Which companies are using Erlang, and why? #MyTopdogStatus

2019-09-11 by Francesco Cesarini

Once upon a time, Cisco, Ericsson, Klarna, Goldman Sachs, T-Mobile, WhatsApp, Amazon and many other top companies kept a secret. Erlang was that badly kept secret. Many have heard of it, but few realise that it controls vast amounts of infrastructure, including the fixed and mobile networks we use on a daily basis. It was monumental when **Cisco** revealed that it ships 2 million devices per year running Erlang at the Code BEAM Stockholm conference in 2018. This translates to 90% of all internet traffic going through routers and switches controlled by Erlang. And have you heard about **Ericsson**? It has Erlang at the core of its GPRS, 3G, 4G and 5G infrastructure. With a market share of 40%, there's a high probability a program written in Erlang assigned the IP address your smartphone is using today (amongst other things).

Since being released as open source, Erlang has been spreading beyond Telecoms, establishing itself in other verticals such as FinTech, Gaming, Healthcare, Automotive, IoT and Blockchain. How can a technology created to switch phone calls make its way into these verticals? And why should you not only care but consider using it for your server-side development?

Erlang and Elixir are the top dogs in any tech stack and it's time we let more people and companies know about the BEAM technologies. We are kicking off with this first blog being part of the #MyTopdogStatus series, to showcase Erlang's success stories. We will follow with the Elixir focus next- so you have the bigger picture of both technologies!

A Solution to a Problem

One of the first things I try to find out when I meet a programming language inventor is what problem were they trying to solve when creating the language. The range of answers is fascinating, and often reveals if it is -or is not- fit for purpose. Ask any of the co-inventors of Erlang and you will get a unanimous response. They were trying to understand how to better build the next generation of telecom systems, the only systems back in the late 90s which had to be scalable, fault-tolerant, predictable and maintainable. They never set off with the intention of inventing a programming language, but the solution to their problem happened to be just that. Programming languages successfully created to solve a problem are the ones to keep an eye out for, as they are, without a doubt, the right tool for the job. If the answer to why a language was invented is experimenting with esoteric computer science concepts, listen, learn, but beware.

At the time of Erlang's conception, telecom systems were the only systems which had to be scalable, handle peak loads predictably and never fail. The internet changed it all. When Erlang was released as open source, telecom grade resilience and scale gave way to web-scale, which in turn gave way to mobile applications and connected devices, which, through the inception of 4G and 5G gave way to IoT.

How was this transition possible, and why was it so seamless? If you are switching phone calls, SMSs, publish/subscribe, instant and mobile messages, credit card transactions, money, stocks, medical records, online gaming command sequences, blockchain propagation or telemetry data, the business logic is still the same. And whilst it might be ok to lose the odd SMS, or message, or for a device to be offline for hours and not transmit any data, it is absolutely not cool to lose a financial transaction which involves money, stocks or cryptocurrencies. Basically, what changes in all these systems are the tradeoffs in scalability, reliability and consistency of the data, all items which are programming language agnostic and taken care of in the business logic of the system. Everything else remains the same.

There is no better example of Erlang's reliability than the English **National Health Service (NHS)**. The NHS obviously requires high availability and handles over 65 million record requests every day through its Spine, a centralised point allowing the exchange of information across national and local systems. Using Riak (written in Erlang), the NHS has managed 99.999% availability for over five years.

From health to the advertising industry - **AdRoll** receives an average of 500,000 real-time bid requests per second (spikes are many times that), with a substantial financial value attached to each. They use Erlang/OTP to manage the real-time bidding platform on the Amazon EC2 platform, pairing up advertisers with users in milliseconds.

Many countries rely on Erlang for their immediate payment switches, allowing instant bank transfers and bill payment services. **Vocalink, a Mastercard company**, is shipping financial switches implemented in Erlang. And they are far from being alone. Other areas where Erlang is used in anger includes private and public blockchain, payment and credit card gateways, banking APIs and more traditional infrastructure management. If you scratch under the surface, you will find an Erlang team in all of the major banks and financial institutions. And those few not using it will be using RabbitMQ, CouchDB, RiakKV or one of the other open source Erlang applications.

