above the mechanics of switch handling from where the various available strategies can be randomly chosen.

Dynamic programming

Dynamic programming is when you use past knowledge to make solving a future problem easier.

With dynamic programming, you store your results in some sort of lookup table. When you need the answer to a problem, you reference the table and see if you already know what it is. If not, you use the data in your table to give yourself a stepping stone towards the answer.

The technique of storing solutions to subproblems instead of recomputing them is called "memoization". A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem.

Tornado

Tornado is a Python web framework and asynchronous networking library. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for applications that require a long-lived connection to each user.

Tornado can be roughly divided into four major components:

- Client and server side implementions of HTTP (HTTPServer and AsyncHTTPClient).
- An asynchronous networking library including the classes IOLoop and IOStream, which serve as the building blocks for the HTTP components and can also be used to implement other protocols.
- A coroutine library (tornado.gen) which allows asynchronous code to be written in a more straightforward way than chaining callbacks.

The Tornado web framework and HTTP server together offer a full-stack alternative to WSGI.

Turbo

Turbo.lua is a framework built for LuaLang to simplify the task of building fast and scalable network applications. It uses a event-driven, non-blocking, no thread design and minimal footprint to high-load applications while also providing excellent support for embedded uses.

It’s main features and design principles are:

- Implemented in straight Lua and LuaJIT FFI on Linux.
- Event driven, asynchronous, threadless design.
- Good documentation, Small footprint.

The framework contains generic nuts and bolts such as; a IOLoop and IOStream classes giving value for everyone doing any kind of high performance network application.

Process management

Traditionally, Tornado and Turbo apps are single-processes and require external management behind a process supervisor and nginx (openresty) for (proxying) load balance.

Organizational programming

An monkey, a building, an automobile, a drone: each is a concrete object and can be easily identified. One difficulty attending the study of organizations is that an organization is not as readily visible or describable.

Exactly what is an organization such as a business concern? It is a building? A collection of machinery? A legal document containing a statement of incorporation? It is hardly likely to be any of these by itself. Rather, to describe an organization requires the consideration of a number of properties it possesses, thus gradually making clear, or at least clearer, that it is.

The purposes of the organization, whether it is formal or informal, are accomplished by a collection of people whose efforts or to use a term to be employed throughout this work, behavior are so directed that they become coordinated and integrated in order to attain sub-goals and objectives.

Perception and behavior

All of us humans at some point or another have had the experience of watching another person do something or behave in a certain way, saying to ourselves, "She/he acts as if she/he thought, ... " and then filling in some supposition about the way the other person looked at things.

Simple as the statement "He acts as if he thought ... " may be, it illustrates two important points.

First, what the person thinks he sees may not actually exist. They could act as if changes in methods as an attempt by management to exploit them.

As long as they had this attitude or belief, any action by management to change any work method would be met, at the very least, with suspicion and probably with hostility.

The second point is that people act on the basis of what they see. In understanding behavior, we must recognize that facts people do not perceive as meaningful usually will not influence their behavior, whereas the things they believe to be real, even though factually incorrect or nonexistent, will influence it.

Organizations are intended to bring about integrated behavior. Similar, or at least compatible, perceptions on the part of organizational members are therefore a matter of prime consideration.

Cues

One of the first things we must recognize is that in learning about things we not only learn what they are, that is, that the round white object is for football, but we also learn what these things mean, that is, football is a sport that the USA men's team don't get and their woman counterpart have master perfectly.

Upon receiving a signal (sight of football) we perform an interpretative step by which a meaning is attached to it.

Many of these "meanings" are so common and fundamental in our understanding of the world that we fail to note them except under unusual circumstances.

One way these meanings are brought home to us is by meeting people from countries different from our own; many of the meanings which things have come from our culture, they are things all people within the culture share.

These common interpretations of things help enormously in communicating, but they sometimes make it difficult to set factors in perspective so that we can really understand the reasons for behavior.

For example "Holding a watch against them" for many people in industry has the same emotional content as "taking a whip to them".

