Mjölnir: The Magical Web Application Hammer

Jelle van den Hooff and David Lazar
MIT CSAIL

James Mickens
Harvard University

Abstract

Conventional wisdom suggests that rich, large-scale web applications are difficult to build and maintain. An implicit assumption behind this intuition is that a large web application requires massive numbers of servers, and complicated, one-off back-end architectures. We provide empirical evidence to disprove this intuition. We then propose new programming abstractions and a new deployment model that reduce the overhead of building and running web services.

1 Motivation

Developing and maintaining a large-scale web application is a daunting task. Programmers and operational staff must understand a complex, multi-tier software stack that contains multiple abstractions (and multiple configuration knobs) for computation, storage, and caching. For example, developers must tune the resource allocations for VMs [41], and worry about whether automatic scaling mechanisms are sufficiently responsive to bursty workloads [28]. Developers must select the proper representation for low-level data (blobs? tables? SQL as a service? files accessed via SMB or HDFS?). Atop that storage layer, developers must create higher-level functionality like text searching [5, 16, 45] and cross-node messaging [1, 3, 35]. The client-facing front-end often requires a load balancer [19, 21, 48]. To detect failures, developers need a distributed monitoring system like Opserver [38], Nagios [34], or Ganglia [18]. Developers must also connect server-side storage and computation to some kind of client-side JavaScript framework [2, 7]. Given this vast array of building blocks, the construction and maintenance of a web service is an intimidating feat of engineering.

Why are developers forced to reason about such low-level details? We believe that there are two reasons.

- Conventional wisdom assumes that massive scalability and maximal performance are primary goals for most web sites. Achieving optimal scalability and performance requires low-level optimizations; thus, developers must be exposed to low-level architectural details.
- Conventional wisdom also assumes that each web application is highly unique with respect to its computation model. According to this assumption, web sites have so many ways of accessing data, and so many ways of computing over that data, that a generic distributed runtime would be too inexpressive or too inefficient. Thus, each application needs a software stack that is tightly customized to that application's data flow.

We believe that both of these assumptions are wrong. For example, intuition suggests that, if a site has millions of users, it must require an enormous number of back-end servers. Empirically speaking, this claim is false. For example, Stack Exchange (which runs sites like Stack Overflow) has 4 million users and 560 million page views a month. However, Stack Exchange only needs 25 servers to handle this load [46], a load that amounts to just 216 requests per second. Bitly, which provides URL shortening, handles 6 billion clicks a month, and 600 million shortening requests each month. Bitly requires 400 servers, but only 30 are dedicated to incoming client traffic, with the rest allocated to asynchronous data organization and analysis [8].

We also dispute the notion that most web services are so structurally unique that they require highly-customized back-ends. Radically different desktop applications are built atop a common set of POSIX abstractions; similarly, web services use common subsystems like key/value stores, load balancers, and publish/subscribe frameworks. In today's world, developers must implement both service-specific logic and the distributed platform that will implement that logic. But if the distributed platform is sufficiently generic to be offered as middleware, it makes no sense for developers to constantly reinvent it, particularly if most sites do not require the ultra-high performance that demands a customized back-ends.

We propose that, for most web applications, the systems community should not focus on performance. Instead, the community should focus on *ease of development, deployment, and management*. Performance is im-

APSys 2015, July 27–28, 2015, Tokyo, Japan.
ACM 978-1-4503-3554-6/15/07.
http://dx.doi.org/10.1145/2797022.2797025

portant, but it must be understood in the context of a scalability target, and observation tells us that fewer than a thousand servers can support complex web sites with millions of users. These observations raise interesting questions: What abstractions should we present to web developers who only need to scale their sites to hundreds of machines, and to thousands of requests per second? What interfaces are simple yet expressive enough to support modern web services? In the rest of this paper, we explore these questions, with the goal of dramatically lowering the burden of developing, deploying, and maintaining a web service. We sketch the design of a new web architecture, called Mjölnir, that tries to resolve the tensions between programmability, scalability, and maintainability, and we describe how Mjölnir can act as a starting point for additional research.

