TurkServer Documentation

Release 0.5.0

Andrew Mao

Nov 14, 2017

Getting Started

For a 2-minute introduction on the purpose of this project, see the *Introduction*.

This documentation is being updated continually and is subject to change. If you see anything missing, use the **Edit on GitHub** button at the top right of every page to suggest improvements.

This site is organized into a few sections:

- *Getting Started*
- *Examples*
- *System Architecture*
- *Building Experiments*
- *Running Experiments*

CHAPTER 1

Introduction

Since the 1950s, social scientists of all disciplines have been studying people in "behavioral labs" at universities. These labs typically involve cramming a group of people into a room for an hour or so and asking them to do something:

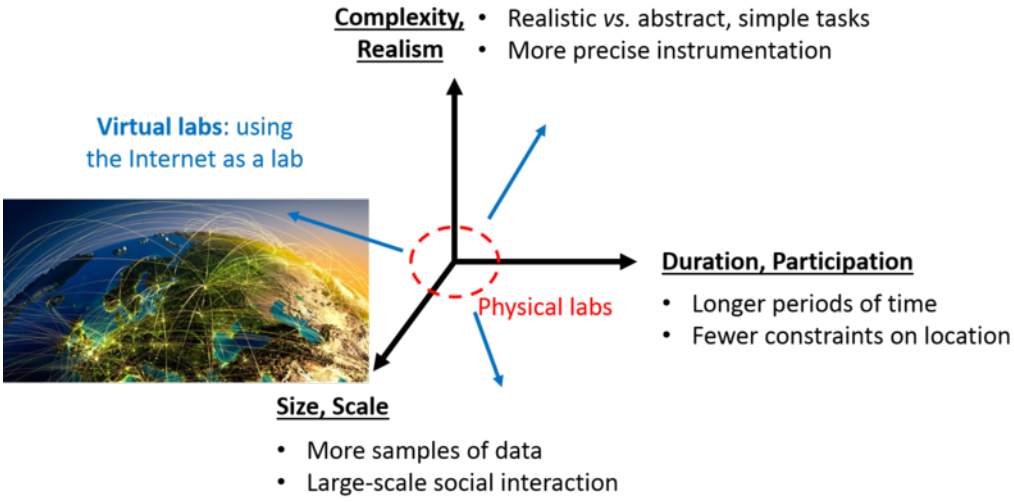
The lab is quite a useful tool in that it offers a great degree of control in doing experiments, and hence the ability to draw correct, causal conclusions. But there are clear limitations of using such an environment to study practically *all* of human behavior:

- it's a pretty **artificial environment**
- the setup generally supports only **simple tasks**
- subjects are **very homogeneous university undergraduates**
- we can only take a **limited number of people** and keep them there for a **short amount of time**
- lab experiments are **expensive**, **difficult to set up**, and **hard to replicate**

CHAPTER 2

Why the Virtual Lab?

The "Virtual Lab" refers to using software-controlled experiments with Internet participants to overcome many of the limitations of brick-and-mortar lab experiments. We can now study more complex tasks and set up interactions that happen over longer periods of time or among larger numbers of people. This allows us to design behavioral experiments that would have been very hard to do in the past.



2.1 What is TurkServer?

TurkServer is a platform for building software-controlled, web-based behavioral experiments using participants from the Internet. We can now study more complex tasks and set up interactions that happen over longer periods of time or among larger numbers of people. This allows us to design more expansive and realistic behavioral experiments.

By taking advantage of the movement toward the web as an all-purpose application platform, and providing a common system on which to build software-based behavioral experiments, TurkServer allows for:

- **Easier real-time programming** using the Meteor web app framework
- Building synchronous experiments and studying *group interaction*
- A live web-based *experimenter console* showing all connected participants
- The ability to create *one-way mirrors* to view behavior in real time

This means we can go from very artificial environments to studying realistic, complex tasks such as teamwork and collective intelligence:

2.2 Sharing and Replication

One significant annoyance of doing experiments is that they're **a lot of work**, yet **this work is often redundant**: it is either not shared or involves reinventing the wheel. So, other than just supporting more advanced experimental designs, another goal of TurkServer is to lower the barrier to both producing and sharing experimental research.

We can do this by making everything **open-source software**: not only is the core platform open-source, but experiments built on it are as well. This means that researchers can now share not only data, but entire **experimental protocols**. For example, check out the code for the teamwork experiment above.

In this way, building off someone else's experiment means you don't have to implement it from scratch: just grab their code and use it!