Here we can see how the same device can have completely different meaning for groups which come quite close together.

Threshold of perception

We all, have certain things (stimuli) to which we are sensitized and that when these appear we are instantly alert and eager to examine them.

There are other stimuli of relative unimportant to us to which we do not pay as much attention and may, in effect, actually block out.

One way of viewing this subject is to suggest that we have thresholds or barriers which regulate what information from the outside world reaches our consciousness.

On some matters the barriers are high and we remain oblivious to them, but on others which are quite important to us we are sensitized and, in effect, we lower the barrier, permitting all the information possible concerning these matters to reach our consciousness.

Resonance

Related to this idea of sensitivity and selectivity is a phenomenon that might be called resonance.

Through experience and what we see ourselves to be, the understanding of a particular item of information may be very similar to that of others.

It is explained this way: since all the people inside a group look upon themselves as peers, they know what a change on the individual means in annoyance and inconvenience.

They can easily put themselves into his shows and, once having done so, probably feel almost as disturbed as he might be.

Internal consistency

One property of the images formed of the world around us is that they are reasonable, or internally consistent.

For instance, we may look at some draw on a page and see a rabbit. One portion along these lines might suggest a duck, but we do not have an image of something half rabbit and half duck.

In fact, if our first impression is of a duck, we may never notice that a portion looks like a rabbit.

We seem to tune out the elements that do not fit.

Differentiable programming

Marrying Deep learning with Reasoning!

You want to test out a new machine learning model for your data. This usually means coming up with some loss function to capture how well your model fits the data and optimizing the loss with respect to the model parameters. If there are many model parameters (neural nets can have millions) then you need gradients. You then have two options: derive and code them up yourself, or implement your model using syntactic and semantic constraints of a system like TensorFlow.

We present a third way: just write down the loss function using a standard numerical library like Torch, Numpy, and Autograd will give you its gradient.

Autograd's takes in a function, and gives you a function that computes its derivative, Your function must have a scalar-valued output (i.e. a float). This covers the common case when you want to use gradients to optimize something.

Chainer

Chainer adopts a *Define-by-Run* scheme, the network is defined on-the-fly via the actual forward computation. Chainer stores the history of computation instead of programming logic.

Chainer represents a network as an execution path on a computational graph. A computational graph is a series of function applications, so that it can be described with multiple Function objects.

Torch

Torch is a scientific computing framework with wide support for machine learning that puts GPUs first. It is easy to use and efficient, thanks to LuaLang and an underlying C/CUDA implementation.

A summary of core features:

- a powerful N-dimensional array
- linear algebra routines
- neural network, and energy-based models
- Fast and efficient GPU support
- Embeddable, with ports to iOS, Android and FPGA backends

Torch comes with a large ecosystem of community-driven packages in machine learning, computer vision, signal processing, parallel processing, image, video and audio among others, and builds on top of the Lua community.

PyTorch

PyTorch is a python package that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autograd system

You can reuse your favorite python packages such as numpy, scipy and Cython to extend PyTorch when needed.

Usually one uses PyTorch either as:

- A replacement for numpy to use the power of GPUs.
- a deep learning research platform that provides maximum flexibility and speed

Supervision trees

A supervisor has a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build a hierarchical process structure called a supervision tree, a nice way to structure a fault-tolerant application.

- Supervisors will monitor their processes through links and trapping exists.
- Supervisors can restart the workers when they terminate.

On production, this usually means a fairly straight-forward combination of external process management, overload monitoring and proxying.

A supervisor is responsible for starting, stopping, and monitoring external processes. The basic idea of a supervisor is that it is to keep its processes alive by restarting them when necessary.

Heart

The purpose of the heart program is to check that the Erlang runtime system it is supervising is still running. If the program has not received any heartbeats within HEART_BEAT_TIMEOUT seconds (defaults to 60 seconds), the Erlang system will be rebooted.

Circus

Circus is a Python program which can be used to monitor and control processes and sockets.

