The Main (flux) man

Jonni Bidwell learns about IoT platforms, embedded programming, messaging protocols and bootloaders from veteran coder and Mainflux founder Draško Drašković.

Draško Drašković is CEO and cofounder of Mainflux, an open source, industrial, Internet of Things Cloud platform written in Go and Erlang. He holds an MSc in electronics, telecommunications and industrial control systems from Belgrade University and is an expert on semiconductors, communication protocols and lots of things we can't even begin to understand.

Draško's worked on a number of FOSS projects, including OpenWRT, U-Boot and OpenOCD, and has worked for a number of major hardware providers. This includes Texas Instruments, where he helped develop the popular OMAP chips which can be found in most every 2G and 3G mobile phone. Lately, he's been working on IoT and 5G technologies, and most recently has been dabbling with Blockchain technology as applied to the domains of security, data integrity and device identity. In a joint effort with two of his Mainflux colleagues, he has just finished a book entitled *Scalable Architecture for the Internet of Things*, which will be available by the end of 2017. We caught up with him at the O'Reilly Software Architecture Conference, held in a swanky hotel in London, to get the lowdown.

Linux Format: How did you get into Linux and open source in general?
Draško Drašković: I got into open source relatively early, back in my student days. The whole computer centre was running Linux mainframes and we used open source software for developing student projects. I recognised the benefits of software freedom and the GNU philosophy corresponded to my own point of view. Since then I've used almost exclusively free software for my own projects. I try and use, promote and ship open source products wherever possible for industry projects.

Just after finishing my studies and obtaining my MSc. I started working at the university's Innovation Centre, which was part of its computer centre and sponsored by the government's Ministry of Technology. One very interesting project was for security in a Serbian-localised distribution. At this point I switched from being an advanced Linux user to becoming a bona fide kernel hacker. We got help from former students who were now professors in other countries, because we wanted as many quality contributions as possible so that our distribution could be as secure as possible.

After this project I joined a French company in Belgrade. It worked with semiconductors, which has always been my area of interest. After a few months we had a big project with Texas Instruments (TI) in Nice on the French Riviera, and I started working with them after this. Since then I've moved between Nice and Paris working for various companies in the semiconductor and wireless communications domains. And practically always used Linux.

LXF: What are some of the highlights of your (considerably impressive) career so far?
DD: Besides my time at TI, I worked at a startup called Sequans Communications where we built a chip that was used in the world's first 4G phone, the HTC Evo. I also worked on the Devialet high-fidelity audio system (**www.devialet.com/en-eu/phantom-speaker**), which is basically a distributed wireless computer system for audio. It achieved worldwide popularity and won lots of prizes. It's interesting in that it's just a Linux machine with a lot of support for wireless streaming and specialised FPGA circuitry for audio processing. Besides that, I've worked on OpenWRT, OpenOCD, U-Boot and Mainflux.

LXF: You have a strong background in engineering and electronics. Is that necessary for low-level kernel programming?
DD: Yes – knowledge of digital electronics and computer architecture is essential here. It's important to understand HW communication and control interfaces and protocols – like GPIO, UART, I2C, SPI, USB and similar, as well as networking (wired and wireless). Linux lower layers are very exciting to hack on, but if you take, for example, just the Linux Wi-Fi subsystem—it's a very complex area and demands a lot of knowledge of how hardware functions and sometimes even physics around radio transmissions over the ether. So you need to understand how the protocol works at the physical layer, but also the layers above that are specific to the device.

ON HIS LIFE-LONG LOYALTY TO LINUX
"I try and use, promote and ship open source products wherever possible"

LXF: Sounds hard. Tell us more about coding for embedded systems.
DD: Embedded programming poses a lot of challenges. But it's extremely exciting. So peripherals connect to the CPU over the standard hardware interfaces I mentioned earlier, and they also have their own controllers with their own internal registers. These are embedded in the SoC at a particular address space. RAM is an important peripheral, and that has its own controller. To boot Linux you need to »

» set up, among others, the memory management unit (MMU) to have virtual/physical address division. Both RAM flash storage are often very limited, so that's a challenge, too. Early on in the development process hardware drivers are probably not working correctly and debugging them is hard, but OpenOCD helps a lot.

