A Software Library Model for the Internet of Things

Ian C. McCormack
Department of Computer Science
University of Wisconsin-Eau Claire
Eau Claire, WI, USA
mccormic2812@uwec.edu

Abstract

Heterogeneity, resource constraints, and scalability are obstacles to making the IoT approachable for non-specialist programmers. To be successful and appealing in these environments, library systems must be as space-efficient and flexible as possible without fundamentally changing the process of creating and maintaining shared software. Existing library frameworks emphasize some but not all of these attributes and rely on a monolithic model that preserves mutable state. We propose a finer-grained approach to software libraries that allows developers to use multiple components of a library concurrently at disjoint versions. This model defines a library as a set of independent functions with immutable global state to avoid maintaining distributed mutable state in dataflow environments. Library code is stored in a data dependency graph, which is traversed to produce a minimal copy of the library containing only what is necessary for a program. This design addresses the constraints of distributed systems and allows developers to quickly customize dependencies for their unique deployment situations.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Domain specific languages**.

Keywords: IoT, Software Library Systems, Data Dependence Analysis, Macroprogramming Systems

ACM Reference Format:
Ian C. McCormack. 2020. A Software Library Model for the Internet of Things. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '20), November 15–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3426430.3428136

SPLASH Companion '20, November 15–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8179-6/20/11…$15.00
https://doi.org/10.1145/3426430.3428136

1 Introduction

The IoT has a staggering scale, spanning from cloud servers to embedded "motes" with tens of kilobytes of storage [8]. IoT developers must ensure that a library is functional *and* that it meets a particular network's storage, memory, and energy limits [8]. This diversity of constraints implies that library maintainers will have less of an incentive to provide support for any singular deployment scenario. Developers may find themselves in a position where an updated library is functional but violates the resource constraints of a deployment. To adapt quickly to these breaking changes, developers need an efficient method for mixing and matching disjointly versioned components of a library within a program.

Developers can unilaterally design IoT applications with *macroprogramming systems* that optimally spread a single program across a network of devices. Existing designs [10, 11, 13, 18] are somewhat modular in syntax, but they do not specify a library system. Existing library models such as Python's PiP[3, 5] and JavaScript's NPM[1, 2] are not adequate for this environment because they allow mutable state to be declared and hidden within a library, which may incur unexpected synchronization costs in the dataflow environments commonly used to enable parallelism for large-scale systems [6, 12]. These models also inefficiently support using disjoint versions of a library within a single program. Non-specialist developers would benefit from a library system that is similar to these solutions but optimized for the orthogonal concerns of the IoT.

We propose a library syntax, storage format, and distribution mechanism for IoT macroprogramming that empowers developers to balance functionality with compatibility without challenging the fundamentals of sharing software.

2 System Design

Libraries are defined using the syntax shown in figure 1. Each definition file begins with a header statement specifying the library name, which is proceeded by a series of declarations *ds*. Library functions are marked by the `export` keyword while helper functions are declared with the `func` keyword. Integers are represented by i, types by τ, identifiers by x, and arbitrary expressions or statements by e. Exports from a library are used via a `require` statement that takes the library name l, a version identifier n, and a bracketed list of function names f. Dependency information is embedded in source code instead of in an external file to avoid aliasing.

Version ranges can be applied similar to NPM's semantic versioning [4].

When published, libraries are translated into an abstract syntax tree that is traversed to produce def-use chains for each top-level declaration. The immediate dependencies for each exported function are identified by finding the intersection of the set of `const` declarations and the set of undeclared variable names within each function body. A node is created for each function and attached to the declaration nodes of its dependencies. This forms a set of data dependence graphs [14, 17] where each node contains an identifier, the snippet of code where it is declared, and a set of antidependent nodes. Figure 2 displays a *library graph* in which nodes *a*, *b*, *c*, and *d* are declared constants used in functions *funcA* and *funcB*

Library graphs are stored as JSON in a centralized repository. When a new graph is uploaded, its nodes are compared with all other nodes for previous versions of the library to identify duplicates, which will not be inserted. A node is a duplicate if there exists another node that has the same identifier and snippet of source code. When a new node is inserted, its list of parent nodes is reformatted into a *dependency set*, which maps library version ranges to lists of parent nodes. This eliminates the need for duplicating nodes for different versions of the library. Instead of copying a node, edges marked with new versions are added to its dependency set. If a version has a severe error or vulnerability, it can be removed by deleting all of its edges and unique nodes.

$$
\begin{array}{rcl}
L & ::= & \textbf{library}\ l@v\ ds \\
ds & ::= & \bullet \mid d\ ds \\
d & ::= & \textbf{require}\ f^{+}\ \textbf{from}\ l[@v]\ \textbf{as}\ x \\
 & \mid & \textbf{export}\ f\{e\} \\
 & \mid & \textbf{func}\ f\{e\} \\
 & \mid & \textbf{const}\ \tau\ x\ =\ e \\
v & ::= & n \mid \hat{}n \mid n - n \mid (< \mid > \mid \leq \mid \geq)\ n \\
n & ::= & m.m.m \mid \textbf{latest} \\
m & ::= & (i \mid \text{“\_”})
\end{array}
$$

Figure 1. The syntax for a library definition.