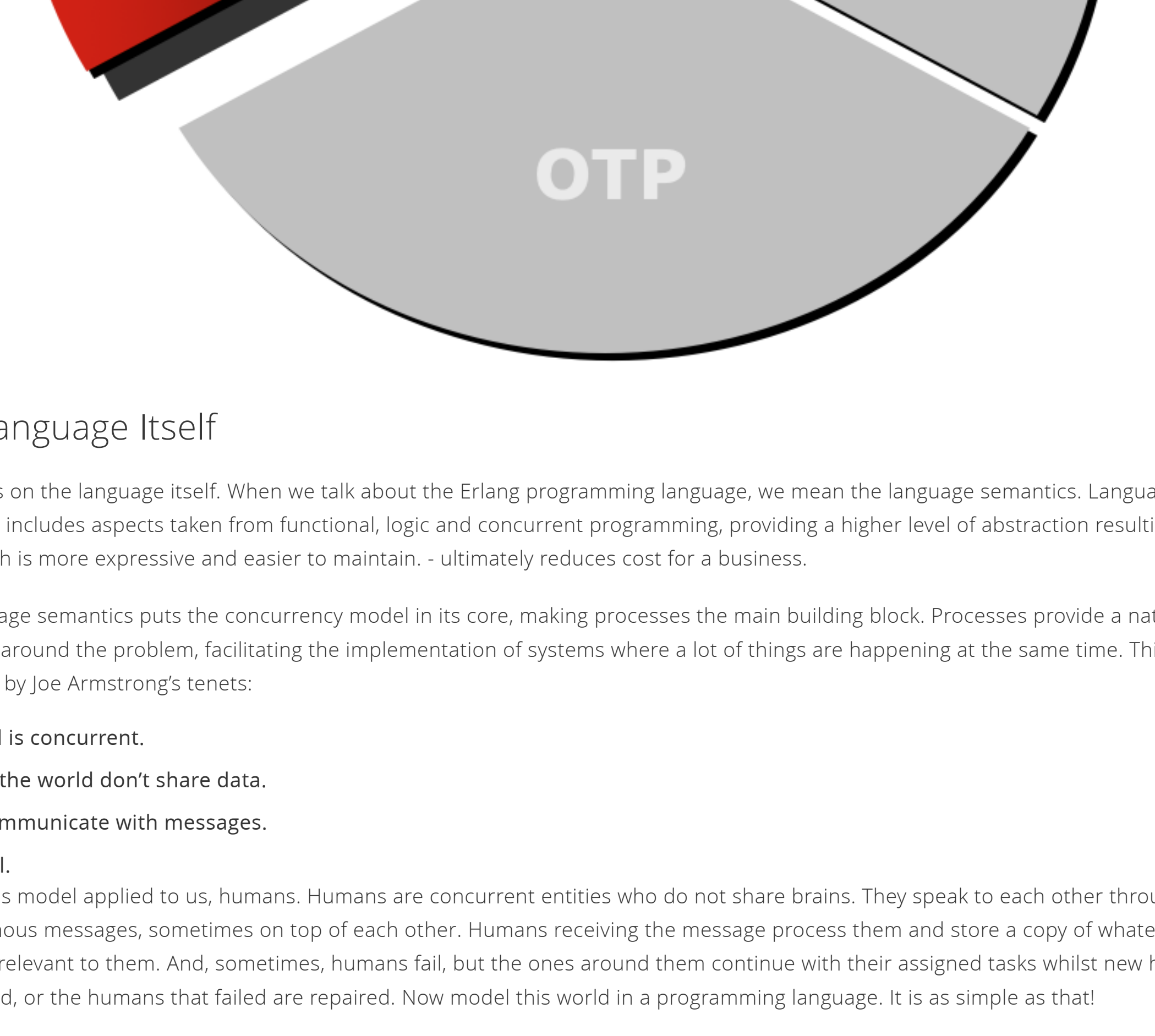
The investment banking giant, **Goldman Sachs**, use Erlang in part of its hedge fund trading platform. The Goldman Sachs platform is a very low latency (microseconds) event-driven market data processing, strategy, and order submission engine. Erlang is used to help them deliver real-time changes in response to market conditions.

What about massively multi-user online gaming? **Nintendo's** Switch has sold over 34 million consoles; the system uses an Erlang based messaging system to handle millions of concurrent connections. Other popular gaming companies such as Riot Games, use Erlang too. League of Legends had up to 7.5 million concurrent players at a time, with an Erlang messaging system allowing them to all chat simultaneously, without interruptions.

