社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Shared Nothing Secure Programming in Erlang/OTP

Kenji RIKITAKE

E-mail: kenji.rikitake@acm.org

Abstract Shared nothing (SN) architecture is a standard design method for distributed systems. In this paper, the author presents how the principles of SN architecture contributes to enhance security of software programming in general, and describe the examples in Erlang/OTP concurrent programming system. The author also shows how SN programming will affect the principles of software security.

Keywords Shared nothing, Erlang, Erlang/OTP, message passing, secure programming

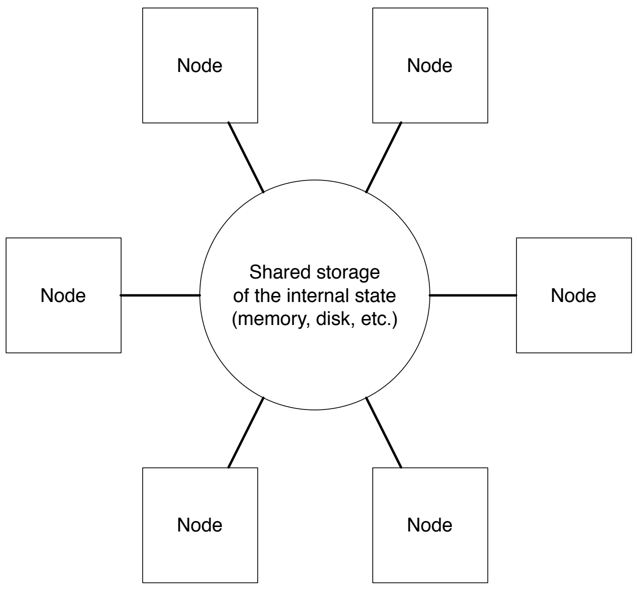

Fig. 1 Shared everything (SE) model.

1. Introduction: shared everything .vs. shared nothing principles

Computers have become an outlet for all media and communication activities. People after the Web social networking service (SNS) era of Facebook [1] and Twitter [2] consider the main purpose of information media is *sharing*. The pace of sharing is even getting faster; more people are involving in chat activities with smartphones through the messaging services such as WhatsApp [3] and LINE [4].

Sharing and global reachability to the storage resources by the computing nodes (hosts, CPUs, etc.) is part of the fundamental design in computer architectures. Figure 1 shows a configuration of a multi-node system, sharing the same storage, under the *shared everything* (SE) principle. The SE principle is firmly ingrained in the modern computing paradigms; programmers assume all resources are readily available from each and every computing nodes.

On the other hand, building an information system based on the assumption which assumes all the involving parties can see everything they have each other as the default state may cause grave problems from the information security perspective. For example, in many SNSes and sharing services, data are only hidden by obscuring the corresponding URLs and has no access control by authenticating the entity who tries to get access to them. This obscurity-based security is weak and can be easily exploited. Dropbox, a leading file-sharing service provider, had to deactivate links before May 5, 2014, due to the disclosure of private links through the referer header [5].

Another problem is that the entities involving in an data exchange in the SE principle are assumed not to accidentally or unintentionally change the shared state; this assumption can be easily broken in the real-world development and operation of the distributed systems. For example, the mutual exclusion problem such as resource locking and database transaction protocols have been one of the most difficult problems to solve, and bugs changing the critical shared state can render the whole protocols useless at once. Deutsch [6] explains the false assumptions of the SE principle partly as: *the network is homogeneous*, *there is one administrator*, and *latency is zero*.

Shared nothing (SN) is a totally opposite characteristic against the SE; in an SN system all involving parties are isolated and cannot see anything which each of them has as the default state. Figure 2 shows an example of SN system, where each node has an independent and isolated storage, and limits the information exchange over the peer-to-peer mesh network.

