Pheme: A real-time user interface for distributed systems

Karl Bo Lopker (klopker@cs.ucsb.edu)
Pete Cappello (cappello@cs.ucsb.edu)

June 1, 2013

Contents

Abstract

The big data era is upon us. To manage it all we increasingly rely on distributed application frameworks like Hadoop, Storm, and jPregel. These frameworks are powerful, but they often have esoteric report and control schemes. Developers rarely have the resources to make something better. Pheme is a simple, drop-in interface for these systems. Users need easy access to distributed applications through a simple interface. Pheme is a real-time standalone web application that is straightforward to use for both application developers and end users. By making big data more accessible to fields outside of computer science, Pheme puts the power of these frameworks in the hands of the people who need it most.

Pheme uses technologies such as Java, the Play Framework, Websockets, and Require.js. This paper explores the design and performance challenges associated with mixing these technologies together to make Pheme a reality.

Introduction

Today's researcher needs to process more data than ever before. They often have more data than can fit in a single commodity machine. Fortunately, we have built excellent tools for coordinating large networks of computers that can process this data. For example, frameworks like Hadoop[A1], Storm[A2], and jPregel[A3] help scientists execute large data processing jobs. However, not everyone possesses the technical background to run these systems. Often, a researcher will have to pay someone to configure computers, write code, and/or run jobs. This alone is enough to halt already underfunded projects. In order to get the power of Big Data into the hands that need it, we want to ease some of these pain points.

To help researchers, we need to figure out where these systems are causing them trouble: We need to find the most effective way to help. Setting up these systems is daunting, but it's done only once. Tasks like monitoring and running jobs, however, need constant attention. This is a barrier for researchers without computer science backgrounds because these tasks usually require the command line. Currently, some systems like Hadoop have spartan web interfaces[A4] for monitoring, but they lack the modern accoutrements of today's webapp. As such, these interfaces tend to only display static information which becomes stale quickly. In contrast, a modern web interface can react to events and show data in real-time. A modern interface can even activate functions on the system; reducing the need for esoteric terminal commands. Also, by using well-known web standards, this interface can be flexible enough to put in any distributed system. Never again does a distributed systems developer have to reinvent the wheel in this respect.

Pheme, who's name comes from the Greek goddess of gossip, is our response to these needs. Researchers will spend less time learning terminal commands and more time getting things done. When a researcher logs into Pheme, they see the dashboard which displays system vitals. On the left, they find a count of the system's active computers and jobs. No more repeatedly hitting the refresh button; these counts update in real-time. Pheme however isn't easy for just the end-user; developers benefit from its simplicity as well.

Figure 1: Pheme's Logo

Integrating Pheme in a distributed system is easy as well. Pheme's only dependency is a recent version of Java. Developers just package the server binary with their framework and activate the `start` script. To communicate with the server, we offer a Pheme API Maven repository. Once imported, this API has a small but powerful set of methods to activate Pheme's functionality. Sending logs, counts, and gauges are just one call away. Furthermore, sending data to the server happens in a separate thread, so it is non-blocking.

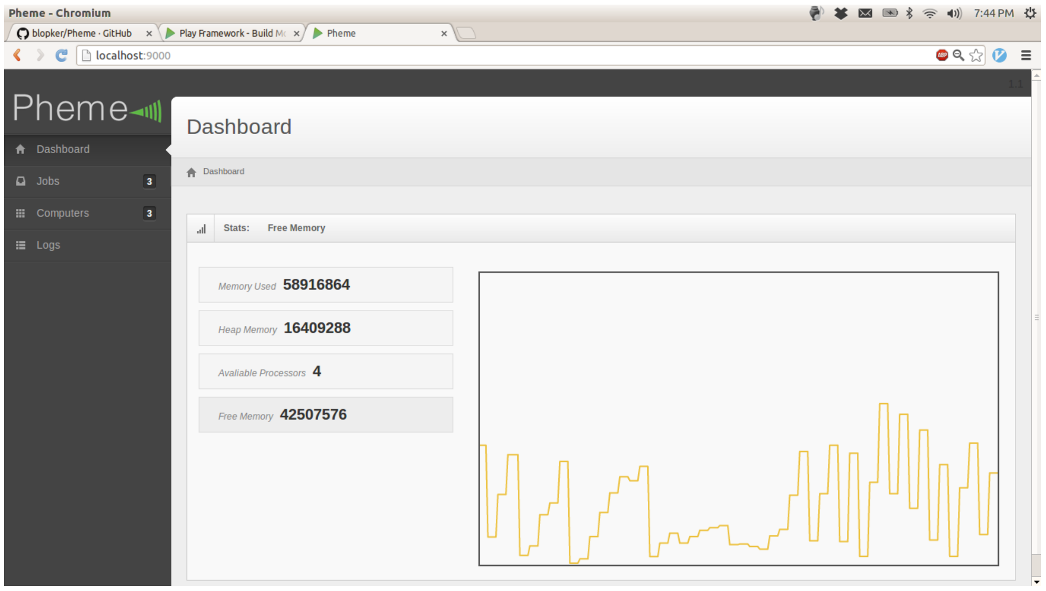
In fact, Pheme's whole architecture is full of modern concurrency and performance considerations. Pheme uses threads, blocking queues, and an asynchronous event bus to move data around. This keeps the server responsive by leveraging all the available CPU cores. In terms of memory usage, Pheme also takes care to keep it to a minimum. It does this by keeping only the data it needs, while using efficient data structures. All these factors result in Pheme's solid real-time performance.

In this paper, we discuss using the web interface and integrating the Java API. We also detail the architecture used on the client (browser), server and API. Finally, we present our performance tests and analysis.

Use

In this section we will go into detail about Pheme's user interface. The user interface is what a researcher sees when they use a distributed system. From here they can check on various subsystems and running jobs. These checks include viewing logs and certain metrics. On the dashboard we can see some of these metrics in action.

