The Computer **Nonsense** Guide

Or the influence of chaos on reason! 01.06.2018

Contents

Abstract 4
- Core ideas 4
- Prerequisites 4

Introduction 5
- Before start 5
- Installation 5

Lua in Erlang 6
- Luerl goal 6
- Embedded language 6
- The result 7

Lisp Flavoured Erlang 8
- LFE goal 8
- The result 8

Why Lisp 3? 9
- Lisp 1 9
- Lisp 2 10
  - Proposals for Lisp 2.0 10
  - The Actor Model 10
- Logic and λ-calculus 11
  - A limitation of logic programming 12
  - Indeterminacy in concurrent computation 12
- Lisp 3 12
  - What isn't 13
  - What is 13
  - Lisp Machine flavors 13

Application containers 15
- Linux containers 15

Demo: Brood War AI 16
- Core ideas 16
- Stages of a game 16
- Show me the money 16

What is an organization? 17
- Perception and behavior 17
- Cues 17
- Threshold of perception 18
- Resonance 18
- Internal consistency 18

Iterative programming 19
- Cumulative adaptation 19
- Dynamic optimization 20
- Overlapping subproblems 20
  - Optimal substructure 20

Bellman's principle 20
Analytical concepts 21

Why Erlang helps? 22
Let it crash! 22
Pattern matching 22
Supervision trees 22
Fault-tolerance 23
Beam virtual machine 23
Load balancing 24
Process stealing 24
Functions and modules 24
Functions 24
Modules 24

McIlroy garden hose 26
Coroutines 26
Subroutines 26
Difference with processes 27
Cooperative multitasking 27
Channeling Shannon 27
Stream Mechanisms 27
Unix System V 28

Why ZeroMQ helps? 29
Protocols 29
Message patterns 29
Device patterns 29
Forwarder 30
Streamer 30
Community 30

Abstract

"My aim is: to teach you to pass from disguised nonsense to something that is patent nonsense." — Ludwig Wittgenstein

This guide is product of the efforts of many people too numerous to list here and of the unique environment of nonsense.ws open-source community.

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 interfaces at all.

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

Our Mission: driven by technical nonsense we focus on provide tools inside a simple unix-like workspace for play, work and science, building information models through observation and action.

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

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.

Core ideas

- Functions are a form of objects.
- Message passing and function calling are analogous.
- Asynchronous message passing is necessary for non-blocking systems.
- Selective receive allow to ignore messages uninteresting now.

Prerequisites

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

Introduction

"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." — Richard P. Gabriel

About 100 years ago there were two schools of thought a clash between two paradigms for how to make an intelligent system, one paradigm was mathematical logic if I give you some true premises and some valid rules of inference you can derive some truth conclusions and people who believe in logic thought that's the way the mind must work and somehow the mind is using some funny kind of logic that can cope with the paradox of the liar or the fact that sometimes you discover things you believed were false.

Classical logic has problems with that and the paradigm said we have these symbolic expressions in our head and we have rules from repairing them and the essence of intelligence is reasoning and it works by moving around symbols in symbolic expressions.

There was a completely different paradigm that wasn't called artificial intelligence it was called neural networks that said we known about an intelligent system it's the mammalian brain and the way that works is you have lots of little processes with lots of connections between them and you change the strengths of the connections and that's how you learn things so they thought the essence of intelligence was learning and in particular how you change the connection strengths so that your neural network will do new things and they would argue that everything you know comes from changing those connection strengths and those changes have to somehow be driven by data you're not programmed you somehow absorb information from data, well for 100 years this battle has gone on and fortunately today we can tell you recently it was won.

Before start

Check your system have the latest Erlang, LuaJIT (with luarocks) and Singularity 3+ installed.

Installation

Then run this command:

```
luarocks install pkg
```

For help using pkg, including a list of commands, run:

```
$ pkg --help
```

Congratulations, you are jacked up and good to go!

Lua in Erlang

"Scripting is a relevant technique for any programmer's toolbox." — Roberto Ierusalimschy

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, 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.

Luerl goal

A proper implementation of the Lua language

- It SHOULD look and behave the same as Lua 5.2
- It SHOULD include the Lua standard libraries
- It MUST interface well with Erlang

Embedded language

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 of 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, Luerl 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.