Stonebraker [7] proposed the notion of *shared nothing architecture system* in 1986 with the multi-processing database engine context, as the architecture which neither memory nor peripheral storage is shared among processors. The SN principle can also be considered as a form of isolation to minimize and to prevent the unnecessary coupling at every layer of the communication levels in a broader information system: nodes/hosts, operating system (OS) processes, modules inside the OS processes or an executable pro-

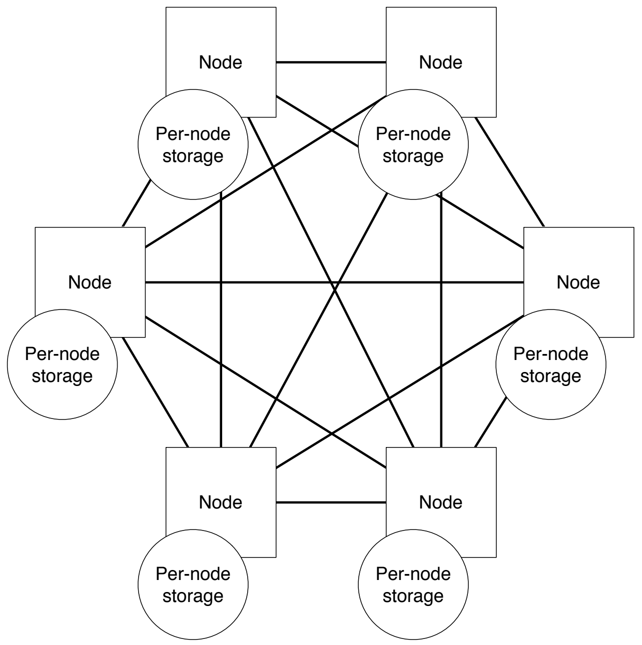

Fig. 2 Shared nothing (SN) model.

gram, and even functions of each module as a part of program.

SN systems were not traditionally chosen for the computer system design, since they are slower than SE systems in a monolithic execution environment, where all the related execution entities are contained in the same physical CPU core and in the same storage device cluster. Exchanging internal state by explicit messaging between the system components requires concurrent programming techniques. On the other hand, SE systems has no way to recover from a partial failure in the system component, while SN systems can be designed to maintain at least reduced or even full of either availability or consistency even when the network partition occurs, according to the CAP theorem (*1) explained by Gilbert and Lynch [8], and Brewer [9], the author of the original conjecture in 2000. Amazon formalizes the methodology of meeting the business requirements with a large-scale distributed key-value database system called Dymano [10], and Basho Technologies (*2) makes Dynamo-based distributed key-value database called Riak [11]. Shapiro et al. [12] proposes *Convergent or Commutative Replicated Data Types* (CRDTs) to formally guarantee *eventual consistency* [13], which is actualized in Riak 2.0 [14].

Erlang/OTP [15] is an SN-based programming system, enforcing the SN principle from the lowest language level. Erlang/OTP is chosen as the core messaging architecture of the large scale sharing and messaging services such as WhatsApp [16] and Basho's Riak [17], because of the robustness and capability of handling massive concurrency without causing the internal state corruption. Erlang has a set of language-level SN properties, and the OTP modules are thoroughly designed to exclude the degradation of reliability caused by sharing for coupling the modules and functions. These fundamental properties of Erlang/OTP largely contributes as the key components of building a robust distributed system.

As all computer systems are moving towards distributed execution environments even within a same physical container, the software developers and system administrators, or the combined role players called *devops* [18], are now daily tackling the problems and issues dealing with the distributed systems running on virtual machines inside the cloud computing infrastructure, for massively concurrent data processing. In such systems, running the programs with reliability is a hard problem to solve, let alone showing the proof of how the systems are secure; the notion of computer security will not firmly stand without the perspective from the reliability issues.

In this paper, the author first presents how the SE principle traditionally affects the detail of programming and system design in Section 2. In Section 3, the author explains how SN principle and architectures will affect the principles of software security, using Erlang/OTP as an example. In Section 4, the author summarizes how and what SN principle has changed the computer programming and system design, and proposes a set of open questions to the computer security community for the further discussions.

2. Shared everything principle and the historical implication on programming

Computer programming is traditionally conducted on imperative languages; early programming languages such as FORTRAN and BASIC are abstraction of the assembly languages. Still most of modern programming languages including JavaScript (JS, also known as ECMAScript [19]), and Lua [20] are based on the imperative, procedural, and structured programming, though many of them also incorporate different programming paradigms.

