Runway
A New Tool for Distributed Systems Design

DIEGO ONGARO

Diego Ongaro is the creator of Runway and is a Lead Software Engineer on the Compute Infrastructure team at Salesforce. He is interested in improving the way people build distributed systems. He received his PhD in 2014 from Stanford University, where he worked on Raft, a consensus algorithm designed for understandability, and RAMCloud, a low-latency storage system.
dongaro@salesforce.com

We strive to build correct systems that are always on and always fast. They must be distributed, yet the complexity inherent in distributed systems poses a major design challenge. Runway is a new tool for distributed systems design, enabling interactive visualizations to help people learn about designs, and simulation and model checking to help evaluate their key properties. This article introduces Runway and discusses key issues in modeling distributed systems.

More than ever, companies are building and deploying distributed systems. They are forced to distribute computation and data across servers to improve the availability, performance, and scale of their services. Unfortunately, this comes at a steep cost of complexity:

- In a distributed system, multiple servers can operate concurrently. Their events can end up happening in orders that are hard to anticipate.
- Due to network latency, by the time a server receives a message, its contents may already be stale.
- Failures such as server crashes and network partitions are common at scale, and they can happen at any time, even while the system is trying to recover from another failure.
- Because servers are separated by a network, visibility into running systems is reduced, and debugging environments are limited.

The best way to manage this complexity is to focus more efforts on system design. In the design phase, we should communicate clearly about a design and also evaluate that design's key properties, such as its understandability and simplicity, correctness, availability, performance, and scalability. Exploring and resolving design issues early, before investing heavily in implementation, should help lower the cost of developing distributed systems and improve their quality.

Many existing tools aim to help with specifying, checking, or simulating distributed system models (some are listed on the Runway wiki [1]). However, none of these seems to be widely used for designing distributed systems in industry. Instead, industry engineers still rely on primitive tools like whiteboards, back-of-the-envelope calculations, and design documents. These are valuable, but they fall short of communicating clearly about a design or evaluating its important properties. Why don't industry engineers use more sophisticated design tools? We can only assume that they are unwilling, existing tools are impractical, or the engineers haven't found the right tools. If it's the former, there is little hope. But if it's the latter two, Runway might have a chance.

Runway is a new design tool for distributed systems. It's not technically superior to existing tools, but it may be better optimized for a chance at widespread adoption in industry. There are three primary reasons for this:

1. **Integration:** Runway combines specification, model checking, simulation, and visualization in the same tool. Integrating many components might tip the cost-benefit calculation in Runway's favor: you can write one system model and get a lot of value from it. Compared to using separate tools, Runway has only a single learning curve. Plus, you can start by specifying and visualizing a model, then decide how to evaluate it later (using model checking, simulation, or both).
2. **Usability:** Runway aims to be approachable, with only a small learning curve. The interactive visualizations allow people with no special knowledge of Runway to learn about a design. For modeling, Runway's specification language is designed to be familiar to most engineers and encourages simple code without many abstractions.
3. **Social:** Runway visualizations run in a Web browser, enabling people to share their models easily. We're currently designing a registry to help people discover other models, as well as a component system to enable using one model within another. We hope a community will grow around modeling systems in Runway and learning about them.

Although Runway is still early in its development, it can already provide significant value. A public instance of Runway is available at https://runway.systems/, and its source code [2] is freely available under the MIT license.

Overview of Runway

A Runway model consists of a specification and a view. The specification describes the model's state and how that state may change over time. Specifications are written in code using a new domain-specific language. This language aims to be familiar to programmers and have simple semantics, while expressing concurrency in a way suited for formal and informal reasoning. A specification describes a *labeled transition system*, which is like a state machine, for how state changes. It can also include *invariants*, properties that must hold for every correct state. The view draws a model's state visually. For example, the view for the Runway model of the Raft consensus algorithm [3] is shown in Figure 1.

Runway includes a compiler for its specification language, and it can execute the specification using a randomized simulator. This produces an *execution*, an ordered history or schedule of events that captures a sequence of state changes. Runway can then visualize or animate these state changes over time. Runway employs the model's view for the main component of the visualization and also adds several generic widgets, including a timeline, an editable table of the model's entire state, and a toolbar of transition rules that can be applied to the state. The visualization is interactive, allowing users to manipulate the state of a model and see how it reacts. It also serves as a great debugger when developing specifications.

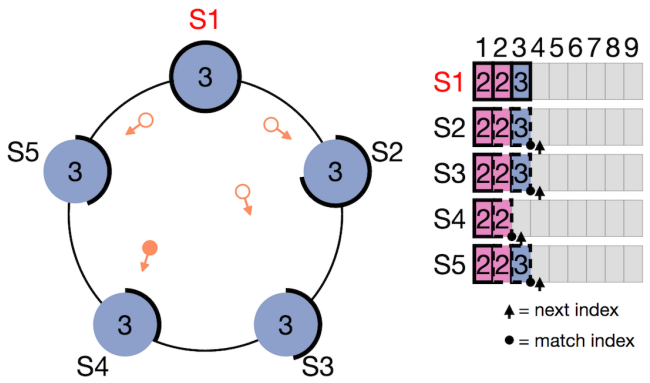

Figure 1: The view of the Raft model. The ring on the left is optimized for understanding leader election. Each server has a randomized election timer, drawn as an arc around the server. Heartbeat messages from the leader reset that timer, and a server begins an election when its timer expires. The servers' logs on the right are optimized for understanding log replication; they are lined up in tidy rows for easy comparisons. This interactive visualization is available at https://runway.systems/.

The simulator can do more than power a visualization: it can also collect data. A single execution can include many interesting data points, and as a planned feature, data could be aggregated across a family of executions. The data can be presented in the form of graphs, and by selecting a point on a graph, the visualization can load and replay the exact event of interest.

The final major component of Runway is the model checker, which can verify that a model will never break an invariant, up to some limit in model size. The model checker begins at the model's starting state and tries to explore all reachable states, evaluating the invariants at each step. It never expands the same state twice, using a hash table to track the states it's already visited. Runway's model checker today is quite slow; we plan to either implement optimizations from the literature or to have Runway invoke an efficient model checker behind-the-scenes. If the model checker finds a bug, it can produce an execution showing how the model reaches a bad state. As a planned feature, this execution could be loaded into the visualization so that a user could easily understand what went wrong.

Runway's Specification Language

Although Runway's specification language is still a work in progress, several basic principles are shaping its design:

- It aims to be easy for industry developers to read and write, with only a small learning curve. Although a functional approach is possible, an imperative, procedural approach is likely to be more familiar.
- It intentionally includes a limited set of language features, favoring specifications with straightforward code, even at the cost of larger specification sizes.

- Its strong type system is intended to help developers avoid silly errors like typos and misused variables.
- It permits modeling concurrency without writing concurrent code. Transition rules are applied atomically, one at a time. To model concurrency, one rule can model the start of a long-lived, concurrent operation, and another can model its completion.
- It keeps all state explicitly in global variables, which simplifies reasoning about the current state. This is in contrast with general-purpose languages, which use the instruction pointer to track information, without giving it a name.
- Although Runway does not yet include an efficient implementation, it must be possible to evaluate Runway models efficiently, and, especially for model checking, their state must be efficient to copy and hash. We hope to store the global state variables contiguously in memory even as the language evolves, although this may need to be relaxed in favor of more flexible models (every variable has a static upper bound on its size today).

The Too Many Bananas problem serves as a good example to illustrate Runway's specification language. It is a simple concurrency problem similar to those taught in introductory systems classes. You live in a house with roommates, and everyone likes to eat bananas. When you run out of bananas, you go to the store to buy more and bring those home. Due to a race condition, it's possible for your roommate to leave for the store while you're already out buying more bananas. When you both return home, you might end up with too many bananas, a critical problem since bananas spoil over time.

This specification models the Too Many Bananas problem:

```
01 var bananas : 0..100;
02 type Person : either {
03   Happy,
04   Hungry,
05   GoingToStore,
06   ReturningFromStore {
07     carrying: 0..8
08   }
09 };
10 var roommates: Array<Person>[1..5];
11 rule step for person in roommates {
12   match person {
13     Happy {
14       person = Hungry;
15     }
16     Hungry {
17       if bananas == 0 {
18         person = GoingToStore;
19       } else {
20         bananas -= 1;
21         person = Happy;
22       }
23     }
24     GoingToStore {
25       person = ReturningFromStore {
26         carrying: urandomRange(0, 8)
27       };
28     }
29     ReturningFromStore(bag) {
30       bananas += bag.carrying;
31       person = Hungry;
32     }
33   }
34 }
35 invariant BananaLimit {
36   assert bananas <= 8;
37 }
```

For the purpose of this model, it's never OK to have more than eight bananas at home. This is checked by the invariant on lines 35–37. More sophisticated models could factor in a rate of decay and rate of consumption, but let's start simple.

Lines 1–10 declare two variables: "bananas" is the number of bananas at home, and "roommates" represents the five people who live there, each of whom is in one of various possible states at any given time. By default, Runway initializes variables to the upper-left possible value, so "bananas" starts at 0 and each person starts out "Happy."

Lines 11–34 declare a state transition rule named "step," which applies to one roommate at a time. If that person is "Happy," they can become "Hungry" (lines 13–15). If they are "Hungry" and a banana is available, they can eat a banana and become "Happy"; if no banana is available, they can go to the store (lines 16–23). If they're going to the store, they can return from the store with a random number of bananas between 0 and 8 (bunches vary in size, and sometimes the store has run out; lines 24–28). And if they are coming back from the store, they can leave the bananas they've purchased at home and return to being "Hungry," where they are likely to eat a banana soon (lines 29–32).

Note that specifications define which state transitions may happen, but they do not say when they should happen or in what order: that's up to the simulator or the model checker. If the specification permits multiple roommates to take a step from a given state, Runway's simulator will pick one at random. Alternatively, Runway's model checker will explore all possibilities, looking for any state that violates the invariant. To use the model checker with this specification, replace the random number on line 26 with a constant.

Inside Runway Views

Runway relies on a model's view to draw the main component of the visualization. Views are built using off-the-shelf Web technologies, so that Runway visualizations can run in a Web browser (the ubiquitous graphical toolkit). Although views are not necessarily constrained to these technologies, we're currently using JavaScript, SVG, and D3.js [4]:

- JavaScript is the scripting language running in every Web browser.
- SVG, Scalable Vector Graphics, is analogous to HTML but used for images instead of text and layout. Just like HTML, SVG is styled with CSS to assign properties like colors and borders.
- D3.js is a JavaScript library that assists with drawing SVG.

At a first approximation, specification and views tend to be similar in size. However, they are very different in nature. A view serves the necessary function of drawing the model's state, and its implementation tends to be uninteresting. You might study a specification to learn about the precise workings of a design, but the only thing you can learn from a view's code is how it draws the state.

Modeling Distributed Systems

Beyond simple banana problems, Runway can be used to model concurrent and distributed systems. Modeling distributed systems, in particular, introduces a new set of challenges. This section describes Runway's approach to modeling failures, networks, and clocks in distributed systems, as well as using invariants and assertions effectively to check properties of distributed system models.

Failures

In most distributed systems, servers can fail, and, down to some limit, the system should remain available. Messages can be delayed and perhaps dropped and reordered. These failures can be extremely important to understanding and evaluating designs, but different systems make different assumptions about their environments and have different requirements. In Runway, failures are encoded the same way as normal events, permitting specifications to model their own assumptions. A server crashing is modeled the same as a client submitting a request.

However, transition rules representing failures and client requests are different from normal transition rules in one regard: typically, they should not be applied all the time. For example, not every message should be dropped, and client requests should arrive at a limited rate. Currently, the specification can limit the rates of these events by imposing additional conditions on them, using random values. For example, when a message is sent, the specification can compute whether or when it will be dropped based on a coin toss. However, this need is recurring and fundamental to modeling, so we're exploring ways to express these event rates intuitively and conveniently in Runway.

Networks

Modeling a distributed system also requires modeling a network. This, too, can be done in Runway using normal state variables and transition rules. For example, the basic Raft model has a flat network modeled as a set. When a server sends a message, the message is added to the set. When a server receives a message, it is removed from the set.

Visualizing a network introduces its own challenge. For example, the Raft view draws each message as it moves from the sender to the recipient. To calculate the position of a message, it needs to know when the message will be received, but that information isn't normally available ahead of time. The Raft model currently takes a simple approach: the specification assigns each message a randomized delay when it is sent and will not deliver the message before then. An alternative, more complex approach would be to delay the visualization until the message's future delivery time had been determined by the simulator; we will implement this in Runway only if the simpler approach is found to be insufficient for common use cases.

In principle, more complex networks with links, switches/routers, and propagation and queueing delays can be modeled the same as simple networks, using variables and rules. However, as the network's wiring complexity increases, it would be tedious to express the wiring in Runway today, and we may explore additional language features to make this more convenient.

Runway also needs a way to import reusable components. This would be useful at various levels of scale, including:

- Choosing from several network models to load into a distributed system model,
- Loading a model of, for example, a coordination service into a model of a larger system, and
- Loading larger system models together into a model of an entire cluster's workload.

We are currently designing the language features to enable this.

Time and Clocks

Runway supports two modes of operation: *synchronous*, which generates events over time, and *asynchronous*, which generates only an ordered sequence of events. These two modes of operation have been useful for different models. For example, the basic Too Many Bananas model is asynchronous, while the model of a building's elevator system is primarily interesting to measure delays in synchronous mode.

The two modes can also be useful for the same model. Many algorithms are designed to maintain safety properties under asynchronous assumptions, making them robust to erroneous clocks and unexpected delays (typically, even a "small" race condition is not acceptable). With Runway, it's possible to

check these properties asynchronously with the model checker and still run timing-based simulations on the same model. For example, we can check that Raft always keeps committed log entries no matter how long communication steps take, then use the same model in synchronous mode to estimate leader election times on a datacenter network.

In Runway, a timer is typically modeled by storing the time when an action should be taken, then guarding a transition rule for the action with an if-statement:

```
rule fireTimer {
  if past(timeoutAt) {
    /* take action, reset timeoutAt */
  }
}
```

This is another example of making all state explicit in Runway. There is no question of whether or not a timer has been set.

When running in synchronous mode, Runway keeps a global clock for the simulation, and "past()" evaluates to true if the given timestamp is earlier than the simulation's clock. In asynchronous mode, however, "past()" always returns true. This has the effect of making all timers fireable immediately after they are scheduled, allowing unlikely schedules to be explored.

We have only tried Runway's current approach to clocks on a handful of models, and it may need further enhancements. Specifically, it may be burdensome to model clock drift across servers using one global clock, as Runway provides today. We plan to revisit this issue based on actual use cases.

Access Restrictions and Distributed Invariants

Runway expects you to follow two ground rules in modeling distributed systems, but to keep the specification language simple, it does not enforce these rules. First, one server should not access another server's internal state. Second, the only shared state should be the network, which should only be accessed in limited ways (such as following send/receive semantics). It would be impossible to implement a real distributed system that violated these rules.

However, accessing unshared state is OK for assertions and invariants. In fact, it's a key advantage to modeling an entire distributed system in a single process. For example, in Raft there should be at most one leader per term. This is easy to check in an invariant by directly accessing and comparing all the servers' states. The alternative, to check this property by exchanging messages as in a truly distributed system, would be much more complex, would be less effective due to message delays, and could interfere with the normal operation of the model.

Defining *history variables* as shared global state is also OK. History variables record information about the past. These variables should not affect the normal execution of the model, but they may be read by assertions and invariants. For example, Raft's property that there is at most one leader per term should actually hold across time. If one server was leader in a particular term, no other server should ever become leader in that term. The Raft model tracks past leaders using a history variable, and when a server becomes leader in some term, it asserts that that term has not yet had a leader:

```
var electionsWon : Array<Boolean>[Term];
rule becomeLeader for server in servers {
  if (/* this candidate has a majority of votes */) {
    assert !electionsWon[server.term];
    electionsWon[server.term] = True;
    /* update local state to become leader */
  }
}
```

Conclusion

Distributed systems are challenging, and their complexity justifies careful design. Using the proper tools, we could be communicating clearly and evaluating our designs thoroughly, even before investing in their implementation. However, existing design tools have not been adopted widely in industry.

Runway hopes to change that. It combines specification, model checking, simulation, and interactive visualization into one tool. This improves Runway's potential benefit without significantly increasing the cost of developing a model. Runway aims to be easy to learn by using a specification language based on imperative, procedural code that discourages unnecessary abstractions. Runway models are also easily shareable on the Web, so others can learn about designs through interactive visualization, even if they have not learned how to read Runway specifications.

At Salesforce, we are redesigning our infrastructure for the next order of scale, and we've already been applying Runway to a few design challenges internally. We found Runway to be effective for concurrent problems as well as distributed ones, and, encouragingly, engineers seem to find value early in specifying their designs more formally and in watching them run.

Runway is open source [2] and still in the early stages of its development. We have made it available early to find out whether other engineers will adopt it, and if not, to learn what is stopping them. We hope you will join us in forming a community around Runway.

Acknowledgments
Thanks to Rik Farrow, Pat Helland, Jennifer Wolochow, and Nat Wyatt (Runway's first user) for their helpful comments on earlier versions of this article, and to David Leon Gil, Steve Sandke, and many others for helping design Runway itself.

References

[1] Runway Wiki, Related Work: https://github.com/salesforce/runway-browser/wiki/Related-Work.

[2] Runway source code: https://github.com/salesforce/runway-browser.

[3] Raft Consensus Algorithm: https://raft.github.io.

[4] D3: Data Driven Documents, JavaScript library: https://d3js.org.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Do you have a USENIX Representative on your university or college campus? If not, USENIX is interested in having one!

The USENIX Campus Rep Program is a network of representatives at campuses around the world who provide Association information to students, and encourage student involvement in USENIX. This is a volunteer program, for which USENIX is always looking for academics to participate. The program is designed for faculty or staff who directly interact with students. We fund one representative from a campus at a time.

A campus rep's responsibilities include:

- Maintaining a library (online and in print) of USENIX publications at your university for student use
- Distributing calls for papers and upcoming event brochures, and re-distributing informational emails from USENIX
- Encouraging students to apply for travel grants to conferences
- Providing students who wish to join USENIX with information and applications
- Helping students to submit research papers to relevant USENIX conferences
- Providing USENIX with feedback and suggestions on how the organization can better serve students

In return for being our "eyes and ears" on campus, the Campus Representative receives access to the members-only areas of the USENIX Web site, free conference registration once a year (after one full year of service as a Campus Representative), and electronic conference proceedings for downloading onto your campus server so that all students, staff, and faculty have access.

To qualify as a campus representative, you must:

- Be full-time faculty or staff at a four-year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

For more information about our Student Programs, please contact office@usenix.org