2 Prior work

A variety of commercial offerings try to ease the pain of developing web applications. For example, Infrastructure as a Service (IaaS) frameworks like OpenStack [36] and CloudStack [4] provide distributed building blocks for more complex applications, supplying components that manage routing tables, provide block storage, and control VMs. However, developers still have to write low-level distributed systems code to connect the modules; furthermore, merely installing and configuring the IaaS service itself can be a daunting challenge [23, 26].

To reduce developer effort, Backend as a Service (BaaS) frameworks like Kony [24] and Parse [39] provide higher-level abstractions like push notifications to end-user devices, and direct support for social network integration. In the extreme, "no backend" frameworks like Hoodie [22] optimize for "front-end driven applications"; all web servers deliver the same content to all clients, with specialized content generated by client-side JavaScript that pulls data from remote storage or uses BaaS services for chatting or notifications. BaaS frameworks and no-backend approaches successfully hide many of the server-side details. However, the level of abstraction is often too high for web applications that require non-trivial, site-specific, server-side computation, like real-time streaming analytics over incoming data.

In summary, prior work strikes an uneasy balance between *ease of development* and *semantic richness* (i.e., the expressiveness of the application logic). In the remainder of this paper, we describe a new model which supports sophisticated applications while keeping development costs low.

3 A New Web Architecture

A major source of developer pain is the distributed nature of a web application. The developer must worry about how servers exchange data, and what happens when

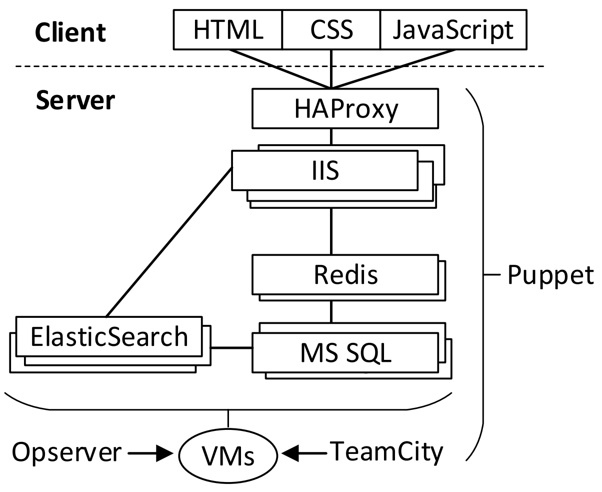


Figure 1: Stack Exchange, a site with millions of users, only requires 25 servers. However, developers have to manage a complex software stack.

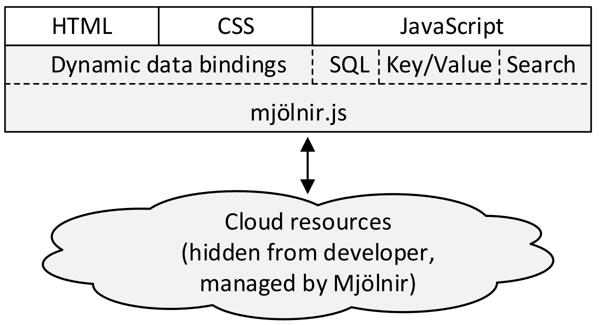


Figure 2: In Mjölnir, developers create a distributed web service simply by writing HTML, CSS, and JavaScript. Mjölnir's runtime library exposes cloud resources using JavaScript objects. Mjölnir translates operations on those objects to server-side interactions; in addition, Mjölnir automatically configures and manages the hidden server-side components.

servers die, and how client requests are synchronized at the servers, and a host of additional low-level concerns. In our new web architecture, we hide this complexity by making the application *obliviously distributed*. From the developer's perspective, a web service should look like a JavaScript program running in a web browser that only accesses local state, even though, behind the scenes, the service consists of multiple clients and servers that are scattered across the network.

