# The Dog Programming Language

**Salman Ahmad**
Massachusetts Institute of Technology
saahmad@mit.edu

**Sepandar D. Kamvar**
Massachusetts Institute of Technology
sdkamvar@mit.edu

## ABSTRACT

Today, most popular software applications are deployed in the cloud, interact with many users, and run on multiple platforms from Web browsers to mobile operating systems. While these applications confer a number of benefits to their users, building them brings many challenges: manually managing state between asynchronous user actions, creating and maintaining separate code bases for each desired client platform and gracefully scaling to handle a large number of concurrent users. Dog is a new programming language that provides an elegant solution to these challenges and others through a unique runtime model. The Dog runtime makes it possible to model scalable cross-client applications as an imperative control-flow — drastically simplifying many development tasks. In this paper we describe the key features of Dog and show its utility through several applications that are difficult and time-consuming to write in existing languages, but are simple and easily written in Dog in a few lines of code.

### Author Keywords

Programming Languages; Application Development

### ACM Classification Keywords

D.3.3. Programming Languages: Language Constructs and Features; H.5.0.Information Interfaces and Presentation

## INTRODUCTION

In the last few years, we have seen an unprecedented growth of the Internet and of mobile devices. Over two billion people now use broadband Internet, up from 50 million a decade ago [3]. And just six years after the original iPhone was introduced, half of Americans own a smartphone [29], and over a quarter of Americans use their phone as their primary Internet device [28]. Consequently, many modern software applications are deployed in the cloud, interact with many users, and run on multiple client platforms such as Web browsers, iOS, and Android.

Unfortunately, these applications are difficult to design, build, and maintain [19]. The interactive applications enabled by modern Javascript and mobile operating systems are inherently stateful, and running them over a stateless protocol like HTTP requires developers to resort to various tricks to manage state between asynchronous web requests. Some of these tricks include: sessions, cookies, and most prominently, mapping application logic onto a persistent data model. This manual bookkeeping drastically increases the complexity of otherwise straightforward applications.

```
define chat_with: user as: username do
  forever do
    listen to user for messages
    on message do
      message = username + ": " + message
      notify: "chatroom" of: message
    end
  end
end

define pick_username_for: user do
  listen to user for usernames
  on username do
    return username
  end
end

listen to anyone for entrances
on each entrance do
  user = person from entrance
  username = pick_username_for: user
  chat_with: user as: username
end
```

**Figure 1. Dog makes it easy to build social applications. This example shows how to create a real-time chatroom application.**

For mobile applications in particular, much of the interaction code ends up being shifted to the client. This further obscures an application's underlying logic, which is now split between the server and the client, and requires new code to be written for each client platform.

The consequence of all of this is that programs that are conceptually simple and easily described even by the nonprogrammer often require sophisticated architectures, the use of advanced programming techniques, and careful planning to implement in practice. This is particularly true for those programs (and parts of programs) that encode interaction flows and social processes.

We developed Dog to address the disconnect between the needs of these programs and the capabilities of current programming systems. Dog is a new dynamically-typed procedural programming language that simplifies creating interactive, multi-user, and cross-client software applications through two key contributions:

Submitted for review.

1. Dog allows server-side code to be written as an imperative control flow that spans multiple requests and shields developers from HTTP's statelessness.

2. Dog facilitates cross-client development by allowing server-side logic to control client-side interactions through a scripting protocol that minimizes code duplication while enabling developers to take advantage of platform-specific UI affordances.

Together, these two features allow programs that take hundreds of lines of code in current languages to be written in just a few lines of Dog code, without sacrificing performance or scalability. To achieve this, we built a unique runtime environment for Dog that incorporates three core features:

1. An asynchronous execution model that allows complex event-driven code to be written in a structured and imperative manner.

2. A concurrency model that features lightweight threading constructs called tasks, which are managed by the runtime, enforce immutable memory semantics, and are backed by persistent continuations so they can pause indefinitely.

3. A distributed runtime model that allows a program's execution to be run across multiple machines.

We begin this paper by describing the two core principles behind Dog: Workflow-Centric Programming and Server-Client Scripting. We then provide an overview of Dog's syntax, and demonstrate Dog's utility through several diverse applications. Then, we describe the details of the Dog runtime and provide a technical overview of its reference implementation, which compiles to JVM bytecode and is available under an Apache 2.0 license. Lastly, we present a user study evaluating Dog's usability in addition to benchmark tests of its runtime performance.

## WORKFLOW-CENTRIC PROGRAMMING

### Conventional Approach: Manual State Management

Conventional cloud-deployed applications break up logic across multiple asynchronous URL handlers. Developers specify server-side logic that is executed in response to an incoming URL request. For example, PHP associates logic with a URL by placing a PHP script at a matching file system path while Java Servlets and Ruby Rack-based applications (including the popular Rails framework) use a more sophisticated routing mechanism that pairs a URL pattern with a handler object. We refer to this style of programming as *resource-centric programming* (Figure 2b).

This style of server-side programming plays a specific and limited role: it inspects incoming HTTP requests and produces an HTTP response. Multistep logic that spans multiple requests needs to manually store state so that the relevant information is available during a future request-response cycle (Figure 2b shows an example using sessions). This manual bookkeeping complicates many development tasks, for example, ensuring confirmation before changing a user's email address, choosing a different billing method before making an online purchase, or viewing the next page from paginated results.