Circus can be driven via a command-line interface, a web interface or through its API.

Supervisor

Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems.

Monit

Monit is a small Open Source utility for managing and monitoring Unix systems. Monit conducts automatic maintenance and repair and can execute meaningful causal actions in error situations.

Pipes

The pipe location in your home is important for proper maintenance and water flow. Many pipes are located in walls, floors and ceilings and are hard to locate.

One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the pipe, as used in a pipeline of commands.

The fundamental idea was by no means new; the pipeline is merely a specific form of coroutine.

Pipes appeared in Unix in 1972, well after the PDP-11 version of the system was in operation, at the insistence of M.D McIlroy, a long advocate of the non-hierarchical control flow that characterizes coroutines.

Some years before pipes, were implemented, he suggested that commands should be thought of as binary operators, whose left and right operand specified the input and output files. Thus a 'copy' utility would be commanded by inputfile copy outputfile.

Multics provided a mechanism by which I/O Streams could be directed through processing modules on the way to (or from) the device or file serving as source or sink.

Thus it might seem that stream-splicing in Multics was the direct precursor of UNIX pipes.

We don't think this is true, or is true only in a weak sense. Not only were coroutines well-known already, but their embodiment as Multics I/O modules required to be specially coded in such a way that they could be used for no other purpose.

The genius of the Unix pipeline is precisely that it is constructed from the very same commands used constantly in simplex fashion.

The mental leap needed to see this possibility and to invent the notation is large indeed.

UNIX

By the 80's users began seen UNIX as a potential universal operating system, suitable for computers of all sizes. Both UNIX and C were developed by AT&T and distributed to government and academics alike.

Coroutines

Are computer-program components that generalize subroutines for non-preemptive multitasking by allowing multiple entry points for suspending and resuming execution at certain locations.

Subroutines

At the same time that assembly languages were being developed, programmers were gaining experience with subroutines.

Subroutines are short programs that perform functions of a general nature that can occur in various types of computation.

A branch sequence of instructions is executed, which jumps the program to the subroutine, the set of instructions in the subroutine is executed using the specified number, and, at completion, the computer goes back to the problem program for its continuation.

A sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

The experience with assemblers and subroutines helped to generate the ideas for the next step, that of a higher level language that would require the programmer to understand the problem he wishes to solve and not the machine that will be used to solve it.

Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.

In different programming languages, a subroutine may be called a procedure, a function, a routine, a method, or a subprogram.

Difference with processes

Processes are independent units of execution instead of a subroutine that lives inside a process.

Cooperative multitasking

Also known as non-preemptive multitasking, is a style of computer multitasking in which the operating system never initiates a context switch from a running process to another process.

Instead, processes voluntarily yield control periodically or when idle in order to enable multiple applications to be run concurrently.

Streams

A stream is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions.

In essence, the stream I/O provides a framework for making file descriptors act in the standard way most programs already expect, while providing a richer underlying behavior, for handling network protocols, or processing the appropriate messages.

Stream Mechanisms

When things wish to communicate, they must first establish communication. The stream mechanism provide a flexible way for processes to conduct an already-begun conversation with devices and with each other: an existing stream connection is named by a file descriptor, and the usual read, write, and I/O control request apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers are independent and separate cleanly.

However, these mechanisms, by themselves, do not provide a general way to create channels between them.

Simple extensions provide new ways of establishing communication. In our system, the traditional UNIX pipe is a cross-connected stream. A generalization of file-system mounting associates a stream with a named file. When the file is opened, operations on the file are operations on the stream. Open files may be passed from one process to another over a pipe.

These low-level mechanisms allow construction of flexible and general routines for connecting local and remote processes.

The work reported on streams describes convenient ways for programs to establish communication with unrelated processes, on the same or different machines.

Why Erlang helps?

Iterative programming is an iterative process, *iterative* is another name for *"Intelligent Trial and Error"*.

Erlang suits iterative development ideally, the BEAM ecosystem offers a variety of languages with different focus all build in top of the BEAM vm and the OTP framework.