In our proposed architecture, called Mjölnir, clients manipulate service state by reading and writing the fields of JavaScript objects; Mjölnir's JavaScript runtime automatically transforms those operations into reads and writes of server-side state. To provide data consistency in the presence of simultaneous client accesses, Mjölnir extends JavaScript's preexisting notion of atomic event handlers. JavaScript already guarantees single-threaded event loop semantics within an `iframe`—at any given time, at most one handler's call chain is executing. Mjölnir applies these semantics to the *distributed* service state as well. Each client-side event handler becomes an atomic transaction over distributed state, with Mjölnir's back-end guaranteeing that state is updated using a linearizable schedule of client-provided transactions. As a result, Mjölnir provides the illusion that the distributed service has a single, global event loop which atomically executes client handlers. JavaScript programmers already use this mental model to deal with client-local concurrency. Thus, developers merely need to continue applying this model, and Mjölnir will allow them to write code that safely and obliviously manipulates distributed state.

A transactional programming model makes it easier to write strongly-consistent distributed applications [10]. However, transactions by themselves are insufficient to hide the server-side complexity shown in Figure 1–developers must still configure, deploy, and monitor a variety of back-end components like databases and web servers. To ease those burdens, Mjölnir leverages the observation that a common set of components are useful to a wide variety of web services. Thus, Mjölnir can deploy and manage a set of those components automatically. For example, Mjölnir might predefine one server-side image that contains Elastic Search, Redis, and Microsoft SQL. Another predefined image might bundle roughly equivalent components (e.g., Lemur, memcached, and PostgreSQL). In both cases, Mjölnir provides the plumbing to bind the client-side `Mjölnir.js` library to the server-side components. Developers merely select the image that would provide the desired performance or durability characteristics for the web service; Mjölnir handles the details of configuring the individual components, deploying them, and dealing with VM management in response to machine failures or changing client load.

The resulting high-level architecture is shown in Figure 2. In this new architecture, developers write *an entire application* using HTML, CSS, and JavaScript. Programmers focus on core application functionality which Mjölnir will translate into concrete distributed operations. Human administrative activity is largely confined to application-specific activities like updating site content and managing user ACLs; Mjölnir automatically handles most of the low-level operational details.

4 Obliviously Distributed Transactions

To demonstrate how Mjölnir obliviously extracts server-side transactions from client-side code, we use a simple bank application (see Figures 3 and 4). This application allows a user to transfer money from their account to another user's account. Like a traditional web service, Mjölnir stores the canonical application state in the cloud. Thus, the transfer operation requires activity on both the client and the server to authenticate the user, performs balance checks on the user's account, and update the user's balance. However, Mjölnir hides these distributed operations from the developer. Furthermore, given the client-side JavaScript code, Mjölnir will *automatically generate* the server-side of the operations, completely freeing the developer from reasoning about distributed abstractions.

Exposing Persistent Storage: To access persistent service state, a web page invokes JavaScript APIs that are defined by `Mjölnir.js`. This library defines two basic storage interfaces, one for key/value storage, and another for row-oriented SQL storage; for example, Figure 4 shows how the `Mjölnir.table` object lets a client fetch a row via the row's primary key. `Mjölnir.js` also defines an interface to perform text search on objects that reside in either store. From the perspective of the developer, all reads and writes appear to be local operations on JavaScript objects; internally, `Mjölnir.js` uses AJAX calls or WebSocket connections [33] to exchange data with remote servers.

Defining Transactions: When a page fires a JavaScript event handler, Mjölnir begins a new transaction. Handlers may fire as a result of user interaction; handlers may also fire as a result of network data arriving, timers expiring, and so on. In the bank example, a transaction starts when the user clicks the "submit" button and triggers the `send()` event handler. That handler transfers money between the two accounts, so Mjölnir must guarantee that `send()` executes atomically with respect to the event handlers of any other client in the distributed system. Otherwise, money may appear or disappear in unexpected ways.