Even one of the world's most used online dating apps used Erlang. Grindr used Erlang in its stack to manage it's 3.2 million daily active users to handle up to 2000 messages per second. Moving to the BEAM helped them maintain a reliable system with a minimal number of outages.

What's in a Programming Language?

A programming language, no matter how good it is, is on its own only of limited use. Combine it with a powerful optimised runtime and middleware which abstracts the scalability and reliability of your system, your developers get the foundation of a very powerful ecosystem. The Erlang ecosystem is not just a programming language. It is a family of programming languages, a virtual machine as powerful as an operating system and a set of middleware libraries which abstracts many of the recurring (and tricky) problems you have to deal with when working with scale and resilience.



The Language Itself

Let's focus on the language itself. When we talk about the Erlang programming language, we mean the language semantics. Language semantics includes aspects taken from functional, logic and concurrent programming, providing a higher level of abstraction resulting in less code which is more expressive and easier to maintain. - ultimately reduces cost for a business.

The language semantics puts the concurrency model in its core, making processes the main building block. Processes provide a natural way to reason around the problem, facilitating the implementation of systems where a lot of things are happening at the same time. This is best described by Joe Armstrong's tenets:

The world is concurrent.

Things in the world don't share data.

Things communicate with messages.

Things fail.
Picture this model applied to us, humans. Humans are concurrent entities who do not share brains. They speak to each other through asynchronous messages, sometimes on top of each other. Humans receiving the message process them and store a copy of whatever they believe is relevant to them. And, sometimes, humans fail, but the ones around them continue with their assigned tasks whilst new humans are created, or the humans that failed are repaired. Now model this world in a programming language. It is as simple as that!

BT Mobile (formerly T-Mobile) in the UK uses this approach for many of their core systems. One of them is the Third-Party Gateway, a system written to handle all machine to human SMSs. This includes alert services such as traffic, financial and weather alerts, TV voting, competitions and reminders. Millions of messages are sent and received each day, with many spikes in traffic during TV votes and special promotions. Each SMS being sent or received is a process. The developer needs to reason around sending and receiving a single SMS, with scale handled through the creation of new processes.

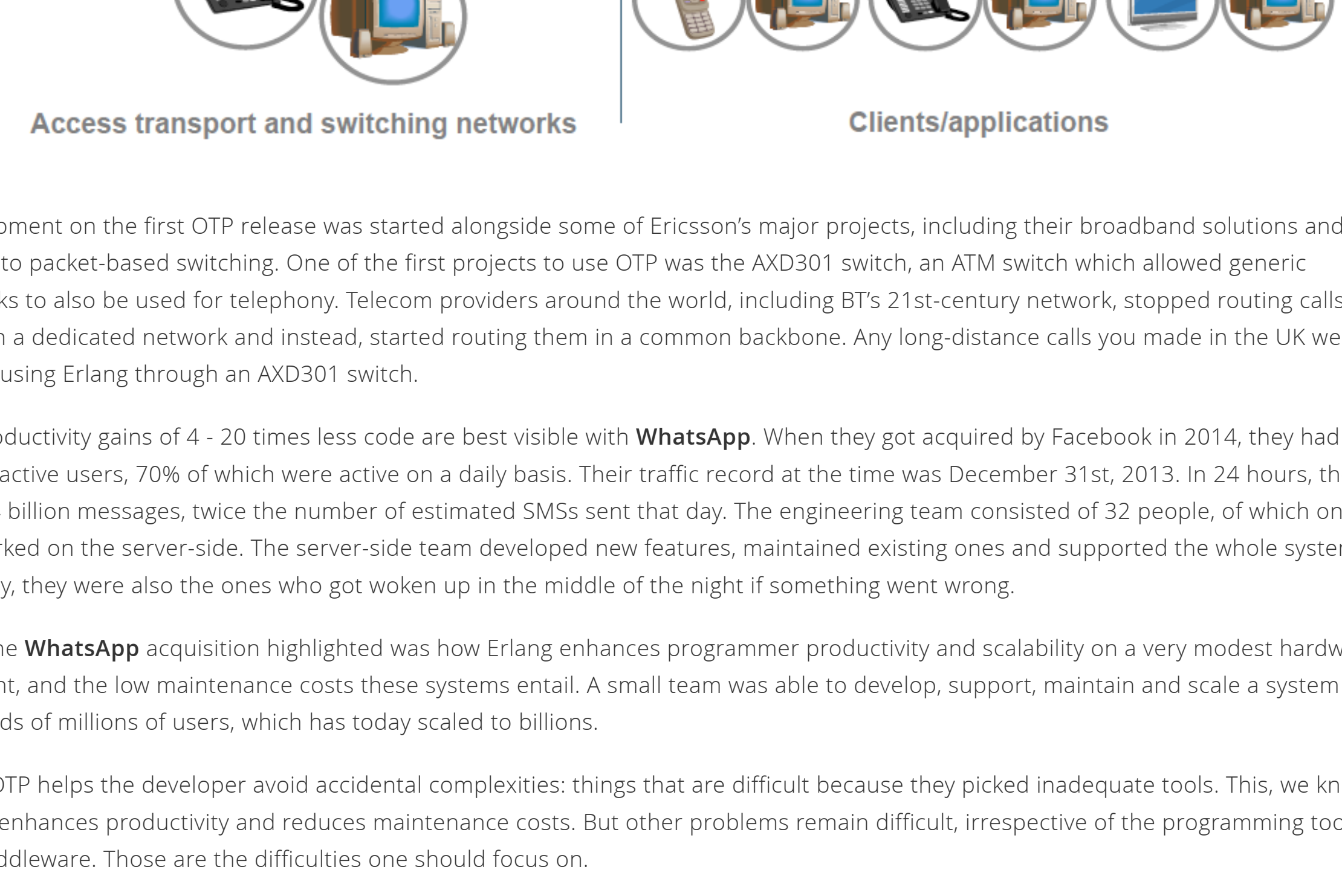
Online betting is another industry with extreme spikes and loads to manage, a busy day for betting agencies is like Black Friday at Amazon, but the prices change every second. Many online betting companies, including **bet365**, use Erlang to manage 100's of thousands of concurrent users.