LXF: Intriguing – can you tell us more about this openOCD.
DD: OpenOCD (open On-Chip Debugger) is a JTAG debugger. JTAG is practically a standard hardware protocol built into every System on Chip (SoC) at the hardware level. It means you can use hardware signals to stop the core, progress instruction by instruction, examine the chip's state and so on. In order to issue these commands to the chip you need PC software that's able to speak and understand the protocol, it needs to send commands and process the chip's responses.

Traditionally this software, and the hardware dongles that went with them, were very expensive. Big companies have enough budget for them, but for individuals – hobbyists and enthusiasts – it's very expensive. The openOCD project uses extremely cheap USB or UART dongles to connect to the chip's sockets, and then the software can do all the low-level debugging. At this level there's not even a serial console. I did a lot of work here on the MIPS architecture. ARM was already pretty well supported on openOCD, but a lot of things were missing for MIPS.

LXF: Tell us more about the work you do now. What open source tools do you use?
DD: Most of the stuff that I work with deals with low-level and embedded programming. I've been working on low-level kernel code both in open-source and commercial products. This has mostly involved working in a Platform team of the company, the team that delivers bootloader and Linux BSP (Board Support Package). This is a bootable Linux image for that architecture including various Linux device drivers.

Throughout my career I've worked mostly in semiconductor companies, where the product is a system on chip (SoC), and then you would start building Linux support from FPGA prototypes to full-blown ASICs. My focus is on hardware and electronics, so I build devices, then write device drivers and so on to give those devices a brain. So starting from the silicon, I build up an abstraction layer, the system layer, and then build applications on top of that. In any case, you start from bare metal, assembly, and then you build your system, slowly, piece by piece until eventually you have the luxury of booting into a Bash shell.

When you deal with low-level programming, your using GCC to compile code, often written in C, sometimes assembler, and GNU Make. Those are used pretty extensively. When you're dealing with constrained devices, microcontrollers for example, you're often programming just a bare metal application. You don't even have an operating system, or more intelligent software. Often here you'll be cross-compiling on your desktop machine for a different target architecture, like ARM or MIPS. Then somehow transferring it (you probably don't have the luxury of networking at this point) to the chip's RAM.

In this line of work Yocto-derived Linux distros (**www.yoctoproject.org**) and the Buildroot (**https://buildroot.org**) framework are standard tools. When you have a slightly more powerful machine, say a Raspberry Pi, or anything that can run Linux, then you need a bootloader in order to load your Linux image. U-Boot is a popular choice here.

LXF: You've contributed to U-Boot – what was the nature of those contributions?
DD: Yes, I've contributed to U-Boot, mostly for ARM devices. My contributions have been dealing with the processor's instruction cache and data cache, and initialising and setting the memory management unit on the chipset.