In Mjölnir, a transaction consists of the reads and writes that are generated by an event handler. If the handler invokes additional functions, the reads and writes of those functions are also included in the transaction. Since clients implement transactions using regular JavaScript code, read and write sets can mention server-side data *as well as client-side JavaScript variables*. Thus, if a transaction is aborted, Mjölnir may have to roll back both server-side storage and client-side JavaScript state.[1] Consider the `send()` function in Figure 4. Lines 5 and 10 update the transaction's read set with the table rows belonging to the sender and recipient of the money. Lines 16 and 17 add these rows to the write set. Ephemeral, function-local variables like `fromName` do not appear in read/write sets, but global JavaScript variables can. Note that a page's HTML structure creates global JavaScript variables, since the DOM model [42] associates each HTML tag with a global JavaScript object. This means that when a transaction modifies a page's HTML structure

[1]Each client-side JavaScript environment is single-threaded, so a transaction which only touches client-side state will never need to be rolled back, since a client cannot simultaneously execute multiple (and potentially conflicting) transactions.

```
1  <html>
2   <body>
3    <!-- show the user's current balance -->
4    Balance: {{ table('users').find(user).balance }}
5
6    <!-- let the user send money -->
7    <form>
8     To: <input id="to" type="text" /><br>
9     Amount: <input id="amount" type="text" /><br>
10    <input type="submit" onclick="send();" />
11   </form>
12  </body>
13 </html>
```

Figure 3: HTML for the bank website. The top of the page shows the user's current balance; the bottom of the page lets the user send money to someone else. To display the user's balance on line 4, the developer uses Django-style templating, where expressions between pairs of curly brackets are dynamically evaluated by `Mjölnir.js`.

```
// transmit some money
function send() {
 var fromName = mjolnir.user;
 var from =
  mjolnir.table('users').find(fromName);

 var toName = document.
  getElementById('to').value;
 var to =
  mjolnir.table('users').find(toName);

 var amount = parseInt(
  document.getElementById('amount').value);

 if (from.balance >= amount) {
  from.balance -= amount;
  to.balance += amount;
 }
}
```

Figure 4: JavaScript code for the bank website. The `send()` function transmits money from the current user to the recipient indicated in the HTML form.

(as in Line 4 of Figure 3), the corresponding DOM nodes are added to the write set.

If a transaction is aborted, Mjölnir may have to roll back both server-side storage and client-side JavaScript state. For example, in the banking application, two users X and Y might simultaneously `send()` money to user Z. Mjölnir would allow one transaction to succeed, while aborting the other and forcing it to retry. This preserves the atomic execution semantics that developers currently associate with event handler call chains, and allows developers to ignore the presence of multiple, concurrent clients.

Implementing Transactions: To determine the remote objects in a transaction's read/write set, `Mjölnir.js` merely has to log which of its remote storage interfaces are invoked by the application. Determining the `JavaScript` variables in the read/write set is more difficult, since accesses to those objects are not mediated by `Mjölnir.js`. In particular, an event handler can modify an arbitrary global variable (or a value that is reachable from a global variable); if the handler's transaction is rolled back, the variable must be reset to its previous value.

There are several ways that Mjölnir could log and rollback the JavaScript values that a transaction accesses. JavaScript rewriting is one such approach. For example, Crom [30] forces a speculative event handler to update shadow JavaScript state that is only committed if the speculated event is actually generated. Crom-style rewriting would allow Mjölnir to log variable accesses without support from the JavaScript engine. However, to make speculative copies of foreground state, Crom must use slow, JavaScript-level reflection interfaces; thus, cloning can be expensive without developer hints about which state the event handler will not touch (and thus does not need to be cloned). To avoid the need for such hints, Mjölnir could represent an application's JavaScript state using immutable functional data structures [9, 32], so that updates create new versions of the structures which Mjölnir can simply discard if a transaction aborts.