User Interface



Memory Used 58916864

Heap Memory 16409288

Avaliable Processors 4

Free Memory 42507576

Figure 2: Dashboard page

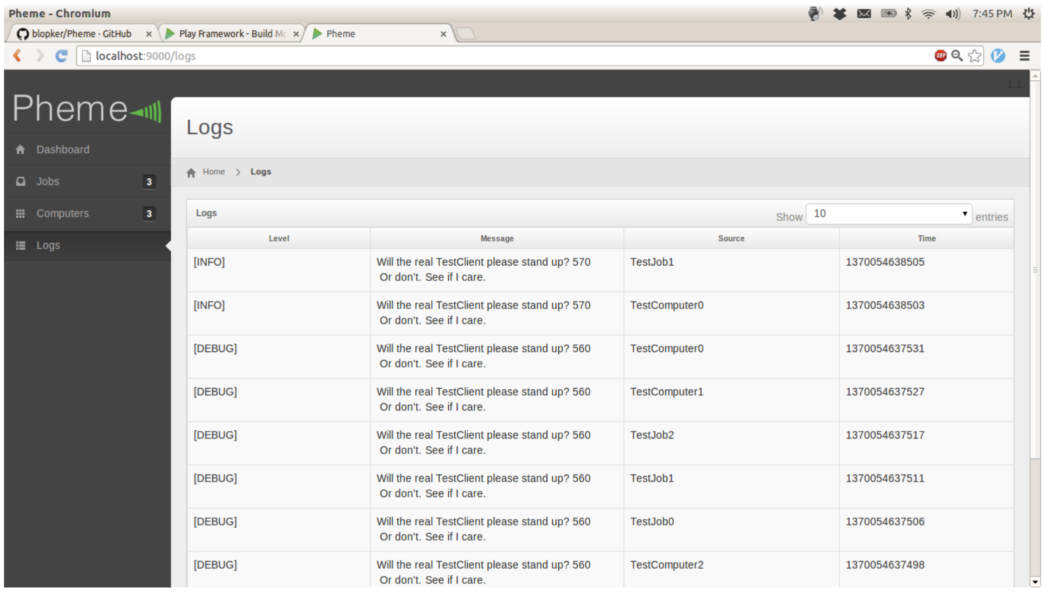
The dashboard is Pheme's index page. We can view metrics about the system on which the Pheme server is running. Down the side of the graph, we see four `count` data-types: `Memory Used`, `Heap Used`, `Available Processors`, and `Free Memory`. These metrics change in real-time as the interface receives them from the server. The graph is currently showing the `Free Memory` metric. There are two indications of this: first, the `Free Memory` button is a darker gray, and, second, the Stats widget says `Free Memory` as the subtitle. If the user clicks on any of the other metrics, these two indicators change to the appropriate label. Looking over to the left, we see the navigation menu.

This navigation menu hosts more real-time functionality. The `Jobs` and `Computers` buttons, contain numbered indicators. When a new job starts or another computer connects to the system, these indicators increment automatically. This gives the user critical feedback about system changes. Aside from those two buttons, we also have `Dashboard`, which takes us to the index, and `Logs` which we discuss next.

The logs page is where users can see every log sent to Pheme. Each log has four properties: level, message, source, and time. The level field shows users how critical that log is. The system developer has full control over these labels, but the usual nomenclature is: INFO, ERROR, WARN, and DEBUG. The message field shows what the log mean, and tries to reproduce the log's message accurately in HTML. The source field provides the name of the job or computer from which it originated. Finally, the time field shows the when the log got to Pheme's server, in milliseconds since epoch. The time stamp could probably be formatted better, but that makes sorting a little tricky. Look for this feature in Pheme 2.0!

Log tables has several advanced features to help sift through data. It supports pagination which only shows some logs at a time. This keeps the interface snappy when in heavy use. However, if the user wants more logs shown, they can select a larger number from the pull-down menu. Another major feature is the instant search box (not shown) at the bottom. Here, a user can start typing and the log table filters out logs that don't contain the text. Sometimes, a user may want logs only for a specific job or computer. For this, they can go to the either the computer or job listing pages.

The listing pages show overviews of the current computers or jobs that have data available. The tables have item's name and times when they first connected to Pheme. The name a unique identifier. This means any part of the system can send data to where it makes sense. For instance, if a job is running on multiple computers, log files generated for that job can go to the same place. To get a more detailed view, the user can click on one of the lists'

