Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Futures The results of asynchronous operations are represented by *futures* which compose to express dependencies between operations.

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

We have deployed this in very large distributed systems with great success. Indeed, Finagle and its accompanying structuring idioms are used throughout the entire Twitter service stack—from frontend web servers to backend data systems.

All of the code examples presented are written in the Scala [17] programming language, though the abstractions work equally well, if not as concisely, in our other principal systems language: Java.

2. Futures

A future is a container used to hold the result of an asynchronous operation such as a network RPC, a timeout, or a disk I/O operation. A future is either *empty*—the result is not yet available; *succeeded*—the producer has completed and has populated the future with the result of the operation; or *failed*—the producer failed, and the future contains the resulting exception.

An immediately successful future is constructed with `Future.value`; an immediately failed future with `Future.exception`. An empty future is represented by a `Promise`, which is a writable future allowing for at most one state transition, to either of the nonempty states. Promises are similar to I-structures [4], except that they embody failed as well as successful computations; they are rarely used directly.

Futures compose in two ways. First, a future may be defined as a function of other futures, giving rise to a dependency graph which is evaluated in the manner of dataflow programming. Second, independent futures are executed concurrently by default—execution is sequenced only where a dependency exists.

Futures are first class values; they are wholly defined in the host language.

All operations returning futures are expected to be asynchronous, though this is not enforced.

Dependent composition It is common to sequence two asynchronous operations due to a data dependency. For example, a search engine frontend, in order to provide personalized search results, might consult a user service to rewrite queries. Given:

```
def rewrite(user: String,
  query: String): Future[String]

def search(query: String): Future[Set[Result]]
```

then, to perform our search, we first need to invoke `rewrite` to retrieve the personalized query, which is then used as a parameter to `search`.

We can phrase this combined operation as a future *transformation*: the operation evaluates to the future that represents the result of `rewrite`, applied to `search`. These transformations are loosely analogous to Unix pipes [18]. With an imagined syntax, we might write `def psearch(user, query) = rewrite(user, query) | search(_)`; `_` being the placeholder argument for the result from the "pipe."

The `flatMap` combinator performs this kind of transformation:

```
trait Future[T] {
  def flatMap[U](f: T => Future[U]): Future[U]
  ...
}
```

(The type `T => Future[U]` denotes a unary function whose argument is of type `T` and whose result is of type `Future[U]`.) Thus `flatMap`'s argument is a function which, when the future succeeds, is invoked to produce the dependent future. `flatMap`, as with all future combinators, is asynchronous: it returns immediately with a future representing the result of the composite operation.

Personalized search is a straightforward application of `flatMap`:

```
def psearch(user: String, query: String) =
  rewrite(user, query) flatMap { pquery =>
    search(pquery)
  }
```

(Scala provides "infix" notation for unary methods; parentheses around the method argument are omitted, so is the dot for method dereference. Function literals are introduced with { `param => expr` }.) `psearch` returns a `Future[Set[Result]]`, thus

```
val result: Future[Set[Result]] =
  psearch("marius", "finagle")
```

issues the personalized search for "finagle;" `result` is the future representing the result of the composed operation.

In this example, `flatMap` is used to resolve a *data dependency* between `search` and `rewrite`; `psearch` merely expresses this, without specifying an execution strategy.

Handling errors `flatMap` short-circuits computation when the outer future fails: the returned future is failed without invoking the given function to produce a dependent future. Indeed, its type bears witness to its semantics: when a `Future` fails, it does not have a value to present to the function producing the dependent future. These error semantics are analogous to traditional, stack-based exception semantics.

It is often useful to recover from such errors, for example in order to retry an operation, or to provide a fallback value. The `rescue` combinator provides this; whereas `flatMap` operates over successful results, `rescue` operates over failures.

```
trait Future[T] {
  ...
  def rescue[B](
    f: PartialFunction[Throwable, Future[B]]
  ): Future[B]
}
```

While `flatMap` demands a total function, `rescue` accepts partial functions, allowing the programmer to handle only a subset of possible errors. For example, we can modify `psearch` from above to skip personalization if the query rewriting system fails to complete within a deadline, so that a degraded result is returned instead of a failure:

```
def psearch(user: String, query: String) =
  rewrite(user, query).within(50.milliseconds) rescue {
    case _: TimeoutError => Future.value(query)
  } flatMap { pquery =>
    search(pquery)
  }
```

(`{ case ...` is a partial function literal in Scala; it may contain multiple `case` clauses. `within` is a method on `Future` that returns a new `Future` which either completes within the given duration, or fails with a timeout error.) If the future returned from `rewrite(..).within(..)` fails, and the partial function is defined for the specific failure—in this case, we match on the exception type `TimeoutError`—the error is recovered by supplying the original query value. Since each combinator returns a future representing the result of the composite operation, we can chain them together as in this example. This style is idiomatic.