Enforcing Consistency: Given client-side machinery for enforcing transactions, how does Mjölnir allow multiple clients to concurrently (and safely) access server-side data? The most straightforward solution is for `Mjölnir.js` to grab server-side locks for the relevant objects. However, this approach injects wide-area latencies into the critical path of transaction processing. To avoid those network penalties, `Mjölnir.js` could use optimistic concurrency control, reading data over the network and queuing client-side writes; the client would ship those writes in bulk to the server at the end of each transaction. The Mjölnir back-end would serialize all transactions and inform clients which ones committed and which ones aborted.

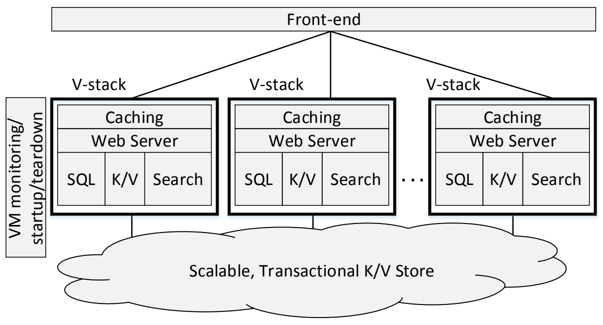
Optimistic concurrency minimizes locking overhead if conflicts are rare, but clients must still block on wide-area network reads if a desired object is not cached locally. To completely eliminate wide-area reads, Mjölnir could run *the entire transaction* on the server-side. In this approach, a client would send the JavaScript code in the transaction to execute, as well as the client-side JavaScript state that the transaction might touch. The server would then execute the JavaScript code (performing all IOs and lock acquisitions locally) and send the resulting client-side state back to the client.[2] Shipping the entire client-side state back and forth for each transaction would require excessive network bandwidth. So, clients and servers can use a delta-encoding protocol to maintain a shared view of the client's state. Immutable functional data structures make it straightforward to generate the deltas.

[2]Web frameworks like Meteor [29] allow developers to write JavaScript code that can run on both clients and servers. However, these frameworks do not deal with transactions, or the more general issue of hiding the service complexity that we discussed in Section 1.

Synthesizing Server-side Code: Given the HTML and JavaScript in Figures 3 and 4, Mjölnir should automatically generate the associated server-side code. If transactions use the third approach mentioned above, whereby the entire transaction runs on the server, then the server code *is* the client code. In this scenario, enforcing access controls and guaranteeing service invariants is straightforward, since the Mjölnir back-end trusts itself to perform these checks faithfully. For example, in Figure 4, the back-end trusts itself to access the tables of both users. However, the back-end does need to verify that the client submitted a whitelisted transaction–clients should not be able to arbitrarily manipulate server-side state. So, instead of supplying raw transaction commands to servers, a client should submit a high-level transaction identifier (e.g., `transferMoney`), the inputs to that transaction, and the identity of the user (e.g., as described in an HTTP cookie). Before the server executes the transaction, the server verifies that the specified user is allowed to execute the specified transaction.

Many web applications require server-side daemon code. This code runs in the background, on behalf of the service instead of a particular end-user; daemons might back-up data, rebuild indices, or garbage collect unneeded resources. Mjölnir represents such code as a "headless web page." Just like a regular Mjölnir page, a headless page is written in HTML, CSS, and JavaScript, and it uses `Mjölnir.js` to interact with server-side resources. However, when a developer submits a headless page to the Mjölnir back-end, the developer sets the page's daemon flag to true. This instructs the back-end to launch a windowless server-side browser to run the web page. The developer can interact with the daemon by detaching it from the server-side browser and running it within a developer-side one. However, a daemon page continues its logical execution on the back-end even if no developer has attached a browser to it.

Generalizing the Architecture: Our discussion of Mjölnir has been couched in terms of traditional web technologies like HTML, CSS, and JavaScript. However, Mjölnir's high-level architecture (a simplified wide-area concurrency model with no explicitly written server-side code) is generalizable to other programming languages and data representations. For example, to detect read/write sets and to rollback client-side state, Mjölnir can use software transactional memory frameworks for languages like Java [12]. and Python [40]. Mjölnir's server-side components (which we describe in the next section) are also compatible with arbitrary client-side technologies. Thus, Mjölnir's architecture can support native code software stacks that do not use HTML, CSS, and JavaScript.



