TECHNOLOGY WHITE PAPER

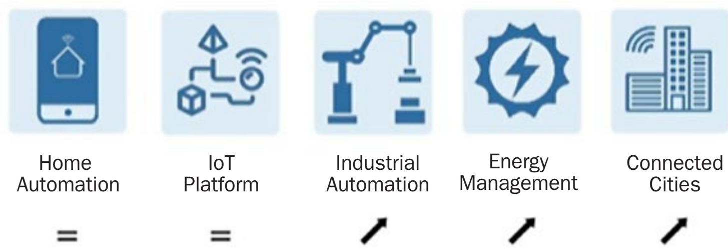
Driving the IoT with Open Source Java

The IoT Brings Challenges for Software Development Teams

The Internet of Things (IoT) is something that will have a profound impact on most enterprises over the next few years. Organizations like Gartner predict that there will be nearly 26 billion connected devices by 2020. The impact of being able to collect and analyze huge volumes of fine-grained data on everything from healthcare to the global climate cannot be underestimated.

Moore's law of increasing transistor density has an inverse, which is that the same complexity chip becomes exponentially cheaper. The cost of providing computing power and network connectivity in virtually any device has now fallen to the point that it is not an issue when considering the bill of materials. To demonstrate this, consider the Raspberry Pi Zero, which has a 32-bit ARM CPU clocked at 1GHz with 512Mb RAM and can drive a quite usable desktop computing environment. All for only $6.

Top 5 IoT Industries – and trends from previous years



Source: Ian Skerrett, IoT Developer Survey 2017

Eliminating the issue of hardware cost is only part of making the Internet of Things a reality. The more significant part is how to develop software that can handle three tasks:

1. Collect data from attached sensors.
2. Send the data, potentially with some preprocessing, to the cloud for aggregation and analysis.
3. Making decisions about how to control actuators, either using sensor data directly or based on results returned from the Cloud.

The volume of data produced by the IoT has led to the widespread adoption of a three-tier architecture rather than the more familiar client-server. With sensor devices generating vast quantities of data it is not feasible (or necessary) to send all this data to the cloud for analysis and storage, even with what appears to be limitless computing resources. To reduce sensor data to manageable levels sensor devices can be connected to gateways. These gateways perform a level of filtering, processing and aggregation on the raw data before sending refined data to the cloud.

Open Source Technologies in the IoT

Developing embedded software is notoriously hard work. There are a number of challenges that make it this way:

The complexity of writing software that interfaces to external sensors and actuators. This often requires developers to have an understanding and ability to use low-level protocols, direct memory access, bus mastering, etc.

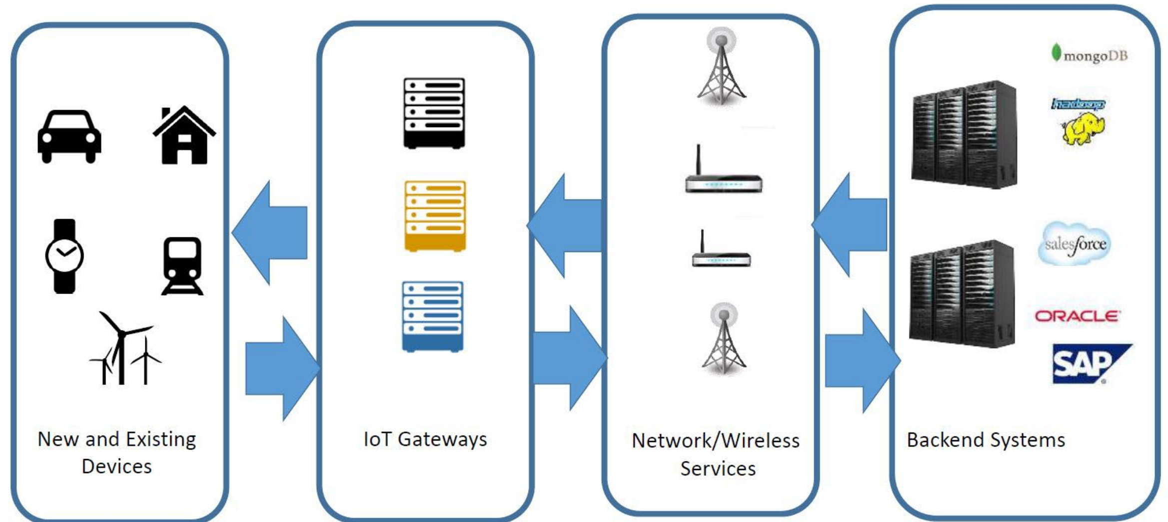
The requirement to modify applications, or port them, to each new revision of hardware that is used. This is time-consuming and introduces the potential for new bugs without adding any new value.