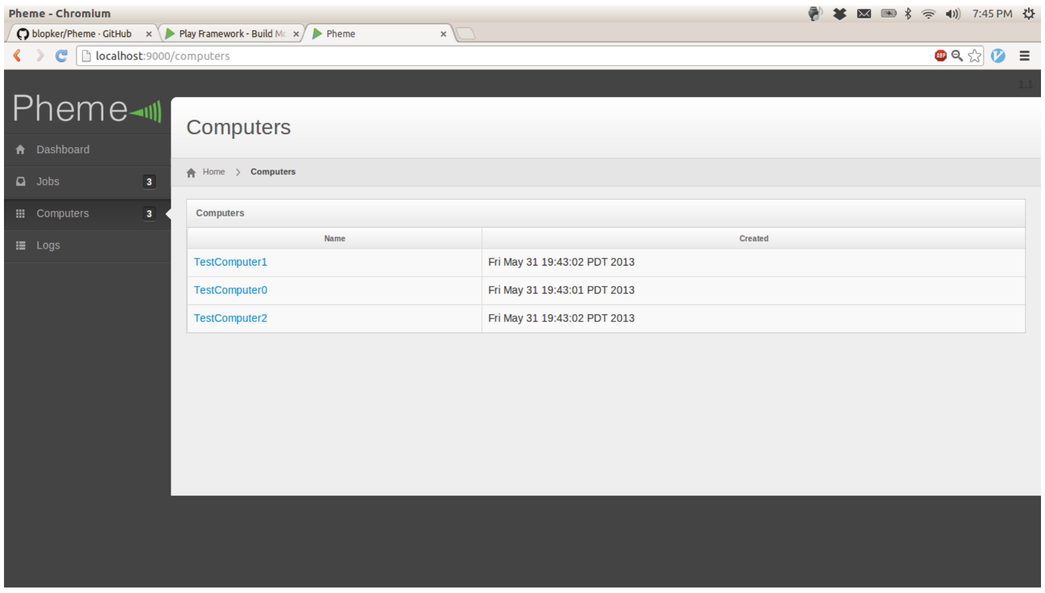


Logs

Logs

Level	Message	Source	Time
[INFO]	Will the real TestClient please stand up? 570 Or don't. See if I care.	TestJob1	1370054638505
[INFO]	Will the real TestClient please stand up? 570 Or don't. See if I care.	TestComputer0	1370054638503
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestComputer0	1370054637531
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestComputer1	1370054637527
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestJob2	1370054637517
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestJob1	1370054637511
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestJob0	1370054637506
[DEBUG]	Will the real TestClient please stand up? 560 Or don't. See if I care.	TestComputer2	1370054637498

Figure 3: Logs page



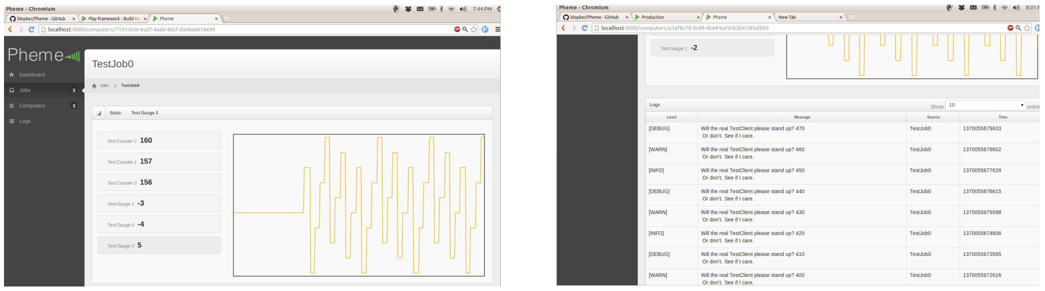
Computers

Computers

Name	Created
TestComputer1	Fri May 31 19:43:02 PDT 2013
TestComputer0	Fri May 31 19:43:01 PDT 2013
TestComputer2	Fri May 31 19:43:02 PDT 2013

Figure 4: Computer listing page

items.



Level	Message	Source	Time
[DEBUG]	Will the real TestClient please stand up? 470 Or don't. See if I care.	TestJob0	1370055679633
[WARN]	Will the real TestClient please stand up? 460 Or don't. See if I care.	TestJob0	1370055678602
[INFO]	Will the real TestClient please stand up? 450 Or don't. See if I care.	TestJob0	1370055677628
[DEBUG]	Will the real TestClient please stand up? 440 Or don't. See if I care.	TestJob0	1370055676615
[WARN]	Will the real TestClient please stand up? 430 Or don't. See if I care.	TestJob0	1370055675598
[INFO]	Will the real TestClient please stand up? 420 Or don't. See if I care.	TestJob0	1370055674606
[DEBUG]	Will the real TestClient please stand up? 410 Or don't. See if I care.	TestJob0	1370055673595
[WARN]	Will the real TestClient please stand up? 400 Or don't. See if I care.	TestJob0	1370055672616

Figure 5: Job detail. Left: Graphs. Right: Logs.

The job detail page combines the dashboard's graph view and a log table. Both of these widgets react in real-time to events for that job. Anyone can configure the system with custom graphs and log messages to suit the job at hand. Users can able to see the system status from across the room, getting constant updates that everything is working. This is in stark contrast to Hadoop's interface which forces users to constantly refresh their browsers.

Requirements

Although Pheme's interface is web-based, it won't run on browsers from yesteryear. Pheme is tested on recent builds of Chrome and Firefox. These are the recommended environments to run the interface because they are the most advanced and standards complaint browsers currently available. In its current state, we cannot recommend Internet Explorer for anything.

Integrate

In this section we will show, with examples, how to send information to Pheme's server for each supported language. Currently there's only a Java API, but Pheme's architecture is such that other language APIs can be developed quickly.

Each language section has Getting Started and Using sections. The Getting Started section will walk developers through setting up a minimal example application that uses Pheme's API. The Using section details some common API calls and what they mean. Think about it like a smaller scoped, but more detailed Javadoc.

Java

Getting Started

In this section, we go over creating a simple Pheme client that can send logs to the Pheme server. To get started, make sure you have all the requirements installed first:

Requirements

- Java 1.6 or better, 1.7 recommended
- Maven 3 or any build tool that can use Maven repositories
- A solid text editor; Eclipse, NetBeans or Sublime Text will do
- Note: these instructions are for Ubuntu 13.04. Watch out for differences in other OSes.

Getting the API The first order of business is to get the API jar in your project. To do this, we must add both Pheme's repository and the API's artifact information.

Pheme's repository is available at: http://blopker.github.com/maven-repo/

To include this repository in your POM.xml file, add this snippet into your `<repositories>` directive:

```
<repository>
    <id>Pheme-Repo</id>
    <name>Pheme Repo</name>
    <url>http://blopker.github.com/maven-repo/</url>
</repository>
```

Next, we must tell the build tool what artifact to look for. The Pheme API artifact `groupId` is `pheme` and the `artifactId` is `pheme-api`. The current `version` is `1.1.0`, but that will change. Check the git repository for the newest version number.