Front-end
V-stack
V-stack
V-stack
VM monitoring/ startup/teardown
Caching
Web Server
SQL
K/V
Search
...
Scalable, Transactional K/V Store

Figure 5: Mjölnir's server-side architecture uses vertical replication of top-to-bottom stacks. This contrasts with traditional architectures which use horizontal replication of individual software components (see Figure 1).

5 Automating Server-side Configuration and Deployment

Abstractly speaking, the server-side of a web application is a black box; this box receives client requests, performs application-specific computations, and returns the results. Ideally, developers would only need to worry about the high-level, application-specific logic, such as table schemas and the associated queries. Unfortunately, modern web architectures force developers to grapple with a host of additional low-level details. As shown in Figure 1, developers have to reason about multiple software components which must be configured, deployed, and monitored for failure.

Software container projects like Docker [13] and CoreOS [11] make it easier to deploy software components to physical machines. However, developers still have to *configure* and *connect* those modules. For example, Docker supplies predefined images [14] for popular building blocks like Redis and Nginx. However, developers still have to stitch those components together to form a larger distributed service. Furthermore, if developers want to create their own Docker component, they must engage in low-level administrative tasks [15]: developers must pick a base image (e.g., a RedHat kernel), and then construct a `DockerFile` which specializes the image (e.g., by setting environment variables, modifying the image's file system, and so on). Difficulty of configuration is a major pain point for users of modular infrastructure-as-a-service frameworks like OpenStack [31]. Adding, removing, or patching a component typically results in the creation, deletion, or modification of inter-component workflows, and it can be difficult to understand how even small changes will cascade throughout the system [6, 20, 37].

To scale out, a traditional web service replicates *individual software components*. For example, Stack Exchange runs multiple instances of the web server, the SQL database, and the in-memory caching layer (see Figure 1). Replication is a standard technique for balancing load and increasing IO parallelism. However, traditional replication only adds to the tangled set of component dependencies that developers must consider. To reduce the management, configuration, and debugging overhead, Mjölnir replicates *vertically* instead of horizontally. Mjölnir uses vertical stacks (or "V-stacks") of preselected software components that are packaged into a single VM image. A single V-stack might contain modules like a SQL engine, a web server, and an in-memory caching layer. To shard data across multiple stacks, Mjölnir leverages scalable, transactional key/value stores like F1 [43]; higher-level storage interfaces like SQL are layered atop the key/value store. The Mjölnir front-end coordinates the activity of clients and servers, enforcing the client-visible abstraction of a single, distributed event loop (§3), and load balancing client requests across V-stacks. The resulting architecture is shown in Figure 5, and has several nice properties:

- Scaling the service is just as easy as with horizontal replication. Mjölnir creates or destroys V-stacks as client load increases or decreases; as the number of V-stacks changes, Mjölnir redistributes data to balance load.
- Vertical partitioning also simplifies configuration and management. Whereas Docker supplies predefined images for individual software components, Mjölnir supplies predefined images for entire server-side stacks. Mjölnir, not the developer, is responsible for properly connecting the individual modules within a vertical stack. Developers simply choose the V-stack whose individual components provide the required characteristics for performance and data durability. Developers tell Mjölnir to deploy the chosen V-stack, and Mjölnir handles the rest. Thus, Mjölnir removes an entire class of potential misconfiguration errors, and allows developers to focus on high-level application abstractions.
- Vertical partitioning also makes it easier for developers to test their web services. Mjölnir can launch an instance of the vertical stack on a developer's local machine; if that machine also has a web browser, the developer can test the full, end-to-end service using only local resources.

Platform-as-a-Service frameworks like AppEngine already expose scale-out, failure-resistant components like SQL databases and key/value stores. However, PaaS frameworks still require developers to write distributed code, often in multiple languages for multiple software components. PaaS systems also tie developers to a particular set of back-end technologies. In contrast, Mjölnir provides a simpler front-end programming model (a global, transactional JavaScript event loop) while still allowing developers to choose their own back-end V-stacks.