When a program `requires` a function, the repository is queried with the library name, version, and requested function names. If the corresponding function nodes are located, their dependencies in the graph are traversed. When a node is reached, its code snippet is recorded and the requested version is used to determine which set of parents to visit. If a node represents a `require` statement, traversal will continue recursively within that library's graphs. Cycles are avoided by assigning unique IDs to nodes and maintaining an efficient mapping of the nodes that have already been visited. This also applies when there are multiple paths upward through the graph, shown with nodes `c`, `d`, and `funcB` in figure 2. Once traversal completes at the root of each graph, the resulting set of versioned snippets is sent to the compiler.

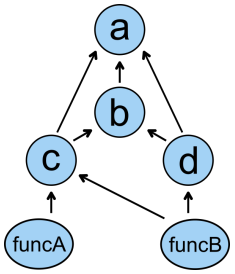

Figure 2. A sample *library graph*

3 Related Work

Northrop et al. [9] outlined several challenges for "ultra large" scale systems, including constant evolution, inconsistent behavior, and heterogeneity. Motta et al. [15] echo these concerns in their survey of IoT stakeholders and emphasize the importance of creating platforms that are easy to understand, extend, and maintain. Newton and Walsh [16] tackle the problem of heterogeneity at scale with a *macroprogramming* paradigm, in which a single program dictates the functionality of an entire network of devices. Existing macroprogramming systems target embedded systems [11, 13] and network switch configurations [10, 18] without including a library system. Foster et al.'s Frenetic language [10] implements a modular design with composable queries and has a potential community of users, but it does not outline a library system for this community. Similarly, Kothari et al.'s Pleiades system [13] defines a program as a set of task-oriented modules containing dependent functions and state, but it does not support imports or exports.

Package management tools such as Python's PiP [3] and JavaScript's Node Package Manager (NPM) [1] treat libraries as inseparable with limited support for concurrent, disjoint versions of a dependency. Methods for working with multiple versions of a package, such as Python's *virtual environments* [5] and NPM's aliasing, require installing each separate version in full. Common Lisp's ASDF uses a "one package per file" method of organization [7] that inspired this model's one version per function capacity.

4 Conclusion

We examined a library syntax, storage model, and distribution mechanism that is optimized for macroprogramming systems and described how this system minimizes code, enforces immutable state, and provides a method for developers to balance compatibility with functionality.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1646235 and 1645578. The author thanks Dr. Robert Iannuci, Dr. Jonathan Aldrich, and Kyle Liang for their extensive mentorship and support.

References

[1] 2020. npm. https://www.npmjs.com/. [Online; accessed 20-July-2020].

[2] 2020. npm-install. https://docs.npmjs.com/cli/install. [Online; accessed 20-July-2020].

[3] 2020. pip - The Python Package Installer. https://pip.pypa.io/en/stable/. [Online; accessed 20-July-2020].

[4] 2020. Semantic Versioning 2.0.0. https://semver.org/. [Online; accessed 12-August-2020].

[5] 2020. Virtual Environments and Packages. https://docs.python.org/3/tutorial/venv.html. [Online; accessed 20-July-2020].

[6] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. 2005. The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. In *Proceedings of the 2005 Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services* (Seattle, Washington) *(EESR '05)*. USENIX Association, USA, 19–24.

[7] Daniel Barlow. [n.d.]. ASDF: Another System Definition Facility - Manual for 3.3.4.

[8] C. Bormann, M. Ersue, and A. Keranen. 2014. Terminology for Constrained-Node Networks. https://tools.ietf.org/html/rfc7228#section-2.1

[9] Peter Feiler, Kevin Sullivan, Kurt Wallnau, Richard Gabriel, John Goodenough, Richard Linger, Thomas Longstaff, Rick Kazman, Mark Klein, Linda Northrop, and Douglas Schmidt. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* Software Engineering Institute, Carnegie Mellon University.

[10] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. *SIGPLAN Not.* 46, 9 (Sept. 2011), 279–291. https://doi.org/10.1145/2034574.2034812

[11] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2003. Giotto: a time-triggered language for embedded programming. *Proc. IEEE* 91, 1 (2003), 84–99. https://doi.org/10.1109/JPROC.2002.805825

[12] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[13] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. 2007. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/1250734.1250757

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia) *(POPL '81)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555

[15] Rebeca C. Motta, Káthia M. de Oliveira, and Guilherme H. Travassos. 2018. On Challenges in Engineering IoT Software Systems. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (Sao Carlos, Brazil) *(SBES '18)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/3266237.3266263

[16] Ryan Newton and Matt Welsh. 2004. Region streams: Functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*. 78–87. https://doi.org/10.1145/1052199.1052213

[17] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. 1991. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/99583.99595

[18] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. 2012. Procera: A Language for High-Level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (Helsinki, Finland) *(HotSDN '12)*. Association for Computing Machinery, New York, NY, USA, 43–48. https://doi.org/10.1145/2342441.2342451