2.3 Next Steps

Ready to jump in? Check out the *quick start*, a basic (but fully functional) *tutorial app*, or read up on how to think about *designing experiments with TurkServer*.

CHAPTER 3

Quick Start

This quick start will guide you to step by step, from install and run a local instance to deploy and test your first Turkserver app in real world.

Example project: Tutorial. For more information, regarding how to build this app, see here.

3.1 *Background*

TurkServer is a framework based on the JavaScript app platform Meteor that makes it easy to build interactive web-based user experiments for deployment on Amazon Mechanical Turk.

This tutorial will explain how to get up and running with TurkServer. The intended audience is someone who:

1. **Is familiar with Amazon Mechanical Turk** and understands how to request and approve work. If you have never used Mechanical Turk before, we recommend you explore a bit first. Go here and create one of the example projects. xs
2. **Knows the basics of the JavaScript app platform Meteor.** If you have never used Meteor before, that's fine – it's quite easy to learn. Before continuing with this tutorial, install Meteor and then complete the to-do app tutorial.

TurkServer is designed around the Meteor framework. There are many reasons that Meteor is especially powerful for web development; see this post for a summary. But more importantly, there are tons of learning resources for new users to get started. The design philosophy of TurkServer is to stick to standard Meteor as much as possible, while minimizing the need to use custom APIs. This means that most of the outstanding Meteor documentation on the Internet will be useful, and that most of the required knowledge is not specific to TurkServer.

All the code for the project that you will build during the tutorial can be found here. If you have questions, comments, or suggestions, please add a Github issue to that repo.

3.2 *The Basics*

Before getting started, it will help to define some terminology used by TurkServer. Many of these concepts are from Mechanical Turk, and should therefore be familiar to you, but others will be new.

- **Worker**: A Mechanical Turk user.
- **HIT**: A task that workers can complete on Mechanical Turk. A HIT can be completed by multiple workers, but each worker can only complete a particular HIT once.
- **External Question**: A HIT with an external question is hosted on your *own website* and displayed in an iframe in the Worker's web browser. TurkServer facilitates the process of building external HITs.
- **HIT Type**: A way to define the properties of a HIT, such as its title, description, and monetary reward. Each HIT has exactly one HIT type, but you can create multiple HITs with the same HIT type (e.g. you might post a HIT on Monday and receive some data, and then post another HIT of the same HIT type on Tuesday to receive additional data).
- **Batch**: Whereas a HIT Type is Mechanical Turk's way of grouping HITs together, a batch is TurkServer's way of grouping HITs together. The HIT Type determines *Mechanical Turk properties* of a HIT; the batch defines *TurkServer properties* of a HIT. When you create a HIT type in TurkServer, you must choose a batch for it. All HITs with that HIT type then belong to that batch and have the corresponding properties.
- **Assignment**: When a worker accepts a HIT, an assignment is created. The assignment keeps track of the work that the worker does on this HIT. You can approve or reject the worker's assignment after he submits the HIT.

In using TurkServer, every Assignment of a HIT will go through one of three states (not necessarily in a linear fashion):

1. **Lobby** – Where a worker goes right after accepting the HIT, and also between participating in experiment instances.
2. **Experiment Instance** – Where the worker actually completes the task. Experiments are a logical way of setting up interaction between a group of workers and can consist of any number of workers, including just one.
3. **Exit Survey** – Where the worker submits the HIT.

   Each participant can be assigned to one instance at a time, but can participate in multiple instances over the course of a HIT, so that each instance can have one or more simultaneous participants. This supports various synchronous and longitudinal experiment designs.

The developer using TurkServer is responsible for building the user interface for each of these stages. In other words, TurkServer assumes you already have a Meteor app that controls the experience workers have after accepting your HIT and before submitting it. TurkServer is just the middleman between your app and Mechanical Turk. The framework will take care of connecting workers to your app when they accept the HIT, and submitting their work to Mechanical Turk when they are done with your HIT. TurkServer also allows you to easily monitor your experiment while in progress. But you first need to build the app that allows workers to complete the desired task.

Haven't build the app yet? Don't worry, we will get you started with one of our basic example projects - Tutorial.

So let's start the journey now! (Click the "Next" button on the right)

3.3 *Install*

1. **Git**

   Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

   How to install? See here.

Using GitHub as the install source is temporary until the Turkserver package reaches "beta".

2. **Meteor**

   TurkServer is built on top of Meteor.

   Meteor.js is a cohesive development platform, a collection of libraries and packages that are bound together in a tidy way to make web development easier.