Composing multiple dependencies Servers commonly perform "scatter-gather," or "fan-out" operations. These involve issuing requests to multiple downstream servers and then combining their results. For example, Twitter's search engine, Earlybird [5], splits data across multiple segments; a frontend server answers queries by issuing requests to a replica of each segment, then combines the results.

The `collect` combinator resolves multiple dependencies. For some value type `A`, `collect` converts a sequence of futures into a future of a sequence of `A`-typed values:

```
def collect[A](fs: Seq[Future[A]]): Future[Seq[A]]
```

(`Seq` is Scala's generic container for sequences.) A sketch of a scatter-gather operation follows. Given a method to query segments,

```
def querySegment(id: Int, query: String): Future[Set[Result]]
```

we use `collect` to relate multiple dependencies; the `search` method from the previous example can thus be defined:

```
def search(query: String): Future[Set[Result]] = {
  val queries: Seq[Future[Result]] =
    for (id <- 0 until NumSegments) yield {
      querySegment(id, query)
    }

  collect(queries) flatMap { results: Seq[Set[Result]] =>
    Future.value(results.flatten.toSet)
  }
}
```

As with `flatMap`, errors in any future passed as an argument to `collect` propagate immediately to the composed future: should any of the futures returned by `querySegment` fail, the collected future will fail immediately.

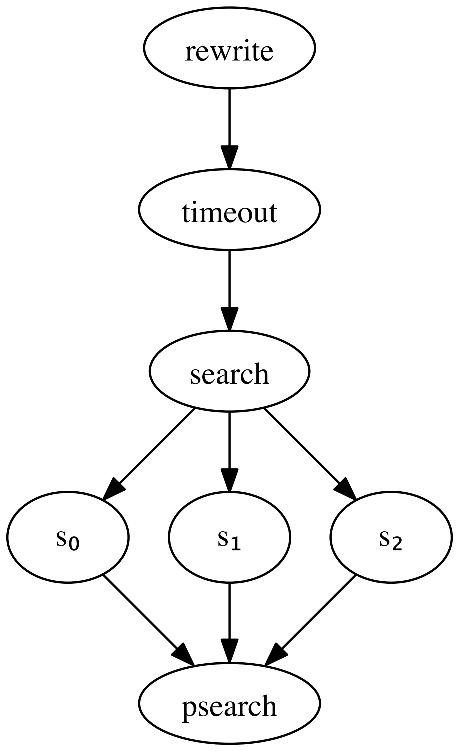

Figure 1. The data flow graph between psearch's constituent futures.

In combination, invoking psearch returns a future composed of a number of sub-operations. The resulting dataflow graph is illustrated in figure 1 (when searching over three segments s_{0-2}).

Recursive composition It is idiomatic to use Future combinators recursively. Continuing with our previous example, we can search iteratively until we have a requisite number of results, perhaps by permuting our query in some way.

```
def permute(query: String): String

def rsearch(user: String, query: String,
    results: Set[Results],
    n: Int): Future[Set[Result]] =
  if (results.size >= n)
    Future.value(results)
  else {
    val nextQuery = permute(query)
    psearch(user, nextQuery) flatMap { newResults =>
      if (newResults.size > 0)
        rsearch(user, nextQuery,
          results ++ newResults, n)
      else
        Future.value(results)
    }
  }
```

The error handling semantics of futures are such that rsearch will short-circuit and fail should any of its constitutent operations fail.

flatMap merges futures, implementing a form of tail-call elimination: the above example will not produce any space leaks; it is safe to define recursive relationships of indefinite length.

3. Services and Filters

A *service* is an asynchronous function, typically representing some remote endpoint to which RPCs are dispatched; it is distinguished from a regular function in Scala by enforcing that its return value is represented by a Future:

```
type Service[Req, Rep] = Req => Future[Rep]
```
[1]

Services represent clients and servers symmetrically and are used by Finagle. Servers implement services to which Finagle dispatches incoming requests; Finagle furnishes clients with instances of Service representing either virtual or concrete remote servers.

