The **Computer** Nonsense Guide

Or the influence of chaos on reason! 06.12.2017

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 operating system of Nonsense Worlds, Inc.

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.

The Computer Nonsense 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.

Nonsense runs an open-source multidisciplinary research laboratory that conducts open work on distributed systems, tropical monkeys, artificial intelligence and high-performance computing.

Our primary base is within the High Performance Computing community. However, after the fact that mostly every mobile device is a supercomputer these days our users come from a wide variety of industries.

Our Mission: Driven by nonsense we focus on multi-dimensional research providing tools inside a simple tiling window environment for play, work and science.

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 Lisp 3, Lua, Erlang and Python.

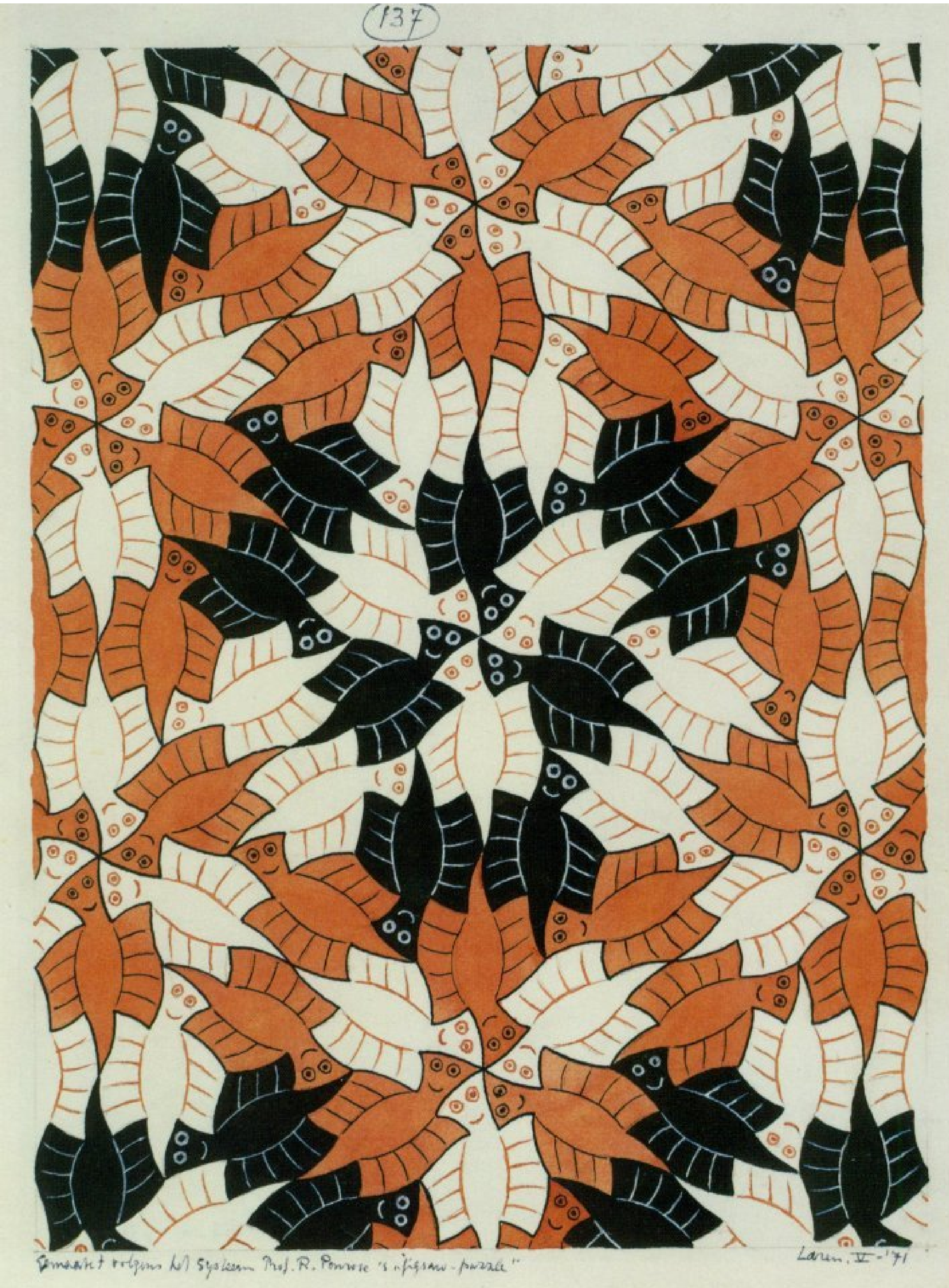
Other native ZMTP implementations are libzmq (C++), NetMQ (C#), JeroMQ (Java), libzmtp (C).