(a) Pseudocode

```
Allow a user to try to login. If the password does not
match, then give them 3 more chances before suggesting
that they submit a 'Forgotten Password' request.
```

(b) Resource-Centric Code (Ruby + Sinatra)

```ruby
get "/login" do
 session.delete("login_tries")
 render "login_form.html"
end

post "/login" do
  user = params["user"]; password = params["pass"]
  if authenticate(user, password) then
    session["user"] = user
    session.delete("login_tries")
    redirect_to("/dashboard")
  else
    if session["login_tries"] == nil then
      session["login_tries"] = 0
    end
    session["login_tries"] += 1

    if session["login_tries"] >= 3 then
      redirect_to("/forgot_password")
    else
      flash["message"] = "Wrong password. Please try again."
      render "login_form.html"
    end
  end
end

```

(c) Workflow-Centric Code (Dog)

```
listen to everyone for logins

on each login do
  user = person from login
  repeat 3 times
    if user: login.user has_password: login.pass then
      signin: login.user
      show_dashboard_for: user
    else
      message = "Wrong password. Try again."
      show message to user
      listen to user for logins
      on login do
        user = person from login
      end
    end
  end
  reset_password_for:user
end
```

**Figure 2. Server-side code to implement login logic. Workflow-centric programming allows for code to be written as a straightforward imperative control flow. State is manage implicitly by the program's execution using familiar techniques like control structures and local variables. In resource-centric programming, state between individual server requests is manually preserved by storing data in the session.**

### Dog: Automatic State Management

Instead of writing code as a series of URL endpoints, Dog applications are written as imperative control flows that directly represent the user interaction flow across multiple request-response cycles. Developers rely on familiar state management features like local variables, condition statements, loops, and functions. State is implicitly captured by the program's execution and developers are afforded compilation warnings, stack-based debugging, well-defined function APIs, as well as easier ways to write test cases. We refer to

this style of programming as *workflow-centric programming* (Figure 2c).

Workflow-centric programming is a natural and familiar way to write applications and allows developers to focus on the interactions they want to create rather than low-level technical details. In fact, many designers and programmers already use workflows as a prototyping tool to describe an application's intended behavior [17]. To confirm our intuition that workflow-centric programming is a good fit for Web programming, we ran a simple study asking participants which of two pseudocode implementations of a popular online service they preferred. One implementation was written using a workflow-centric programming model and the other was resource-centric. The service described in the study included YouTube, ZipCar, and Tumblr. Participants overwhelmingly preferred the workflow-centric model by a margin of 21 to 2.

Operationally, workflow programming is similar to console-based programming. Instead of calling `print` and `readLine` to send and receive data from a terminal, Dog developers use Dog's built-in `show` and `listen` commands to send and receive data to a Web-accessible API. The `show` command (Figure 2c line 11) makes a particular variable visible to the client, and the `listen` command (Figure 2c lines 1 and 12) opens a "channel" that clients can use to send data back into the program.

A program can open multiple channels and selectively wait until information is sent using the `on` block (Figure 2c line 13). Execution will stop when it reaches an `on` block and wait for a message to be sent over the specified channel. In certain cases, it is convenient to delegate work to a handler to avoid complicating the application's primary control flow. To address this, Dog includes the `on each` block (Figure 2c line 3), which "forks" the program when data arrives over a channel.

Authentication and security are built into `listen` and `show`. Both commands accept a predicate[1] that designates which users can view the shown data or send input on the listen's channel (Figure 2c line 1 shows an example where anyone is allowed to login). By default, a client is assumed to be an anonymous user. Users are logged into the system when the application calls the `signin:` function (Figure 2c line 7).

Dog's workflow-centric programming model provides a structured approach for managing state that helps to elucidate application logic and reduce informal state management conventions that require additional code. The resulting code is not only more writable, but also more understandable.

## SERVER-CLIENT SCRIPTING

With the rise of mobile OSes, developers are increasingly deploying native client applications on multiple platforms. While most of the client applications are often similar, they currently need to be reimplemented for each platform.

With Dog, an application's high-level logic is written server-side rather than client-side. Clients connect to the server and

[1]Predicates are discussed later in the syntax section.

```
1 <script id="login" type="text/x-dog-template">
2   <h3>Please Login</h3>
3   {{#listenForm listens.login}}
4     <input type="text" name="login">
5     <input type="password" name="pass">
6     <input type="submit" name="submit" value="Log In">
7   {{/listenForm}}
8 </script>
```

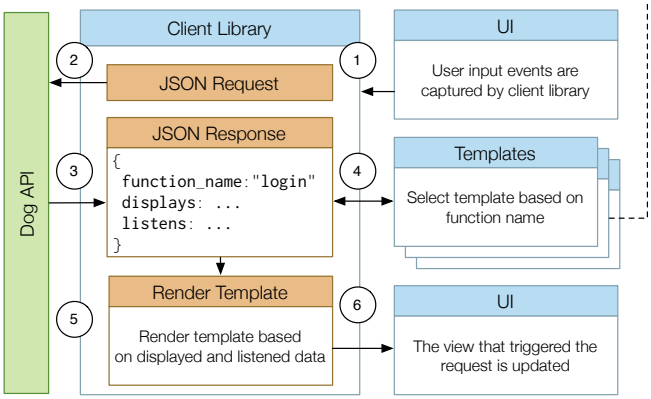


**Figure 3. Dog's client-side libraries are built around templates. Here, we show the Javascript client library. Developers create templates using Handlebars.js but other templating languages are supported as well.**

the server "tells" the client what they should do and when they should do it. Communication between the client and the server is done through a server-client scripting protocol that exposes what a program has `shown` and `listened` for.

Dog's model provides a clean separation between back-end business logic and front-end design. The client has complete control over the application's aesthetics and is free to leverage platform-specific UI features and design conventions. However, the client is not burdened with reimplementing the application's high level interaction and navigational flows. We felt that this was a sensible separation of concerns that reduces duplicated code while providing engaging experiences on a variety of platforms.

Server-client scripting is also a key differentiating feature from Web frameworks like Seaside, Tir, and Cocoon that use continuation passing style for processing requests [23, 26, 30]. These frameworks provide a similar model to workflow-centric programming but return HTML intended only for Web browsers. Furthermore, browsers can only send data back to the server through special form tags that force the entire screen to be re-rendered.

In terms of its implementation, the server-client-scripting protocol is built on top of a JSON-based RESTful API. The API allows clients to inspect the runtime state of any *task*[2] in a Dog program that is waiting for user input. Tasks are identified by a special 128-bit *id*, but if the client is connecting for the first time, it asks the API for a special root task, which corresponds to the top-level scope in the Dog program. The server responds with the name of the current function that the task is waiting in, the data that the function wants the client to show (specified by `show`) and the inputs that the

[2]Tasks are a special concurrency construct in Dog and are discussed in the Runtime section.

function wants back from the user (specified by `listen`). When the client sends input to the API, the API passes that data into the task and continues its execution until the task waits again. The API then returns the new state of the task to the client, which likely contains a new set of `shows` and `listens` (Figure 3).

To further simplify development, Dog provides a series of client libraries to automatically talk to the API. The official client libraries included in the Dog distribution are available for Web browsers (Javascript), iPhone, and Android. These libraries are primarily template based. Whenever the client library gets a response, it inspects the "function_name" field (Figure 3) and automatically renders an associated template. If no template is available, the client library will autogenerate one based on the information that has been `shown` and `listened`. Thus, client-side development only requires developers to create a template for the functions they define in their program. Of course, developers are not restricted to only using templates and can opt to use the API directly. However, this is considered an advanced use case.

Server-client scripting is primarily intended for interactive portions of Web applications. Browser-based Dog applications are typically "single-page apps" - they redraw themselves by updating their DOM with Javascript instead of navigating to new pages. This does complicate certain tasks like server-side cacheing and search engine visibility.

## SYNTAX

### Overview

Our goal in designing Dog's syntax was to make it possible to write code that is understandable even to a non-programmer who has never seen Dog. In practice, this meant designing a syntax that allows and encourages English-like readability [21], minimizes intimidating punctuation [20] and includes built-in libraries and language constructs that are already easy-to-read.

A hallmark feature of Dog's syntax is its use of named parameters with functions, a feature popularized by Smalltalk [7]. The definition of the "substring" function in Dog looks like:

```
define substring: str from: start to: finish do
  ...
end
```

The argument names are actually part of the function name itself. Thus, the function above is actually called `substring:from:to:`. This allows the definition of another substring function with different semantics in an unambiguous way. For example, it is possible to have another substring function called `substring:from:length:`.

Calling a function requires specifying all of the arguments:

```
hello = substring: "Hello, World" from: 0 to: 5
```

While this syntax may seem verbose, it is designed to be understandable. Other languages may use:

```
substring("Hello, World", 0, 5)
```

In this case, it is unclear if the "5" is the length of the substring or the ending syntax. Dog's function invocation is unambiguous.

Dog's data model has four built in primitive types: nulls, booleans, numbers, and strings. Values can be assigned to variables using the equals operator. Primitives can be grouped into a composite data type called structures. Structures are dynamic containers for key-value pairs. They can be nested and have either strings or numbers as keys. A structure's contents can be accessed using both the "dot" and "bracket" operators (similar to Javascript objects). Dog does not have an explicit array data type, however, since the keys of a structure can be numbers, structures can be used to store array information.

```
my_struct = {
  string = "String Value"
  number = 3.14159
}

my_struct.string == my_struct["string"] # true
```

All values in Dog are immutable and there are no shared variable references in the language. Consequently, the only data a Dog function can access are its arguments and all arguments follow "pass-by-value" semantics. This is an important feature of Dog that supports concurrent programming and is discussed at length later in the paper.

Almost every Web application incorporates the use of a database of some sort [24, 6, 9, 19]. While Dog eliminates shared variable references, modeling applications around a centralized data store is common in practice. Therefore, as an added convenience, Dog features "collections". Collections are globally scoped containers that hold structures. They can be thought of as a database but do not require pre-defined columns and can hold structures of any structure type. Since there are no shared *variable* references, a value in a collection does not change until the program explicitly re-saves it.

The example below shows the basic usage for collections in Dog:

```
define collection games

sonic = {
  title = "Sonic The Hedgehog"
  platform = "Genesis"
}

insert: sonic into: games
```

To query collections, Dog has built-in syntactic support for SQL-like predicates similar to LINQ [15]. Predicates in Dog are created using the `where` keyword. When a predicate is paired with a collection identifier, it is compiled as a `query` that can be passed to the built-in `find:` function to retrieve matching structures:

```
sega_games = find: games where platform == "Genesis"
```

Built-in support for predicates allows some of the most common operations in Web applications to be written in a manner that is easily readable. In particular, predicates are used

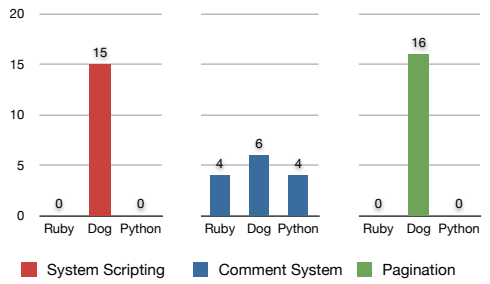


**Figure 4. Results from asking users to identify the code they prefer for solving different tasks.**

heavily with the `listen` command. The following snippet opens a channel that is only visible to users that have a their "teacher" property set to true:

```
listen to people where teacher == true for assignments
```

The `listen` command accepts a predicate a query into the `people` collection, a special system managed collection that stores all user profiles. When a client submits a request, the API ensures the client is authenticated on behalf of a user that is matched by that query. To accept all users, developers can use `everyone` instead of a query.

Dog's core language constructs and named-parameter function invocation syntax allow developers to write English-like programs that reduce code ambiguity to simplify development. While much of a language's ease-of-use is out of the hands of the designers and depends on the authors of libraries, by emphasizing understandability in the standard libraries we hope to cultivate a community that values readability and intuitive API design.

### Syntax Evaluation

To evaluate our syntax decisions, in particular our choice to use named parameters we ran a small study that compared programs written in Dog to the same program written in either Ruby or Python. The programs were explained in plain English and participants were asked which implementation they preferred and if they had any prior experience with programming. Three programs were used during this experiment. The first was a portion of a Web application that handled pagination, an example that lends itself to workflow-centric programming. The second was a commenting system that allowed users to post comments on an online board, an example that lends itself to resource-centric programming. The third was a system-scripting program that sequentially renamed all JPG files in a directory, an example of something that Dog was explicitly not designed for. The Ruby and Python implementations were written using best practices, widely accepted idioms, and relevant frameworks. Participants preferred the Dog implementations by a wide margin (Figure 4). Participants with little to no programming experience frequently commented that the English-like language constructs (like `on` and `listen`) as well as the "long but descriptive" function names were the reason for their preference of Dog. On the flip side, participants with programming experience, frequently commented favorably about how Dog programs exhibit little "hidden state" and that they didn't need to guess the purpose of particular function calls.

## USING DOG

In this section, we present a series of applications to demonstrate the usefulness of Dog and its programming model.

### Real-Time Messaging: GroupChat

GroupChat is a real-time chatroom application. The program begins by `listening` for users to enter a chatroom:

```
define chat_with: user do
  ...
end

listen to anyone for entrances
on each entrance do
  user = person from entrance
  chat_with: user
end
```

Once the user enters, we use the `person from` command to get the current user and then call the `chat_with:` function that loops forever listening for messages from that user:

```
define chat_with: user do
  forever do
    listen to user for messages
    on message do
      notify: "chatroom" of: message
    end
  end
end
```

Notice the use of `on` instead of `on each` when waiting for messages. When waiting for entrances, we used `on each` because we wanted to track each user independently of one another. Whereas, in this case, we used `on` because we wanted to continue execution with the current user.

Once we receive a message, we will want to broadcast it to all other users in the chatroom. Real-time applications like this call for "push" networking rather than HTTP's typical "pull" request-response cycle. As a solution to this common and difficult problem, Dog provides a built-in notification library that allows the server to push messages to all subscribed clients. All the official client libraries fully support push notifications either through a background network connection or by polling the server. The call to `notify:of:` will push the message to all clients subscribed to the "chatroom" channel.

To build a browser-based client for GroupChat, we use dog.js, the Javascript client library. Dog.js, like the other client libraries, is template-driven — all we need to do is create templates that the library will render automatically. The UI is specified in an `index.html` file that is placed in a directory called `public` that is located along side the Dog program. The Dog runtime will automatically serve any files inside the `public` directory, including resources like images and stylesheets. `index.html` is shown below:

```
<html>
  <script src="/dog/dog.js" />
  <script id="root" type="text/x-dog-template">
    <h2>Welcome to GroupChat</h2>
    {{#listenLink listens.entrances}}Enter{{/listenForm}}
  </script>
```

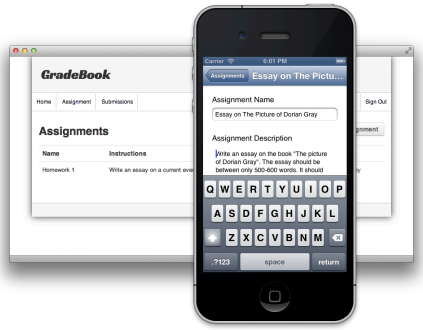


**Figure 5. GradeBook is a peer assessment application written in Dog. Dog's cross-client development model allows server-side code to drive multiple client side applications.**

```
  <script id="chat_with:" type="text/x-dog-template"
    data-notifications="chatroom">
    <ul>
      {{#each notifications.messages}}
        <li>{{this}}</li>
      {{/each}}
    </ul>
    {{#listenForm listens.messages}}
      <input type="text" name="message">
      <button type="button">Send</button>
    {{/listenForm}}
  </script>

  <div data-dog-viewport="true" />
</html>
```

Dog.js uses Handlebars.js, a logic-less template system, for rendering templates [13]. Templates are contained inside `script` tags with a content type of `text/x-dog-template`. The template has access to the variables that have been `shown` and `listened`. Templates are defined on a per-function basis. Thus, in this case, we have two: one for the root function and another for the `chat_with:` function. Templates are associated with a function using their `id` attribute. The `chat_with:` template is decorated with the `data-notifications` attribute, which tells the client library to re-render the template whenever a "chatroom" notification is received. Templates are rendered and placed inside "viewports". When a template sends a request to a server-side listen, the server's response is rendered by a new template and replaces the viewport that sent the original request. In most application, only a single viewport is needed, as is the case with GroupChat. The full source code is shown in Figure 1.

### Native and Browser Development: GradeBook

GradeBook is a peer assessment application. When a student submits an assignment, the application will require another student to grade the submission. Once the grader is done, the assignment is forwarded to the professor who can take the grader's feedback into consideration before providing the final grade. The main application logic is shown below. Some of the application's code is abstracted into functions that are not shown.

```
listen to people where role == "student" for assignments
```

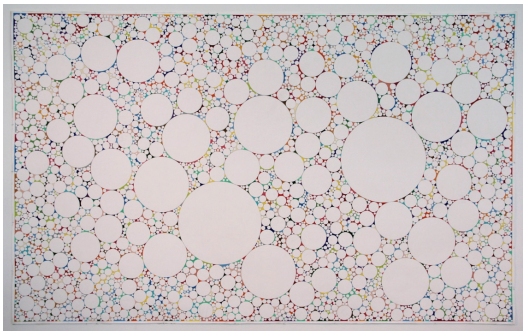

**Figure 6. Tangent is a socially created art installation where a Dog program coordinated the activities of many individual contributors. Zoom for detail. Graphite on canvas. 8ft. × 5ft.**

```
on each assignment do
  submitter = person from assignment
  if user:submitter already_submitted:assignment then
    error = "You may only submit your assignment once..."
    show error to submitter
  else
    grades = ask_peers_of:submitter to_grade:assignment
    teacher = people where role == "teacher"
    final_grade = ask:teacher to_grade:assignment using:grades
    assignment.grade = final_grade
    save: assignment
  end
end
```

The application can be easily changed to handle new requirements. For example, the code for having 3 students grade each assignment is:

```
grades = []
repeat 3 times
  grade = ask_peers_of:submitter to_grade:assignment
  grades = append:grade to:grades
end
```

In addition to a browser-based client, we also created an iPhone application using the the Cocoa Dog client library (Figure 5).

### Crowdsourcing: Tangent

Dog originated as a domain-specific language for parallel human computation [2]. It has evolved into a more general-purpose programming language, but is still useful for programming crowdsourcing systems. As an example, we used Dog to create a number of crowdsourced art pieces for a recent exhibition.

One such installation was a geometric art piece called Tangent. Participants used "child-friendly" tools like stencils and color pencils to draw circles that were tangent to each other and then color the negative space between them. The installation can be seen in Figure 6. The entire code for the application is shown below, although certain instructional message strings have been shortened for brevity.

```
# Functions

define confirm: message index: index do
  show index to everyone
  show message to everyone
```

```
  message = "Is there any space available?"
  listen to everyone for confirmation
  wait on confirmation
  confirmation == "no"
end

define confirm: message do
  show message to everyone
  listen to everyone for confirmation
  wait on confirmation
end

define draw_circle do
  confirm: "...draw a circle using a stencil..."
end

define ask: user to_draw_circle_tangent: index do
  confirm: "...draw a circle tangent
    to another circle..." index: index
end

define ask: user to_color_negative_space: index do
  confirm: "...color in any enclosed
    space that is not a circle..." index: index
end

# Main

draw_circle

count = 1
while !done do
  done = ask: someone to_draw_tangent_circle: count
  count = count + 1
end

count = 1
while !done do
  done = ask: someone to_color_negative_space: count
  count = count + 1
end
```

During the time of the exhibit, Tangent has had hundreds of collaborators. The application ran on a kiosk next to a canvas and gave visitors instructions on how they could contribute. The kiosk connected to a centrally hosted server which enabled remote tweaks to the application while it was deployed in another country.

### Social Networking: Aardvark

Dog is also capable of building applications using protocols other than HTTP. For example, we built an Aardvark-clone in Dog that allowed users to interact with the application over IM and email. Aardvark [11] is a social question-answer site that routes user submitted questions to somebody in the user's extended social network. Instead of submitting questions using a Web page, Aardvark allows users to naturally submit questions via instant message or email. Dog's standard library has built-in support for handling SMTP and XMPP connections by overloading the `listen` command with `via instant_message` making Aardvark's implementation straightforward. The follow code also makes use of the `wait on` command, which is the same as the `on` block except inline.

```
listen to everyone via instant_message for questions
on each question do
  message = "Hey! I'll try to find someone to answer that..."
  send_message: message to: asker

  category = classify: question
  ranked_users = people where expertise == category
  answered = false
  for each user in ranked_users do
    response = ask:user if_able_to_answer_question_on:category
    if response == "yes" then
      send_message: "Great, here it is: " to: user

      listen to user via instant_message for response
      wait on response

      send_message: "Thanks. I'll send that along." to: user
      send_message: "I got an answer from #{user}." to: asker
      send_message: response to: asker

      answered = true
      break
    else
      send_message: "Thanks for your help." to: user
    end
  end

  if !answered then
    message = "I couldn't find anyone. Please try again."
    send_message: message to: asker
  end
end
```

### RUNTIME

Most existing languages have runtimes that provide coarse-grained concurrency primitives, execute instructions in a blocking manner, and require application-level coordination for distributed execution. These properties make it difficult to efficiently implement Dog's workflow programming model. To address this, we built a custom runtime environment for Dog that provides solutions to these limitations.

### Existing Limitation: Thread-Only Concurrency

Many applications need to coordinate the activities of multiple concurrent and independent users. Threads are the conventional solution to concurrency. In theory, a developer could create a separate thread to manage each user. Unfortunately, threads require significant amounts of system resources. If an application creates a new thread whenever it is waiting for a user to submit input, the system will quickly run out of memory.

Some languages, like Ruby and Python, provide green-threads or coroutines that execute at the language level rather than at the OS level and generally use less memory. However, even green-threads are not suitable for coordinating individual users. First, despite their reduced memory consumption, green-threads cannot handle Web-scale traffic if each user is backed by a thread. Second, since thread state is stored in memory, all user progress will be lost when the application server is restarted (during a crash or routine maintenance). Third, green-threads do not change the fact that creating multi-threaded applications and coordinating access to shared memory is challenging for even the most experienced developers [18]. Fourth, at a practical level, most languages that provide green-threads also impose a Global Interpreter Lock (GIL) which only allows a single thread to run at a time

to avoid problems with libraries that are not thread safe (e.g. Python, Ruby, Javascript). As a result, these languages are unable to take advantage of multicore CPUs.

#### Dog's Solution: Task-Based Concurrency

Dog provides a concurrency primitive called *tasks*. A task is a green-thread that is backed by a persistent continuation. This means that a task waiting for input can be saved to disk to reduce memory contention and survive past server restarts. Saved continuations are automatically deleted from disk after a configurable timeout, which, by default, is 1440 seconds. An exception to this is the root task, which is never deleted.

Additionally, all values in Dog are immutable and the runtime prevents shared memory references. This memory model not only simplifies the understandability and predictability of Dog programs but also facilitates multithreaded programming by eliminating the need for locks and avoiding unintended side effects. Cross-task communication is coordinated by special, uni-directional, FIFO channels. Lastly, since there is no shared state, Dog does not impose a GIL and Dog programs take advantage of multicore CPUs. While immutability makes parallel execution easier, certain sequential programming tasks are harder. As an example, iterators are less efficient and can be awkward since changes to enumerated elements are lost unless explicitly re-saved.

#### Existing Limitation: Blocking Execution Semantics

Almost all existing languages' runtimes are synchronous and execute instructions in a blocking manner that waits for the current instruction to finish before proceeding. In conjunction with thread-only concurrency, when an application attempts to access input from a user, the entire thread will block. This has two negative consequences. First, as previously mentioned, it increases the memory demand on the machine. Second, this will lead to a large number of system threads, result in significant context switching overhead, and slow the machine.

Additionally, many applications may need to respond to multiple sources of user input. For example, a survey application may want to send out a single task to hundreds of people and wait for the responses. With blocking execution semantics, only a single task can be sent at a time even though the tasks are independent of one another and can be done in parallel. Using a separate thread (or even a Dog task) to solve this problem requires application control to be broken up into multiple worker threads and joined back, which is challenging for even experienced developers [18].

#### Dog's Solution: Asynchronous Execution

Synchronous runtimes provide a straightforward, "strictly consistent" guarantee: the program always finishes executing a statement before moving to the next. Dog's runtime, on the other hand, is asynchronous and provides a slightly different, "causally consistent" guarantee: a statement's output will be written by the time that the output is needed. In general, this nuanced distinction can be ignored as it often does not impact the *correctness* of a program[3]. However, it does allow the runtime to execute code in non-deterministic ways to improve performance. Certain function calls in Dog do not immediately return an output value but rather return a special future that promises to eventually hold the output. When a program attempts to access a future the current task will automatically block and the runtime will swap it out so another task can execute. The moment the promised value is provided, all tasks waiting on that future will resume execution. This allows Dog programs to wait for multiple users in the same task without fear of blocking the task from doing other operations.

[3]This is only true because of Dog's immutable memory model.

#### Existing Limitation: Process-Bound Execution

A key problem facing current applications is distributing traffic across multiple machines to handle large numbers of users [19]. This poses a problem for workflow-programming with existing languages because virtually all existing runtimes are process-bound and only execute in a single process on a single machine. Distributing execution over many nodes requires architecting applications around a centralized database and manually managing state between machines.

#### Dog's Solution: Dynamic Distributed Load Balancing

The Dog runtime is designed to naturally scale across multiple machines and readily tolerate machine failures. When a Dog program begins, the runtime starts, connects to a distributed task queue and waits for a task to be scheduled. Adding new machines to a Dog cluster is as simple running the Dog program on a new machine and specifying the task queue as a command line flag. When a task is scheduled for execution, the task queue will evenly distribute them across the available machines. Tasks can be distributed across machines without worry of deadlock, race conditions, or consistency errors, because of the runtime's non-shared and immutable memory semantics. Thanks to persistent continuations, machines in a Dog "cluster" can fail without worry. If a machine crashes in the middle of executing a function, some other machine will eventually pick up where the machine left off and continue execution. In fact, the runtime has a "crash-only" design — gracefully stopping a runtime instance and yanking the power cable from the box have the same effect. This is also another difference between Dog and continuation passing style Web frameworks. These frameworks store continuations in memory which means that special load balancing is needed to route subsequent requests to the correct process on the correct machine and that custom fault tolerance schemes are necessary to recover from a machine crashing or going offline for maintenance.

### IMPLEMENTATION

The standard reference implementation for Dog compiles to Java Virtual Machine (JVM) bytecode. Targeting the JVM was a natural choice given its performance, stability, and widespread adoption [14]. As a result, developers can natively interoperate with any existing Java library.

#### Compiler

The Dog compiler is written in Java and uses an LL(*) parser [25] generated from a grammar written using ANTLR [22].

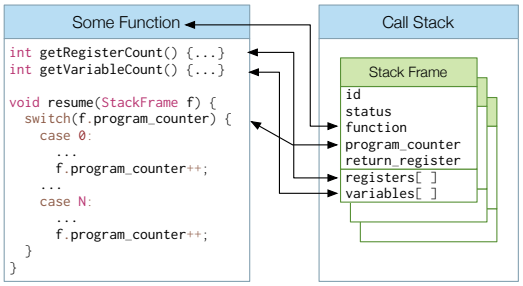


**Figure 7. Dog implements persistent continuations by subverting the JVM stack and using custom, heap-allocated, stack frames. Every function is backed by a class that reports the number of necessary registers and variable slots. A stack frame object is instantiated accordingly and passed into the function's resume method.**

The parser produces a Dog abstract syntax tree that the compiler transforms into a flat array of Dog Virtual Machine Instructions. The Dog Virtual Machine is a register-based machine [27] and has an instruction set modeled after Lua [12]. The assembler emits the appropriate JVM bytecode for each Dog VM instruction. Internally, each Dog VM instruction is modeled as a class that overrides the `toJVMBytecode` method. This code architecture allows reuse of the compiler front-end by multiple runtime back-ends, making it possible to target other runtime environments in the future[4]. Lastly, the generated JVM bytecode is written to disk as a *bark* file, which is a ZIP file that holds metadata used by the Dog runtime to begin execution.

### Persistent Continuations

The JVM (like most languages) does not provide support for continuations that are both serializable and portable. Thus, Dog subverts the JVM stack and forces the runtime to use a continuation passing style with custom stack frames. All Dog functions are compiled to Java classes that override the `resume` method. When the resume method is called, it receives a heap-allocated stack frame object that is dynamically created with the correct amount of registers and variable slots that the function needs to run (Figure 7). During compilation, all the JVM instructions are instrumented so they interact with the Dog stack instead of the JVM stack.

To resume a function the compiler inserts special *checkpoints* into the JVM bytecode stream. These checkpoint atomically increment a special *program-counter* in the stack frame (Figure 7). At the start of the function, the compiler inserts a jump table (implemented as a JVM `tableswitch` instruction) that inspects the program-counter and jumps to the correct instruction in the function's bytecode. The checkpoints are inserted such that the code between checkpoints is guaranteed to be idempotent. Thus, if the machine crashes while the runtime is between checkpoints, the function can still be safely resumed.

### Scheduler

The scheduler connects to the distributed task queue, waits for tasks to be scheduled, and then coordinates their execution.

[4]Currently, Dog has an existing implementation in Ruby. Additionally, a Javascript runtime that targets V8 and Node.js is also in the works.

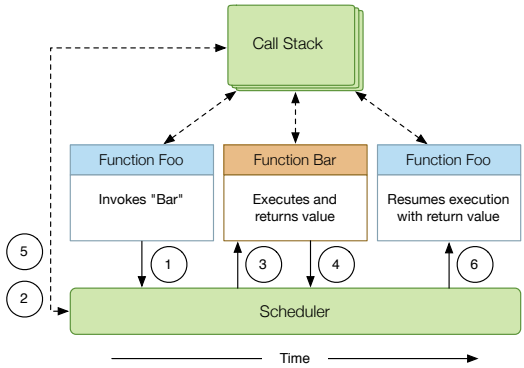


**Figure 8. Dog uses a "trampoline" to call and return from functions. Function Foo calls Bar by returning a signal object to the scheduler (1). The scheduler inspects the signal, allocates a new stack frame for Bar and pushes it onto the call stack (2). The scheduler then calls Bar and passes the current call stack as an argument (3). Function Bar executes and returns another signal object with its return value (4). The scheduler pops the stack frame from the call stack, and inserts the return value into Foo's stack frame (5). The scheduler then resumes execution of Foo from where it left off (6).**

Each runtime has at least one scheduler but, by default, the reference implementation creates single scheduler for every core that is available on the system to maximize performance on multicore CPUs. The task queue is implemented on top of MongoDB [1].

An important part of the scheduler is managing function invocations using a technique called a "trampoline" [4]. Instead of a caller function directly invoking a callee function, the caller simply returns a "signal" to the scheduler with a reference to the callee (Figure 8). The scheduler inspects the signal, resolves the function reference, dynamically creates a custom stack frame for the callee, sets the callee's stack frame's parent to the caller's stack frame, and then invokes the callee. When the callee returns, the scheduler takes the return value, inserts it into the caller's stack frame's `return_register` (Figure 7), and resumes execution of the caller. The use of a trampoline greatly simplifies the implementation of Dog's persistent continuations.

When a task attempts to access a future or begins waiting on a channel it throws a `WaitException` which is caught by the scheduler. The task is then persisted to the task queue until the data become available.

## PERFORMANCE EVALUATION

While Dog was not designed specifically for execution speed, it is important to achieve a certain threshold of performance to be useful in practice. Consequently, we ran a standard set of benchmarks on Dog and compared its results with other languages:

| Benchmark | Dog v0.4 | Ruby v1.9 | Python v2.7 |
|---|---|---|---|
| fib | 1.722s | 2.357s | 6.385s |
| prime | 14.974s | 3.985s | 15.993s |
| mandelbrot | 11.410s | 8.780s | 1.080s |
| substring | 15.222s | 28.016s | 22.724s |

As shown in the table above, Dog is competitive with Ruby and Python, faster on the benchmarks that involve frequent

memory access (substring) and slow on the benchmarks that require complex mathematics (mandelbrot).

