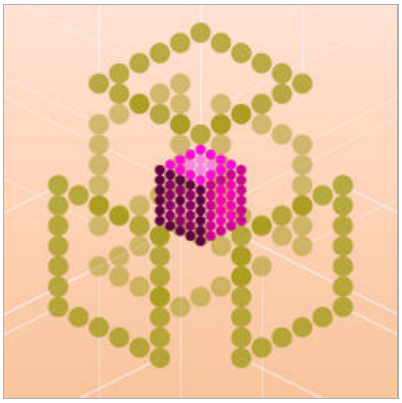
PRACTICE

Functional at Scale

By Marius Eriksen

Communications of the ACM, Vol. 59 No. 12, Pages 50-55
10.1145/2980985

Comments



Applying functional programming principles to distributed computing projects.

Modern server software is demanding to develop and operate: It must be available at all times and in all locations; it must reply within milliseconds to user requests; it must respond quickly to capacity demands; it must process a lot of data and even more traffic; it must adapt quickly to changing product needs; and, in many cases, it must accommodate a large engineering organization, its many engineers the proverbial cooks in a big, messy kitchen.

What's more, the best computers for these applications—whether you call them clouds, datacenters, or warehouse computers—are really bad. They are complex and unreliable, and prone to partial failures. They have asynchronous interconnects and deep memory hierarchies, and leave a lot of room for operator error.[1]

Cloud computing thus forces us to confront the full complexity of distributed computing, where seemingly simple problems require complicated solutions. While much of this complexity is inherent—the very nature of the problem precludes simpler solutions—much of it is also incidental, a simple consequence of using tools unfit for the purpose.

At Twitter, we use ideas from functional programming to tackle many of the complexities of modern server software, primarily through the use of higher-order functions and effects. *Higher-order functions*, or those that return other functions, let us combine simpler functions to define more complex ones, building up application functionality in a piecemeal fashion. *Effects* are values that represent some side-effecting operation; they are used in conjunction with higher-order functions to build complex effects from simpler ones.[7]

Higher-order functions and effects help build scalable software in two ways: First, building complex software from simple parts makes it easy to understand, test, reuse, and replace individual components; second, effects make side-effecting operations tractable, promoting modularity and good separation of concerns.

This article explores three specific abstractions that follow this style of functional programming: *futures* are effects that represent asynchronous operations; *services* are functions that represent service boundaries; and *filters* are functions that encapsulate application-independent behavior. In turn, higher-order functions provide the glue used to combine these to create complex systems from simple parts.

Futures, services, and filters are thus combined to build server software in a piecemeal fashion. They let programmers build up complex software while preserving their ability to reason about the correctness of its constituent parts.

ARTICLE CONTENTS:

By consistently applying these principles, programmers can construct systems that are at once simpler, more flexible, and performant.

Back to Top

Concurrent Programming with Futures

"Concurrent programs wait faster."—Tony Hoare

Concurrency is a central topic in server-software development.[1][2] Two sources of concurrency prevail in this type of software. First, scale implies concurrency. For example, a search engine may split its index into many small pieces (shards) so the entire corpus can fit in main memory. To satisfy queries efficiently, all shards must be queried concurrently. Second, communication between servers is asynchronous and must be handled concurrently for efficiency and safety.

Concurrent programming is traditionally approached by employing threads and locks[3]—threads furnish the programmer with concurrent threads of execution, while locks coordinate the sharing of (mutable) data across multiple threads.

In practice, threads and locks are notoriously difficult to get right.[9] They are hard to reason about, and they are a stubborn cause of nasty bugs. What's more, they are difficult to compose: you cannot safely and arbitrarily combine a set of threads and locks to construct new functionality. Their semantics of computation are wrapped up in the mechanics of managing concurrency.

At Twitter, we instead structure concurrent programs around futures. A future is an effect that represents the result of an asynchronous operation. It's a type of reference cell that can be in one of three states: *incomplete*, when the future has not yet taken on a value; *completed with a value*, when the future holds the result of a successful operation; and *completed with a failure*, when the operation has failed. Futures can undergo at most one state transition: from the incomplete state to either the success or failure state.

In the following example, using Scala, the future count represents the result of an integer-valued operation. We respond to the future's completion directly: the block of code after respond is a callback that is invoked when the future has completed. (As you will see shortly, we rarely respond directly to future completions in this way.)

```
val count: Future[Int] = getCount ()
count.respond {
  case Return(value) =>
    println(s"The count was $value")
  case Throw(exc) =>
    println(s"getCount failed with $exc")
}
```

Futures represent just about every asynchronous operation in Twitter systems: RPC (remote procedure call), timeout, reading a file from disk, receiving the next event from an event stream.

With the help of a set of higher-order functions (called *combinators*), futures can be combined freely to express more complex operations. These combinations usually fall into one of two composition categories: sequential or concurrent.

Futures, services, and filters are combined to build server software in a piecemeal fashion. They let programmers build up complex software while preserving their ability to reason about the correctness of its constituent parts.

Sequential composition permits defining a future as a function of another, such that the two are executed sequentially. This is useful where data dependency exists between two operations: the result of the first future is needed to compute the second future. For example, when a user sends a Tweet, we first need to see if that user is within the hourly rate limits before writing the Tweet to a database. In the Figure 1 example, the future done represents this composite operation. (For historical reasons, the sequencing combinator is called flatMap.)

This also shows how failures are expressed in futures: `Future.exception` returns a future that has already completed in a failure state. In the case of rate limiting, done becomes a failed future (with the exception `RateLimitingError`). Failure short-circuits any further composition: if, in the previous example, the future returned by `isRateLimited`(user) fails, then done is immediately failed; the closure passed to flatMap is not run.

Another set of combinators defines concurrent composition, allowing multiple futures to be combined when no data dependencies exist among them. Concurrent combinators turn a list of futures into a future of a list of values. For example, you may have a list of futures representing RPCs to all the shards of a search index. The concurrent combinator collect turns this list of futures into a future of the list of results.

```
val results: List[Future[String]] = ...
val all: Future[List[String]] =
  Future.collect(results)
```

Independent futures are executed concurrently by default; execution is sequenced only where data dependencies exist.

Future combinators never modify the underlying future; instead, they return a *new* future that represents the composite operation. This is an important tool for reasoning: composite operations do not change the behavior of their constituent parts. Indeed, a single future can be used and reused in many different compositions.

Let's modify the earlier `getCount` example to see how composition allows building complex behavior piecemeal. In distributed systems, it is often preferable to degrade gracefully (for example, where a default value or guess may exist) than to fail an entire operation.[8] This can be implemented by using a timeout for the `getCount` operation, and, upon failure, returning a default value of 0. This behavior can be expressed as a compound operation among different futures. Specifically, you want the future that represents the winner of a timeout-with-default and the `getCount` operation (see Figure 2).

The future `finalCount` now represents the composite operation as described. This example has a number of notable features. First, we have created a composite operation using simple, underlying parts—futures and functions. Second, the constituent futures preserve their semantics under composition—their behavior does not change, and they may be used in multiple different compositions. Third, nothing has been said about the mechanics of execution; instead, the composite computation is expressed as the combination of a number of underlying parts. No threads were explicitly created, nor was there any explicit communication between them—it is all implied by data dependencies.

This neatly illustrates how futures liberate the application's semantics (what is computed) from its mechanics (how it is computed). The programmer composes concurrent operations but need not specify how they are scheduled or how values are communicated. This is a good separation of concerns: application logic is not entangled with the minutiae of runtime concerns.

Futures can sometimes free programmers from having to use locks. Where data dependencies are witnessed by composition of futures, the implementation is responsible for concurrency control. Put another way, futures can be composed into a dependency graph that is executed in the manner of dataflow programming.[11] While we need to resort to explicit concurrency control from time to time, a large set of common use cases are handled directly by the use of futures.