For Maven, this goes in the `dependencies` directive:

```
<dependency>
    <groupId>pheme</groupId>
    <artifactId>pheme-api</artifactId>
    <version>1.1.0</version>
</dependency>
```

Once you've added this information to your build tool, try building. The API should download and be in your classpath.

Connecting to Pheme Once we have the API in our classpath, we need to create a new `Pheme` object with the hostname where Pheme's server is. If the server is running on the same machine, it will be `localhost`. But, any URL like `blopker.com` where Pheme is running will work. Currently, Pheme has no authentication. So, it's generally best to host the server inside a closed network.

Here we demonstrate the simplest case where the client connects, but does not send any information. Create a new class file with a main method like:

```
public static void main(String[] args) {
    // Connect to Pheme
    pheme = new Pheme('localhost');
}
```

This is enough to alert Pheme of this client's presence, but Pheme will not show anything on the web interface because no `Components` have been registered yet. Pheme does not base any information on a single connection. In fact, we encouraged developers to make only one `Pheme` object per JVM to keep connection counts low.

As we already hinted, we now will register a `Component` with Pheme. `Component` is the base class `Job` and `Computer`. The two objects from which Pheme can currently take data. Let's make a `Job` object.

First we must register a new `Job` with the `Pheme` object. Components are uniquely identified by their names. If another JVM registers a `Job` with the same name, Pheme aggregates data from both sources to the same `Job`. This allows multiple machines to report on the same components. After we register a `Job`, we will send a log to Pheme.

```
public static void main(String[] args) {
    pheme = new Pheme('localhost');
    Component job = pheme.registerJob('My First Job');
    job.log('info', 'Hello Pheme!');
}
```

Running this tells Pheme to create a `Job` named 'My First Job' on the web interface and store a new log for it. Going to `localhost` in the browser should confirm that.

We now have a basic client up and running. In the next section, we explore more of the Pheme API.

Using the API

This section will discuss the main methods to use in the Pheme API. For a more complete look check out the javadoc. We divided the subsections into the specific classes to interact with.

`Pheme` Create the `Pheme` object with Pheme server's hostname. When this object is created a new thread handles the connections and sending. Thus the constructor returns immediately and the connection happens in the background.

The Pheme object has several methods:

- `registerX(String)`: The register methods take a name and returns a component of the type you requested. This is the way you create a Computer or Job on the Pheme server.

- `startServer()`: This starts a Pheme server on the machine you ran it on. In the background this method downloads the Pheme server binary to the `/tmp` directory, unzips it and runs it. If the zip file is already in your `/tmp` directory this method will skip the download part. If a server is already running this method is a noop. This method also takes an optional `boolean` value which will tell the server not to shut down when the starting JVM dies.
- `killServer()`: This kills the Pheme server started with `startServer()`

`Compoonent` A component is either a Computer or a Job. Both objects have the same API, they are just put different places on the web interface. To get a component call the appropriate `registerX()` method on a `Pheme` object. All these methods are non-blocking. A completed method call does not guarantee delivery of the message.

The methods of `Component` are:

- `count(String, long)`: A count is a running record of the numbers sent to it. Two calls to `count("counter", 1)` will show a value of `2` on the interface.
- `gauge(String, long)`: A gauge is an instant record of the number sent to it. Two calls to `gauge("gauge", 1)` will show a value of `1` on the interface.
- `log(String type, String message)`: A log needs a type and a message. This method will tell Pheme to store a log of this type and show the user. The type is any arbitrary string.

That concludes the main objects of Pheme's Java API. Be sure to check out the javadoc for more information.

Develop

In the Develop section we delve into how Pheme works internally. This section is for developers who would like to modify or extend Pheme. First we will go over some of Pheme's feature and performance goals. Then we explain the terminology used in the code base. Next we will talk about Pheme's architecture, how messages get from clients to users. Then we describe how to extend Pheme. Finally we put Pheme through its paces and run some performance tests.

Requirements

To make Pheme a viable option for system developers, it must meet some requirements. These requirements are split into two categories: functional and performance. Functional requirements are key features Pheme must have to make it useful. Features however are only useful if they are fast. Pheme also needs to meet some performance goals. This ensures that Pheme doesn't interfere with a system's performance and doesn't irritate its users.

There are functional requirements for both the server and the interface. The server must have an API that is easy to understand and get working. We chose an adapter based API system for this reason. With adapters, we are able to make APIs that are idiomatic for every system we want to support. For example, the Java API uses RMI and object orientation. A JavaScript API might use HTTP and callbacks. Pheme's interface has functional requirements too. It must reliably take data from the server and present it in an organized way. Logs, for example, might be formatted in a specific way by the client. If care is not taken, the formatting could be lost when it's displayed on the browser. This could make the logs useless. Pheme takes great care to ensure that this doesn't happen.

Pheme's performance requirements are focused mainly on the API. If Pheme added any noticeable overhead to a distributed system, it would probably not be used. With this in mind, we made all Pheme's API methods non-blocking, and minimized client side processing. Pheme's API works by using blocking queues and a separate thread for sending data across the network. Each method in the API uses the producer-consumer pattern: It's arguments are put into a queue by the invoking thread, and consumed (processed) by another thread, enabling API method invocations to return control immediately to their invoking thread.

Terminology

As with any decent sized project, developers must familiarize themselves with terminology to ease communication of concepts. Pheme is no different. This section introduces the terms a developer working on Pheme needs to understand. We start with terms from the API and move to the server. First, is the `Component` concept.

`Component` A component is an object that sends datatypes. It is an umbrella term, an abstract class, if you will, for component types like `Job` and `Computer`. Every component type has an entry on the left side of Pheme's interface. Clicking on the type displays a list with access to the type's details.

Components are uniquely identified by their names. For instance, a `Job` could be named 'Word Count'. Any client that registers a `Job` with that name can send datatypes to the same `Job` object on the server. This makes it trivial to get updates from different parts of a cluster.

`Datatype` A datatype represents any kind of message that a `Component` may want to store. Logs, count, and gauges are all examples of datatypes. Pheme handles transferring datatypes automatically. Though, each datatype must define its persistence on the server. For example, counts and gauges only store the last value sent, but logs store the last 1000 values for each `Component`.

`Count` Counts are datatypes that can either be incremented or decremented. Pheme evaluates the last count sent and adds it to the new count value. Counts show up on the interface in graph form.

`Gauge` A `Gauge` is like a count, except that it replaces the old value with the new one. Gauges show up on the interface in graph form.

`Log` `Logs` are more complex datatypes that have more persistence and hold more data than counts or gauges. Logs notify users of events. Logs have types, messages, and time-stamps. Logs appear in a table on the interface.

`Adapter` Once a datatype leaves the client, it arrives at an adapter. Adapters form a plugin-type system for connecting Pheme to clients via different transport protocols. Essentially, it's an adapter's job to translate datatypes it receives from clients to Pheme's internal API. Currently, the only adapter facilitates communication with Java's RMI.

Architecture

We divided Pheme's architecture into three parts; Interface, Server, and API. The interface is what users see in their browsers. Pheme's interface is a sophisticated webapp with its own logic, so much so that it might be considered a separate project. The server is the conduit between a distributed system and its users. The server processes and stores the data on its way through. The API is the conduit between a system and Pheme. It must perform well and tolerate faults.

Interface

The interface is what Pheme users see in their browser. It uses JavaScript, CSS, and HTML. This code gets its data from the server through either standard HTTP or via Websockets[B1]. Besides Websockets, the interface uses several other technologies: jQuery[B2], Twitter Bootstrap[B3], PubSubJS[B4], AutoBahnJS[B5], and Require.js[B6].

Interface Dependencies

- jQuery
- Twitter Bootstrap
- PubSubJS
- AutoBahnJS
- Require.js

Websockets provide a bidirectional communication layer between the interface and the server. The interface initiates a persistent Websocket connection with the server. This connection allows the sever to push user data in real-time. All data is transfered using JSON.

We use jQuery throughout the front end to simplify JavaScript. Although it has some negative performance implications, jQuery excels at hiding browser quirks inherent in web programming. This library also makes code more concise by providing convenience functions for common patterns. jQuery also eases the creation and management of Websockets.

Twitter Bootstrap is a CSS framework. Bootstrap provides a CSS grid system based on columns. This gives Pheme a solid and structured feel. It also comes with an impressive JavaScript library for creating flexible usability paradigms, like modal pop-ups or buttons. Bootstrap also helps with browsers' visual quirks by providing a normalizing CSS file.

PubSubJS gives Pheme a publish/subscribe mechanism for the interface. This allows Pheme's graph and log views to use a decoupled observer pattern. They tell the pub/sub system what data they are interested in. When the Websocket connection gets that data, they are notified.

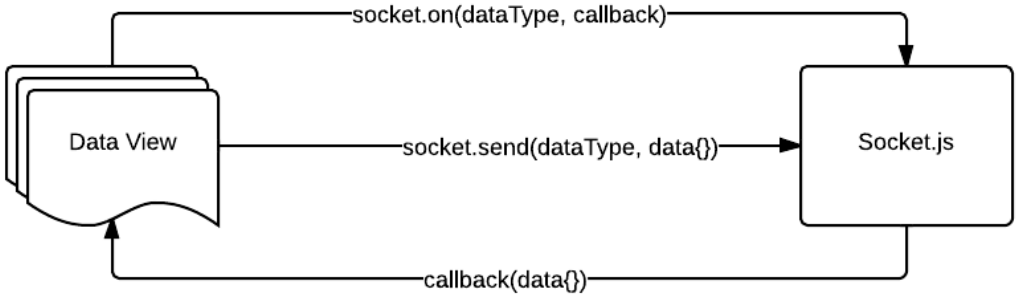


Figure 6: How data views tell the Websocket what data they want.

AutoBahnJS is a WAMP[B7] implementation for JavaScript. WAMP is a specification of how to send data through Websockets. We'll go into more detail later. AutoBahnJS gives the interface seamless compatibility with WAMP-enabled servers like Pheme.

Finally, to manage all this JavaScript, we need Require.js. Traditionally, JavaScript is written in one giant file. This is necessary because browsers cannot import JavaScript modules like other languages. Browsers can only use code that is either in the same file or is loaded through a `script` tag in the DOM. Require is a JavaScript loading library that fixes this issue. With Require, Pheme is able to keep its JavaScript organized for development. When Pheme is deployed, Require concatenates all the code into one file for browsers to use.

Require has another feature that Pheme uses: HTML templates. Widgets like the log table or the graph area need HTML to generate their part of the DOM. Instead of sticking the HTML in JavaScript strings, we load HTML files with Require. These snippets can be found in the `/public` folder with the CSS and image assets.

The interface gets all of its data from the server. This communication happens through either dynamically generated HTML or with WebSockets. Let's see how that works.

Server

The server is the most sophisticated part of Pheme. It follows the classic Model-View-Controller (MVC) separation of responsibilities. The interface communicates with Pheme through controllers. These controllers contain business logic that knows how to compile data from the models into the static view files. In this section, we go over how each part works.

Server Dependencies

- Google Guava
- WAMPlay
- Play Framework 2.1
- WAMPlay

Views comprise of static files and Scala/HTML templates. Essentially, they are HTML files with a little Scala logic to show controllers where to put data. When a client first connects to Pheme, they are sent a compiled HTML page from the template engine. A compiled HTML page includes a main template and a page template. The main template has HTML common to all Pheme pages, like navigation. Changing this file affects the entire interface. Page-specific templates are unique to the URL a user requests. For instance, the dashboard and log pages share the same main template, but have different page templates.

Pheme's datatypes are examples of models. When a model's save function is called (either from a change or creation), it saves the data in a data structure specific to that model. The model then notifies Guava's[B8] event bus of the

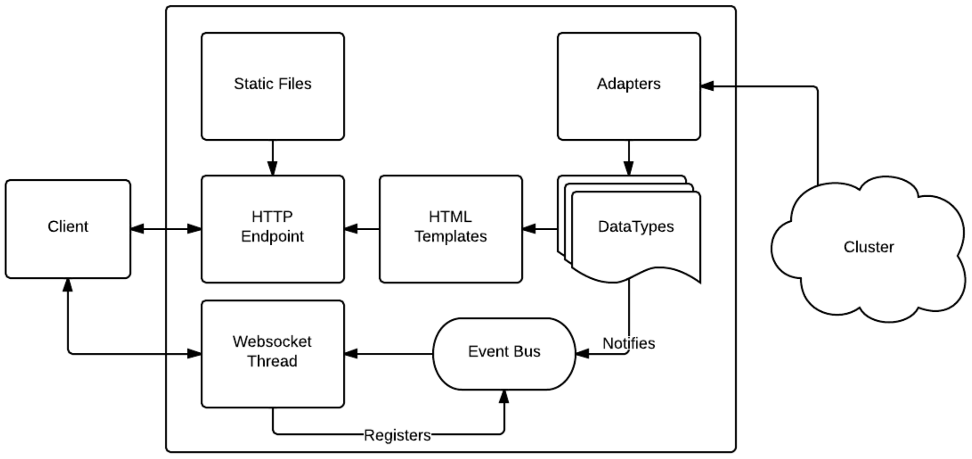


Figure 7: Server architecture

change. Here's an example of the `Count` model telling the event bus a new count was created:

```
public void publishCount(Count count) {
    EventBus.post(count);
}
```

Another object can then listen for count events with this:

```
@Subscribe
public void countListener(Count count) {
    publish(getTopic("counts"), count);
}
```

Now, the event bus will notify that object every time a new `Count` is made. The main object that listens to the event bus is WAMPlay[B9].

WAMPlay To make Pheme a reality, a Websocket based RPC and publish/subscribe library also was created. This library is needed because, although the Play Framework[B10] can use Websockets to send arbitrary strings, it defers message structure to the developer. Without a message specification, bidirectional communication becomes ad-hoc, and quickly can get messy. Luckily, a solid Websocket subprotocol specification called WAMP already existed. To facilitate Pheme's real-time requirements, the WAMP specification is implemented in Play's context to create WAMPlay. This library already has garnered interest in both the WAMP and Play communities, revealing this as a common pain point for developers.

WAMPlay is considered a controller, moving data between the interface and the models.

Besides the interface, Pheme also has to communicate with distributed systems. It does this through adapters. An adapter's job is to listen, on whatever communication layer it knows, for data. When new data arrives, the adapter must convert that data into Pheme models. Currently, the only adapter uses RMI for communication. Again, when an adapter creates a new model, an alert is sent to the event bus. Adapters have associated APIs that clusters use to make communications simple.

API

Each adapter in the server has an API. Since the RMI adapter is the only one for now, we focus on its API.

The RMI API is designed to be non-blocking. It does this by backing any method invocation with a blocking queue. The API spawns its own thread to consume items from this queue and send the data to Pheme. This way, a component can send a log to the API, but not have to wait for the log to actually be sent. If the API loses connection with Pheme, it buffer the logs and tries to reconnect every couple of seconds.

The API is also simple to package with an existing application. It is designed to be compiled into a jar file so that it easily can be imported into a classpath. It has no dependencies outside of Java. Since the API uses RMI for communication, client applications can be on any Internet connected device and still send logs to Pheme. The API is also conveniently available as a Maven artifact.

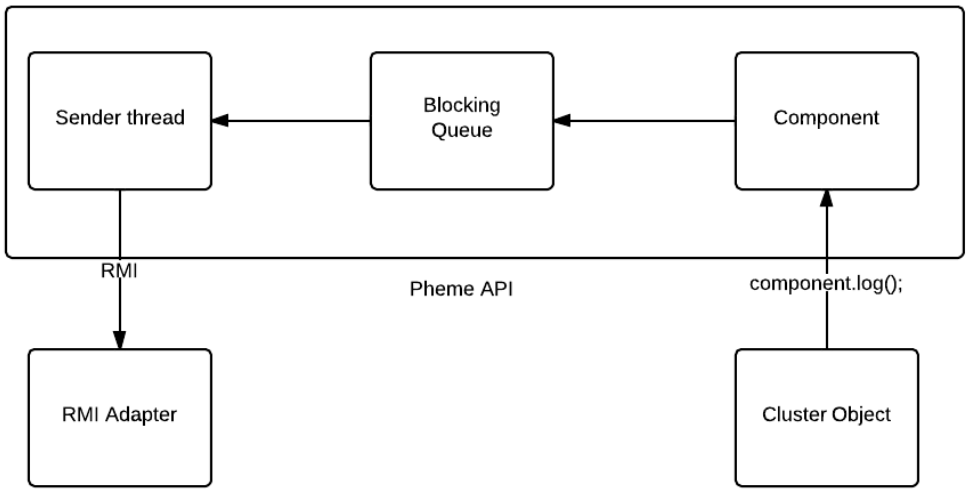


Figure 8: Pheme's API, a call to log()

Extending

Datatypes

Datatypes are the messages clients send Pheme to display to users. Making new datatypes requires a couple of steps. We use the `Count` class as an example.

1. Make a new class in the models.datatypes package. Have it implement DataType.
2. Create a way to store the datatype values. `Count` uses a Multimap from Goolge's Guava library. This allows `Count` to store multiple counts for each component.
3. Add a way to create the datatype. This method is used by adapters when they translate the datatypes from their APIs. `Count` has `public synchronized static DataType create(Component component, String counterName, long addToCount)`.
4. Make a way to get all the data for the datatype. This is useful when a interface first connects, and it needs all the stored information. `Count`

has `public static List<Count> getAll()`

5. Make sure all the adapters can make the datatype. For instance, a new method might have to be added to the Java API so that clients can make it. This means that the adapter also has to know where to find and create the datatype. The RMI adapter does this mapping in `adapters.rmi.DTOMapper.java`.
6. Add the datatype name to the enum list in `DataTypes.java`. This is used by some classes to identify with the datatypes.

That should cover most of it. Remember to make any data you want the interface to use a `public` field. This is how the Java to JSON converter serializes data. It's may be easiest to use the `Count` class as an example, and modify it.

Performance Test

Performance tests are used to figure out limitations of a system. The main characteristics of this system are it's real-time properties. If Pheme is going to be useful, it has to relay messages from components to a user's web browser within a reasonable time frame. To measure how well Pheme can accomplish this, we define a metric called *realtimey-ness*.

Realtimey-ness

This new metric is a ratio between a lower bound on the transmission time for a sequence of messages vs. the actual time it takes (i.e., lower bound/actual). With this ratio, we can remove the inherent constant latency in these systems, and focus on decreased responsiveness over time. A system that always lags by 20ms is fine, as long as the lag is not increasing.

In a perfect world, realtimey-ness is as close to one as possible. A value of one means the system is keeping up with the amount of information given to it. For example, if we send Pheme eleven messages at one second intervals, we should expect the last message to appear exactly ten seconds after the first. If the transmission takes longer than ten seconds, then Pheme was unable to keep up. This behavior correlates with a decrease in realtimey-ness.

Testing

With our new metric defined and validated, we need to put Pheme through some tests. In this section, we detail the testing environment, method, results, and analysis.

Environment The test bed is a late model Mac Mini with Ubuntu 13.04 installed. This machine has 16GB of ram, a 256GB SSD, and an Intel i7 CPU, which makes it relatively high end. With respect to software, the operating system has all the newest updates as of May 27th, 2013. Both the Pheme server and API were run on the same machine to reduce the effects of network latency. All other programs, save for a small music player, were disabled during the tests.

Specifications

- 2013 model Mac Mini
- 2.6GHz Intel i7
- 16GB RAM
- Ubuntu x64 13.04
- Pheme 1.1
- Pheme API 1.1.0
- JavaJDK 1.7
- Play Framework 2.1.1

Method For our experiments, we controlled the transmission time while varying the delay between messages. In particular, we told the client to send messages for ten seconds per delay variation. The client would wait three second before sending the next batch of messages. Here is the basic procedure:

1. Start up Pheme server.
2. Start up client.
3. Client initiates connection to server, waits three seconds for the connection to finalize.
4. Client runs `runTest` with the initial delay:

```java
private void runTest(int delay) {
    component.log("INFO", "Starting test with delay of " + delay);
    long total_count = TEST_RUN_SEC * 1000 / delay;
    for (int count = 0; count < total_count; count++) {
        component.count("Performance Count " + delay, 1);
        try {
            Thread.sleep(delay);
        } catch (Exception e) {
        }
    }
    component.log("INFO", "Finished test with delay of " + delay);
}
```

5. Within `runTest` the client first sends a log message. The log messages get time stamped by the server on arrival. They are used to calculate the actual transmission time.
6. The client then calculates how may messages should be sent in the ten second time frame.
7. For each message, the client sends a count to Pheme, adding one to the counter each time.
8. The client then sleeps for the delay interval.
9. Repeat 7 and 8 for `total_count`.
10. Finish that delay test with a final log message.
11. Repeat 4-11 for each delay needed.

Since the Pheme API is non-blocking, the messages are not actually done sending when the methods return. This allows us to ignore the time taken for each `count()` and `log()` invocation.

After every test completed, the log's time stamps were taken off Pheme's website and realtimey-ness was calculated.

Results and Analysis The results show how Pheme's real time attributes are affected by different message loads. The first test was conducted from a message delay of 50ms to 5ms in 5ms decrements (50, 45, 40...). This shows a healthy realtimey-ness of about 99% until delay gets down to about a 20ms. In other words, as long as a system does not average more than one message every 20ms Pheme has no trouble keeping up. When delay drops below 20ms, we see a sudden drop in realtimey-ness, however.

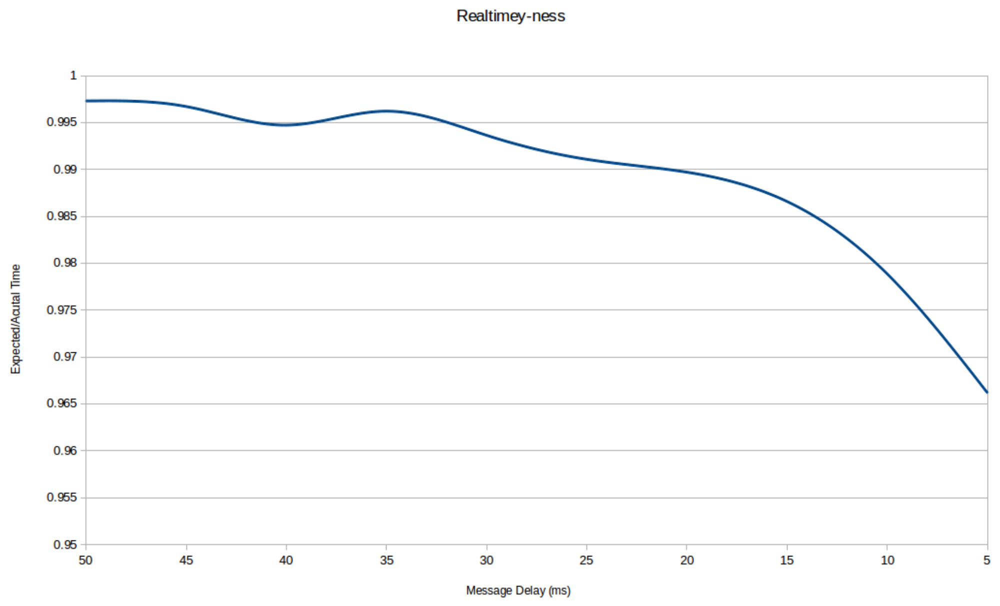


Figure 9: 50ms start delay, 5ms resolution

The second chart shows message delays from 19ms to 1ms with intervals of 2ms. In this range, Pheme starts to show some slowness. We start with 99% real time at 19ms, but end just below 94% at 1ms delay. While 94% isn't that bad, it's the logarithmic drop that warrants investigation.

Clearly, there is a bottleneck somewhere in Pheme. It may be on the client side: perhaps the sending mechanism is unable to send enough data at a time. Although the sender is buffered and it batch sends data, it also is limited to a maximum message count per batch. It is possible that increasing this limit would increase Pheme's performance.

Another possibility is a slowdown in Pheme's message processing. Pheme is multi-threaded. But, it needs to process the messages in order so they don't get mixed up. Only one thread is responsible for taking a message out of Pheme's receive buffer and committing it to memory. This thread may be getting overwhelmed, causing work to back up.

There are several plausible reasons why we see a decrease in realtimey-ness,

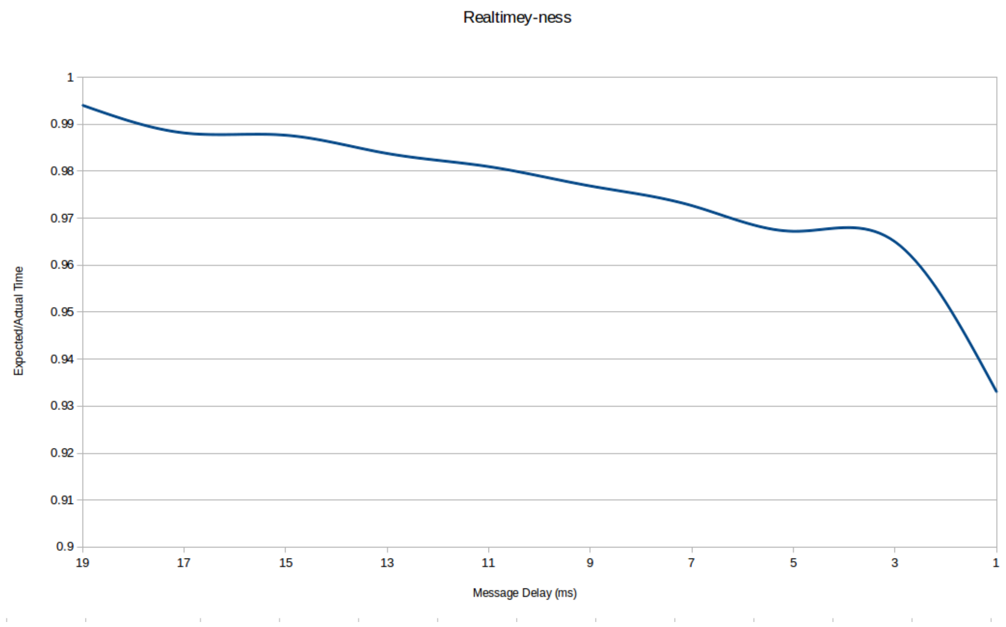


Figure 10: 19ms start delay, 2ms resolution

though further tests are required to know for sure. A test that measures the queue lengths of both the client and Pheme would be a good place to start. Also measuring the heap size of each JVM may provide more insight.

Future Work and Conclusions

To end this paper, we consider Pheme's future. While Pheme has met all the requirements we set for it, there are more features we want to see. Pheme can send several datatypes from clients to users. However, there is no reason why Pheme could not relay user input to clients. Through WAMP, we could send function invocations to remote clients, for instance, telling them to start up or shut down. This would give users unparalleled control over their systems. Aside from bidirectional communication, we would also like to add more adapters. Pheme could talk to Python, Haskell, or C++ clients. This could give Pheme a large market for adoption. Pheme also could use theming functionality. Developers who integrate Pheme in their projects may want to individualize the look to their project.

This wraps up this lengthy paper on Pheme. We discussed why Pheme exists, how to use it, and how to modify it. We also briefly discussed where Pheme could be going. We hope to see a wider adoption of Big Data in general. It will be fueling scientific discovery for the foreseeable future. We want to make these systems as accessible as possible. We hope that Pheme will facilitate that vision.

References

[A1] Hadoop http://hadoop.apache.org/. May 30, 2013

[A2] Storm https://github.com/nathanmarz/storm. May 30, 2013

[A3] jPregel http://charlesmunger.github.io/jpregel-aws/. May 30, 2013

[A4] Hadoop screenshot http://ostatic.com/files/images/Hadoop-web-interface-screenshot.png. May 30, 2013

[B1] Websockets http://www.websocket.org/. May 30, 2013

[B2] jQuery http://jquery.com/. May 30, 2013

[B3] Twitter Bootstrap http://twitter.github.io/bootstrap/. May 30, 2013

[B4] PubSubJS https://github.com/mroderick/PubSubJS. May 30, 2013

[B5] AutoBahnJS http://autobahn.ws/js. May 30, 2013

[B6] Require.js http://requirejs.org/. May 30, 2013

[B7] WAMP http://wamp.ws/. May 30, 2013

[B8] Google Guava https://code.google.com/p/guava-libraries/. May 30, 2013

[B9] WAMPlay https://github.com/blopker/WAMPlay. May 30, 2013

[B10] Play Framework 2.1 http://www.playframework.com/. May 30, 2013