For example, this HTTP service dispatches a request to the Twitter web site, returning a future representing the eventual reply:

```
val client: Service[HttpReq, HttpRep] =
  Http.newService("twitter.com:80")
val f: Future[HttpRep] = client(HttpReq("/"))
```

An HTTP echo server may be implemented thus:

```
Http.serve(":80", { req: HttpReq =>
  Future.value(HttpRep(Status.OK, req.body))
})
```

Putting client and server together neatly illustrates the symmetry of the service abstraction: The following is a primitive HTTP proxy, forwarding HTTP traffic from the local port 8080 to twitter.com.

```
Http.serve(":8080",
  Http.newService("twitter.com:80"))
```

Services are used to represent logical application components like an HTTP server responsible for part of Twitter's public API, a Thrift [3] RPC server providing a user authentication service, a memcached [11] client representing a cluster balanced with consistent hashing [15], etc. However, a number of *application agnostic* concerns arise when building server software; these include timeouts, retry policies, service statistics, and authentication.

Filters implement application-independent functionality; they are composed with services to modify service behavior. As with services, filters are simple functions:

```
type Filter[Req, Rep] =
  (Req, Service[Req, Rep]) => Future[Rep]
```

That is, a filter receives a request and a service with which it is composed; function application evaluates to a future. It follows that an identity filter simply defers to the given service:

```
val identityFilter =
  { (req, service) => service(req) }
```

A filter that performs request timeout can be implemented thus:

```
def timeoutFilter(d: Duration) =
  { (req, service) => service(req).within(d) }
```

[1] Read: A Service, parameterized by the types Req and Rep, is a unary function taking a function parameter of type Req and returning a value of type Future[Rep]. This is a simplified version of the one implemented in Finagle, which extends Scala's Function trait without declaring a type alias.

Filters provide a combinator, andThen, which is used to combine filters with other filters—producing composite filters—or with services—producing a new service whose behavior is modified by the filter.

```
val httpClient: Service[HttpReq, HttpRep] = ...
val httpClientWithTimeout: Service[HttpReq, HttpRep] =
  timeoutFilter(10.seconds) andThen httpClient
```

Since services are symmetric, filters may be applied to both clients and servers.

Filters can also transform requests and responses, enabling the programmer to use the static typing facilities of the programming language to enforce certain guarantees. For example, an HTTP server may use a separate request type to indicate authenticated requests; the method

```
def authReq(req: HttpReq): Future[AuthHttpReq]
```

might authenticate the given request via an authentication service, returning an "upgraded" request on success, failing otherwise. The filter

```
val auth: (HttpReq, Service[AuthHttpReq, HttpRes])
  => Future[HttpRep] = {
    (req, service) =>
      authReq(req) flatMap { authReq =>
        service(authReq)
      }
  }
```

composes with a service demanding authentication, yielding a new service to which unauthenticated requests may be dispatched.

```
val authedService: Service[AuthHttpReq, HttpRep] = ...
val service: Service[HttpReq, HttpRep] =
  auth andThen authedService
```

Thus we can express authentication requirements in static types, which are in turn enforced by the compiler. This reduces the surface area of authentication mishandling to authReq, providing an effective, type safe firewall between the components dealing with unauthenticated and authenticated requests.

4. Discussion

4.1 Declarative programming with futures

The style of declarative programming encouraged by futures forces the programmer to structure his system as a set of components whose data dependencies are witnessed by the various future combinators. This is a sort of systems *description*, divorcing the semantics of an operation, which are described by the programmer, from execution details, handled by Finagle.

This has been enormously beneficial, freeing the programmer from the tedium of managing threads, queues, resource pools, and resource reclamation, allowing him instead to focus on application semantics.

This achieves a kind of modularity as we separate concerns of program semantics from execution details. We focus our efforts on efficient execution in Finagle, and indeed employ different execution strategies for different types of servers. For example, Finagle can implement thread affinity, so that all I/O belonging to a logical server request are handled on a single operating system thread, reducing context switching costs. Intriguing opportunities lurk here: How can we use runtime information to improve execution strategies?

Because Finagle implements the runtime, we were able to add features like Dapper-style RPC tracing [20] without changing APIs or otherwise modify any existing user code.

Additionally, the style encourages the programmer to think about data-flow over control-flow, which in turn tends to lead to code whose semantics are preserved under non-deterministic concurrent computation: synchronization concerns are subsumed by the data-flow, as expressed by future combinators. The emphasis on data-flow encourages the programmer to structure his software in terms of transformations of immutable values, not as a sequence of mutations of shared data. We believe this makes it simpler to reason about shared data, especially in the presence of concurrency. This is perhaps the principal advantage of Future-based concurrency.

