The Zen of Erlang

"My bad opinions"

2016/02/08

This is a loose transcript (or long paraphrasing?) of a presentation given at ConnectDev'16, a conference organized by Genetec in which I was invited to speak.

I assume most people here have never used Erlang, have possibly heard of it, maybe just the name. As such, this presentation will only cover the high level concepts of Erlang, in such a way that it may be useful to you in your work or side projects even if you never touch the language.

If you've ever looked at Erlang before, you've heard about that "Let it crash" motto. My first encounter with it had me wondering what the hell this was about. Erlang was supposed to be great for concurrency and fault tolerance, and here I was being told to let things crash, the entire opposite of what I actually want to happen in a system. The proposition is surprising, but the "Zen" of Erlang is related to it directly nonetheless.

In some ways it would be as funny to use 'Let it Crash' for Erlang as it would be to use 'Blow it up' for rocket science. 'Blow it up' is probably the last thing you want in rocket science — the Challenger disaster is a stark

reminder of that. Then again, if you look at it differently, rockets and their whole propulsion mechanism is about handling dangerous combustibles that can and will explode (and that's the risky bit), but doing it in such a controlled manner that they can be used to power space travel, or to send payloads in orbit.

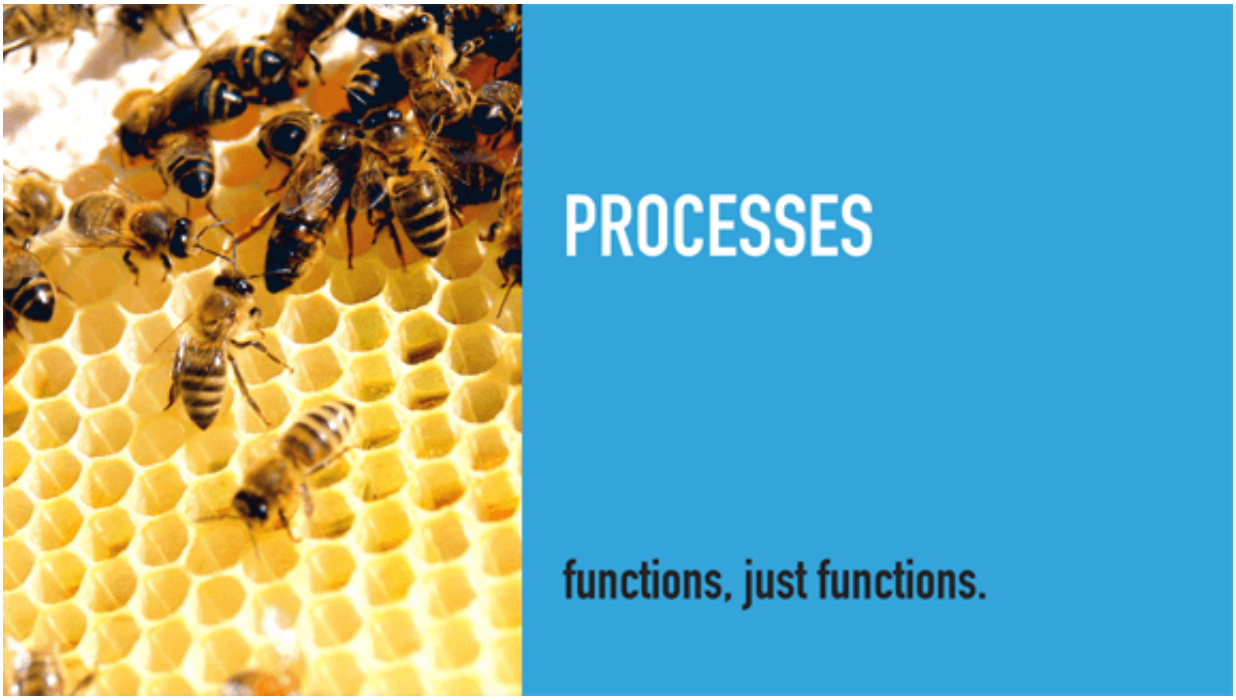
The point here is really about control; you can try and see rocket science as a way to properly harness explosions — or at least their force — to do what we want with them. Let it crash can therefore be seen under the same light: it's all about fault tolerance. The idea is not to have uncontrolled failures everywhere, it's to instead transform failures, exceptions, and crashes into tools we can use.

Back-burning and controlled burns are a real world example of fighting fire with fire. In Saguenay–Lac-Saint-Jean, the region I come from, blueberry fields are routinely burnt down in a controlled manner to help encourage and renew their growth. To prevent forest fires, it is fairly frequent to see unhealthy parts of a forest cleaned up with fire, so that it can be done under proper supervision and control. The main objective there is to remove combustible material in such a way an actual wildfire cannot propagate further.

In all of these situations, the destructive power of fire going through

crops or forests is being used to ensure the health of the crops, or to prevent a much larger, uncontrolled destruction of forested areas.

I think this is what 'Let it crash' is about. If we can embrace failures, crashes and exceptions and do so in a very well-controlled manner, they stop being this scary event to be avoided and instead become a powerful building block to assemble large reliable systems.



So the question becomes to figure out how do we ensure that crashes are enablers rather than destructors. The basic game piece for this in Erlang is the process. Erlang's processes are fully isolated, and they share nothing. No process can go and reach into another one's memory, or impact the work it's doing by corrupting the data it operates on. This is good because it means that a process dying is essentially guaranteed to keep its issues to itself, and that provides very strong fault isolation into your system.

Erlang's processes are also extremely lightweight, so that you can have thousands and thousands of them without problem. The idea is to use as many processes as you *need*, rather than as many as you *can*. The common comparison there is to say that if you had an object-oriented language where you could only have 32 objects running at a any given time, you'd rapidly find it overly constraining and quite ridiculous to build

programs in that language. Having many small processes does ensure a higher granularity in how thing break, and in a world where we want to harness the power of these failures, this is good!

Now it can be a bit weird to picture how these processes work exactly. When you write a C program, you have one big `main()` function that does a lot of stuff. This is your entry point into the program. In Erlang, there is no such thing. No process is the designated master of the program. Every one of them runs a function, and that function plays the role of `main()` within that single process.

We now have that swarm of bees, but it is probably very hard to direct them to strengthen the hive if they cannot communicate in any way. Where bees dance, Erlang processes pass messages.

Message passing is the most intuitive form of communication in a concurrent environment there is. It's the oldest one we've worked with, from the days where we wrote letters and sent them via couriers on horse to find their destination, to fancier mechanisms like the Napoleonic semaphores shown on the slide. In this case you'd just take a bunch of guys up into towers, give them a message, and they'd wave flags around to pass data over long distances in ways that were faster than horses, which could tire. This eventually got replaced by the telegraph, which got

replaced by the phone and the radio, and now we have all the fancy technologies to pass messages really far and really fast.

A critical aspect of all this message passing, especially in the olden days, is that everything was asynchronous, and with the messages being copied. No one would stand on their porch for days waiting for the courier to come back, and no one (I suspect) would sit by the semaphore towers waiting for responses to come back. You'd send the message, go back to your daily activities, and eventually someone would tell you you got an answer back.

This is good because if the other party doesn't respond, you're not stuck doing nothing but waiting on your porch until you die. Conversely, the receiver on the other end of the communication channel does not see the freshly arrived message vanish or change as by magic if you do die. Data *should* be copied when messages are sent. These two principles ensure that failure during communication does not yield a corrupted or unrecoverable state. Erlang implements both of these.

To read messages, each process has a single mailbox. Everyone can write to a process' mailbox, but only the owner can look into it. The messages are by default read in the order they arrived, but it is possible, through features such as pattern matching [which were discussed in a prior talk during the day] to only or temporarily focus on one kind of message and drive various priorities around.

Some of you will have noticed something in what I have mentioned so far; I keep repeating that isolation and independence are great so components of a system are allowed to die and crash without influencing others, but I did also mention having conversations across many processes or agents.