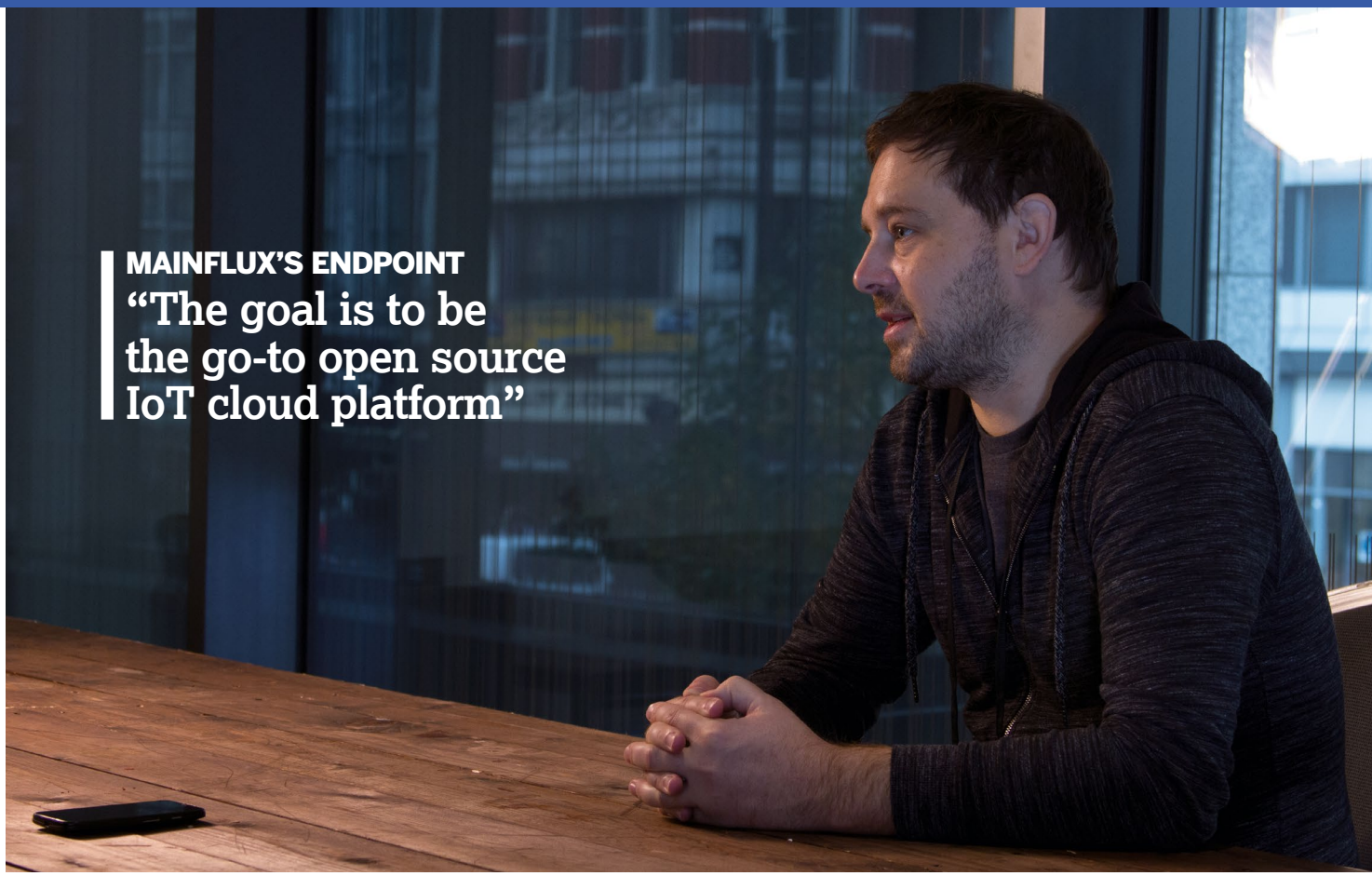
LXF: I think most of our readers are familiar with GRUB, and possibly some other bootloaders on x86 devices. But U-Boot always seemed more hardcore?
DD: Well U-Boot supports x86. The point is when you're starting from scratch, and this is what U-Boot in general has to do: it starts from the first instruction at the entry point of the chip. It's set up in hardware to jump to this address when it's powered up, and it expects to find instructions there that make sense and that lead toward configuration of your SoC: the processor, the peripherals... everything you need to boot Linux. The most important thing to configure is the memory, because without this being correctly set up you won't be able to load a Linux image into it.

Next you must configure the flash drive, or whatever non-volatile storage the device has, so that the kernel image can be read. So U-Boot's goal is to configure the system, fetch an image from non-volatile storage and load it into RAM. U-Boot then jumps to this address and from here Linux takes over. There are a lot of applications where people are using Intel's architecture in an embedded context where you don't need a fully featured bootloader like GRUB. You need to use U-Boot here, in these industrial contexts, and it does the same on x86 as it does on ARM, MIPS or any other architecture.

LXF: Okay, so I guess it's just that with desktop PCs the BIOS or UEFI does a lot of the initial configuration for you...
DD: That's it, so the low-level stuff U-Boot does is equivalent to what happens in the BIOS.

LXF: Mainflux sounds interesting, but – and do forgive our ignorance here – what is it?
DD: Mainflux is an open-source, Apache-2.0 licensed IoT platform with the ambition of building an industrial infrastructure. It's a middleware that can be used for building vertical IoT solutions and bringing smart connected products to the market faster.

The idea is to build a multi-protocol system with a modern architecture based on a set of microservices. It's built in Go and deployed in Docker. The Mainflux system can be deployed in the Cloud or on-premise and is designed to be highly scalable, to process connections from lots of sensors. The system has been built with security in mind, and respects modern standards such as JSON Web Signature (JWT) and TLS, as well as fine-grained, policy-based authorisation. All work is published on Github: **https://github.com/Mainflux**.