Sometimes, I hear complaints that the Erlang way of thinking requires a serious mind shift. The problem is not the mind-shift; it is unlearning the unnatural models other programming languages have taught us and going back to basics using Joe's tenents. If you understood them, understanding Erlang will be just as easy.

The OTP Middleware

OTP is a set of frameworks, principles, and patterns that guide and support the structure, design, implementation, and deployment of Erlang systems. The innovation in OTP which other languages are copying are the abstract principles used to describe the software architecture. Processes are given a design pattern such as servers, finite state machines, event handlers or supervisors, all packaged in reusable libraries. These libraries have built-in support for debugging, software upgrade and generic error handling. They also abstract and take care of all tricky edge cases which occur with concurrent programming, providing a solid and tried approach to problem-solving. It makes the code easier to understand and maintain, reduces maintenance costs and stops developers from reinventing a square wheel.

Motorola funded a study which involved rewriting a telecommunication system used by the emergency services from C++ to Erlang, focusing on productivity gains. Depending on how you calculated, the code reduction in the Erlang system achieved a result of 4-20 times less code. The 20 times reduction assumed the OTP libraries to be part of the Erlang standard libraries, which they are. As C++ did not have a generic OTP, the original project had to implement a good part of it. That seems to be the norm with any project dealing with concurrency at scale. The norm also being that this implementation is often bug-ridden.



Development on the first OTP release was started alongside some of Ericsson's major projects, including their broadband solutions and foray into packet-based switching. One of the first projects to use OTP was the AXD301 switch, an ATM switch which allowed generic networks to also be used for telephony. Telecom providers around the world, including BT's 21st-century network, stopped routing calls through a dedicated network and instead, started routing them in a common backbone. Any long-distance calls you made in the UK were routed using Erlang through an AXD301 switch.

The productivity gains of 4 - 20 times less code are best visible with **WhatsApp**. When they got acquired by Facebook in 2014, they had 450 million active users, 70% of which were active on a daily basis. Their traffic record at the time was December 31st, 2013. In 24 hours, they sent 54 billion messages, twice the number of estimated SMSs sent that day. The engineering team consisted of 32 people, of which only ten worked on the server-side. The server-side team developed new features, maintained existing ones and supported the whole system. Basically, they were also the ones who got woken up in the middle of the night if something went wrong.