   Meteor supports OS X, Windows, and Linux, and is simple to install.

```
| OS      | Version              | Status |
|---------|----------------------|--------|
| Windows | 10                   |        |
| Windows | 8.1                  |        |
| Windows | 7                    |        |
| Windows | Server 2012          |        |
| Windows | Server 2008          |        |
|  Linux  | X86                  |        |
|  Linux  | x86_64               |        |
|   OSX   | 10.7(Lion) and above|        |
```

   On Windows?

   Download the official Meteor installer

   On OS X or Linux?

```
curl https://install.meteor.com/ | sh
```

3.4 *Getting Started*

1. **Checkout the project**

```
git clone https://github.com/TurkServer/tutorial.git
```

2. **Create settings file**

```
cd tutorial
```

   Under the root of your project, create a `settings.json` file, follow `settings-template.json`.

   Fill in "adminPassword", "accessKeyId", and "secretAccessKey".

   (**Note**: To retrieve your Amazon Mechanical Turk `accessKeyId` and `secretAccessKey`, you need sign up AWS Account, see here)

   The file should look like this:

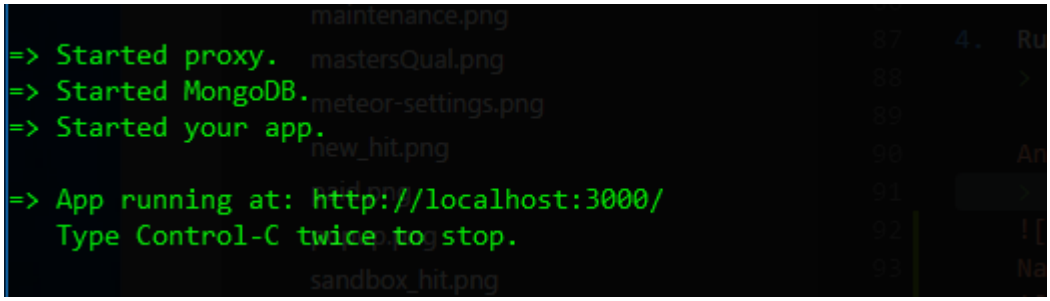
```
{
  "turkserver": {
    "adminPassword": <String>, // Password you choose for the TurkServer admin␣
↪deshboard
    "experiment": {
      "limit": {
        "batch": <Number> // Max number of times a user can accept HITs in one␣
↪batch
      }
    },
    "mturk": {
      "sandbox": <Boolean>, // Flag to turn on and off for Amazon Mechanical Turk␣
↪Sandbox test
```

```
      "accessKeyId": <String>, // Amazon Mechanical Turk account accessKeyId
      "secretAccessKey": <String>, // Amazon Mechanical Turk account␣
↪secretAccessKey
      "externalUrl": <String>, // Root url of your app
      "frameHeight": <Number> // Height of the iframe in M-Turk
    }
  }
}
```

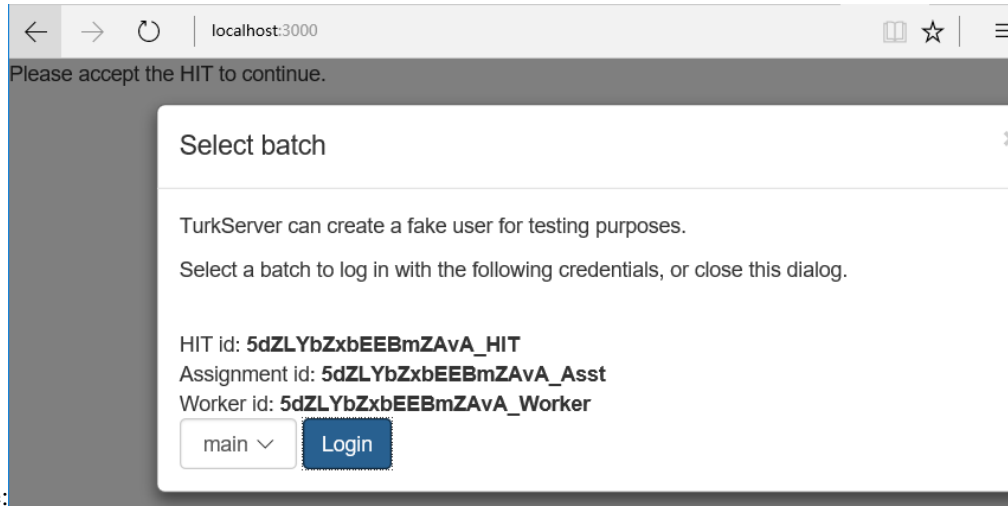
3. **Run your app**

```
meteor --settings settings.json
```

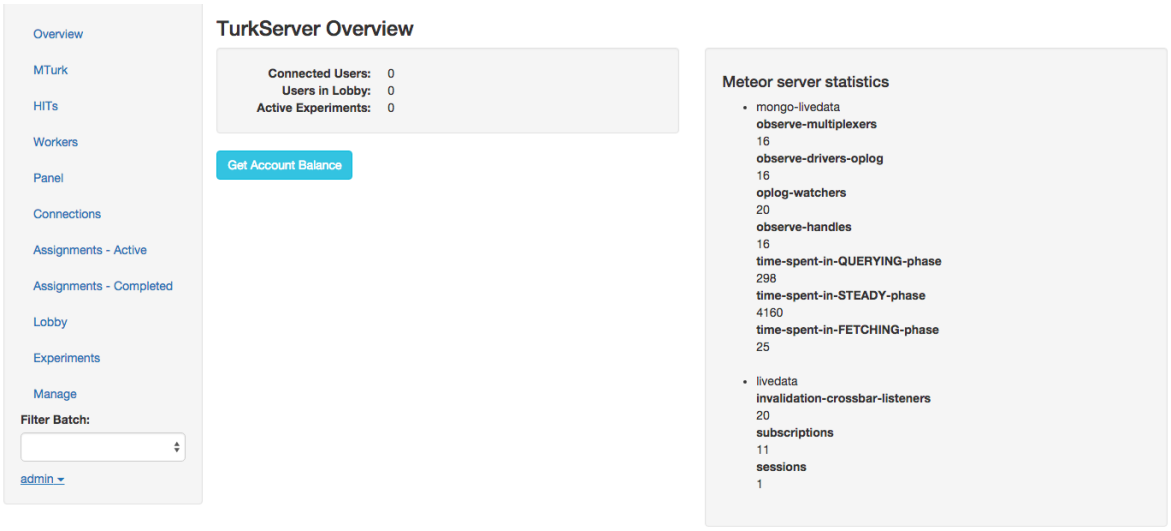
In your console, you should see:



Open your browser, you should see:



Navigate to via http://localhost:3000/turkserver, login with your password, enter the admin dashboard:



3.5 *Deployment*

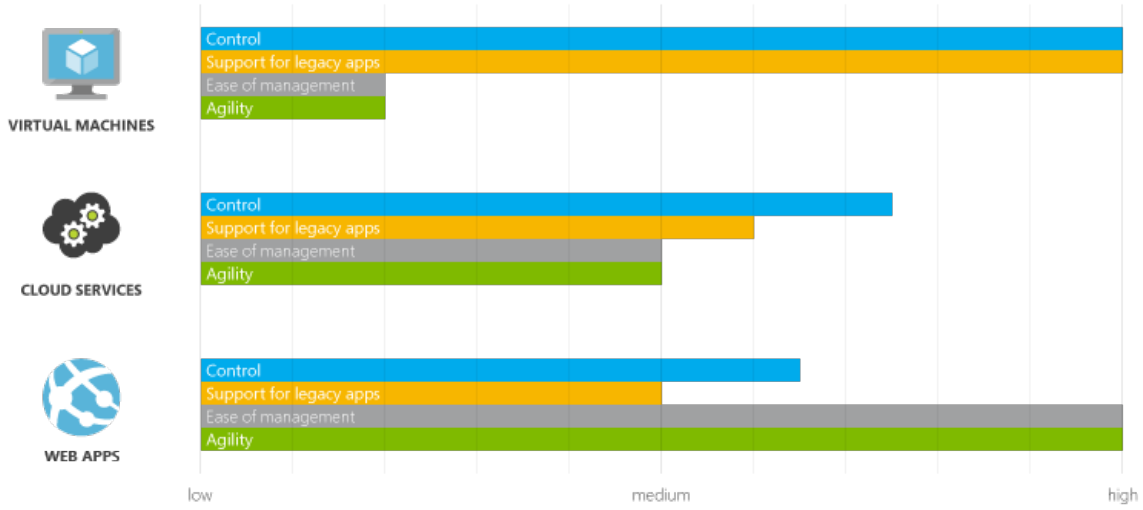
To deploy your meteor app in cloud, there are many choices.