In terms of network performance, which is more important for Web applications, Dog has a clear advantage thanks to the JVM's robust network stack. The GradeBook application written in Rails with various performance optimizations is able to process an average of 229.08 requests per second while the Dog version is able to process 937.62 requests per second.

All tests were run on a 2.4 GHz Intel Core i5 CPU with 8GB of RAM. It is important to note that performance-critical portions of a Dog program can always be written natively with Java.

## RELATED WORK

Dog was inspired by many different languages and systems. Its syntax with named parameters heavily influenced by Smalltalk, its data model with only a single composite "structure" type by Lua, its register-based virtual machine instruction set by Lua, its immutable memory model by Clojure, and its runtime concurrency features like tasks and channels by Go and Stackless Python [7, 12, 10, 8, 31].

Dog is philosophically related to prior work on languages that facilitate Web programming tasks, although they target other aspects of the Web development stack: S (RESTful services), Flapjax (client-side AJAX), and Atomate (information processing) [5, 16, 32]. Dog is also related to and inspired by many widely-used Web development frameworks that prioritize ease of use and rapid development like Rails and Django [24, 6]. Additionally, there are a handful of Web development frameworks that are built around continuation passing. These include Seaside, Apache Cocoon, and Tir [23, 30, 26]. While these frameworks make it easy to use workflows in Web development they do not support persistent continuations for load balancing and are not built around a server-client-scripting protocol to target multiple client platforms.

## CONCLUSION

Dog introduces a new way of building applications that simplifies many development tasks. Its unique runtime model provides elegant solutions to asynchronous state management, cross-client development, and scalable deployment. As such, Dog represents a useful and timely tool for developing modern applications, which emphasize Web accessible services that coordinate the activities of many users across multiple different client platforms.

## REFERENCES


1. 10Gen Inc. MongoDB. **http://www.mongodb.com**.
2. Ahmad, S., Battle, A., Malkani, Z., and Kamvar, S. The Jabberwocky Programming Environment for Structured Social Computing. In *Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology*, ACM (2011), 53–64.
3. Andreesen, M. Why Software Is Eating The World. *The Wall Street Journal* (2011).
4. Bothner, P. Kawa: Compiling Dynamic Languages to the Java VM. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1998), 41–41.
5. Daniele Bonetta et. al. S: A Scripting Language for High-Performance RESTful Web Services. In *PPoPP '12* (2012).
6. Django Software Foundation. Django. **https://www.djangoproject.com**.
7. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
8. Google Inc. Go. **http://golang.org**.
9. Google Inc. Google web toolkit. **https://developers.google.com/web-toolkit/**.
10. Hickey, R. Clojure. **http://www.clojure.org**.
11. Horowitz, D., and Kamvar, S. D. The Anatomy of a Large-Scale Social Search Engine. In *Proceedings of the 19th International Conference on World Wide Web* (2010), 431–440.
12. Ierusalimschy, R., De Figueiredo, L. H., and Celes, W. The implementation of Lua 5.0. *Journal of Universal Computer Science 11*, 7 (2005), 1159–1176.
13. Katz, Y. Handlebars.js: Minimal Templating on Steroids. **http://handlebarsjs.com**.
14. Lindholm, T., and Yellin, F. *Java Virtual Machine Specification*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1999.
15. Meijer, E., Beckman, B., and Bierman, G. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of ACM SIGMOD* (2006), 706–706.
16. Meyerovich, Leo A. et. al. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN conference on OOPSLA* (2009), 1–20.
17. Myers, B., Park, S. Y., Nakano, Y., Mueller, G., and Ko, A. How Designers Design and Program Interactive Behaviors. In *Visual Languages and Human-Centric Computing* (2008), 177–184.
18. Ousterhout, J. Why Threads Are A Bad Idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, vol. 5 (1996).
19. Ousterhout, J. Fiz: A Component Framework for Web Applications, 2009.
20. Pane, J. F., Myers, B. A., and Miller, L. B. Using HCI Techniques to Design a More Usable Programming System. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments* (2002), 198–206.
21. Pane, J. F., Ratanamahatana, C., Myers, B. A., et al. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies 54*, 2 (2001), 237–264.
22. Parr, T. J., and Quong, R. W. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience 25* (1994), 789–810.
23. Perscheid, M., Tibbe, D., Beck, M., Berger, S., Osburg, P., Eastman, J., Haupt, M., and Hirschfeld, R. *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008.
24. Rails Core Team. Ruby on Rails. **http://www.rubyonrails.org**.
25. Rosenkrantz, D. J., and Stearns, R. E. Properties of deterministic top down grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, ACM (1969), 165–180.
26. Shaw, Z. Tir Web Framework. **http://tir.mongrel2.org**.
27. Shi, Y., Casey, K., Ertl, M. A., and Gregg, D. Virtual Machine Showdown: Stack versus Registers. *ACM Transactions on Architecture and Code Optimization 4*, 4 (2008), 2:1–2:36.
28. Smith, A. Smartphone Adoption and Usage. *Pew Research Center* (2011).
29. Smith, A. Nearly half of American adults are smartphone owners. *Pew Research Center* (2012).
30. The Apache Cocoon Project. Cocoon. **http://cocoon.apache.org/**.
31. Tismer, C. Continuations and Stackless Python. In *Proceedings of the 8th International Python Conference* (2000), 2000–01.
32. Van Kleek, et. al. Atomate it! End-user Context-sensitive Automation using Heterogeneous Information Sources on the Web. In *WWW '10* (2010).