MAINFLUX'S ENDPOINT

"The goal is to be the go-to open source IoT cloud platform"

LXF: So Mainflux originally grew out of another project connected to OpenWRT (the open source firmware for routers)?

DD: My work on OpenWRT and community contributions was done through a fascinating project called WeIO (**www.we-io.net**). It's a Linux prototyping board that we designed from scratch in Paris, both HW and SW. The idea was to democratise Linux prototyping and development of connected IoT objects through an intuitive Python-based SDK that could interface with the underlying HW.

We wanted to design something that was similar to the Raspberry Pi, but we were doing this before the Pi existed, around 2010-11. Trying to find inexpensive Wi-Fi chips was difficult at this time. There weren't really Linux boards or the maker spaces that we have now – all that existed was Arduino. I knew that Linux could help here, but back then the maker community wasn't so interested in complex devices, they were more about simple microprocessors.

So the challenge was to somehow wrap that complexity that goes with Linux and present a simple API, and an application that exposes that API, that resembles programming a microcontroller with Arduino. We successfully crowdfunded this project (**www.indiegogo.com/projects/weio-platform-for-web-of-things#**) and built the hardware. That worked very well locally: on the LAN you could connect to your boards and program them easily.

But we wanted to go a step further and connect them to the Internet, so that the boards could be programmed remotely. To do this you need some sort of remote cloud to which those devices would connect. Then your application would also connect to this centralised server, which would then serve as a bridge, relaying messages between the application and those devices. It turns out that this sort of centralised IoT solution, with all the capabilities we wanted and the licenses we wanted to see, didn't exist in open source form. So we started building it, and this became Mainflux. This was about two and a half years ago. In the meantime, other members of the community had the potential to build something beyond the enthusiast and maker mindset, and build something targeting the industry. The goal is to be the go-to open source IoT cloud platform.

LXF: Mainflux just joined the Linux Foundation's EdgeX Foundry group. We've been hearing lots about Edge computing lately. Can you explain to us a little about it?

DD: The whole ecosystem of IoT and connected devices is rich and there are a lot of strategies for how to interconnect those devices. This isn't something new: machine-to-machine communication has been around for decades, but new protocols have been invented for these workloads in the past couple of years. Different use cases call for different strategies, so when you go to the industrial context – in the factory for example – there are latencies that must be respected and the responses from the network must be extremely fast.

This is the case in the factory, and it's also the case for autonomous vehicles. If such a vehicle doesn't receive an answer from processing some sensor readings, then it might not stop at a red light, it might crash or worse. So those latencies are extremely important.

The second thing that is important is that we're expecting a huge number of devices to be connected soon. Some are predicting 50 billion devices by 2020, some not quite so high, but a huge number nevertheless. So the cloud won't be able to accept all the data sent by these devices and we need some kind of filtering, processing and aggregation on the Edge. And for economical purposes it makes sense to send only the necessary data. This is what's known as Edge Computing, or Fog Computing if you like – the Cloud comes down to the ground.

The Edge ecosystem hadn't really been standardised. There were a lot of implementations – practically every company that made an IoT gateway was doing it their own way with their own software. These machines mostly run on Linux, but the software doing the connecting, security and the filtering and processing layers I just mentioned, this was all done with no standards in place.

At the beginning of this year, Mainflux was presenting at the Open Networking Summit in Santa Clara in Silicon Valley. We spoke to the Linux Foundation, in particular Dell, about joining what would soon become the EdgeX Foundry. This involved big players like AMD, Canonical, VMware and many more. I think there are more than 60 in the consortium now. Mainflux joined as an open source project, as an affiliate member that was there from the start. »

» What was recognised was that a set of technologies used to implement Mainflux, which is mostly written in the Go programming language, was very interesting. At least the architectural patterns; Mainflux is written as a set of microservices that you can easily port to the gateway. This kind of architecture is very scalable, and is suitable for Cloud and on the IoT gateway itself. So we contributed our code to the EdgeX Foundry project. Now we want to see it moving towards a complete Go implementation and set up an industrial standard for how IoT Edge devices should communicate in this ecosystem, and enable the industry to produce quality gateways and ultimately solve this mess on the Edge.