The imperative programming model consists of statements or instructions which *directly change the internal state*, usually *commonly shared and accessible between all involving functions and modules*, and inherently based on the SE principle. The data storage of imperative languages are designed for reusing and sharing, for gaining execution speed.

For example, variables in the imperative languages are considered as reusable pieces of boxes with a size value in bits or bytes. This characteristic is the most simply represented by the following notation: `a = a + 1`, which increments the variable `a` by one. This notation is confusing due to the dual meaning of *assignment* and *comparison* on the same operator = (equal sign), which is another common syntax practice on many imperative languages.

On the other hand, when a complex data structure is allocated by a constructor, the allocated object in the memory is referred by *the memory address of the structure*. Each structure must be explicitly constructed; assignment of an existing structure into a variable *does not make a copy* of the structure. Also, when a data structure

(*1) : The word *CAP* stands for Consistency, Availability, and Partition tolerance.

(*2) : Disclaimer: the author Kenji Rikitake was a Senior Software Engineer at Basho Japan KK, a subsidiary of Basho Technologies, Inc., from February to September 2013.

```
> a = {1, 2, 3}
> b = a
> print(a[1], a[2], a[3])
1       2       3
> a[3] = 4
```
b[3] is changed because a[3] is re-assigned.
```
> print(b[1], b[2], b[3])
1       2       4
> print(b == a)
true
```
Printing a table returns the address value.
```
> print(a)
table: 0x7f8751c12cf0
> print(b)
table: 0x7f8751c12cf0
```
A constructor makes a new table; note the return value of the second constructor call is different from the first constructor call.
```
> print({1,2,4})
table: 0x7f8751d00c20
> print({1,2,4})
table: 0x7f8751f002b0
```
Comparison of the tables only compares the *address of the table*, and does not compare the each and every element.
```
> print(b == {1,2,4})
false
```
Fig. 3 Variable and table assignment example on Lua 5.2.3 REPL.

```
> var a = {first: 1, second: 2, third: 3}
undefined
> b = a
{ first: 1, second: 2, third: 3 }
> a.third = 4
4
```
b.third is changed because a.third is re-assigned.
```
> b
{ first: 1, second: 2, third: 4 }
> b == a
true
```
The right-side object is *newly* constructed, so is *different from* the left-side value.
```
> b == { first: 1, second: 2, third: 4 }
> false
```
Fig. 4 Table assignment example on node.js v0.10.28 REPL.

containing multiple elements such as JS *object* and Lua *table* is assigned to a variable, the variable contains the reference of the data structure.

Figures 3 and 4 show how an element of a data structure is treated in REPL [*3] of Lua and node.js (a JS application platform [21]). A few points should be noted:

- variables assigned to data structures only store the reference of the structures, not the value or the complete set of them;
- changing an element of the structure assigned to the source variable will cause the corresponding change of the element in the destination variable; and
- comparing two data structures means comparing the memory address only, and will not compare the the each and every elements; this may result in an unintuitive result of *false* even if the contents of the data structure represented by a valuable is compared to another data structure which has the exactly same element values and hierarchy.

These characteristics of the modern imperative languages require cautious operation practices when the programmers need to perform the following operation:

- extracting the data structure contents between two or more computing nodes which do not share the memory addresses with each other;
- isolating a copy of a data structure from the source, i.e., making the contents of the copied structure will not be affected even the elements in the source are later modified; and
- ensuring two *different* data structures have the exact same structures and the element values; this requires external *deep comparison* functions [22], [23].

In summary: under the SE principle, the entire programming language is designed to minimize copying and to implicitly enforce the programmers to share the data structures which does not fit into a variable. This design makes the isolation between the data structures extremely difficult.

3. Shared nothing principle in Erlang/OTP and how it affects the secure programming

3.1. Erlang/OTP and the shared nothing properties

The Erlang/OTP programming system is based on functional, declarative, and message passing paradigms, which are completely on the opposite perspective from the modern imperative languages, based on the SN principle.