Every time two processes start having a conversation, we create an implicit dependency between them. There is some implicit state in the system that binds both of them together. If process A sends a message to process B and B dies without responding, A can either wait forever, or give up on having a conversation after a while. The latter is a valid strategy, but it's a very vague one: it is unclear if the remote end has died or is just taking long, and off-band messages can land in your mailbox.

Instead Erlang gives us two mechanisms to deal with this: monitors and links.

Monitors are all about being an observer, a creeper. You decide to keep an eye on a process, and if it dies for whatever reason, you get a message in your mailbox telling you about it. You can then react to this and make decisions with your newly found information. The other process will never have had an idea you were doing all of this. Monitors are therefore fairly decent if you're an observer or care about the state of

a peer.

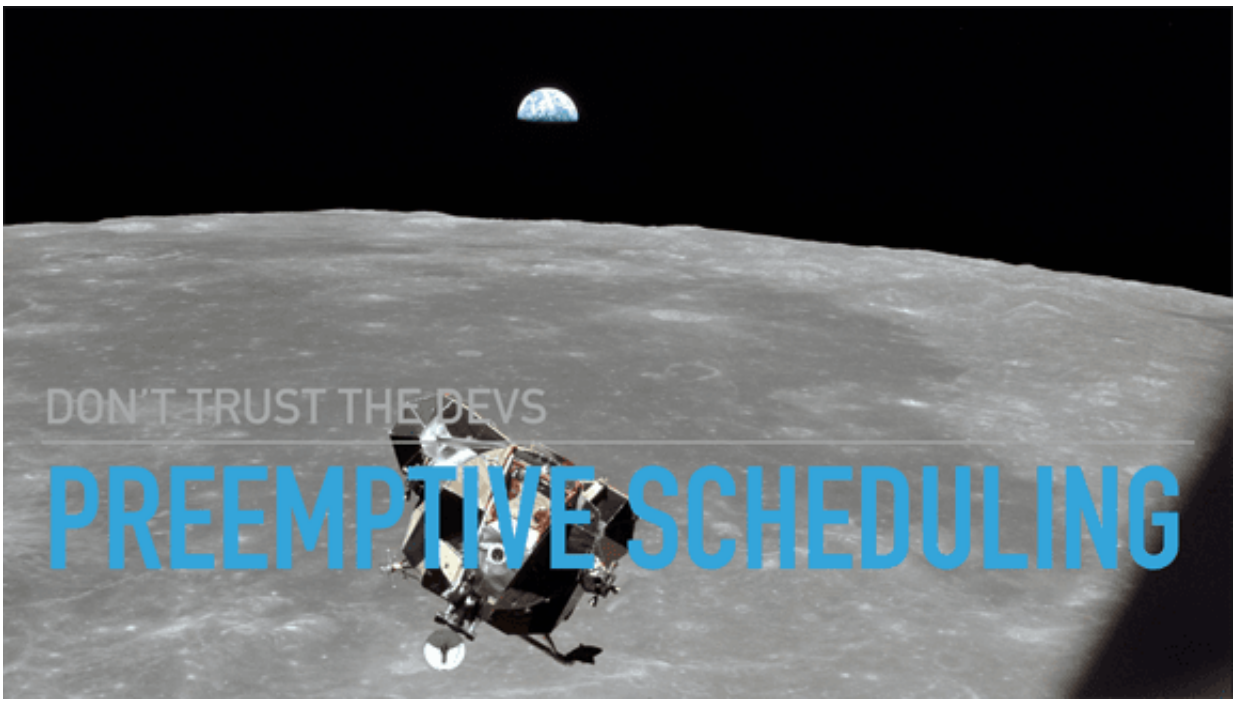
Links are bidirectional, and setting one up binds the destiny of the two processes between which they are established. Whenever a process dies, all the linked processes receive an exit signal. That exit signal will in turn kill the other processes.

Now this gets to be really interesting because I can use monitors to quickly detect failures, and I can use links as an architectural construct letting me tie multiple processes together so they fail as a unit. Whenever my independent building blocks start having dependencies among themselves, I can start codifying that into my program. This is useful because I prevent my system from accidentally crashing into unstable partial states. Links are a tool letting developers ensure that in the end, when a thing fails, it fails entirely and leaves behind a clean slate, still without impacting components that are not involved in the exercise.

For this slide, I picked a picture of mountain climbers roped together. Now if mountain climbers only had links between them, they would be in a sorry state. Every time a climber from your team would slip, the rest of the team would instantly die. Not a great way to go about things.

Erlang instead lets you specify that some processes are special and can be flagged with a `trap_exit` option. They can then take the exit signals sent over links and transform them into messages. This lets them recover faults and possibly boot a new process to do the work of the former one. Unlike mountain climbers, a special process of that kind cannot prevent a peer from crashing; that is the responsibility of that peer by using `try ... catch` expressions, for example. A process that traps exits still has no way to go play in another one's memory and save it, but it can avoid dying of it.

This turns out to be a critical feature to implement supervisors. If you haven't heard of them, we'll get to them soon enough.



Before going to supervisors, we still have a few ingredients to be able to successfully cook a system that leverages crashes to its own benefit. One of them is related to how processes are scheduled. For this one, the real world use case I want to refer to is Apollo 11's lunar landing.

Apollo 11 is the mission that went to the moon in '69. In the slide right there, we see the lunar module with Buzz Aldrin and Neil Armstrong on board, with a photo taken by a person I assume to be Michael Collins, who stayed in the command module for the mission.

While on their way to land on the moon, the lunar module was being guided by the Apollo PGNCS (Primary Guidance, Navigation and Control System). The guidance system had multiple tasks running on it, taking a carefully accounted for number of cycles. NASA had also specified that the processor was only to be used to 85% capacity, leaving 15% free.

Now because the astronauts in this case wanted a decent backup plan in case they needed to abort, they had left a rendezvous radar up in case it would come in handy. That took up a decent chunk of the capacity the CPU had left. As Buzz Aldrin started entering commands, error messages would pop up about overflow and basically going out of capacity. If the system was going haywire on this, it possibly couldn't do its job and we could end up with two dead astronauts.

This was mostly because the radar had known hardware bugs causing its frequency to be mismatched with the guidance computer, and caused it to steal far more cycles than it should have had otherwise. Now NASA people weren't idiots, and they reused components with which they knew the rare bugs they had rather than just greenfielding new tech for such a critical mission, but more importantly, they had devised priority scheduling.

This meant that even in the case where either this radar or possibly the commands entered were overloading the processor, if their priority were too low compared to the absolutely life-critical stuff, the task would get killed to give CPU cycles to what really, really needed it. That was in 1969; today there's still plenty of languages or frameworks that give you *only* cooperative scheduling and nothing else.

Erlang is not a language you'd use for life-critical systems — it only respects soft-real time constraints, not hard real time ones and it just wouldn't be a good idea to use it in these scenarios. But Erlang does provide you with preemptive scheduling, and with process priorities. This means that you do not *have* to care, as a developer or system designer, about making sure that absolutely everyone carefully counts all the CPU usage they're going to be doing across all their components (including libraries you use) to ensure they don't stall the system. They just won't have that capacity. And if you need some important task to always run when it must, you can also get that.

This may not seem like a big or common requirement, and people still ship really successful projects only with cooperative scheduling of concurrent tasks, but it certainly is extremely valuable because it protects you against the mistakes of others, and also against your own mistakes. It also opens up the door to mechanisms like automated load-balancing, punishing or rewarding good and bad processes or giving higher priorities to those with a lot of work waiting for them. Those things can end up giving you systems that are fairly adaptive to production loads and unforeseen events.

The last ingredient I want to discuss in getting decent fault tolerance is network awareness. In any system we develop that we need to stay up for long periods of time, having more than one computer to run it on quickly becomes a prerequisite. You don't want to be sitting there with your own golden machine locked behind titanium doors, unable to tolerate any disruption with effecting your users in major ways.

So you eventually need two computers so one can survive a broken other, and maybe a third one if you want to deploy with broken computers part of your system.

The plane on the slide is the F-82 twin mustang, an aircraft that was designed during the second world war to escort bombers over ranges most other fighters just couldn't cover. It had two cockpits so that pilots could take over and relay each other over time when tired; at some point they also fit it so one would pilot while the other would operate radars in an interceptor role. Modern aircrafts still do something similar; they have countless failovers, and often have crew members sleeping in transit during flight time to make sure there's always someone who's alert ready to pilot the plane.

When it comes to programming languages or development environments, most of them are designed ignoring distribution altogether, even though

people know that if you write a server stack, you need more than one server. Yet, if you're gonna use files, there's gonna be stuff in the standard library for that. The furthest most languages go is giving you a socket library or an HTTP client.

Erlang acknowledges the reality of distribution and gives you an implementation for it, which is documented and transparent. This lets people set up fancy logic for failing over or taking over applications that crash to provide more fault tolerance, or even lets other languages pretend they are Erlang nodes to build polyglot systems.

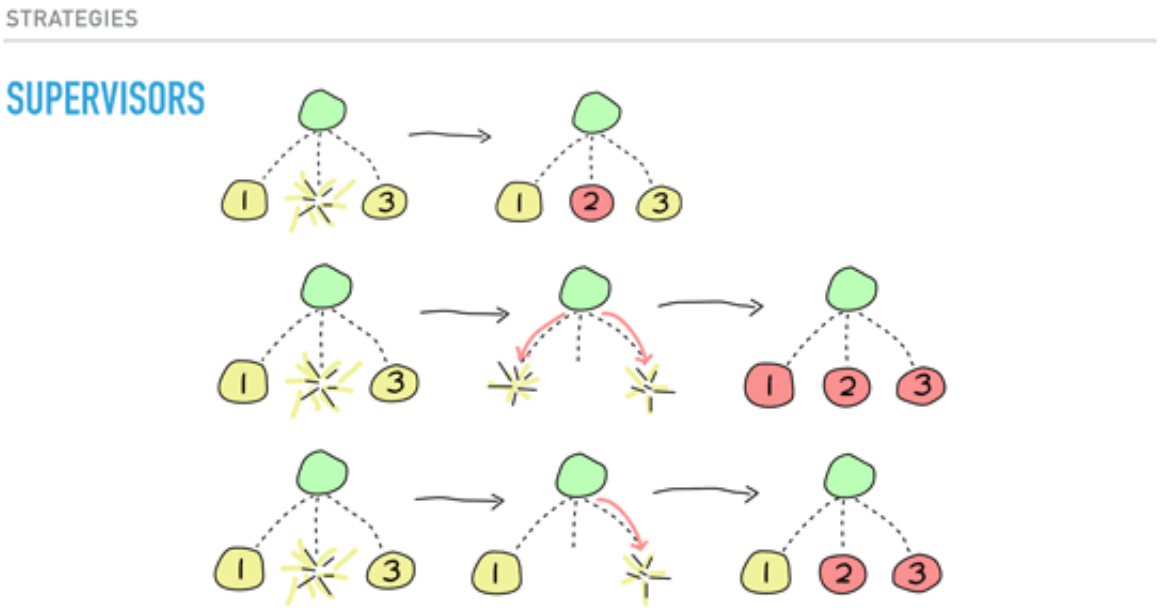
So those are all the basic ingredients in the recipe for Erlang Zen. The whole language is built with the purpose of taking crashes and failures, and making them so manageable it becomes possible to use them as a tool. Let it crash starts making sense, and the principles seen here are for the most part things that can be reused as inspiration in non-Erlang systems.

How to compose them together is the next challenge.

Supervision trees is how you impose structure to your Erlang programs. They start with a simple concept, a supervisor, whose only job is to start processes, look at them, and restart them when they fail. By the way, supervisor are one of the core components of 'OTP', the general development framework used in the name 'Erlang/OTP'.

The objective of doing that is to create a hierarchy, where all the important stuff that must be very solid accumulate closer to the root of the tree, and all the fickle stuff, the moving parts, accumulate at the leaves of the tree. In fact, that's what most trees look like in real life: the leaves are mobile, there's a lot of them, they can all fall down in autumn, the tree stays alive.

That means that when you structure Erlang programs, everything you feel is fragile and should be allowed to fail has to move deeper into the hierarchy, and what is stable and critical needs to be reliable is higher up.



Supervisors can do that through usage of links and trapping exits. Their job begins with starting their children in order, depth-first, from left to right. Only once a child is fully started does it go back up a level and start the next one. Each child is automatically linked.

Whenever a child dies, one of three strategies is chosen. The first one on the slide is 'one for one', enacted by only replacing the child process that died. This is a strategy to use whenever the children of that supervisor are independent from each other.

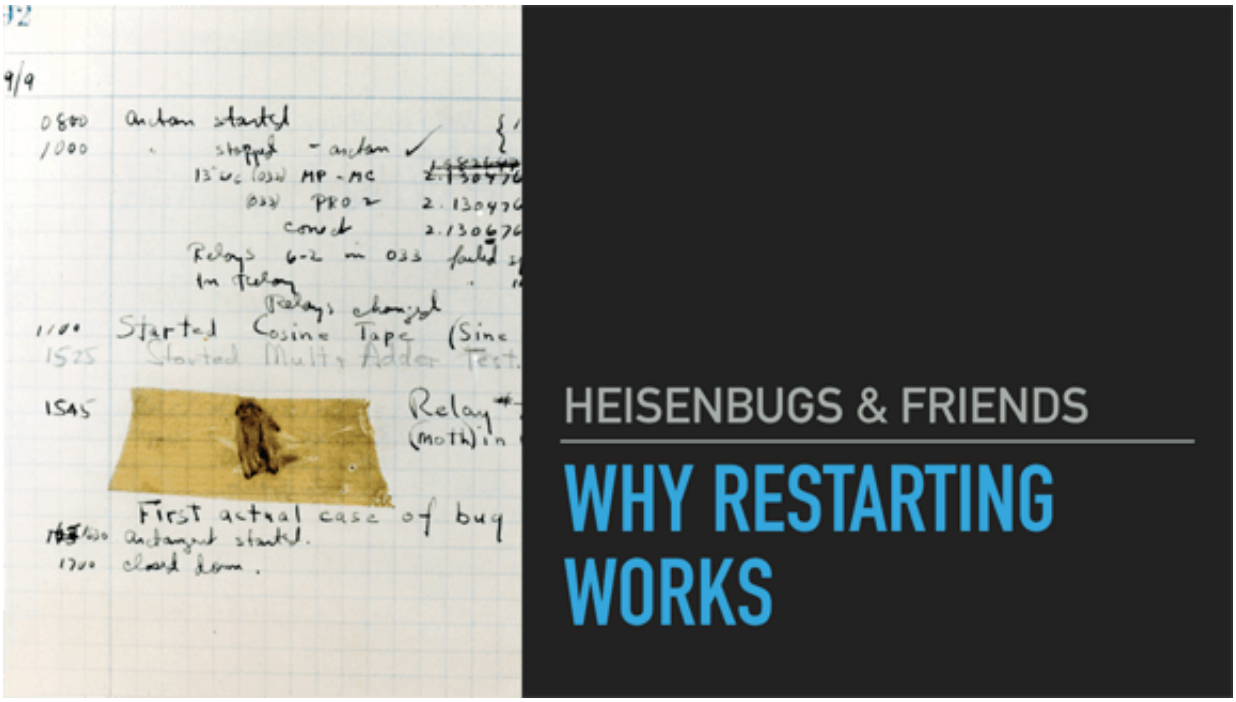
The second strategy is 'one for all'. This one is to be used when the children depend on each other. When any of them dies, the supervisor then kills the other children before starting them all back. You would use this when losing a specific child would leave the other processes in an uncertain state. Let's imagine a conversation between three processes that ends with a vote. If one of the process dies during the vote, it is possible that we have not programmed any code to handle that. Replacing that dead process with a new one would now bring a new peer to a table that has no idea what is going on either!

This inconsistent state is possibly dangerous to be in if we haven't really defined what goes on when a process wreaks havoc through a voting procedure. It is probably safer to just kill all processes, and start afresh

from a known stable state. By doing so, we're limiting the scope of errors: it is better to crash early and suddenly than to slowly corrupt data on a long-term basis.

The last strategy happens whenever there is a dependency between processes according to their booting order. Its name is 'rest for one' and if a child process dies, only those booted after it are killed. Processes are then restarted as expected.

Each supervisor additionally has configurable controls and tolerance levels. Some supervisors may tolerate only 1 failure per day before aborting while others may tolerate 150 per second.



The comment that usually comes right after I mention supervisors is usually to the tune of "but if my configuration file is corrupted, restarting won't fix anything!"

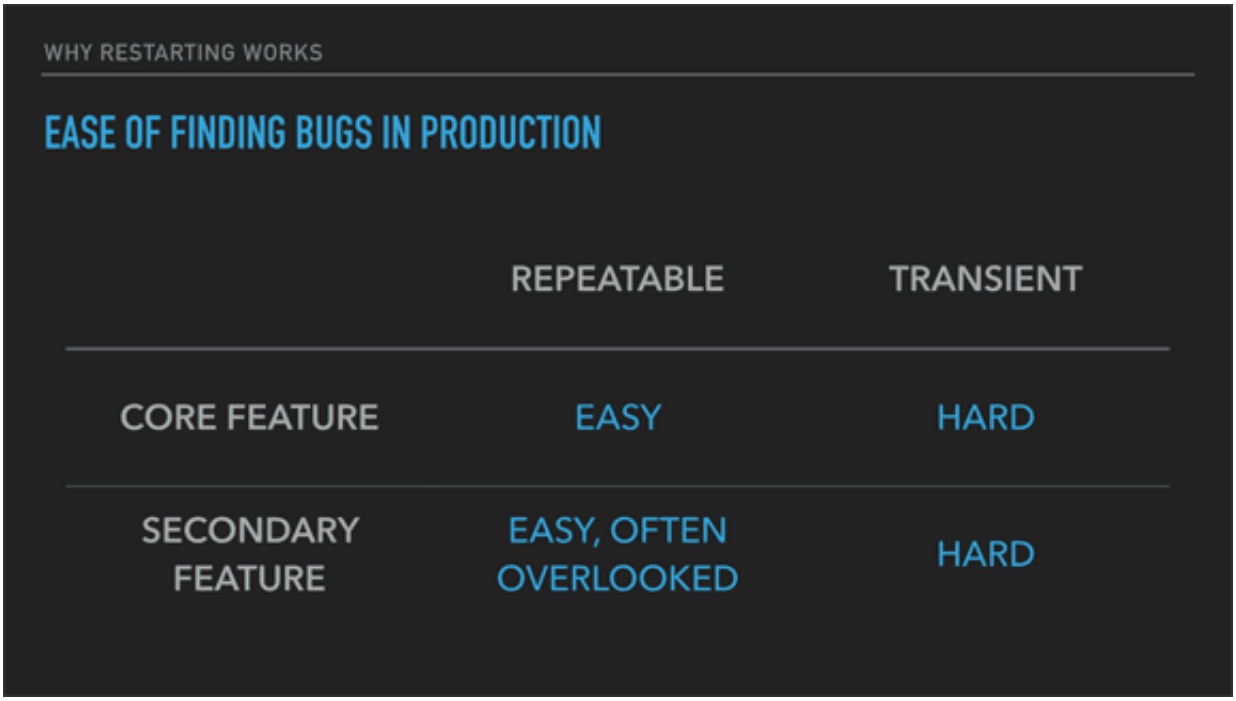
That is entirely right. The reason restarting works is due to the nature of bugs encountered in production systems. To discuss this, I have to refer to the terms 'Bohrbug' and 'Heisenbug' coined by Jim Gray in 1985 (I do recommend you read as many Jim Gray papers as you can, they're pretty much all great!)

Basically, a bohrbug is a bug that is solid, observable, and easily

repeatable. They tend to be fairly simple to reason about. Heisenbugs by contrast, have unreliable behaviour that manifests itself under certain conditions, and which may possibly be hidden by the simple act of trying to observe them. For example, concurrency bugs are notorious for disappearing when using a debugger that may force every operation in the system to be serialised.

Heisenbugs are these nasty bugs that happen once in a thousand, million, billion, or trillion times. You know someone's been working on figuring one out for a while once you see them print out pages of code and go to town on them with a bunch of markers.

With these terms defined, let's look at what should be their frequencies.



Here I'm classifying bohrbugs as repeatable, and heisenbugs as transient.

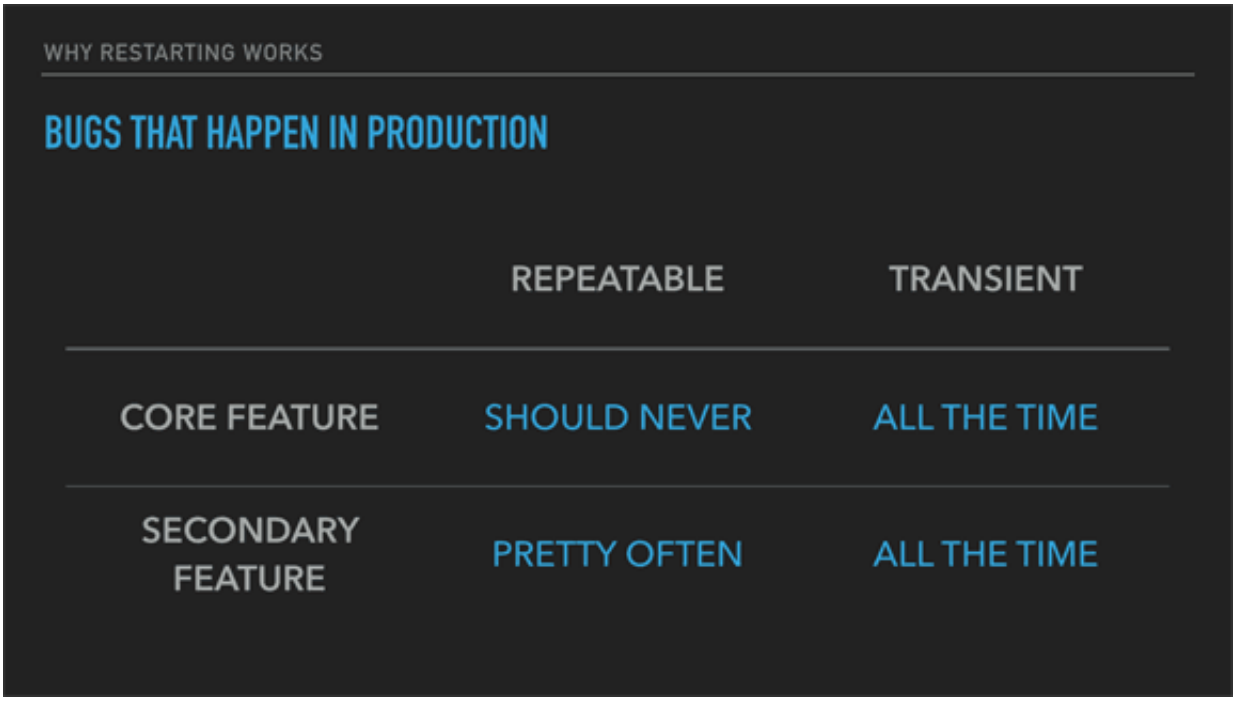
If you have bohrbugs in your system's core features, they should usually be very easy to find before reaching production. By virtue of being repeatable, and often on a critical path, you should encounter them sooner or later, and fix them before shipping.