At Twitter, we implement the machinery required to make concurrent execution with futures work in our open-source Finagle[4,5] and Util[6] libraries. These take care of mapping execution onto OS threads through a pluggable scheduler mechanism. Some teams at Twitter have used this capacity to construct scheduling strategies that better match their problem domain and its attendant trade offs. We have also used this capability to add features such as maintaining runtime statistics and Dapper-style RPC tracing to our systems, without changing any existing APIs or modifying existing user code.

Back to Top

Programming with Services and Filters

"We should have some ways of coupling programs like garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of I/O also."—Doug McIlroy

Modern server software abounds with essential complexity—that which is inherent to the problem and cannot be avoided—as well as complexity of the more self-inflicted variety (did we really need that product feature)?

Anyhow, since we can't seem to keep our applications simple, we must instead cope with their complexities.

Thus, the modern software engineering practice is centered on how to contain and manage complexity—to package it up in ways that allow us to reason about our application's behavior. In this endeavor the goal is to balance simplicity and clarity with reusability and modularity.

What's more, server software must account for the realities of distributed systems. For example, a search engine might simply omit results from a failing index shard so that it can return a partial result rather than failing the query in its entirety (as seen in the `getCount` example). In this case, the user would not usually be able to tell the difference—the search engine is still useful without a complete set of results. Applying application-level knowledge in this way is often essential to creating a truly resilient application.

Distributed applications are therefore organized around *services* and *filters*. Services represent RPC service endpoints. They are a function of type A => Future[R] (that is, a function that takes a single argument of type A and returns an R-typed future). These functions are asynchronous: they return immediately; deferred actions are captured by the returned future. Services are symmetric: RPC clients call services to dispatch RPC calls; servers implement services to handle them.

The example in Figure 3 defines a simple dictionary service that looks up a key in a map, which is stored in memory. The service takes a String argument and returns a String value, or else null if the key is not found.

Note that the service, dictionary, is again a value like any other. Once defined, it can be made available for dispatch over the network through a favorite RPC protocol. On the server, services are exported as

```
Rpc.serve(dictionary, ":8080")
```

while clients can bind and call remote instances of the service:

```
val service: Service[String, String] =
  Rpc.bind("server:8080")
val result: Future[String] =
  service ("key")
```

Observe the symmetry between client and server: both are conversing in terms of services, which are location transparent; it is an implementation detail that one is implemented locally while the other is bound remotely.

Services are easily composed using ordinary functional composition. For example, the service in Figure 4 performs a scatter-gather lookup across multiple dictionary services. The dictionary can therefore be distributed over multiple shards.

This example has a lot going on. First, each underlying service is called with the desired key, obtaining a sequence of results—all futures. `Future.collect` composes futures concurrently, turning a sequence of futures into a single one containing a sequence of the values of the (successful) completion of the constituent futures. The code then looks for the first nonnull result and returns it.

Filters are also asynchronous functions that are combined with services to modify their behavior. Their type is `(A, Service[A, R]) => Future[R]` (that is, a function with two arguments: a value of type A and a service `Service [A, R])`; the filter then returns a future of that service's return type, R. The filter's type indicates that it is responsible for satisfying a request, given a service. It can thus modify both how the request is dispatched to the service (for example, it can modify it) and how it is returned (for example, add a timeout to the returned future).

Filters are used to implement functionality such as timeouts and retries, failure-handling strategies, and authentication. Filters may be combined to form compound filters—for example, a filter that implements retry logic can be combined with a filter implementing timeouts. Finally, filters are combined with a service to create a new service with modified behavior.

Building on the previous example, Figure 5 illustrates a filter that downgrades failures from lookup services to null. This kind of behavior is often useful when constructing resilient services—it's sometimes better to return partial results than to fail altogether.

The dictionary lookup service `resilientService` implements scatter-gather lookup in a resilient fashion. The functionality was built from individual well-defined components that are composed together to create a service that behaves in a desirable way.