The difficulty of making applications reliable. The use of low-level languages like C and C++ expose developers to many potential problems. The use of explicit pointers and memory management can lead to abrupt application termination as well as memory leaks leading to the need to restart devices on a regular basis.

TABLE OF CONTENTS

The IoT Brings Challenges for Software Development Teams 2
Open Source Technologies in the IoT 2
Zulu: An Open Source Build of OpenJDK – Java for the IoT 3
Zulu Advantages for IoT Development Teams 4
What's Holding Java Back in the IoT? 5
Runtime Performance 5
Memory Footprint 5
Low-Level Operations For Device Access 6
Conclusion – Why Open Source Java is the Best Choice for the IoT 6

IoT Architectures



Ian Skerrett, Eclipse Foundation

The use of explicit pointers in languages like C and C++ also presents would-be hackers with greater opportunities to subvert devices. The use of buffer overruns is a standard technique for gaining unauthorized access.

Access to resources. Skilled embedded developers are a rare breed and can be difficult to recruit and retain. Salary expectations are higher than for many other developer groups.

Adequate tooling to help speed the development of embedded applications. Most embedded development still relies on command line tool chains with complex cross-compiler setups making it considerably slower than other forms of development.

Lack of easy code reuse. Enterprise development relies heavily on libraries and frameworks to eliminate most of the common code required for typical applications. In the case of embedded device development, there are very few libraries and frameworks and those that do exist tend to be commercial with expensive licensing fees. The impact of this is that the average embedded developer spends a lot of time replicating work for each new project.

The complexities of parallel programming. Moore's law combined with the physical limitations of the energy required to make clock speeds higher and higher has led to the widespread use of multi-core processors, even in embedded devices. Languages like C and C++ do not have the concept of multithreaded programming built in, requiring the use of external libraries. Writing code for co-operating threads, which is almost always needed, poses further challenges for the development of reliable, robust code.

Remote management. Once devices are deployed into the field, they may well be spread out over a large area in potentially inaccessible places (think of devices that might be used to monitor and control street lighting). Remote management of these devices can be built into applications but adds complexity and size to the end product. An even greater challenge is how to update application code over a network without physical access.

Zulu: An Open Source Build of OpenJDK – Java for the IoT

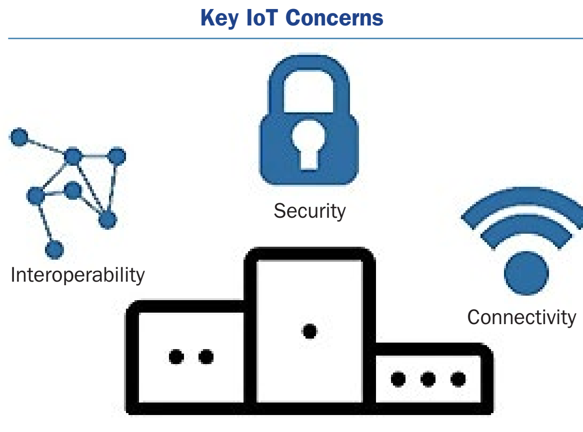
An ideal solution that addresses all these challenges is the Java platform. The most flexible and cost-effective open source binary distribution is Azul Zulu. Azul Zulu also has significantly less-encumbered licensing than other commercial alternatives.

Let's look at what Zulu is and how it makes the development of IoT embedded software simpler and quicker with more reliable and secure deployments.

The source code for the Java Development Kit (JDK) was released by Sun Microsystems under an open-source license when they created the OpenJDK project in 2006. Over the space of two years, the complete JDK source code was released, so it is now possible to build a JDK, conformant to the Java SE specification without the use of any closed source.

Azul uses the OpenJDK source to create binary distributions of Zulu; Zulu is a fully-tested and supported build of OpenJDK. Versions are available for Intel and ARM processors in both 32 and 64 form as well as PowerPC 64 bit. OpenJDK is the reference implementation of the

Key IoT Concerns



Source: Ian Skerrett, IoT Developer Survey 2017

Java Standard Edition (SE) specification, as defined by the Java Community Process (JCP). Azul runs the full set of tests provided by the Technology Compatibility Kit (TCK) to guarantee that Zulu fully conforms to the specification. Azul has also performed a meticulous analysis of the license terms of every source file to ensure our customers are not liable for the use of unlicensed intellectual property.

Zulu Advantages for IoT Development Teams

Zulu provides a number of features that address the challenges of embedded, IoT, software development:

"Write once, run anywhere": By compiling to bytecodes rather than native machine instructions, it is the JVM that is responsible for running application code. Once the JVM has been ported to a particular hardware platform, any Java code can be run on it. No more endless porting of applications each time there is a new revision of the hardware.