Those that happen in secondary, less used features, are far more of a hit and miss affair. Everyone admits that fixing all bugs in a piece of software

is an uphill battle with diminishing returns; weeding out all the little imperfections takes proportionally more time as you go on. Usually, these secondary features will tend to gather less attention because either fewer customers will use them, or their impact on their satisfaction will be less important. Or maybe they're just scheduled later and slipping timelines end up deprioritising their work.

In any case, they are somewhat easy to find, we just won't spend the time or resources doing so.

Heisenbugs are pretty much impossible to find in development. Fancy techniques like formal proofs, model checking, exhaustive testing or property-based testing may increase the likelihood of uncovering some or all of them (depending on the means used), but frankly, few of us use any of these unless the task at hand is extremely critical. A once in a billion issue requires quite a lot of tests and validation to uncover, and chances are that if you've seen it, you won't be able to generate it again just by luck.



The next connection I want to make is regarding the frequency each of these types of bugs have in production (in my experience). There's no obvious proof that there is any connection between the use of finding bugs and their incidence in production systems, but my gut feeling would

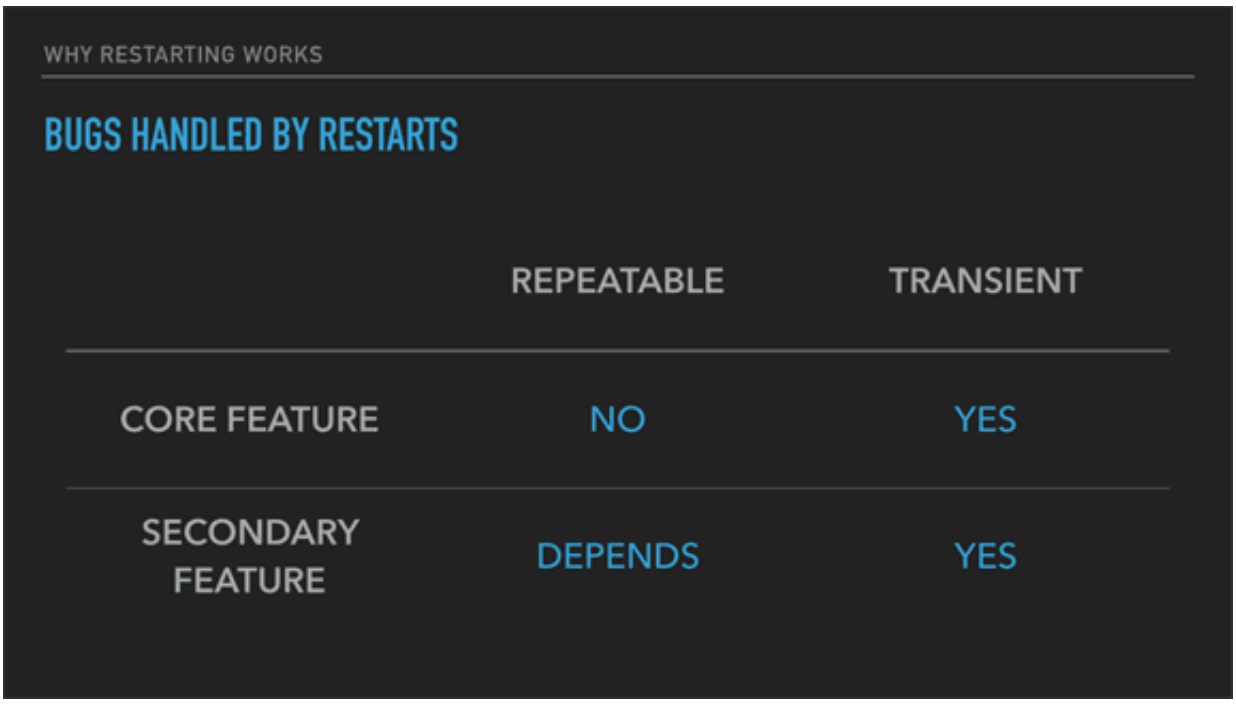
tell me that such a connection does exist.

First of all, easy repeatable bugs in core features should just not make it to production. If they do, you have essentially shipped a broken product and no amount of restarting or support will help your users. Those require modifying the code, and may be the result of some deeply entrenched issues within the organisation that produced them.

Repeatable bugs in side-features will pretty often make it to production. I do think this is a result of not taking the time to test them properly, but there's also a strong possibility that secondary features often get left behind when it comes to partial refactorings, or that the people behind their design do not fully consider whether the feature will coherently fit with the rest of the system.

On the other hand, transient bugs will show up all the damn time. Jim Gray, who coined these terms, reported that on 132 bugs noted at a given set of customer sites, only one was a Bohrbug. 131/132 of errors encountered in production tended to be heisenbugs. They're hard to catch, and if they're truly statistical bugs that may show once in a million times, it just takes some load on your system to trigger them all the time; a once in a billion bug will show up every 3 hours in a system doing 100,000 requests a second, and a once in a million bug could similarly show up once every 10 seconds on such a system, but their occurrence would still be rare in tests.

That's a lot of bugs, and a lot of failures if they are not handled properly.



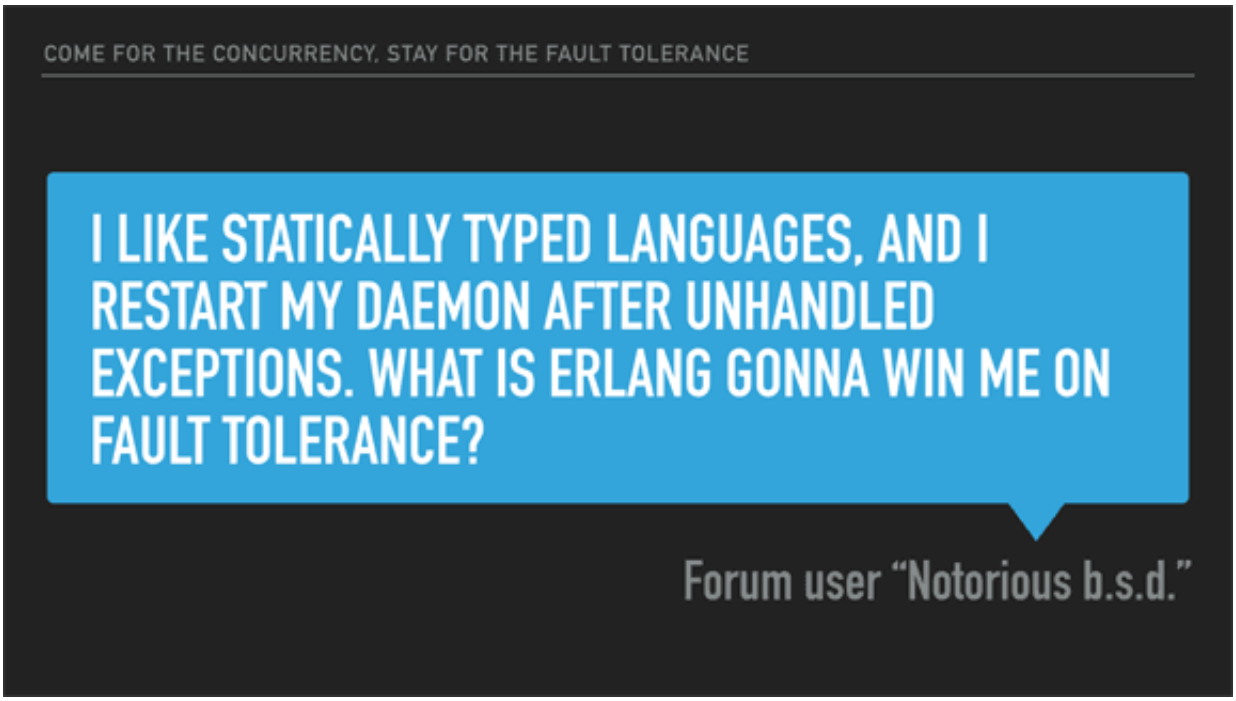
So really, how efficient is restarting as a strategy?

Well for repeatable bugs on core features, restarting is useless. For repeatable bugs in less frequently used code paths, it depends; if the feature is a thing very important to a very small amount of users, restarting won't do much. If it's a side-feature used by everyone, but to a degree they don't care much about, then restarting or ignoring the failure altogether can work well. For example, if the facebook 'poke' feature were to be broken (would it still exist), not too many users would notice or see their experience ruined by its failure.

For transient bugs though, restarting is extremely effective, and they tend to be the majority of bugs you'll meet live. Because they are hard to reproduce, that their showing up is often dependent on very specific circumstances or interleavings of bits of state in the system, and that their appearance tends to be in a very small fraction of all operations, restarting tends to make them disappear altogether.

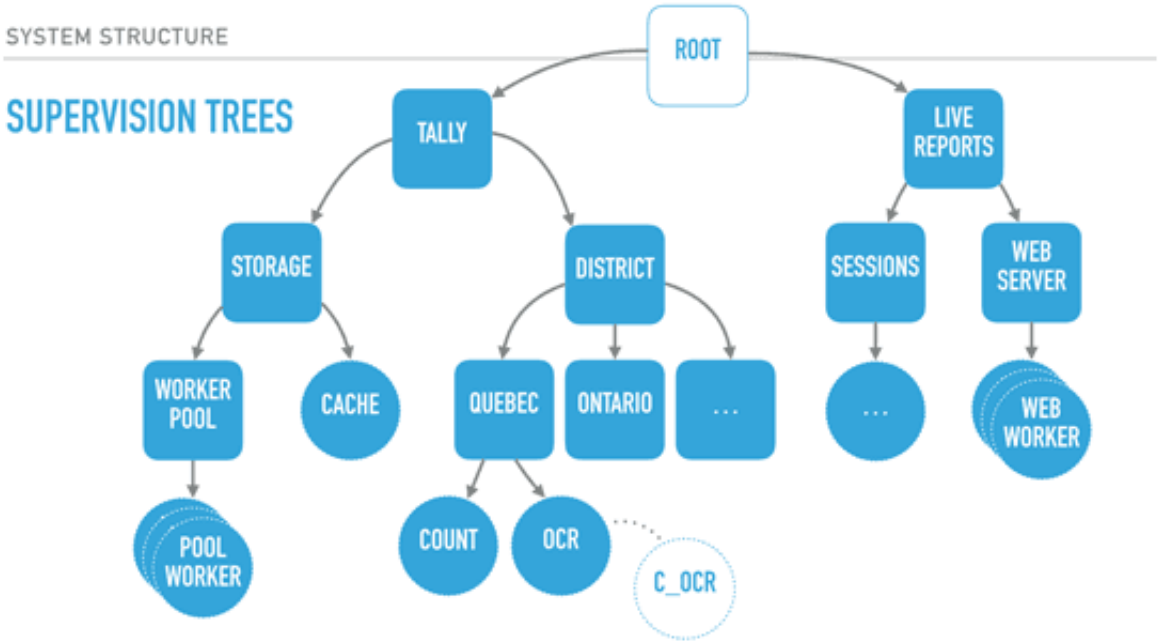
Rolling back to a known stable state and trying again is unlikely to hit the same weird context that causes them. And just like that, what could have been a catastrophe has become little more than a hiccup for the system, something users quickly learn to live with.

You can then make use of logging, tracing, or a variety of introspection tools (which all come out of the box in Erlang) to later find, understand, and fix the issues so they stop happening. Or you could just decide to tolerate them were the effort required to fix the issues too large.



This question was asked to me on a forum where I was discussing programming stuff and discussing the Erlang model. I copied it verbatim because it's a great example of a question a lot of people ask when they hear about restarting and Erlang's features.

I want to address it specifically by giving a realistic example of how a system could be designed in Erlang, which will highlight its peculiarities.



With supervisors (rounded squares), we can start creating deep hierarchies of processes. Here we have a system for elections, with two trees: a tally tree and a live reports tree. The tally tree takes care of counting and storing results, and the live reports tree is about letting people connect to it to see the results.

By the order the children are defined, the live reports will not run until the tally tree is booted and functional. The district subtree (about counting results per district) won't run unless the storage layer is available. The storage's cache is only booted if the storage worker pool (which would connect to a database) is operational.

The supervision strategies I mentioned earlier let us encode these requirements in the program structure, and they are still respected at run time, not just at boot time. For example, the tally supervisor may be using a one for one strategy, meaning that districts can individually fail without effecting each other's counts. By contrast, each district (Quebec and Ontario's supervisors) could be employing a rest for one strategy. This strategy could therefore ensure that the OCR process can always send its detected vote to the 'count' worker, and it can crash often without impacting it. On the other hand, if the count worker is unable to keep and store state, its demise interrupts the OCR procedure, ensuring nothing breaks.

The OCR process itself here could be just monitoring code written in C, as a standalone agent, and be linked to it. This would further isolate the faults of that C code from the VM, for better isolation or parallelisation.

Another thing I should point out is that each supervisor has a configurable tolerance to failure; the district supervisor might be very tolerant and deal with 10 failures a minute, whereas the storage layer could be fairly intolerant to failure if expected to be correct, and shut down permanently after 3 crashes an hour if we wanted it to.

In this program, critical features are closer to the root of the tree, unmoving and solid. They are unimpacted by their siblings' demise, but their own failure impacts everyone else. The leaves do all the work and can be lost fairly well — once they have absorbed the data and operated their photosynthesis on it, it is allowed to go towards the core.

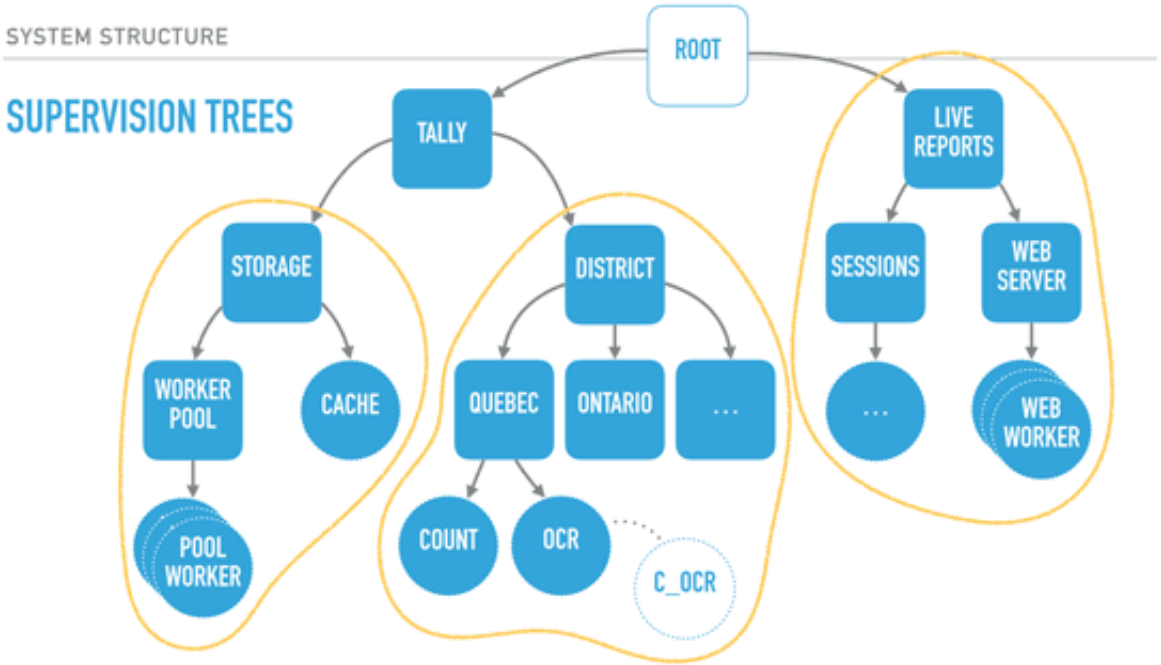
So by defining all of that, we can isolate risky code in a worker with a high tolerance or a process that is being monitored, and move data to stabler process as information matures into the system. If the OCR code in C is dangerous, it can fail and safely be restarted. When it works, it transmits its information to the Erlang OCR process. That process can do validation, maybe crash on its own, maybe not. If the information is solid, it moves it to the Count process, whose job is to maintain very simple state, and eventually flush that state to the database via the storage subtree, safely independent.