What the **WhatsApp** acquisition highlighted was how Erlang enhances programmer productivity and scalability on a very modest hardware footprint, and the low maintenance costs these systems entail. A small team was able to develop, support, maintain and scale a system with hundreds of millions of users, which has today scaled to billions.

Using OTP helps the developer avoid accidental complexities: things that are difficult because they picked inadequate tools. This, we know, always enhances productivity and reduces maintenance costs. But other problems remain difficult, irrespective of the programming tools and middleware. Those are the difficulties one should focus on.

The Virtual Machine

Erlang uses a Virtual Machine (VM) when executing code; an approach similar to NodeJS, Java and Ruby. Programs are compiled to low-level instructions called Byte Code. The BEAM is the most commonly used Erlang VM which executes this byte code. It can be seen as an operating system running in a container, a virtual instance, on another operating system or directly on the bare metal. The BEAM has, for the last two decades, been optimised and under heavy load rendered predictable for the types of problems Erlang is good at solving. It is capable of concurrently handling millions of processes, ensuring each process displays soft real-time properties and is fairly treated. Throughput remains constant irrespective of load. If your system handles 100,000 requests per second, it will take a second per request if 100,000 are being served simultaneously. If the number of requests increases to 200,000, throughput will remain the same, but latency will increase to 2 seconds. What is important is that all requests are served with no degradation of throughput, even when spikes happen. With many other Virtual Machines, an increase in requests often leads to a degradation of service, possibly grinding to a halt. Not with the BEAM, as it was built for scale.

As processes do not share memory, memory management is done on a per-process basis. Schedulers allow a process to execute a predefined number of instructions before suspending it, ensuring fair execution. If the process hadn't completed its tasks, it will be scheduled to continue executing after all other processes have had their go. This will result in a constant execution time for all requests, unlike the Java Virtual Machine, where garbage collecting affects all requests going through the system at that particular point in time or NodeJS, where a request will execute until completion, blocking other requests on that thread until it is done. The RubyVM has not had decades of engineering resources put into it, and as a result, will not scale.

There are no other Virtual Machines around with the properties of the BEAM, making it unique in its capability to predictably handle massively concurrent spikes of traffic. It is not the fastest VM, but it is the most stable and predictable one. How do the advantages of using the BEAM apply in practice?

Imagine a VM handling a million TCP/IP connections, each managed by a process serving a customer, where load share and CPU allocation for each process is fair and constant. This equates to a predictable user-experience, where a million users can be served on a single instance. All the programmer has to worry about is developing code to handle a single user (a process), and the VM will automatically scale it to millions. And if a request fails because of corrupt data or a bug in the software, all other requests will continue executing independently of it, isolating the failure.

WhatsApp achieved 2 million TCP/IP connections (each managed by a process) on a single BEAM instance back in 2012, using the connections to serve messages and notifications whilst minimising its hardware footprint. This, in turn, greatly reduced their infrastructure costs and the size of the team needed to maintain it. Similar numbers were achieved using Elixir and the Phoenix framework in 2015. From messaging, vertical scalability on a single node we move to the web.

Bleacher Report, a news app and website focusing on sports, had a similar impact on operational and infrastructure costs when changing Virtual Machine. When they migrated from Ruby to the BEAM, they were able to reduce their hardware requirements from 150 servers to 5! And Quicken Loans launched a new mortgage offering with a Super Bowl ad, knowing with confidence that their servers would handle the traffic with ease. In fact, they had trouble load testing it with traditional tools, in the end, they needed to fall back to using Erlang to load test Erlang.