A single V-stack contains multiple components, and in many cases, Mjölnir can leverage off-the-shelf software for these components. For example, a V-stack might use commodity Redis for a key-value store, and commodity Lemur to perform text search. However, Mjölnir will require new innovations to automate V-stack configuration, deployment, and failure recovery. Indeed, the aggressiveness with which Mjölnir can automate these tasks will determine the extent to which Mjölnir can protect developers from low-level details about server-side components. Our idealistic vision for Mjölnir is that developers *only* need to write JavaScript, HTML, and CSS, with Mjölnir synthesizing (and hiding) the entire server-side implementation. In practice, Mjölnir will have to expose some view of server-side components, if only to allow developers to 1) choose which V-stacks to deploy, and 2) performance-debug those stacks to see if different stacks should be used. Devising such management and debugging interfaces is a key area for future research, but we can already see some potential avenues for investigation. For example, since each V-stack has a cleanly defined, message-driven interface between other V-stacks and client-side code, distributed event tracking frameworks like Domino [27], XTrace [17], and Dapper [44] should help Mjölnir developers to understand bugs. To debug Mjölnir's automatically generated server-side code, developers can leverage source maps [47] which describe the translation process.

Another challenge involves the partial scaling of a V-stack. Abstractly speaking, a single V-stack is an exemplar for the entire server-side workflow–as client load increases and decreases, Mjölnir creates and destroys V-stacks. However, what happens if client workloads increase the stress on some (but not all) components in a V-stack? For example, if a V-stack contains a search module and a SQL module, what happens if there is a surge in client requests that skew towards the search module? At first glance, naïvely spawning entire V-stacks seems wasteful, since the resources that are allocated to the SQL modules will lie fallow. However, using transparent page sharing [49], Mjölnir can coalesce shared V-stack memory pages into a single copy, reducing overall memory pressure. Newly created VMs can also fault-in memory pages lazily [25], eliminating copy operations for cold V-stack resources that will never be touched.

6 Conclusions

Designing a complex system is an exercise in trade-offs. However, the systems community has been making the wrong trade-offs with respect to large-scale web services. Conventional wisdom says that ease of development and deployment must be sacrificed for the sake of scalability and performance. We disagree. By abandoning the seductive (yet often inappropriate) goal of optimal performance, we can dramatically simplify the development, deployment, and management of surprisingly large web sites.

References

[1] Aeron. Efficient reliable unicast and multicast transport protocol, 2015. https://github.com/real-logic/Aeron.

[2] Angular. Superheroic JavaScript MVW framework, 2015. https://angularjs.org/.

[3] Apache Kafka. A high-throughput distributed messaging system., 2015. http://kafka.apache.org/.

[4] Apache Software Foundation. Open Source Cloud Computing, 2015. https://cloudstack.apache.org/.

[5] Apache Solr. Apache solr 5.2.1, 2014. http://lucene.apache.org/solr/.

[6] R. H. K. Ariel Rabkin. How Hadoop clusters break. In *IEEE Software, Vol. 30, No. 4*, 2013.

[7] Backbone. Backbone.js, 2015. http://backbonejs.org/.

[8] Bitly. Lessons learned building a distributed system that handles 6 billion clicks a month, 2014. http://highscalability.com/blog/2014/7/14/bitly-lessons-learned-building-a-distributed-system-that-han.html.

[9] ClojureScript. Clojure to JS compiler, 2015. https://github.com/clojure/clojurescript.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of OSDI*, 2012.

[11] CoreOS. CoreOS is Linux for Massive Server Deployments, 2015. http://coreos.com.

[12] Deuce STM. Java Software Transactional Memory, 2014. https://sites.google.com/site/deucestm/.

[13] Docker. Build, ship, and run any app, anywhere, 2015. https://www.docker.com.