Another, perhaps surprising, benefit is that since future types are "infectious"—any value derived from a future must itself be encapsulated with a future—asynchronous behavior is witnessed by a program's static types. A programmer can then tell simply by a method signature whether dispatching it is likely to be expensive.

Futures are cheap in construction and maintenance. Our current implementation allocates 16 bytes for the central data structure, and our runtime library multiplexes operations onto several underlying OS threads, using efficient data structures (for actions like timeouts), and the operating system I/O multiplexing facilities (for I/O actions.)

While most of our engineers find the programming model unusual, they tend to internalize it quickly.

4.2 Futures in practice

Futures, as presented, are read-only, "pure" constructs: the producer of a value is separated from its consumer. This enhances modularity and makes the program simpler to reason about; however the real, messy world of distributed systems inevitably complicates matters.

Let's consider a simple example: timeouts. Imagine an HTTP client represented by a service to which we apply a timeout filter

```
val httpClient: Service[HttpReq, HttpRep] =
  Http.newService("twitter.com:80")
val timeoutClient: Service[HttpReq, HttpRep] =
  timeoutFilter(1.second) andThen httpClient

val result: Future[HttpRep] =
  timeoutClient(HttpReq("/"))
```

If the request fails to complete within 1 second, result fails. However, futures are read-only: the underlying operation, as initiated by the HTTP client, is not terminated. This becomes problematic when connections are scarce, or if the remote server is partitioned from the network.

The read-only, data-flow semantics of futures seemingly force consumers of futures to have no knowledge of their producers. This is good abstraction, but as we've seen, it can also introduce a form of resource leaking. (This is not unlike how languages with lazy evaluation semantics, such as Haskell [2], may introduce space leaks.)

We introduced an *interrupt* mechanism to bridge the gap. Interrupts enable consumers of a future to notify the asynchronous operation responsible for populating it, typically because the result is no longer needed. Interrupts flow in the opposite direction of the data carried by futures, and they are advisory. Interrupts don't directly change the state of the future, but a producer may act on it. We added interrupt handling to the bottom-most part of our network clients. In practice, only a handful of places in our code base, such as our timeout filter, were modified to raise interrupts.

Interrupts also allowed us to implement end-to-end cancellation in all of our servers. First, we added a control message to our RPC protocol to instruct the server to cancel an in-flight request. A client, when interrupted, issues this control signal. When a cancellation signal is received by a server, it raises an interrupt on the pending future (as returned by the server-supplied `Service`). This allows us to interrupt a request in our frontend HTTP server, canceling all ongoing work on behalf of that request throughout our distributed systems. As with our tracing system, this was implemented without any API modifications or other changes to user code.

While interrupts violate the pure data flow model presented by futures, consumers are still oblivious to their producers. Interrupts are advisory, and do not directly affect the state of the future.

Interrupts are not without problems. They introduce new semantic complications: Should combinators propagate interrupts to all futures? Or only the outstanding ones? What if a future is shared between multiple consumers? We don't have great answers to these questions, but in practice interrupts are used rarely, and then almost exclusively by Finagle; we have not encountered any problems with their semantics or their implementation.

4.3 Filters

Filters have held their promise of providing clean, orthogonal, and application-independent functionality. They are used universally: Finagle itself uses filters heavily; our frontend web servers—reverse HTTP proxies through which all of our external traffic flows—use a large stack of filters to implement different aspects of its responsibilities. This is an excerpt from its current configuration:

```
recordHandletime     andThen
traceRequest         andThen
collectJvmStats      andThen
parseRequest         andThen
logRequest           andThen
recordClientStats    andThen
sanitize             andThen
respondToHealthCheck andThen
applyTrafficControl  andThen
virtualHostServer
```

Filters are used for everything from logging, to request sanitization, traffic control, and so on. Filters help enhance modularity and reusability, and they have also proved valuable for testing. It is quite simple to unit test each of these filters in isolation—their interfaces are simple and uniform—without any set up, and with minimal mocking. Furthermore, they encourage programmers to separate functionality into independent modules with clean boundaries, which generally leads to better design and reuse.

We have also used filters extensively to address lower-level concerns. For example, we were able to implement a sophisticated backup request mechanism using a simple filter in about 40 lines of code (see Appendix A). Engineers at Tumblr, who also use Finagle, report [16] the use of a low level filter to deduplicate request streams.

4.4 The cost of abstraction

High level programming languages and constructs do not come for free. Future combinators allocate new futures on the garbage collected heap; closures, too, need to be allocated on the heap, since their invocation is deferred. While we've focused on reducing the allocation footprints—and indeed created many tools for allocation analysis—it is an ongoing concern.

The tail latencies of most of our servers are governed by minor heap garbage collections. In isolation, this implies only a small service degradation. However our large fan-out system amplifies such effects as overall request latency is governed by the slowest component; with large request distribution—often 100s of systems—encountering minor garbage collection in the request path is common. Dean and Barroso [7] describe similar experiences at Google.

A frequent source of unintentional garbage collection pressure is the ease with which space leaks can be introduced by the inadvertent capturing of references in closures. This is amplified by long-lived operations, for example, closures that are tied to lifetime of a connection, and not of a request. Miller et.al.'s Spores [14] proposes to mitigate these types of leaks by giving the programmer fine-grained control over the environment captured by a closure.

In most of our servers, major collections are rare. This gives rise to another kind of space leak: if a `Promise` is promoted to the major heap (for example because the operation it represents took an unexpectedly long time), its referent value, even if its useful lifetime is miniscule, survives until the next major garbage collection.

Development discipline is an important mitigating factor. In order to ensure that allocation regressions aren't introduced, we have developed a tool, JVMGCPROF [9] which runs regularly along with our tests, providing reports on per-request allocation rates and lifetimes.

This is an area of ongoing effort with many intriguing possibilities. Since Finagle controls logical-to-physical thread multiplexing and is aware of request boundaries, it can bias allocation. This opens up the possibility that, with the cooperation of the underlying JVM, we may make use of region allocation techniques [13].

4.5 Futures, Services, and Filters at Twitter

These techniques are used together with our RPC system [10], throughout our production runtime systems. Almost all of our modern server software is implemented in this way, including our frontend serving system, application servers, web crawling systems, database systems, and fan-out and data maintenance systems. The systems are employed in a majority of machines across multiple datacenters.

We've found these techniques excel especially in middleware servers, whose interface is a `Service` and whose dependencies are other `Services`. Such middleware reduces to effectively big "future transformers": it's common to express the entire system in a declarative fashion.

The advantages of uniformity extend beyond individual server software. Statistics, tracing data, and other runtime information is observed with the use of a common set of filters. These in turn export such runtime information in a uniformly, so that operators can monitor and diagnose our systems without knowing specifics.

Our offline data processing systems have not yet adopted these techniques as they are often built on top of other frameworks such as Hadoop.

5. Related work

Lwt [22] is a cooperative threading library for OCaml whose chief abstraction, the lightweight thread, is similar to our `Future`.

Dataflow programming languages [8, 21, 23] also emphasize dependencies between computations, performing concurrent graph reduction in their runtimes. However, dataflow concurrency demands determinacy, requiring, among other things, timing independence and freedom from nondeterminate errors (most such languages require freedom from any errors). Thus in its raw form, dataflow languages are unsuitable for systems programming. Roy et.al. [19] propose introducing nondeterminate *ports* to split a dataflow program into pure (determinate) parts, connected by nondeterministic channels. This is intriguing, but in systems programming, nearly all concurrency is nondeterministic. (Indeed, you could argue that deterministic concurrency is better described as parallelism.)

Haskell [2] and Go [1] provide cheap user-space threads, reducing the overhead of thread-based concurrency. These runtimes manage threads as a cheap resource, and frees the programmer from the obligation of manually managing threads. However, they are distinct from futures in two ways. First, they do not provide a clean data flow model—their threads do not compose as naturally as do futures. Second, the management of threads is built into their runtimes, and thus limit the amount of runtime specialization that can be done by a separate library like Finagle [10].

Filters are a special case of the "decorator pattern" [12].

6. Conclusions

We have described an approach to structuring server software by using *futures*, *services*, and *filters*. Futures are our base abstraction for expressing the relationships between concurrent, asynchronous operations. Services and filters are used to structure servers and clients—they are symmetric—in a modular fashion.

Taken together, these abstractions form an orthogonal basis with which we construct server software in our demanding environment. Picking a few, well-thought-out abstractions to form such a basis has been highly profitable: In addition to making our software simpler and easier to reason about—services are constructed piecemeal by composing smaller parts—developers are encouraged to structure their applications along the same lines. This leads to the emergence of small, orthogonal, reusable components that compose well—a "software tools" approach to creating server software. As well, the emphasis on immutability leads to code that is easier to reason about, and with clearer module boundaries.