Figure 5 shows how an element of a data structure is treated in the Erlang Shell (a REPL). The programming language Erlang incorporates the following characteristics, which *enforces the programmer to avoid unnecessary sharing at all costs*:

- *No variables can be reused* in the function scope; each variable can be assigned *only once*. A statement such as `"A = A + 1."` [*4] causes a syntax error.
- A variable can contain *a whole data structure and the elements*, and an assignment *makes a new copy*; even if an element of the source structure changes, that will not affect the

(*3): The word *REPL* stands for read-eval-print loop, an interactive execution environment equivalent to UNIX shell.

(*4): In Erlang Shell, all statements must be terminated by "." (period).

```
Eshell V6.0  (abort with ^G)
1> A1 = {1,2,3}.
{1,2,3}
2> B1 = A1.
{1,2,3}
```
The `setelement/3` function *makes a modified copy*.
```
3> A2 = setelement(3,A1,4).
{1,2,4}
```
Comparison of the tuples compares the each and every element.
```
4> B1 =:= {1,2,3}.
true
5> B1 =:= A1.
true
```
Fig. 5 Tuple assignment example on Erlang/OTP 17.0 Shell.

assigned structure at all. This characteristic affects how a data structure is manipulated; a modified copy of the *entire structure* is newly constructed, even only one element is modified in the structure. (See the function `setelement/3` in Figure 5.)

- Comparing two data structures mean comparing the each and every elements and the hierarchy.

Erlang's language semantics restriction demands the programmer to *explicitly* introduce data sharing among different functions and Erlang processes [*5]. The programmer has to explicitly handle the cost of assignment and copying.

In Erlang, the variables are strictly local within the function. Preserving the internal state of an Erlang function before and after the execution has to be conducted explicitly by passing it in and out of the function as the return values and the arguments.

Erlang/OTP does have the way of distributed sharing; to share data among multiple functions and processes, Erlang has the process-level shared storage called *process dictionary* which enables sharing data between two or more functions in the same process. Erlang/OTP also has the Erlang Term Storage (ETS) and the disk-based term storage (DETS), as well as the distributed database called Mnesia. The important point about data sharing policy in Erlang/OTP, nevertheless, is that the sharing-based libraries and system services are *not implicitly* used and completely excluded from the default execution of the language statements.

Table 1 summarizes the difference between the SE principled languages and Erlang/OTP. The table shows that the two groups stand on the completely opposite positions with each other.

3.2. How shared nothing principle contributes to make computer programs more secure

Sharing states is inevitable for the computer programming, but the unnecessary sharing may cause errors and malfunctions due to the unexpected I/O coupling between concurrently running modules and functions. Erlang/OTP's design philosophy to minimize the sharing will contribute to make computer programs more secure, including but not limited to, the following aspects:

- *Memory allocation* : while modern programming languages usually employ the garbage collector (GC) and do not require programmers to explicitly allocate and free the memory, Erlang/OTP's single assignment restriction on variables eases the the memory management and a simply and fast algorithm such as one pass real-time generational mark-sweep GC is applicable [24]. Being able to mark unused memory blocks in soft real-time manners keep the active memory footprint small.
- *Process isolation* : On Erlang, the granularity of GC is comparably finer than other languages, since cross referencing between multiple processes will never implicitly occur, and the complexity of each process is much smaller than in other language systems. The path of sharing between multiple processes should be explicitly specified, or be carried through the message passing between the processes. This will contribute to reduce the unnecessary coupling between the concurrently running processes.
- *Referential transparency* : Erlang/OTP employs the functional programming paradigm, to enforce referential transparency which guarantees the same output will be obtained for the function calls with the same arguments. This enables independent module and function unit testing before coupling them into a complex system, and other formal verification methods.
- *Idempotency* : the immutable characteristic of Erlang functions based on the SN principle contributes to increase the idempotency of each function, which means the entire system will be more fault tolerant. Erlang/OTP's supervisor also helps automatic restart and process pool management [25] with the *fail fast principle* for the fast containment of failure [26].

Erlang/OTP has been successfully working on various pieces of software, from large-scale chat servers such as ejabberd [27] and

Table 1 Comparison of SE principled languages and Erlang/OTP.