Our main goal is building modern-edge systems and IoT gateways based on innovative EdgeX technology that would be connected and managed by the Mainflux cloud. This way we will have an end-to-end vertical solution for industrial IoT, based on completely open-source components, published under the Apache-2.0 license. Mainflux and EdgeX go really well together: Mainflux as the IoT cloud and EdgeX as an IoT gateway on the network edge.

LXF: We're particularly concerned about IoT security, mostly in the sense of manufacturers deciding to produce cheap consumer devices and then not issuing security updates...

DD: Security is extremely important because as soon as certain kinds of physical objects are connected to the Internet they have the potential to cause injury or even death. That's one side of it, but last year we also saw a huge hack of Internet-connected cameras that were used in a DDoS attack. The cameras acted as malicious clients, attacking major websites, and practically bringing down half of the Internet (see **www.forbes.com/sites/briansolomon/2016 /10 /21/hacked-cameras-cyber-attack-hacking-ddos-dyn-twitter-netflix**). Why is this possible? Because there are so many of them, we've never had this many connected clients before. Once someone hijacks these they have a considerable army of machines at their disposal.

Security is certainly one of the biggest challenges and must be addressed on many levels – both on firmware running on chips and in the cloud. Hardware must be secured via secure-boot and hardware fuses, and all firmware must be encrypted. Also, anti-tampering hardware mechanisms must be implemented. All communication must be encrypted and certificates must be in place.

In the cloud there must be proper authentication and authorisation services and secret keys must be kept in secured vaults. Secure, remote update over-the-air must be implemented, and this can be very challenging with embedded Linux devices. We're trying to do all of these things right with EdgeX Foundry and Mainflux systems.

LXF: How does Linux fit into the general IoT scheme? Some of our readers would like the idea of just apt updating their fridge for piece of mind, but that's probably not the future. For small devices that just need to send sensor readings down a wire it seems crazy to be running a full Linux kernel, even one that's thoroughly pared back.

DD: Right. For constrained devices a realtime OS such as Contiki (**www.contiki-os.org**), RIOT (**www.riot-os.org**), or the Linux Foundation's new Zephyr Project (**www.zephyrproject.org**) would be a better fit. More powerful devices will probably run embedded Linux using hand-crafted distros via the tools mentioned earlier.

LXF: How can Mainflux help here, what challenges remain and where do you think viable solutions lie?

DD: One of the main problems with IoT is the lack of standardisation, in the sense of intercommunication and interoperability. There are a lot of inadequate protocols in use right now: On the one hand we see people trying to use heavyweight protocols, like Websockets or HTTP, in small microcontrollers. These are protocols designed for a web browser running on a [relatively] powerful computer. On the other hand people are trying to revive old protocols like MQTT and use them in a modern context. We're also seeing new protocols like CoAP (Constrained Application Protocol).

Because of this diversity we try and build communication protocols interoperability into Mainflux. We had to build several servers that act as microservices and then build a bridge between them so that we can natively bridge those protocols. So when a machine speaks MQTT, for example, then an application connected over, say, Websocket, can understand it. Having this kind of interoperability, at least on the protocol level, will help connect a diverse set of objects. On the security level, what we're doing now is something that resembles what Microsoft is doing with Azure IoT and what AT&T is doing with its platform.

But we feel there's still room for improvement, particularly through the use of public key cryptography and symmetric keys, which exists in, for example, the mobile telephony sphere. This kind of infrastructure would bring more security, but they're way more complicated to implement. So with this complexity comes a corresponding increase in price in building the product and maintaining the public key infrastructure, which is currently hard. We feel there's a need for innovative solutions in this sphere, how we handle public key infrastructure in general in the industry, not specifically relating to IoT.

At Mainflux we have a feeling that this will be enabled through blockchain technology. Blockchains are suitable for being immutable databases for storing certificates, or hashes of your certificates. Through the blockchain you can be sure of the device's identity, which you can then use to authenticate and authorise the device's access to some resources. So through the blockchain a device can bring its identity with it and connect to different service providers. This is one possible direction and we feel there'll be innovations in this sphere, but that's something for the future! LXF

ON THE CHALLENGE OF SECURITY

"In the cloud there must be proper authentication and authorisation service"