Here we are going to use Microsoft Azure as example to show you how easily you can deploy your app in a cloud.(*Free one-month trial: sign up for free and get $200 to spend on all Azure services)

3.5.1 Understand Azure

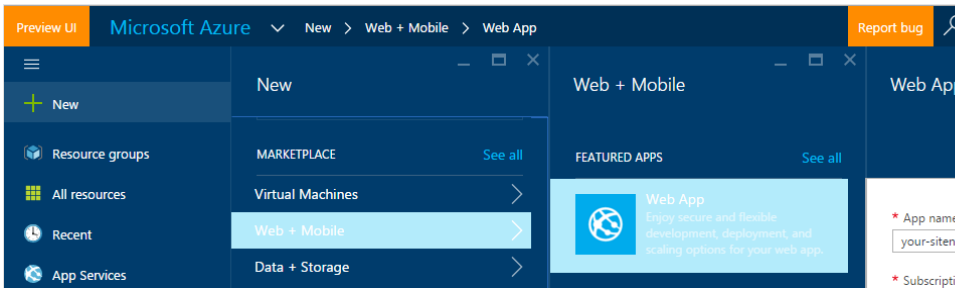
Azure offers several ways to host web apps:

- App Service (Very beginner-friendly, less manual)
- Cloud Services (Beginner-friendly, and flexible)
- Virtual Machines (Less beginner-friendly, but much more flexible).
- Service Fabri (Less beginner-friendly, microservice-based application development)



For more detailed comparison, see here.

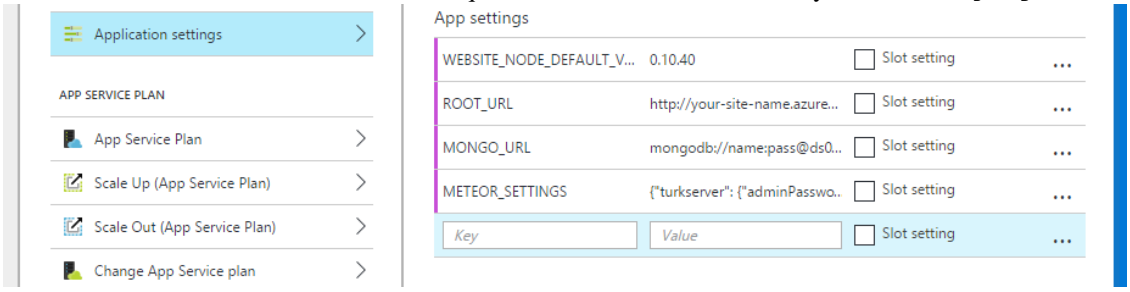
3.5.2 Deploy to Azure App Service



1. **Build App Service using Azure Portal**
   Or try it here. (Note: Please don't forget to set your `deployment credentials`.)
2. **Configure App Service**

   Add the following enironment variables in your Application Settings:

   - WEBSITE _ NODE _ DEFAULT _ VERSION : `0.10.40`
   - ROOT _ URL : `http://{sitename}.azurewebsites.net` or your custom domain if you've set that up
   - MONGO _ URL : (Mongo DB connection string from a MongoDB hosted on MongoLab or a VM)
   - METEOR _ SETTINGS : JSON value equals to the entire contents of your `settings.json` file



3. **Deploy**

   Azure has native support for Node.js app, Azure App Service hasn't enable native support for Meteor, you can track insider here.

   However, we can still deploy to Azure App Service by **demeteorizer** - Which is a great tool to convert a Meteor app into a "standard" Node.js application.

   In this way, you can take advantage of the native support with features like `high availability with auto-patching`, `built-in autoscale` and `load balancing`.

   For **Windows** users, we recommend to use Azure-demeteorizer. Make sure you install and meet all the pre-requisites first, (Note: Visual Studio 2013 would be the most comaptible choice for most users.) then Azure-demeteorizer can do all the magic for you:

   (a) `npm install -g christopheranderson/azure-demeteorizer` to install azure-demeteorizer globally via npm:

   (b) 'cd tutorial

   (c) `azure-demeteorizer build`

   (d) `azure-demeteorizer install`

   (e) `azure-demeteorizer zip`

   (f) `azure-demeteorizer deploy -s [sitename] -u [username] - p [password]` - sitename: the name of your App. - username: username for your site's [deployment credentials]. - password: password for your site's [deployment credentials].

   (Note: For a full respect of your changes in **Continuous Deployment**, you need stop server during step 6.)