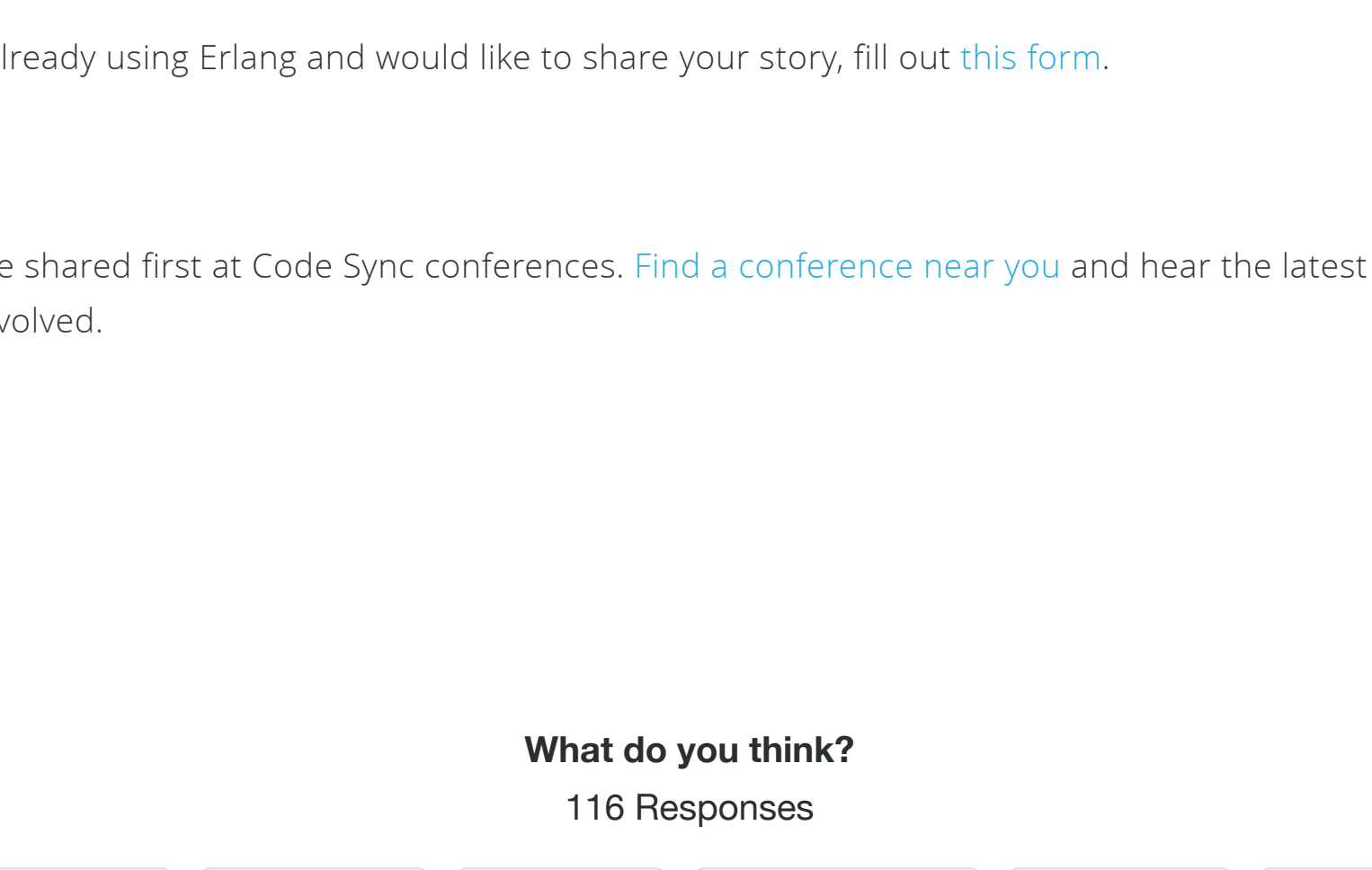
You can copy OTP and its libraries, but if it does not run on the BEAM, you can not emulate the semantics. This is why using OTP ported to .net or the JVM will not work. On the other hand, many languages are being ported to the BEAM VM, making full use of OTP, Elixir being the most popular. But let's not forget Luerl, Lisp Flavoured Erlang, Efene and at least a dozen more.

Is it still a badly kept secret?

In this blog post, we are just scratching the surface. Many companies are using Erlang to power their server-side infrastructure. They are doing it to reduce development costs whilst ensuring their systems scale and are resilient. So resilient that their end-users, often focusing on the glitzy app or website, do not even know it is there. Isn't it time you looked into Erlang as well?

if you're part of the pack already using Erlang and would like to share your story, fill out this form.

MyTopdogStatus



Want to see more from our #MyTopdogStatus campaign? Check out our blog on which companies are using elixir, or head to our hub for videos, news from our team and more Erlang & Elixir news. Check it out now!

Be included in this blog

If you're part of the pack already using Erlang and would like to share your story, fill out this form.

Hear it first

Many of these stories were shared first at Code Sync conferences. Find a conference near you and hear the latest Erlang success stories, directly from the teams involved.

Go back to the blog ›

Tags: *OTP, Erlang*

Share