Automatic memory management: When programming in C and C++ the developer is responsible for all memory management, allocating memory through APIs like malloc and deallocating it through free. If memory being used is not explicitly deallocated the application exhibits a memory leak. Assuming memory continually needs to be allocated the application will eventually fail when free memory is exhausted. In Java all objects are allocated by the JVM in a heap, references to objects are tracked and when no longer required the memory is reclaimed by the garbage collector, which runs automatically in the background.

No explicit pointers: Again, using languages like C and C++ have drawbacks because of the use of pointers to reference explicit memory addresses. Buffer overrun security flaws and references to invalid memory addresses causing application crashes are both consequences of this. In Java you only have implicit pointers to objects. These cannot be manipulated, other than to point at a different (valid) object. This eliminates a broad range of potential application flaws before you even start writing code.

Comprehensive Libraries: The current release of Java has over 4,000 classes available by default to developers. This rich set of functionality means developers are saved a lot of time not having to reinvent the wheel by writing their own list class, for example. If more application specific APIs are required there is most likely a third-party library available; most of the time this will be both free and open source.

Flexible Deployment: A comprehensive set of libraries is great, but in an embedded environment this can mean the runtime takes up too much storage space. The current version of the JDK provides a way to subset the standard libraries to reduce the storage they require. Java 9 extends this capability further to a fully modular runtime.

Multi-threading support: From its first version, Java has included a standard way to create new threads of execution. That provided fundamental threading so, in recent versions, Java has added increasingly sophisticated, yet simple ways to organize groups of threads that work together. Higher level constructs like semaphores, mutexes and atomic operations were followed by the fork-join framework for decomposing large tasks and assigning them to a pool of threads. Most recently, the introduction of streams in JDK 8 provides a powerful, functional style of programming that can utilize multiple threads by using a single API to make the thread parallel.

Remote management built in: Java has a set of Java Management Extensions (JMX), which includes MBeans (Managed Beans). This provides an architecture that can remotely manage resources dynamically at runtime. This solves the problem of dealing with devices that are hard to access physically.

Remote Application Updates: OSGi started as the Open Services Gateway Initiative and has evolved to become a popular module system for use in enterprise Java applications. Despite this, it retains its relevance to the embedded and IoT space by providing a clean and simple way to deliver services to remote devices. 'Bundles' can be created and configured for deployment

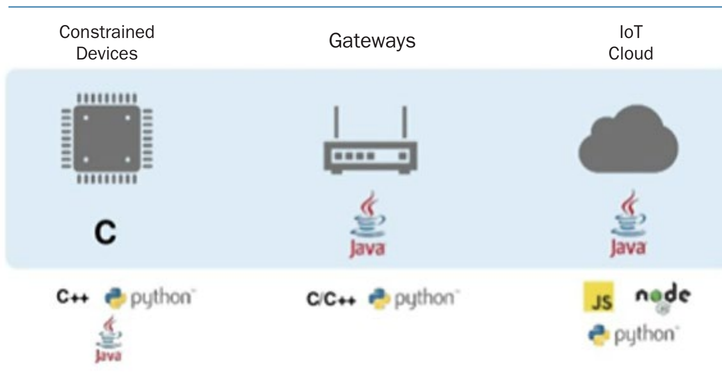
to a remote JVM and can make use of versioning information so updates are only downloaded when required.

A huge pool of trained developers: According to some estimates there are nine million Java developers in the world. Regardless of exact numbers it's fair to say that there are millions of developers trained in Java. Most universities use Java as a teaching language for object-oriented techniques because of its appeal to employers and practicality. What this all means is that when you want to staff a project being developed in Java you have a big pool of talent to draw on.

Access to powerful Development Tools: Having skilled developers is good, but for them to be really productive they need great tools. Again, Java has what's needed. There are several great integrated development environments (IDEs) like Eclipse (Azul is a member of the Eclipse Foundation), NetBeans and IntelliJ, all of which have free and open source versions. In addition, there are many useful tools covering all aspects of Java software development with free and open source versions: VisualVM, JUnit, Jenkins, Gradle and GC Viewer to name but a few.

Simplified Licensing: Azul Zulu is built from OpenJDK source code that has been released under a GPL license with the classpath exception. Primarily, this means that you can ship an application with a Java Runtime Environment (JRE) without being required to make the source code of your application public (as is generally required by the viral nature of the GPL). However, this also means that Azul Zulu is unencumbered by the traditional field-of-use (FOU) restriction that has required users to license embedded Java from Oracle in the past.