If the OCR process dies, it gets restarted. If it dies too often, it takes its own supervisor down, and that bit of the subtree is restarted too — without affecting the rest of the system. If that fixes things, great. If not, the process is repeated upwards until it works, or until the whole system is taken down as something is clearly wrong and we can't cope with it through restarts.

There's enormous value in structuring the system this way because error handling is baked into its structure. This means I can stop writing outrageously defensive code in the edge nodes — if something goes

wrong, let someone else (or the program's structure) dictate how to react. If I know how to handle an error, fine, I can do that for that specific error. Otherwise, just let it crash!

This tends to transform your code. Slowly you notice that it no longer contains these tons of if/else or switches or try/catch expressions. Instead, it contains legible code explaining what the code should do when everything goes right. It stops containing many forms of second guessing, and your software becomes much more readable.

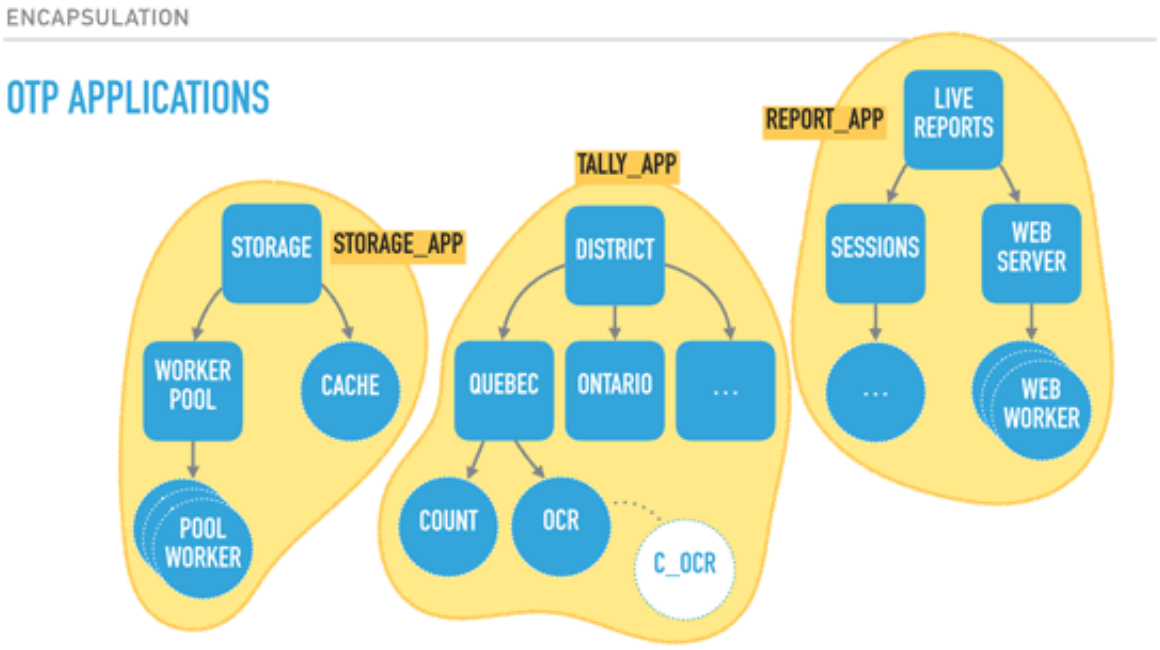


When taking a step back and looking at our program structure, we may in fact find that each of the subtrees encircled in yellow seem to be mostly independent from each other in terms of what they do; their dependency is mostly logical: the reporting system needs a storage layer to query, for example.

It would also be great if I could, for example, swap my storage implementation or use it independently in other systems. It could be neat, too, to isolate the live reports system into a different node or to start providing alternative means (such as SMS for example).

What we now need is to find a way to break up these subtrees and turn them into logical units that we can compose, reuse together, and that we

can otherwise configure, restart, or develop independently.



OTP applications are what Erlang uses as a solution here. OTP applications are pretty much the code to construct such a subtree, along with some metadata. This metadata contains basic stuff like version numbers and descriptions of what the app does, but also ways to specify dependencies between applications. This is useful because it lets me keep my storage app independent from the rest of the system, but still encode the tally app's need for it to be there when it runs. I can keep all the information I had encoded in my system, but now it is built out of independent blocks that are easier to reason about.

In fact, OTP applications are what people consider to be libraries in Erlang. If your code base isn't an OTP application, it isn't reusable in other systems. [Sidenote: there are ways to specify OTP libraries that do not actually contain subtrees, just modules to be reused by other libraries]

With all of this done, our Erlang system now has all of the following properties defined:

- what is critical or not to the survival of the system
- what is allowed to fail or not, and at which frequency it can do so before it is no longer sustainable

- how software should boot according to which guarantees, and in what order
- how software should fail, meaning it defines the legal states of partial failures you find yourself in, and how to roll back to a known stable state when this happens
- how software is upgraded (because it can be upgraded live, based on the supervision structure)
- how components interdepend on each other

This is all extremely valuable. What's more valuable is forcing every developer to think in such terms from early on. You have less defensive code, and when bad things happen, the system keeps running. All you have to do is go look at the logs or introspect the live system state and take your time to fix things, if you feel it's worth the time.

With all of this done, I should be able to sleep at night, right? Hopefully yes. What I included here is a small pixelated diagram from a new software deploy we ran at Heroku a couple of years ago.

The leftmost side of the diagram is around September. By that time, our new proxying layer (vegur) had been in production for maybe 3 months, and we had ironed out most of the kinks in it. Users had no problem, the transition was going smoothly and new features were being used.

At some point, a team member got a very expensive credit card bill for the logging service we were using to aggregate exceptions. That's when we took a look at it and saw the horror on the leftmost side of the diagram: we were generating between 500,000 to 1,200,000 exceptions a day! Holy cow, that was a lot. But was it? If the issue was a heisenbug, and our system was seeing, say 100,000 requests a second, what were the odds of it happening? Something between 1/17000 and 1/7000. Somewhat very frequent, but because it had no impact on service, we didn't notice it until the bandwidth and storage bill came through.

It took us a short while to figure out the error, and we fixed it. You can see that there is still a low rate of exceptions after that, maybe a few dozen thousands a day. They're all things we know of, but are impact-free. Two years later and we haven't bothered to fix it because the system works fine despite that.

At the same time, you can't always just sleep at night. Failures can be out of your control despite your best design efforts.