JavaScript, Lua	Erlang/OTP
Variables are reusable and reassignable	Variables are assigned *only once* in the function scope
Global-scoped variables exist	Variables are function local
Data structures are assigned by reference	Data structures are assigned by the value
Assignment of a variable for data structures only copy the memory address and *shares the same memory contents*	Assignment of a variable for data structures always copy the contents *as an independent entity*
Comparison of two data structures means comparing the memory address only	Comparison of two data structures means the deep comparison of each and every element and the hierarchy

(*5) : Erlang processes are light weight with small amount of internal storage data, reclaimed entirely after the execution ends.

MongooseIM [28] to an OpenFlow software defined network router called LINC [29]. While reliability and fault tolerance do not necessarily guarantee security, no software is secure without the reliability and fault tolerance. The author thinks the Erlang/OTP's reliability approach will contribute to make a more secure software system.

4. Summary and open questions to the computer security community

In this paper, the author first proposed the general difference and real-world issues of *shared everything* (SE) and *shared nothing* (SN) models in Section 1, and described how the traditional SE programming language systems focused on reusing and sharing the existing data structures, and the unintuitive irregularities by showing a simple example, in Section 2. The author then described how Erlang/OTP worked differently by incorporating the SN principle with the same example, and how the change of characteristics would contribute to writing a secure software in Section 3.

Traditionally, writing computer software means a battle for more speed from the same hardware, with the limited memory and storage resources. While this still holds the truth in the world of supercomputing, more and more computer systems become distributed, and the traditional idealistic assumption of SE principle are eventually but surely being substituted by SN principle-based assumptions, which Rotem-Gal-Oz [30] explains: the network is *un*reliable, latency is *significant*, and the network is *far from being secured*. This transitional trend from the SE to the SN paradigm poses a fundamental question: *can pursuing execution speed and maintaining the level of security in the same system coexist?* The author of this paper believes that speed and security do *not* coexist and the two factors are in the trade-off relationship.

For many production services, the traditional SE principle is no longer applicable; these services are running on virtual machines (VMs) or cloud computing systems, where the actual physical server systems are constantly failing and replaced, to mitigate the deployment issues on the failure recovery to reduce the recovery time; and the immutability demand has been raised at the computing infrastructure level.

Fowler [31] proposed an enhancement of the notion of Erlang/OTP's immutable variables into the deployment of the production system itself, called *immutable infrastructure*, which uses disposable components such as pre-built VMs with precompiled servers. Deployment tools for the immutable infrastructure such as docker [32] and a lightweight Linux distribution called CoreOS [33], with the deployment tools such as Chef [34] and Puppet [35] supporting automated system administration for devops. The immutability and automation process are also applicable to mitigate the security threats; if a server is compromised, discarding it and restarting a new uncompromised instance will suffice. In this sense, security incidents can be treated the same as other reliability incidents, such as network or hardware outages, though the mutable user data integrity check should be regularly performed.

The author would like to conclude this paper with a list of random open questions to the computer security community:

- Are the traditional malware detection and other security mitigation methodologies still effective in the world of massively distributed systems?
- Is there any way for the programming language community to contribute to raise the level of security, to prevent the small but fatal incidents caused by programming bugs, such as *goto-fail* [36] and *heartbleed* [37] cases?
- Should the computer science and engineering community as a whole maintain putting higher priority to the execution speed than security? Is there any effective way to reverse the trend to put priority to security?
- Are we ready to accept the inability of the traditional shared everything paradigm, and to move on to the shared nothing architecture?
- How can research, education, and academic communities contribute to incorporate the security first culture into the computer scientists and engineers?

References

[1] Facebook, Inc., "Facebook." `https://www.facebook.com/`.

[2] Twitter, Inc., "Twitter." `https://twitter.com/`.

[3] WhatsApp Inc., "WhatsApp." `http://www.whatsapp.com/`.

[4] LINE Corporation, "LINE." `http://line.me/en/`.

[5] Dropbox, Inc., "The Dropbox Blog: Web vulnerability affecting shared links." `https://blog.dropbox.com/2014/05/web-vulnerability-affecting-shared-links/`.

[6] P. Deutsch, "The Eight Fallacies of Distributed Computing." `https://blogs.oracle.com/jag/resource/Fallacies.html`.

[7] M. Stonebraker, "The Case for Shared Nothing," Database Engineering, vol.9, pp.4–9, 1986. `http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf`.