As with futures, combining filters and services does not change the underlying, constituent components. It creates a new service with new behavior but does not change the meaning of either underlying filter or service. This again enhances reasoning since the constituent components stand on their own; we need not reason about their interactions after they are combined.

Futures, services, and filters form the foundation upon which server software is built at Twitter. Together they support both modularity and reuse: we define applications and behaviors independently, composing them together as required. While their application is pervasive, two examples nicely illustrate their power. First, we implemented an RPC tracing system à la Dapper[10] as a set of filters, requiring no changes to application code. Second, we implemented backup requests[2] as a small, self-contained filter.

Back to Top

Conclusion

"Don't tie the hands of the implementer."—Martin Rinard

Functional programming promotes thinking about building complex behavior out of simple parts, using higher-order functions and effects to glue them together. At Twitter we have applied this line of thinking to distributed computing, structuring systems around a set of core abstractions that express asynchrony through effects and that are composable. This allows building complex systems from components with simple semantics that, preserved under composition, makes it easier to reason about the system as a whole.

This approach leads to simpler and more modular systems. These systems promote a good separation of concerns and enhance flexibility, while at the same time permitting efficient implementations.

Functional programming has thus furnished essential tools for managing the complexity that is inherent in modern software—untying the hands of the implementer.

Back to Top

Acknowledgments

Thanks to Jake Donham, Erik Meijer, Mark Compton, Terry Coatta, Steve Jenson, Kevin Oliver, Ruben Oanta, and Oliver Gould for their guidance on earlier drafts of this article. Early versions of the abstractions discussed here were designed together with Nick Kallen; numerous people at Twitter and in the open source community have since worked to improve, expand, and solidify them.

Related articles
on queue.acm.org

Scalable SQL
Michael Rys
http://queue.acm.org/detail.cfm?id=1971597

Evolution and Practice: Low-latency Distributed Applications in Finance
Andrew Brook
http://queue.acm.org/detail.cfm?id=2770868

Distributed Development Lessons Learned
Michael Turnlund
http://queue.acm.org/detail.cfm?id=966801

Back to Top

References

1. Barroso, L. A., Clidaras, J. and Hölzle, U. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture 8*, 3 (2013), 1–154.

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

3. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun, ACM 8*, 9 (Sept. 1965), 569.

4. Eriksen, M. Your server as a function. In *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*. ACM (Nov. 2013), 5.

5. Eriksen, M. and Kallen, N. Finagle, 2010; http://twitter.github.com/finagle.

6. Eriksen, M. and Kallen, N. Util, 2010; http://twitter.github.com/util/.

7. Hughes, J. Why functional programming matters. *The Computer Journal 32*, 2 (1989), 98–107.

8. Netflix. Hystrix; https://github.com/Netflix/Hystrix.

9. Ousterhout, J. Why threads are a bad idea (for most purposes). In presentation given at the Usenix Annual Technical Conference, 1996.

10. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google. 2010, 36.

11. Smolka, G. The Oz programming model. *Computer Science Today*. Jan van Leeuwen, ed. *Lecture Notes in Computer Science 1000* (1995), 324–343. Springer-Verlag, Berlin.

12. Sutter, H. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's J. 30*, 3 (2005): 202–210.

Back to Top

Author

Marius Eriksen (marius@twitter.com) is a principal engineer in Twitter's systems infrastructure group. He works on all aspects of distributed systems and server software and is currently working on data management and integration systems; he also chairs Twitter's architecture group. Twitter @marius.

Back to Top

Figures

Figure 1. Admit request only when the user is within their rate limits.

Figure 2. Degrade with a default value after timeout.

Figure 3. A simple dictionary service.

Figure 4. Scatter-gather across multiple dictionaries.

Figure 5. Scatter-gather lookup with resiliency measures.

Back to top

No entries found

Comment on this article

Signed comments submitted to this site are moderated and will appear if they are relevant to the topic and not abusive. Your

comment will appear with your username if published. View our policy on comments

(Please sign in or create an ACM Web Account to access this feature.)

Create an Account

SUBMIT FOR REVIEW