Let it crash!

Robust systems must always be aware of errors but avoid the need of error checking code everywhere. We want to be able to handle processes crashes among cooperative processes.

- If one process crashes all cooperating processes should crash
- Cooperating processes are linked together
- Process crashes propagate along links

System processes can monitor them and rest them when necessary but sometimes we do need to handle errors locally.

Pattern matching

Functions use pattern matching to select clauses, this is a BIG WIN™

The BEAM VM

The BEAM virtual machine runs as one OS process. By default it runs one OS thread per core to achieve maximum utilisation of the machine. The number of threads and on which cores they run can be set when the VM is started.

Erlang processes are implemented entirely by the VM and have no connection to either OS processes or OS threads. So even if you are running a BEAM system of over one million processes it is still only one OS process and one thread per core, in this sense the BEAM is a "process virtual machine" while the Erlang system itself very much behaves like an OS and Erlang processes have very similar properties to OS processes.

- Process isolation
- Asynchronous communication
- Error handling, introspection and monitoring
- Predefined set of datatypes
- Immutable data
- Pattern matching
- Functional, soft real-time, reactive, message-passing system
- Modules as function containers and the only way of handle code

We just worry about receiving messages.

Load balancing

The goal is to not overload any scheduler while using as little CPU as possible.

Compacting the load to fewer schedulers is usually better for memory locality, specially on hyperthreads, the primary process is in charge of balance the workloads on the rest of the schedulers.

Process stealing

Process stealing is used by artists of all types and computers alike, on the BEAM is the primary mechanism to load balance and spread processes.

- A scheduler with nothing runnable will try to "steal processes" from adjacent schedulers, then next beyond that.
- We only steal from run-queues, never running or suspended processes.
- Schedulers changes on other schedulers run-queues.
- Each scheduler has its own run-queue.
- Processes suspend when waiting for messages, this is NOT a busy wait.
- Suspended processes become runnable when a message arrives.

By this mechanism the BEAM suspend unneeded schedulers. Once every period of 20k function calls is reach a new primary process inside a node scheduler is chosen. Primary processes balance workloads on schedulers.

Functions and modules

Modules contain functions, its a flat module space with just functions they only exist in modules there are no dependencies between running modules they can come and go as they please.

Functions

Functions cannot have a variable number of arguments! Erlang/OTP assumes functions with same name but different arities, each function has only a fixed number of arguments.

Modules

Modules can have functions with the same name and different number of arguments (arity), inside the virtual machine they are different functions.

Modules can consist of

- Declarations
- Function definitions
- Macro definitions
- Compile time function definitions

Macros can be defined anywhere, but must be defined before used.

The system only has compile code there is no build-in interpreter just compile code in modules. Everything is in modules the module is the unit of code handling, you compile modules, load modules, delete modules, update modules, everything run though modules there are no living functions outside modules.

We can have multiple versions of modules in the system at the same time, all functions belong to a module, this handle of modules means there is no inter-module dependency of modules at all, they just come and go when the system is running.

In this sense a running BEAM instance has no notion of a system, and can be described more like a collection of running modules.

Why ZeroMQ helps?

ZeroMQ (also known as ØMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like inter-process and TCP.

Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing subroutines.

Multilingual Distributed Messaging thanks to the ZeroMQ Community.

- Carries messages across inproc, IPC, TCP, multicast.
- Smart patterns like pub-sub, push-pull, and request-reply.
- Resource Property (NEW in ZMTP 3.1!)
- High-speed asynchronous I/O engines, in a tiny library.
- Backed by a large and active open source community.

You can connect sockets N-to-N with patterns like push-pull, pub-sub, request-reply and the new ZMTP 3.1 resource property.

Any message through the socket is treated as an opaque blob of data. Delivery to a subscriber can be automatically filtered by the blob leading string.

ZeroMQ implements ZMTP, the ZeroMQ Message Transfer Protocol. ZMTP defines rules for backward interoperability, extensible security mechanisms, command and message framing, connection metadata, and other transport-level functionality.