You can easily talk to any other ZeroMQ socket or a more standard MPI stack on your network.

This integration enables millions of machines to communicate with Torch, PyTorch, OpenResty, LÖVE, Tornado, Turbo or your favorite Singularity instance and focus around mutualism and balance inside distributed systems of C, Python, Lua and BEAM ecosystem processes.

We make a stable workspace that anyone can use today, at nonsense these things work together into one unified environment with native support for Luerl and LFE.

This guide describes research done at the AI Laboratory of Nonsense Worlds, Inc.



M.C. Escher, Penrose 'ghosts' - 1971

His last tessellation was a solution to a puzzle sent to him by Roger Penrose, the mathematician. Escher solved it and, true to form, changed the angular wood blocks into rounded 'ghosts'.

Prerequisites

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

Document conventions

In this guide, the following terminology is used:

A sequence is one or more items. For example, a clause body consists of a sequence of expressions. This means that there must be at least one expression.
A list is any number of items. For example, an argument list can consist of zero, one, or more arguments.

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.

Messages and functions

In its pure essence message passing is a form of function calling or function calling is a form of message passing and objects are a form of function or functions are a form of objects.

Metamorphoses

My mind is bent to tell of bodies changed into new forms. Ye humans, for you yourselves have wrought the changes, breathe on these my undertakings, and bring down my song in unbroken strains from the world's very beginning even unto the present time.

Before the sea was, and the lands, and the sky that hangs over all, the face of nature showed alike in her whole round, which state have you called chaos: a rough, unordered mass of things, nothing at all save lifeless bulk and warring seeds of ill-matched elements heaped in one.

No sun as yet shone forth upon the world, nor did the waxing moon renew her slender horns; not yet did the earth hand poised by her own weight in the circumambient air, nor had the ocean stretched hes arms along the far reaches of the lands. And, though there was both land and see and air, no one could tread that land, or swim that see; and the air was dark. No form of things remained the same; all objects were at odds, for within one body cold things strive with hot, and moist with dry, soft things with hard, things having weight with weightless things.

Nature composed this strife; for she rent asunder land from sky, and see from land, and separated the ethereal heavens from the dense atmosphere. When this she had released these elements and freed them from the blind heap of things, she set them each in its own place and bound them fast in harmony.

The fiery weightless element that forms heaven's vault leaped up and made place for itself upon the topmost height.

Next came the air in lightness and in place. The earth was heavier that these, drawing with it the grosser elements, sank to the bottom by its own weight. The streaming water took the last place of all, and held the solid land confined in its embrace.

When nature had thus arranged in order and resolved that chaotic mass, and reduced it, thus resolved, to cosmic parts, she first molded the earth into the form of a mighty ball so that it might be of like form on every side.

Then she bade the waters to spread abroad, to rise in waves beneath the rushing winds, and fling themselves around the shores of the encircled earth.

Springs, too, and huge, stagnant pools and lakes she made, and hemmed down-flowing rivers within their shelving banks, whose waters, each far remote from each, are partly swallowed by the earth itself, and partly flow down to the sea: and being thus received into the expanse of a freer flood, beat now on shores instead of banks.

Then did she bid plains to stretch out, valleys to sit down, woods to be clothed in leafage, and the rock-ribbed mountains to arise. And as the celestial vault is cut by two zones of the right and two of the left, and there is a fifth zone between hotter than these, so did the providence of nature mark off the enclosed mass with the same number of zones, and the same tracts were stamped upon the earth.

The central zone of these may not be dwelt in by reason of the heat; deep snow covers two, two she placed between and gave them temperate climate, mingling heat with cold.

The air hung over all, which is as much heavier than fire as the weight of water is lighter than the weight of earth. There did nature bid the mists and clouds to take their place, and thunder, that should shake the hearts of men, and winds which with the thunderbolts make chilling cold.

To these also the world's creator did not allot the air that they might hold it everywhere. Even as it is, they can scarce be prevented, though they control their blasts, each in his separate tract, from tearing the world to pieces. So fiercely do these brothers strive together.

Above these all she placed the liquid, weightless ether, which has naught of earthy dregs.

Scarce had she thus parted off all things within their determined bounds, when the stars, which had long been lying hid crushed down beneath the darkness, began to gleam throughout the sky. And, that no region might be without its own forms of animate life, the stars and divine forms occupied the floor of heaven, the sea fell to the shining fishes for their home, earth received the beasts, and the mobile air the birds.

A living creature of finer stuff than these, more capable of lofty thought, one who could have dominion over all the rest, was lacking yet. Then she was born.

So, then, the earth, which had but lately been a rough and formless thing, was changed and clothed itself with forms before unknown.

Getting started

First double check that your system have the latest releases of Erlang, LuaJIT (with luarocks) and Singularity 2.4+.

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, IoT devices, AI bots, machine learning and scientific computing research.

Luerl supports procedural, object-oriented, functional, data-driven, reactive, actor-model, 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 as well.

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 LFE, Erlang or Elixir functions, Luerl can be augmented to cope with a wide range of different domains, creating a customized language sharing a syntactical framework.

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 seemlesly with the rest of the ecosystem.

It's a language out of a language out of Sweden that can be used to build web scale, asynchronous, non-blocking, sharded, 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.

Pattern matching

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

Unhygienic macros

Macros are unhygienic, but is not as bad as it sounds, all variables are scoped and cannot be changed.

Macros can have any other number of arguments but only one macro definition per name.

Macros can have multiple clauses like functions the arguments are then the list passed to the macro.

The BIG Question

Apart from 42 the answer to Life, the Universe, and Everything...

Q. Will LFE end the moaning and wining about Erlang syntax and how bad all the things?

A. NO!

Modules and functions

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

Modules

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

LFE modules can consist of

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

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

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.

Singularity Containers

Singularity enables its users to have full control of their environment on whatever host they are on. This includes HPC, resource managers, file systems, GPUs and/or IoT devices, etc.

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

Singularity does this by enabling several keys:

- Encapsulation of the environment
- Containers are image based
- No user contextual changes or root escalation allowed
- No root owned daemon processes

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.

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**

Documentation

Learn the build environment, including changing the cache and specifying credentials for Docker.

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.

> M. Čertický, D. Churchill. The Current State of StarCraft AI Competitions and Bots . In Proceedings of the AIIDE 2017 Workshop on Artificial Intelligence for Strategy Games. 2017.

Installation

cube-cli install spqr

Meet the Marian Bots

Go Read the Quick Start and our current list of Heuristics

Tsunami engine

One of our goals is to produce fully autonomous processes that interact with their environment to learn optimal behaviors, improving over time through trial and error.

A framework full of pure technical nonsense for experienced, experience-driven autonomous, augmented, self-learning groups of entities is the tsunami engine and its squirrel monkey launcher.

Speed monkeys embarrassingly learn to exploit parallel computation. In particular stuff for training networks through asynchronous gradient updates have been use for in both single machines with multiple CPU's or GPU's and distributed systems with high number of clusters and machines.

Model-based

The key idea is to learn a model that allow for simulation of the environment without interacting with the environment directly.

Model-based functions do not assume specific prior knowledge however, in practice, we can incorporate prior knowledge like physics to speed up learning.

Model learning plays, an important role in reducing the amount of required interactions with the real environment, which may be limited in practice.

Installation

luarocks install tsunami

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.

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.

Trial and Error

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, humans don't like things which are unpredictable.
- Some of us hate being asked to predict the unpredictable.

Error handling

Errors will always occur!

Methodology

Successful with simple problems and games, is often resorted to when no apparent rule applies.

This does not mean that the approach need to be careless for an individual can be methodical in manipulating the variables in an attempt to sort through possibilities that may result in success.

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.

Hierarchies

In the Ashby-and-Cybernetics tradition, the word "trial" usually implies random-or-arbitrary, without any deliberate choice.

However amongst non-cyberneticians, "trial" will often imply a deliberate subjective act by some adult human agent; (e.g. in a court-room, or laboratory). So that has sometimes led to confusion.

Ashby's develops this "meta-level" idea, and extends it into a whole recursive sequence of levels, successively above each other in a systematic hierarchy. On this basis he argues that human intelligence emerges from such organization: relying heavily on trial-and-error (at least initially at each new stage), but emerging with what we would call "intelligence" at the end of it all.

Ashby-hierarchy coincides with Piaget's theory of developmental stages. After all, it is part of Piagetian doctrine that children learn by first actively doing in a more-or-less random way, and then hopefully learn from the consequences — which all has a certain random "trial-and-error".

Supervisors

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.

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.

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

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.

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.

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.

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 1980s users began seen UNIX as a potential universal operating system, suitable for computers of all sizes. Both UNIX and the C were developed by AT&T and distributed to government and academics alike.

Organizations

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.

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.

Pura LFE

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, [Abrahams 1966] a concerted language developed effort funded by ARPA that represented a radical departure from Lisp 1.5. *In contrast to most languages, in which the language is first designed an then implemented Lisp 2 was an implementation in search of a language*.

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).

Why Lisp?

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.

Lisp 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 object 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.

Torch and PyTorch

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

Tornado and Turbo

Tornado

Tornado is a Python web framework and asynchronous networking library, originally developed at FriendFeed. 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:

- A web framework (including RequestHandler which is subclassed to create web applications, and various supporting classes).
- 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 LuaJIT 2 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:

- Simple and intuitive API (much like Tornado).
- Low-level operations is possible if the users wishes that.
- Implemented in straight Lua and LuaJIT FFI on Linux, so the user can study and modify inner workings without too much effort.
- Good documentation
- Event driven, asynchronous and threadless design
- Small footprint
- SSL support (requires OpenSSL)

Proxying

We include an example for proxing tornado and turbo behind nginx (openresty) as a load balancer.

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.

Why Erlang?

Erlang is designed for massive concurrency. Erlang processes are lightweight green (grow and shrink dynamically) with small memory footprint, fast to create and terminate, and the scheduling overhead is low.

Primitives

- Isolated processes
- Asynchronous communication
- Selective receive

Error handling

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.

BEAM properties

The Erlang VM 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, for example isolation.

- Process isolation
- Asynchronous communication
- Error handling, introspection and monitoring
- Predefined set of datatypes
- Immutable data
- Pattern matching
- Functional, actor model, 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 VM it 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 VM 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.

Why ZeroMQ?

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 tasks.

You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, request-reply and the new ZMTP 3.1 resource property.

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.

Go Read the guide and Learn the Basics

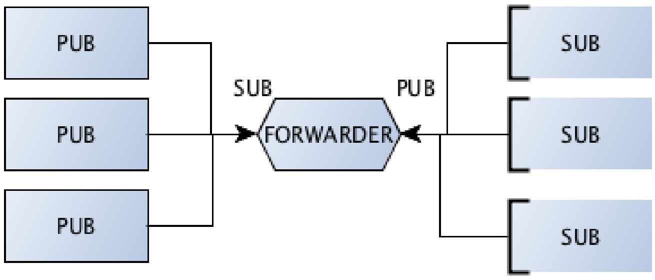
ZMQ devices

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 (server) will BIND on a specific port and have the more dynamic parts (client) CONNECT to that.

ZMQ provides certain basic proxy processes to build a more complex topology with basic device patterns this implementation 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.