[8] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," SIGACT News, vol.33, no.2, pp.51–59, 2002. `http://doi.acm.org/10.1145/564585.564601`.

[9] E. Brewer, "Cap twelve years later: How the "rules" have changed," Computer, vol.45, no.2, pp.23–29, 2012.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, New York, NY, USA, pp.205–220, ACM, 2007.

[11] Basho Technologies, Inc., "Riak." `http://basho.com/riak/`.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Rapport de recherche RR-7506, INRIA, Jan. 2011. `http://hal.inria.fr/inria-00555588`, `http://hal.inria.fr/inria-00555588/PDF/techreport.pdf`.

[13] W. Vogels, "Eventually Consistent," Queue, vol.6, no.6, pp.14–19, 2008. `http://doi.acm.org/10.1145/1466443.1466448`.

[14] J. Jacobson, "Riak 2.0: Data Types." March 23, 2014, `http://blog.joeljacobson.com/riak-2-0-data-types/`.

[15] Ericsson AB, "Erlang Programming Language and The Open Telecom Platform (Erlang/OTP)." `http://www.erlang.org/`.

[16] R. Reed, "Scaling to Millions of Simultaneous Connections," Erlang Factory SF Bay 2012, Burlingame, CA, USA, Erlang So-

lutions, 2012. http://www.erlang-factory.com/conference/SFBay2012/speakers/RickReed.

[17] J. Sheehy, "Erlang at Basho, Five Years Later." Basho Blog, July 2, 2013, http://basho.com/erlang-at-basho-five-years-later/.

[18] M. Loukides, "What is DevOps?." O'Reilly Radar, June 7, 2012, http://radar.oreilly.com/2012/06/what-is-devops.html.

[19] Ecma International, "Standard ECMA-262: ECMAScript Language Specification Edition 5.1." June, 2011, http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[20] R. Ierusalimschy, Programming in Lua, Third Edition. 2013, ISBN 978-85-903798-5-0, http://www.lua.org/pil/.

[21] Joyent, Inc., "node.js." http://nodejs.org/.

[22] "spankmaster79", "Object comparison in JavaScript [duplicate]." http://stackoverflow.com/questions/1068834/object-comparison-in-javascript.

[23] S. Donovan, "Function pl.tablex.deepcompare at Penlight Lua Libraries 1.3.1." https://github.com/stevedonovan/Penlight/blob/1.3.1/lua/pl/tablex.lua#L123.

[24] J. Armstrong and R. Virding, "One Pass Real-Time Generational Mark-Sweep Garbage Collection," In International Workshop on Memory Management, pp.313–322, Springer-Verlag, 1995.

[25] F. Hébert, "Learn You Some Erlang - Building an Application With OTP." http://learnyousomeerlang.com/building-applications-with-otp.

[26] J. Gray, "Tandem TR 85.7: WHY DO COMPUTERS STOP AND WHAT CAN BE DONE ABOUT IT?," 1985. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.9127.

[27] "ejabberd community site." http://www.ejabberd.im/.

[28] Erlang Solutions, Inc., "MongooseIM." https://www.erlang-solutions.com/products/mongooseim-massively-scalable-ejabberd-platform.

[29] flowforwarding.org, "LINC OpenFlow Software Switch." https://github.com/FlowForwarding/LINC-Switch.

[30] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained (The more things change the more they stay the same)." http://www.rgoarchitects.com/Files/fallacies.pdf.

[31] C. Fowler, "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components." June 23, 2013, http://chadfowler.com/blog/2013/06/23/immutable-deployments/.

[32] Docker, Inc., "Docker." https://www.docker.io/.

[33] CoreOS, Inc., "CoreOS." https://coreos.com/.

[34] Chef Software, Inc., "Chef." http://www.getchef.com/.

[35] Puppet Labs, Inc., "Puppet Open Source." http://puppetlabs.com/puppet/puppet-open-source.

[36] "ImperialViolet: Apple's SSL/TLS bug (22 Feb 2014)." https://www.imperialviolet.org/2014/02/22/applebug.html.

[37] "The Heartbleed Bug." http://heartbleed.com/.