Read the guide and Learn the Basics

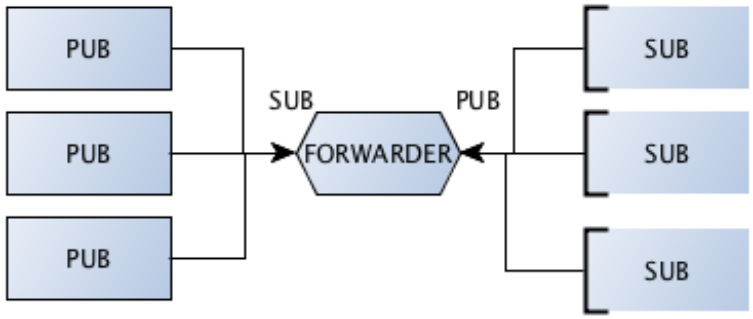
ZMQ devices

You must have noticed that you can bind a port to any of the ZeroMQ Socket types. In theory, most stable part of the network (the server) will **bind** on a specific port and have the more dynamic parts (the client) **connect** to that.

ZMQ provides certain basic proxy processes to build a more complex topology with basic device patterns our work this guide focus on *Forwarder* and *Streamer*.

Forwarder

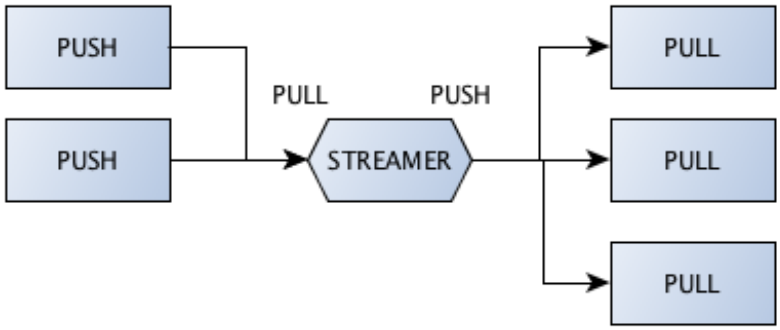
Forwarder device is like the pub-sub proxy server. It allows both publishers and subscribers to be moving parts and it self becomes the stable hub for interconnecting them.



This device collects messages from a set of publishers and forwards these to a set of subscribers.

Streamer

Streamer is a device for parallelized pipeline messaging. Acts as a broker that collects tasks from task feeders and supplies them to task workers.



References

- The Enchiridion (Epictetus)
- Metamorphoses (Ovid)
- Tractatus Logico-Philosophicus (1918, Wittgenstein)
- A Mathematical Theory of Communication (1948, Shannon)
- Organization of Behavior (1948, Hebb)
- What's most important (McIlroy, 1964)
- Video Ergo Scio (1973, Marr, Hewitt)
- A Universal Modular ACTOR Formalism for Artificial Intelligence (1973, Hewitt)
- Viewing Control Structures as Patterns of Passing Messages (1976, Hewitt)
- The Analysis of Organizations (1965, Litterer)
- The Evolution of Lua (Ierusalimschy, Figueiredo, Celes (date?))
- StarCraft: Brood War (Blizzard Entertainment, 1998)
- Torchnet: An Open-Source Platform for (Deep) Learning Research (2016, Collobert, Joulin)
- Organizations of Restricted Generality (2009, Hewitt)
- The Scientific Community Metaphor IEEE Transactions on Systems, Man, and Cybernetics. (1981, Kornfeld, Hewitt)
- Transforming Auto-encoders (Hinton, 2011)
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Hinton(date?))
- The role of learning in chuck call recognition by squirrel monkeys (1999, Mccowan, Newman)
- ZeroMQ: The Guide (Hintjens, 2013)
- Matrix Capsules With EM routing (2017, Sabour, Frosst, Hinton)
- Dynamic Routing Between Capsules (2017, Sabour, Frosst, Hinton)