[14] Docker. Docker registry, 2015. https://registry.hub.docker.com/.

[15] Docker. Understanding Docker, 2015. https://docs.docker.com/introduction/understanding-docker/.

[16] ElasticSearch. Open source distributed real time search & analytics, 2015. http://www.elasticsearch.org/.

[17] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of NSDI*, 2007.

[18] Ganglia. Ganglia monitoring system, 2014. http://ganglia.info/.

[19] Google Compute Engine. Load balancing, 2015. https://cloud.google.com/compute/docs/load-balancing/.

[20] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure Recovery: When the Cure is Worse Than the Disease. In *Proceedings of HotOS*, 2013.

[21] HAProxy. The reliable, high performance TCP/HTTP load balancer, 2015. http://www.haproxy.org/.

[22] Hoodie. Hoodie is for you, 2015. http://hood.ie/.

[23] T. Jones. Beware the steep OpenStack install learning curve, April 28, 2015. TechTarget. http://searchcloudcomputing.techtarget.com/news/4500245245/Beware-the-steep-OpenStack-install-learning-curve.

[24] Kony. Rapidly deliver great mobile apps, 2015. http://www.kony.com/.

[25] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of EuroSys*, 2009.

[26] D. Laube. Why We Threw 4 Months of Work in the Trash; or How we Failed at OpenStack, January 12, 2015. https://www.packet.net/blog/how-we-failed-at-openstack/.

[27] D. Li, J. Mickens, S. Nath, and L. Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causalities in Web Applications. In *Proceedings of SOCC*, 2015.

[28] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012.

[29] Meteor. Meteor: The JavaScript App Platform, 2015. https://www.meteor.com.

[30] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster web browsing using speculative execution. In *Proceedings of NSDI*, April 2010.

[31] Microsoft. Meet Microsoft's new Azure CTO, Mark Russinovich, 2014. http://www.cio.in/topstory/meet-microsoft%27s-new-azure-cto,-mark-russinovich.

[32] Mori. ClojureScript's persistent data structures and supporting API from the comfort of vanilla JavaScript, 2015. http://swannodette.github.io/mori/.

[33] Mozilla Developer Network. WebSockets, 2015. https://developer.mozilla.org/en-US/docs/WebSockets.

[34] Nagios. The Industry Standard in IT Infrastructure Monitoring, 2015. http://www.nagios.org/.

[35] NSQ. A realtime distributed messaging platform, 2015. http://nsq.io/.

[36] OpenStack. Open source software for creating private and public clouds, 2015. https://www.openstack.org/.

[37] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USENIX Association, 2003.

[38] Opserver. Stack Exchange's monitoring system, 2015. https://github.com/opserver/Opserver.

[39] Parse. Build your perfect app on any platform, 2015. https://www.parse.com/.

[40] Pyrsistent. Persistent/Immutable/Functional data structures for Python, 2015. https://github.com/tobgu/pyrsistent.

[41] Redis. Five tips for running Redis over AWS, 2014. https://redislabs.com/blog/5-tips-for-running-redis-over-aws.

[42] J. Robie. What is the Document Object Model?, 1998. http://www.w3.org/TR/REC-DOM-Level-1/introduction.html.

[43] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *SIGMOD*, 2012.

[44] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, 2010. http://research.google.com/archive/papers/dapper-2010-1.pdf.

[45] Sphinx. Sphinx open source search engine, 2015. http://sphinxsearch.com.

[46] StackOverflow. 560M Page Views a Month, 25 Servers, and It's All About Performance, 2014. http://highscalability.com/blog/2014/7/21/stackoverflow-update-560m-pageviews-a-month-25-servers-and-i.html.

[47] Traceur compiler. SourceMaps, 2014. Google. https://github.com/google/traceur-compiler/wiki/SourceMaps.

[48] Varnish. Varnish makes websites fly!, 2015. https://www.varnish-cache.org/.

[49] VMware. Understanding Memory Resource Management in VMware vSphere 5.0, 2011. http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf.