How Languages Can Save Distributed Computing

Andrew C. Myers

Department of Computer Science
Cornell University
Ithaca, New York, United States
andru@cs.cornell.edu

1. Current abstractions are failing us

Current networked computing platforms make available to users an immense and growing amount of computing power and storage. This should be an exciting time for computing, and it is. To users, it might seem an infinitely powerful Internet Computer exists with a manifestation on each of their several computing devices.

However, distributed applications are still relatively primitive and unreliable; they are too centralized on clusters provisioned by individual organizations; users find it hard to share code or even data without opening themselves up to attacks.

The grit in the gears of progress is the difficulty of building applications for the Internet Computer. Our programming models are simply too difficult and low-level for programmers to use effectively. We need higher-level programming models and programming languages that get us closer to programming the Internet Computer directly.

The software stack Industry-standard programming models involve developing the application logic in a programming language such as Java. (The situation is not very different in most other programming languages, except that library and tool support are typically weaker.) This application logic sits atop a large stack of abstractions, whose abbreviations seem to proliferate: SQL, HTTP, EJB, RMI, XML, AJAX, SOAP, JSON, etc. Because these abstractions don't hide the software layers below, applications must be written in a way that is aware of too much of the stack. This structure makes the application complex and unreliable, and more likely to have security vulnerabilities. Further, as it filters through these layers, information within the program is transformed through multiple representations as it migrates from persistent storage to remote clients and back, creating fragility and inefficiency.

Of course, there are reasons why the software stack has evolved to this complex state. Distributed programs must handle distributed code and data securely and efficiently. They usually span multiple domains of trust: a user client and a server, at least, though applications that bring together information and code from multiple domains are becoming more common.

If past history is any guide, industry will keep incrementally adding new layers to the software stack. On the current trajectory, our computers will become increasingly difficult and unpleasant to program, and less secure. Something better is needed, and languages can help.

2. A higher-level abstraction: Fabric

In Fabric [1, 6], we have been exploring a very different approach to building distributed applications, centered around a high-level programming language in which code and data can be used in a uniform and transparent way, regardless of where that code and data is located on the network. Objects at the language level may be persisted at remote nodes in the network. Analogously to the current use of JavaScript, the code for those objects may be located at remote, untrusted nodes and dynamically loaded by the using node. Dynamic loading is important because we want to (securely) support the kinds of activities already common in the web ecosystem, in which programmers share and swap code and data freely.

There are two challenges in realizing this vision: first, to develop a programming model that in an abstract way exposes to the programmer the underlying systems issues: persistence, distribution, failure, and security. Second, to realize this model in an implementation that, even in a completely decentralized system, respects the properties that the language-level abstraction purports to offer. These properties can easily be in tension with each other. For example, the distributed protocols needed to ensure consistency can easily enable distrusted participants to infer confidential information.

The language abstraction The Fabric programming language closely resembles Java, but differs in some important respects.

- Similarly to the Jif programming language [7], it adds information security annotations that specify the confidentiality and integrity of information manipulated by the program. The security of the system is enforced both statically and dynamically by using these policies to control information flow. Fabric goes beyond Jif by adding annotations that control new attacks on confidentiality and integrity that arise in a distributed setting: covert channels created by requests for remote objects, and attacks on integrity that might be launched by downloaded mobile code.
- All information, including distributed and persistent data and code, is presented to the programmer as language-level objects. The system automatically takes care of caching, migrating, and persisting these objects to support efficient, secure, consistent execution. There is no need to have a relational database as part of the software stack.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.4.6 [*Operating Systems*]: Security and Protection—information flow controls

Keywords security, information flow, abstractions

POPL'13, January 23–25, 2013, Rome, Italy.
ACM 978-1-4503-1832-7/13/01.

- Distribution is exposed explicitly to the programmer through annotations on constructors and method calls that indicate where allocation or computation, respectively, should occur. For example, a method call `o.m@n()` transfers control to network node `n` to run the code.
- The consistency and durability of computations of these distributed, persistent objects is supported by a language-level transaction mechanism. The statement `atomic {`S`}` executes statement S as though in isolation from everything else happening on the network, even when S contains distributed computation.

Implementing the abstraction Fabric aims to provide this high-level programming abstraction, yet with an implementation that runs code securely and efficiently on a completely decentralized collection of nodes. The threat model is similar to that of the web, so malicious nodes are assumed to be part of Fabric and to be able to join and then provide code and data to other nodes. The Fabric compiler and run-time system together ensure that information does flow to any node that is not trusted to enforce its confidentiality policy, and that information does not flow from any node that is not trusted to enforce its integrity policy. The inherent compositionality of information flow control then ensures that all information flows across the distributed system respect the information security policies, even though these policies are interpreted and enforced in a decentralized way.

Cooperation between nodes requires explicit trust, so information security policies are expressed in terms of principals. Nodes are principals in Fabric, and like all other principals, are represented by objects that can describe their trust in other principals. Complex trust relationships can thus be expressed, enabling enforcement of complex information security controls.

Nodes serve three roles in Fabric: as *workers*, which do computation over cached copies of objects persistently stored elsewhere; as *stores*, which record the authoritative versions of persistent objects and ensure durability; and as *dissemination nodes*, which make object data highly available. These different kinds of nodes support a variety of different distributed architectures, including web-like client–server structures, but also more general structures, such as clients that directly interact or clients that use information and code from multiple servers.

Fabric works differently from middleware systems such as RMI or CORBA because worker nodes cache copies of remote objects rather than representing remote objects as proxies. This implementation choice is important for security and for performance. Execution stays on the current node unless there is an explicit transfer of control through a remote method call. Computing on object copies means that remote nodes do not learn when objects they store are used. A hierarchical two-phase commit protocol ensures consistency while protecting the confidentiality and integrity of information from untrusted participants.

The prototype implementation of Fabric is publicly available [5].

3. Conclusions

The Fabric project is developing a new language and system to support the kinds of activities now happening on the web: the free exchange of code and data across a decentralized, distributed system. Fabric eliminates much of the complexity of the current web-based distributed programming model, by raising the level of abstraction at which applications are developed. This makes the resulting system simpler overall, by eliminating software layers and data conversions that are not essential to application functionality. By exposing security considerations in information flow policies, it supports secure composition of code and data from different trust domains.

Fabric is one approach to raising the level of abstraction, and clearly could be extended usefully in various ways. There have been several other recent language-based attempts to raise the level of abstraction for programming distributed systems, particularly for web applications [2–4, 8]. Some of these show that the Fabric abstraction is not the only candidate for an effective high-level programming abstraction. For example, abstractions centered around streams are likely to be useful for some applications.

If we are to obtain a clean, high-level language-level abstraction for programming future distributed computing systems, it is to be hoped that the programming language community will be involved. However, this will require that language researchers confront some traditionally "systems" issues such as persistence, distribution, and security, and to develop effective abstractions for handling these issues. To program the Internet Computer, programming-language research will need to break out of its abstraction layer.

References

[1] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE Symposium on Security and Privacy*, pages 191–205, May 2012.

[2] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, October 2010.

[3] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, October 2007.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, November 2006.

[5] Jed Liu, Owen Arden, Michael D. George, K. Vikram, and Andrew C. Myers. Fabric 0.2. Software release, `http://www.cs.cornell.edu/projects/fabric`, October 2012.

[6] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.

[7] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[8] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proc. 1st Dynamic Languages Symposium*, pages 975–985, October 2006.