Finally, because of the high level of abstraction afforded by futures, services, and filters, program semantics are (mostly) liberated from execution mechanics: A separate runtime is implemented by our RPC system, Finagle, allowing developers to focus on their applications. Separating the runtime in this way also enhances modularity as code isn't tied to an execution strategy.

Acknowledgments

This is the work of many. Most of the core abstractions introduced here were designed together with Nick Kallen. Steve Gury, Arya Asemanfar, Wilhelm Bierbaum, Jeff Smick, Sonja Keserovic, Ruben Oanta, Rao Fu, David Helder, Jeremy Cloud, Glen Sanford, Blake Matheny, and Johan Oskarsson have made substantial contributions.

Jake Donham, Marcel Molina, Steve Gury, Jeff Smick, Nathan Taylor, Ruben Oanta, Berk Demir, Arya Asemanfar, Brian Wickman, and Adam Belay provided valuable feedback on this paper.

References

[1] Go. http://www.golang.org.

[2] Haskell. http://www.haskell.org.

[3] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.

[5] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1360–1369, Washington, DC, USA, 2012. IEEE Computer Society.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407. ACM, 2007.

[7] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[8] S. Doeraene and P. Van Roy. A new concurrency model for scala based on a declarative dataflow core. 2013.

[9] M. Eriksen. jvmgcprof. https://github.com/twitter/jvmgcprof, Nov. 2012.

[10] M. Eriksen and N. Kallen. Finagle. http://twitter.github.com/finagle, Nov. 2010.

[11] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, 1990.

[14] M. O. Heather Miller and P. Haller. SIP-21 - Spores. http://docs.scala-lang.org/sips/pending/spores.html, 2013.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[16] B. Matheny. Eliminating duplicate requests. http://tumblr.mobocracy.net/post/45358335655/eliminating-duplicate-requests, Mar. 2013.

[17] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[18] D. Ritchie. The evolution of the Unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 25–36, London, UK, UK, 1980. Springer-Verlag.

[19] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[20] 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, Inc., 2010.

[21] G. Smolka. The Oz programming model. In *COMPUTER SCIENCE TODAY, LECTURE NOTES IN COMPUTER SCIENCE*, pages 324–343. Springer-Verlag, 1995.

[22] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM.

[23] M. Zissman, G. O'Leary, and D. Johnson. A block diagram compiler for a digital signal processing mimd computer. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '87.*, volume 12, pages 1867–1870, 1987.

A. Backup request filter

```
class BackupRequestFilter[Req, Rep](
    quantile: Int,
    range: Duration,
    timer: Timer,
    statsReceiver: StatsReceiver,
    history: Duration,
    stopwatch: Stopwatch = Stopwatch
) extends SimpleFilter[Req, Rep] {
  require(quantile > 0 && quantile < 100)
  require(range < 1.hour)

  private[this] val histo = new LatencyHistogram(range, history)
  private[this] def cutoff() = histo.quantile(quantile)

  private[this] val timeouts = statsReceiver.counter("timeouts")
  private[this] val won = statsReceiver.counter("won")
  private[this] val lost = statsReceiver.counter("lost")
  private[this] val cutoffGauge =
    statsReceiver.addGauge("cutoff_ms") { cutoff().inMilliseconds.toFloat }

  def apply(req: Req, service: Service[Req, Rep]): Future[Rep] = {
    val elapsed = stopwatch.start()
    val howlong = cutoff()
    val backup = if (howlong == Duration.Zero) Future.never else {
      timer.doLater(howlong) {
        timeouts.incr()
        service(req)
      } flatten
    }

    val orig = service(req)

    Future.select(Seq(orig, backup)) flatMap {
      case (Return(res), Seq(other)) =>
        if (other eq orig) lost.incr() else {
          won.incr()
          histo.add(elapsed())
        }

        other.raise(BackupRequestLost)
        Future.value(res)
      case (Throw(_), Seq(other)) => other
    }
  }
}
```

The backup filter issues a secondary request if a response has not arrived from the primary request within a given time, parameterized by the observed request latency histogram. When a response arrives—the first to arrive satisfy the future returned by `Future.select`—the other request is cancelled in order to avoid unnecessary work. (Services furnished by Finagle are load balanced, so that repeated invocations to `service` are likely to be dispatched to different physical hosts.)