The result

Luerl is a native Erlang implementation of standard Lua 5.2 written for the BEAM ecosystem.

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

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

Lisp Flavoured Erlang

"Lisp: Good News Bad News How to Win Big ™." — Richard P. Gabriel

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

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

For all of us in general the bad news is that almost everything that what we have been using is WORNG! no one can denied the respect that Black Mesa Research deserves but even with it's limited concurrency implemented by a CSP model with monads or static types have yet it classical boundaries; the λ-calculus low concurrent ceiling.

We were not that into Lisp until reading some tweets from certain no-horn viking, the short story is that Scheme is Lisp 2, the goal of Scheme was to implement a Lisp following the actor model but they discover closures instead got hyped with them and forget about Hewitt.

Erlang was born to the world on Stockholm Sweden in 1998, Jane Walerud, Bjarne Däcker, Mike Williams, Joe Armstrong and Robert Virding open-source a language that implement this model of universal computation based in physics without even know or care about it, just pure engineering powers 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.

LFE 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 3?

"Lisp is the greatest single programming language ever designed." — Alan Kay

A lot has changed since 1958, even for Lisp it now 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 and ideas to draw from.

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.

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.

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 2

The Lisp 2 project was 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 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

- 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

The Actor Model

Actors are the universal primitive of concurrent digital computation. In response to a message that it receives, an actor can make local decisions, create more Actors, send more messages, and designate how to respond to the next message received.

Unbounded nondeterminism is the property that the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources while still guaranteeing that the request will eventually be serviced.

Arguments for unbounded nondeterminism include the following:

There is no bound that can be placed on how long it takes a computational circuit called an Arbiter to settle.

- Arbiters are used in computers to deal with the circumstance that computer clocks operate asychoronously with input from outside, "e.g, keyboard input, disk access, network input, etc..."
- So it could take an unbounded time for a message to sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.

The following were the main in uences on the development of the actor model of

computation:

- The suggestion by [Alan] Kay that procedural embedding be extended to cover data structures in the context of our previous attempts to generalize the work by Church, Landin, Evans, and Reynolds on \functional data structures."
- The context of our previous attempts to clean up and generalize the work on coroutine control structures of Landin, Mitchell, Krutar, Balzer, Reynolds, Bobrow-Wegbreit, and Sussman.
- The infuence of Seymour Papert's \little man" metaphor for computation in LOGO.
- The limitations and complexities of capability-based protection schemes. Every actor transmission is in e ect an inter-domain call eciently providing an intrinsic protection on actor machines.
- The experience developing previous generations of PLANNER. Essentially the whole PLANNER-71 language (together with some extensions) was implemented by Julian Davies in POP-2 at the University of Edinburgh.

In terms of the actor model of computation, control structure is simply a pattern of passing messages.

We have quoted Hewitt at length because the passage illustrates the many connections among different ideas floating around in the AI, Lisp, and other programming language communities; and because this particular
point in the evolution of ideas represented a distillation that soon fed back quickly and powerfully into the evolution of Lisp itself.

Logic and λ-calculus

Logic programming is the proposal to implement systems using mathematical logic.

Perhaps the first published proposal to use mathematical logic for programming was John McCarthy's Advice Taker paper.

Planner was the first language to feature "procedural plans" that were called by "pattern-directed invocation" using "goals" and "assertions". A subset called Micro Planner was implemented by Gerry Sussman, Eugene Chariak and Terry Winograd and was used in Winograd's natural language understanding program SHRDLU, and some other projects.

This generated a great deal of excitement in the field of AI. It also generated controversy because it proposed an alternative to the logic approach one of the mainstay paradigms for AI.

The upshot is that the procedural approach has a different mathematical semantics based on the denotation semantics of the Actor model from the semantics of mathematical logic.

There were some surprising results from this research including that mathematical logic is incapable of implementing general concurrent computation even though it can implement sequential computation and some kinds of parallel computation including the lambda calculus.

Classical logic blows up in the face of inconsistent information that is kind of ubiquitous with the growth of the internet.

This change enables a new generation of systems that incorporate ideas from mathematical logic in their implementation, resulting on some reincarnation of logic programming. But something is often transformed when reincarnated!

A limitation of logic programming

In his 1988 paper on early history of Prolog, Bob Kowalski published the thesis that "computation is controlled deduction" which he attributed to Pat Hayes.

Contrary to Kowalski and Hayes, Hewitt's thesis was that logical deduction was incapable of carrying out concurrent computation in open systems because of indeterminacy in the arrival order of messages.

Indeterminacy in concurrent computation

Hewitt and Agha [1991] argued that: The Actor model makes use of arbitration for determining which message is next in the arrival ordering of an Actor that is sent multiple messages concurrently.

For example Arbiters can be used in the implementation of the arrival ordering of an Actor which is subject to physical indeterminacy in the arrival order.

In concrete terms for Actor systems typically we cannot observe the details by which the arrival order of messages for an Actor is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere.

Instead of observing the internals of arbitration processes of Actor computations, we await outcomes.

Physical indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no alternative because of indeterminacy.

According to Chris Fuchs [2004], quantum physics is a theory whose terms refer predominately to our interface with the world. It is a theory not about observables, not beables, but about 'dingables' we tap a bell with our gentle touch and listen for its beautiful ring.

It is important to distinguish between indeterminacy in which factors outside the control of an information system are making decision and *choice* in which the information system has some control.

It is not sufficient to say that indeterminacy in Actor systems is due to unknown/unmodeled properties of the network infrastructure. The whole point of the appeal to quantum indeterminacy is to show that aspects of Actor systems can be unknowable and the participants can be entangled.

The concept that quantum mechanics forces us to give up is: the description of a system independent from the observer providing such a description; that is the concept of the absolute state of a system. I.e, there is no observer independent data at all.

According to Zurek [1982], "Properties of quantum systems have no absolute meaning. Rather they must always characterized with respect to other physical systems."

Does this mean that there is no relation whatsoever between views of different observers? Certainly not. According to Rovelli [1996] "It is possible to compare different views, but the process of comparison is always a physical interaction (and all physical interactions are quantum mechanical in nature)".

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).

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in LFE and used by the Blueberry HPC package 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.

Lisp Machine flavors

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 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.

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

Application containers

"Awaken my child, and embrace the glory that is your birthright." — The Overmind

Singularity: Application containers for Linux enables full control of the environment on whatever host you are on. This include distributed systems, your favorite blockchain, HPC centers, microservices, GPU's, IoT 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.

Linux containers

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!

Demo: Brood War AI

"Send colonies to one or two places, which may be as keys to that state, for it is necessary either to do this or else to keep there a great number of lings and hydras." — Sarah Kerrigan

We present Blueberry a TorchCraft bot system build for online competition and AI research on the real-time strategy game of StarCraft; ours is a message-passing, asynchronous system that exploits the hot swap loading and parallelism of Luerl and the concurrency of the BEAM VM.

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

Unlike synchronous turn-based games like chess and go, StarCraft games are played in real-time, the state continue to progress even if no action is taken, actions must decided in fractions of a second, game frames issue simultaneous actions to hundreds of units at any given time, players only get the information about what they units observe, there is a fog of information present in the environment and hidden units that require additional detection.

Core ideas

- StarCraft is about information, the smoke of rare weeds and silver for tools.
- Strong armies vs mobile armies.
- Defense units are powerful but immobile, offense units are mobile but weak.
- Efficiency is not the number one goal.

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.

Information, colonies, rare weed and silver for better tools.

Show me the money

Install the bot,

```
$ pkg install blueberry
```

play a game.

```
$ pkg run blueberry
```

What is an organization?

"ORGs is a paradigm in which people are tightly integrated with information technology that enables them to function with an organizationally relevant task or problem." — Carl Hewitt

A monkey, a building, 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 members 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 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.

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.

Iterative programming

"Programming is an iterative process, *iterative* is another name for intelligent trial and error."
— Michael C Williams

In cybernetics, the word "trial" usually implies random-or-arbitrary, without any deliberate choice.

Programming 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.

Trial and error is also a heuristic method of problem solving, repair, tuning, or obtaining knowledge.

In the field of computer science, the method is called generate and test. In elementary algebra, when solving equations, it is "guess and check".

Cumulative adaptation

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.

Suppose N events each have a probability *p* of success, and the probabilities are independent. An example would occur if N wheels bore letters A and B on the rim, with A's occupying the spun and allowed to come to rest; those that stop at an A count as successes. Let us compare three ways of compounding these minor successes to a Grand Success, which we assume, occurs only when every wheel is stopped at an A.

Case 1: All N wheels are spun; if all show an A, Success is recorded and the trials ended; otherwise all are spun again, and so on till 'all A's' come up at one spin.

Case 2: The first wheel is spun; if it stops at an A it is left there; otherwise it is spun again. When it eventually stops at an A the second wheel is spun similarly; and so on down the line of N wheels, one at a time, till all show A's.

Case 3: All N wheels are spun; those that show an A are left to continue showing it, and those that show a B are spun again. When further A's occur they also are left alone. So the number spun gets fewer and fewer, until all are at A's.

The conclusion that Case 1 is very different from Cases 2 and 3, does not depend closely on the particular values of p and N.

Comparison of the three Cases soon shows why Cases 2 and 3 can arrive at Success so much sooner than Case 1: they can benefit by partial successes, which 1 cannot. Suppose, for instance, that, under Case 1, a spin gave 999 A's and 1 B. This is very near complete Success; yet it counts for nothing, and all the A's have to be thrown back into the melting-pot. In Case 3, however, only one wheel would remain to be spun; while Case 2 would perhaps get a good run of A's at the left-hand end and could thus benefit from it.

The examples show the great, the very great, reduction in time taken that occurs when the final Success can be reached by stages, in which partial successes can be conserved and accumulated.

We can draw, then, the following conclusion. A compound event that is impossible if the components have to occur simultaneously may be readily achievable if they can occur in sequence or independently.

Dynamic optimization

Dynamic optimization (also known as dynamic programming) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed result, saving computation time at the expense of (it is hoped) a modest expenditure in storage space.

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimization literature this is called the Bellman equation.

Overlapping subproblems

Like Divide and Conquer, Dynamic optimization combines solutions to sub-problems mainly used when they are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common subproblems because there is no point storing the solutions if they are not needed again.

Optimal substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming.

The application of dynamic programming is based on the idea that in order to solve a dynamic optimization problem from some starting period t to some ending period T, one implicitly has to solve subproblems starting from later dates s, where $t<s<T$. This is an example of optimal substructure. The Principle of Optimality is used to derive the Bellman equation, which shows how the value of the problem starting from t is related to the value of the problem starting from s.

Bellman's principle

The dynamic programming method breaks this decision problem into smaller subproblems.

In the context of dynamic game theory, this principle is analogous to the concept of subgame perfect equilibrium, although what constitutes an optimal policy in this case is conditioned on the decision-maker's

opponents choosing similarly optimal policies from their points of view.

Analytical concepts

To understand the Bellman's principle, several underlying concepts must be understood. First, any optimization problem has some objective: minimizing travel time, minimizing cost, maximizing profits, maximizing utility, etcetera.

The mathematical function that describes this objective is called the objective function.

Dynamic programming breaks a multi-period planning problem into simpler steps at different points in time. Therefore, it requires keeping track of how the decision situation is evolving over time. The information about the current event that is needed to make a correct decision is called the "state".

Why Erlang helps?

"If somebody came to me and wanted to pay me a lot of money to build a large scale message handling system that really had to be up all the time, could never afford to go down for years at a time, I would unhesitatingly choose Erlang to build it in." — Tim Bray

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™

Supervision trees

"Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang." — Robert Virding

Too often, developers try to implement their own error-handling and recovery strategies in their code, with the result that they increase the complexity of the code and the cost of maintaining it, how many times have you seen catch statements with nothing more than TODO comments to remind some future, better smarted developer to finish the job on the error handling.

This is where the supervisor process makes it entrance. It takes over the responsibility for the unexpected-error-handling and recovery strategies from the developer.

The behavior, in a deterministic and consistent manner, handles monitoring, restart strategies, race conditions, and borderline cases most developers would not think of.

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.

Fault-tolerance

Fault-tolerance is achieved by creating supervision trees, where the supervisors are the nodes and the workers are the leaves of this analogy. Supervisors on a particular level monitor and handle children in the subtrees they have started.

If any worker terminates abnormally, the simple supervisor immediately restart it. If the process instead terminate normally, they are removed from the supervision tree and no further action is taken.

Stopping the supervisor results in all the processes in the tree being unconditionally terminated. When the supervisor terminates, the run-time ensures that all processes linked to it receive an EXIT signal.

It is a valid assumption that nothing abnormal should happen when starting your system. If a supervisor is unable to correctly start a process, it terminates all of its processes and aborts the startup procedure. While we are all for a resilient system that tries to recover from errors, startup failures is where we draw the line.

Beam virtual machine

"Links were invented by Mike Williams and based on the idea of a *C-wire* a form of electrical circuit breaker." — Joe Armstrong

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

Inside the BEAM ecosystem, 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 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.

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.

McIlroy garden hose

"Streams means something different when shouted." — Dennis Ritchie

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.

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.

Channeling Shannon

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.

Unix System V

In this framework, a stream is a chain of coroutines that pass messages between a program and a device driver (or between a pair of programs).

An important concept is the ability to push custom code modules which can modify the functionality of a network interface or other device – together to form a stack. Several of these drivers can be chained together in order.

Why ZeroMQ helps?

"Diversity is not a political slogan. It's the basis for collective intelligence." — Pieter Hintjens

ZeroMQ is a community of projects focused on decentralized message passing. They agree on protocols (RFCs) for connecting to each other and exchanging messages. Messages are blobs of useful data of any reasonable size.

You can use this power to queue, route, and filter messages according to various "patterns."

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.

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.
- Backed by a large and active open source community.

Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing subroutines. Read the guide and learn the basics.

Protocols

The ZeroMQ protocols live as RFCs on http://rfc.zeromq.org. The main one would be RFC 23, the ZeroMQ Message Transport Protocol (ZMTP), which describes how two pieces talk, over TCP or IPC. RFC 23 provides backwards compatibility to all older stable releases of ZeroMQ.

ZMTP defines rules for backward interoperability, extensible security mechanisms, command and message framing, connection metadata, and other transport-level functionality.

Message patterns

Device patterns

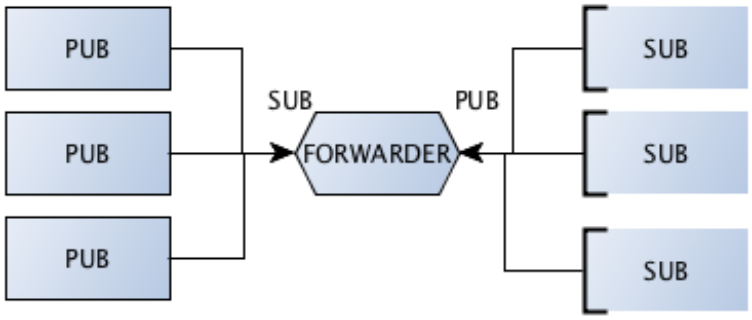
"To cope with blind spots we have to make use of another vocabulary or system outside the one in which the doubt exists." — Joseph. A Litterer

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

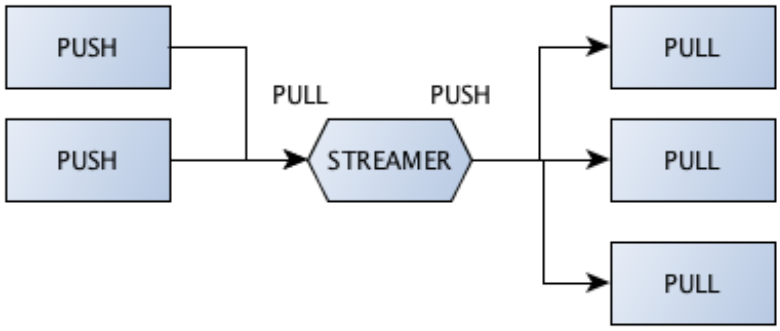
Forwarder device is like the pub-sub proxy server. It allows both publishers and subscribers to be moving parts and it self becomes the stable hub for interconnecting them.



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.



Community

The ZeroMQ community uses a collaboration contract, C4.1. This is an RFC (of course), at http://rfc.zeromq.org/spec:22. It defines how the community works together and has been the main factor for the happy growth and stability of the community.