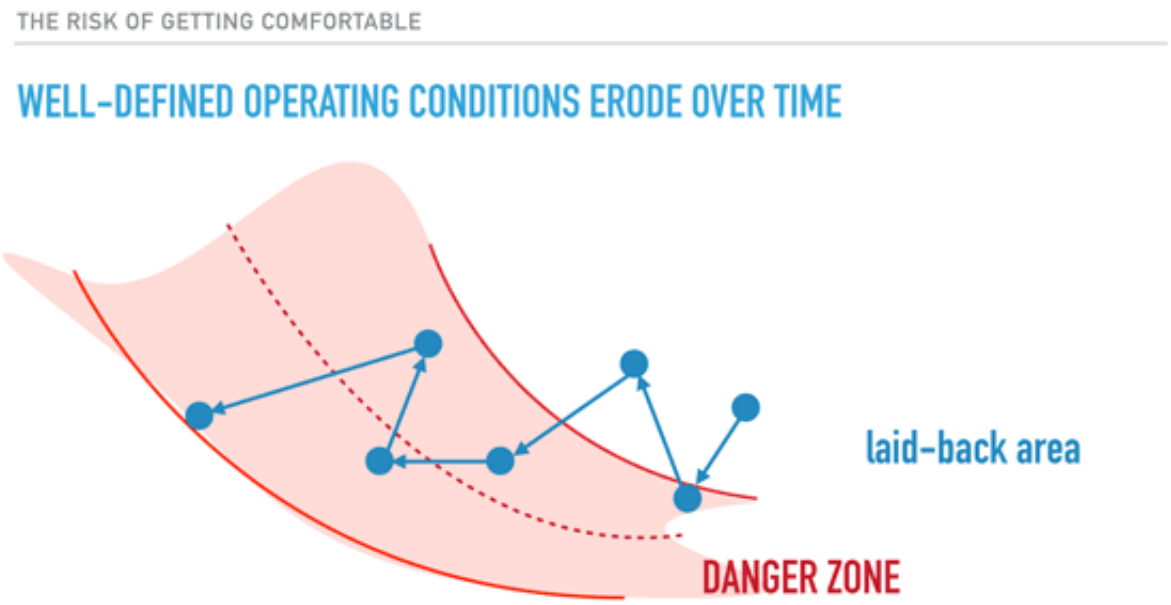
A couple of years ago I was on a flight to Vancouver starting on its descent when the pilot used the intercom to tell us something a bit like this: "this is your captain speaking, we will be landing shortly. Do not be alarmed as we will stay on the tarmac for a few minutes while the fire

department looks over the plane. We have lost some hydraulic component and they want to ensure there is no risk of fire. There are two backup systems for the broken one, and we should be fine."

And we were fine. In this case the airplane was amazingly well designed.

The image for this slide isn't that flight though, it's another one I was on two weeks ago, while the Eastern US were being burrowed under 24 inches of snow. The plane (flight United 734), which I'm sure was as reliable, landed on the runway. When it came time to break though, it made a loud noise, what I assume is the ABS equivalent of aircrafts, but it still kept going.

We ran over the red lights at the end of the runway you see on the picture, and at the end of the tarmac, the plane skid off the runway, missed the onramp, and the front wheel ended up in the grass. Everyone was fine, but this is an example of why great engineering cannot save the day every time.



In fact, operations will always remain a huge factor in successful systems being deployed. This slide is heavily inspired (pretty much stolen in fact) from presentations by Richard Cook. If you don't know him, I urge you to go watch videos of his talks on youtube, they're pretty much all fantastic.

Proper system architecture and development practices can still not replace, or can be broken by inadequate operations; the efficiency and usefulness of tools, playbooks, monitoring, automation, and so on, all tend to implicitly rely on the knowledge and respect of well-defined operating conditions (throughput, load, overload management, etc.) If defined at all, these operational limits let you know when things are about to go bad, and when they are good again.

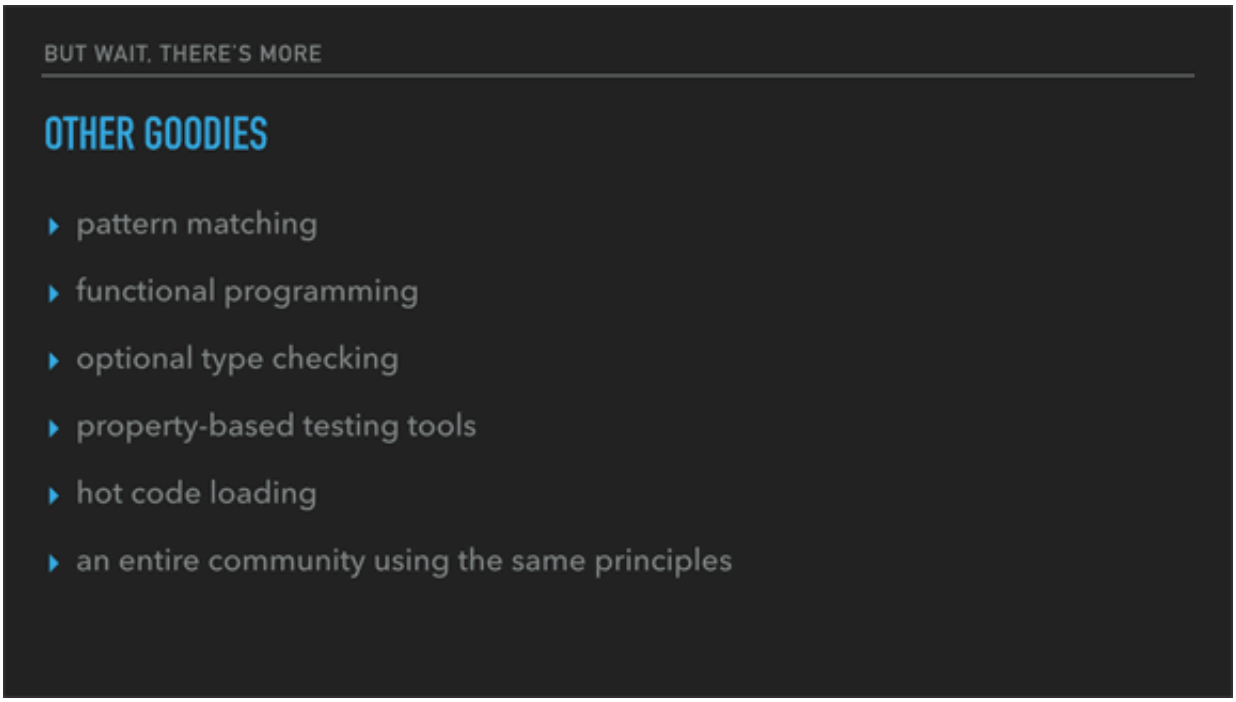
The problem with these limits is that as operators get used to them, and get used to frequently breaking them without negative consequences, there is a risk of slowly pushing the envelope towards the edge of the danger zone, where nasty large-scale failures happen. Your reaction time and margin to adapt to higher loads erodes, and you eventually end in a position where things are constantly broken with no respite in sight.

So we have to be careful and aware of this kind of thing, and to the importance that people using and operating the software has on it. It is always harder to scale up a good team than it is to scale up a program. Plan for emergencies even if they don't happen; they will some day and you'll be happy you ran simulations and have recipes to follow to fix it all up.

In the case of my flight, as I said, nobody was injured. Still, this is the

circus that was deployed for it all: busses to escort passengers back to the terminal, since moving a stranded plane could be risky. Pick-up trucks to escort the busses safely from the runway to the terminal. Police cars, a whole lot of fire trucks, and that black car that I don't know what it does but I'm sure it's super useful.

They deploy all of that despite everyone being fine, despite planes being super reliable. They do things right.



Here's another bunch of things you gain by using Erlang. I don't really have much to say about them, just that I do tend to have some kind of interest in you switching to use it, so here it is.

The last point is worth commenting though. One of the risks that happen in languages that are very flexible in their approach in system design is that libraries you use may not want to do things the way you feel would be appropriate in your case, and you're left either not using the lib, or having to operate codebases with an incoherent design. This doesn't happen in Erlang as everyone uses the same proven approach to do things.

In a nutshell, the Zen of Erlang and 'let it crash' is really all about figuring out how components interact with each other, figuring out what is critical and what is not, what state can be saved, kept, recomputed, or lost. In all cases, you have to come up with a worst-case scenario and how to survive it. By using fail-fast mechanisms with isolation, links & monitors, and supervisors to give boundaries to all of these worst-case scenarios' scale and propagation, you make it a really well-understood regular failure case.

That sounds simple, but it's surprisingly good; if you feel that your well-understood regular failure case is viable, then all your error handling can fall-through to that case. You no longer need to worry or write defensive code. You write what the code should do and let the program's structure dictate the rest. Let it crash.

That’s the Zen of Erlang: building interactions first, making sure the worst that can happen is still okay. Then there will be few faults or failures in your system to make you nervous (and when it happens, you can introspect everything at run time!) You can sit back and relax.