Top IoT Programming Languages



Source: Ian Skerrett, IoT Developer Survey 2017

What's Holding Java Back in the IoT?

Despite its maturity and popularity for enterprise application development, Java tends not to be as widely used in IoT and embedded applications as would be expected.

A survey of Embedded developers asked about their reasons for using or not using Java.

The top three reasons developers cited for not using Java were:

1. Run time speed is slow
2. Memory requirements are large
3. Inability to perform low-level operations, such as device IO

We'll examine each of these to see how Azul Zulu overcomes these objections.

Runtime Performance

In the early days of Java, the JVM implementation was simplistic, using an interpreter to convert bytecodes to native instructions and rudimentary algorithms for memory management (both space allocation and garbage collection). Now that Java is into its third decade of development things have improved considerably.

The JVM uses adaptive compilation through a Just-In-Time (JIT) compiler to compile frequently used sections of code into native instructions at run time. These sections of code are cached and reused to avoid repeatedly interpreting the same set of bytecodes. Using a JIT has advantages that can enable Java to not just match the performance of natively compiled code but in certain cases exceed it. To understand how this is possible it is important to be aware that Java supports dynamic class loading as part of its design. For Ahead-Of-Time, statically compiled, code this restricts the level of optimizations that can be applied due to the lack of absolute knowledge about what classes will be loaded at runtime. The JIT compiles code at runtime, and so knows exactly what classes are loaded; allowing it to optimize code more aggressively. Another powerful technique that can only be used at runtime, when profiling information is available, is the use of speculative optimizations.

Memory Footprint

Again, much work has been done over the lifetime of the Java platform to improve its performance and reduce its overall footprint. Since Java provides a managed environment for resources such as memory, there may be occasions where the runtime footprint will be bigger. Given the size of memory in use by the majority of today's devices, this difference will be negligible in real terms.

Azul Zulu provides the complete set of programming APIs available in Java. This is a real bonus for developers since they do not need to implement many common classes such as a linked list, semaphore and so on. Having a rich set of APIs to choose from does, however, have the drawback of increasing the size of the runtime platform. To address this, the current release (JDK 8) allows the use of subsets of the standard APIs called compact profiles, of which there are three, the smallest being less than 11Mb in size. For greater flexibility and a much-reduced footprint, the next release will support a full module system. The JDK 9 runtime libraries have been divided into 26 modules, which can be included or excluded as necessary for the functionality of the application.

Low-Level Operations For Device Access

As previously mentioned, one of the dominant features of the Java platform is the rich variety of libraries and frameworks available, either commercially or as open source. As such, embedded systems libraries that simplify device interaction are already available. For the Raspberry Pi boards, there is an open source library, Pi4J that provides a simple API for accessing GPIO, I2C, SPI and UART pins on the board. For a more general-purpose solution, there is the Device IO APIs. Originally designed for the Java Micro Edition (ME) version, these have now been ported to Java SE and provide similar functionality to Pi4J on any development board that supports it.

For low-level access that requires support beyond these libraries, there is the Java Native Interface (JNI). Java API wrappers can be created for native libraries written in C and C++ with the simple movement of data between the two. Once the wrapper has been created, use of the native code becomes just like any other Java API. Looking to the future, there are plans for a project called Panama to make this process even simpler.

Conclusion – Why Open Source Java is the Best Choice for the IoT

As you can see Azul Zulu provides an ideal solution for the development of IoT and embedded device applications.

Here's a summary of Azul Zulu's advantages for the IoT:

- Built from OpenJDK source, tested to be fully conformant to the Java SE specification.
- As a JDK, millions of developers know how to use it and can take advantage of the wide availability of commercial and open source libraries.
- Scalable through the use of profiles and the upcoming module system to fit demanding resource constraints of edge devices.
- Faster time to market: having more developers to recruit from, better tooling and libraries with a simpler overall development process
- Lower on-going costs: Java's more reliable code structure combined with simpler remote management and deployment all reduce costs once development is complete.
- Released under the GPL license with classpath exception eliminating concerns of complex licensing and reporting issues.

Find out more

To learn more about Azul and Zulu Embedded, contact Azul Systems today.

Azul Systems, Inc.
385 Moffett Park Drive, Suite 115
Sunnyvale, CA 94089 USA
+1.650.230.6500
www.azul.com
www.azul.com/products/zulu-embedded

Download Zulu free:

www.azul.com/downloads/zulu-embedded
www.zulu.org/download

Send a note to info@azul.com if you'd like to discuss custom builds of Zulu for your next IoT project.

© 2017 Azul Systems, Inc.
385 Moffett Park Drive, Suite 115. Sunnyvale, CA 94090 USA.