   For all **Linux/Mac OSX/Windows** users, you can use Demeteorizer, however, you have to demeteorize first, and meet the meteor install requirements (Use Below command to install meteor on local environment: "curl https://install.meteor.com/ | sh" Then, Check for your meteor version. we highly recommend using > 1.4), then you can do a continuous deployment with Git, TFS, GitHub, or Visual Studio Team Services (Here use Git as example):

   (a) `$ npm install -g demeteorizer` to install Demeteorizer globally via npm

   (b) `$ cd tutorial`

   (c) `$ demeteorizer`

   (d) `$ cd .demeteorized/bundle/programs/server`

   (e) `$ npm install` (Please note: Use npm > 3 if you are getting any long path issues.. Use nvm to easily change node/npm version in local environment.)

   (f) `$ MONGO_URL=mongodb://localhost:27017/test PORT=3000 ROOT_URL=http://localhost:3000 npm start` If this is successful, then you can continue the step using `git remote` to push your app to Azure. (If you didn't choose continuous deployment using Git when you build your Azure App Service, you can manually set it up here). Don't forget to add below app setting to your web app inside Azure portal App Settings Key: ROOT_URL Value: web app url(ex: http://{appname}.azurewebsites.net/) Then, create a web.config file under .demeteorized/bundle/ and insert this link content. Now you can navigate to .demeteorized/bundle/ folder and Commit your changes:

   (g) `$ git init`

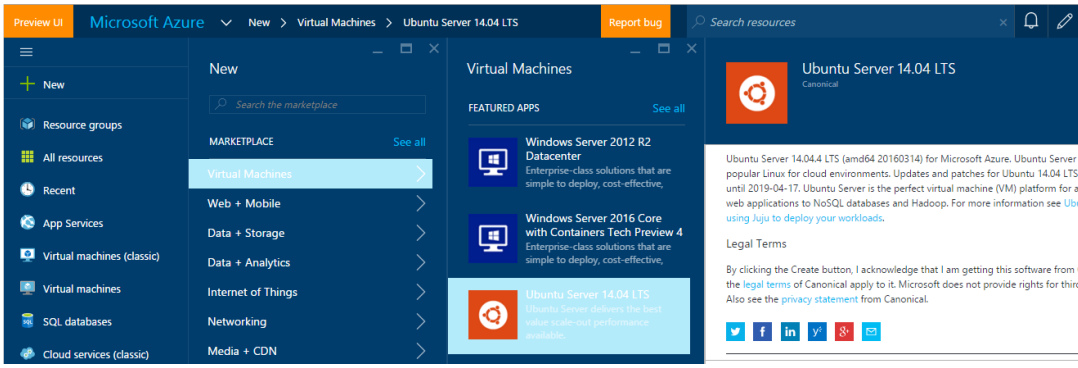
   (h) `$ git remote add azure https://username@{sitename}.scm.azurewebsites.net:443/{appname}.git`

   (i) `$ git add .`

(j) `$ git commit -am "Initial Commit"`

(k) `$ git push azure master`

3.5.3 Deploy to Azure Virtual Machine



1. **Build VM by Azure Portal**
   Or try it here. (Note: Please don't forget to set your deployment credentials.)

2. **Deploy**

   If you build Windows VMs, (i.e. Windows Server 2012) please check previous chapter **"Deploy to Azure App Service"** about how to demetoerize and deploy your app to Azure.

   If you build Linux VMs, (i.e. Ubuntu 14.04 LTS) you can use Mupx, please continue the following steps:

   (a) `npm install -g mupx` to install mupx globally via npm

   (b) `cd tutorial`

   (c) `mupx init` to generate or manually add:* mup.json - Meteor Up configuration file, see exmpale here * settings.json - Settings for Meteor's settings API

   (d) `mupx setup`

   (e) `mupx deploy`

3.5.4 Deployment Choices

Client OS	Server OS	Solutions(Recommended)	Status
Windows	Windows	Azure-demeteorizer	
Windows	Linux	Mupx	
Linux	Windows	Demeteorizer	
Linux	Linux	Mupx	

***Note**: For other deployments on Modulus, DigitalOcean, Galaxy, please check the official deployment book here.

3.6 *Test*

This step is showing you how to do your sandbox test in Amazon Mechanical Turk, according to Amazon payment requirements: `ExternalURL` - The URL of your web app, to be displayed in a frame in the Worker's web browser.

This URL must use the **HTTPS** protocol. It might take some time for you to setup your own SSL server if you are not using Azure, however, we strongly recommand you complete the test before actually make your project go live.

3.6.1 Enable SSL

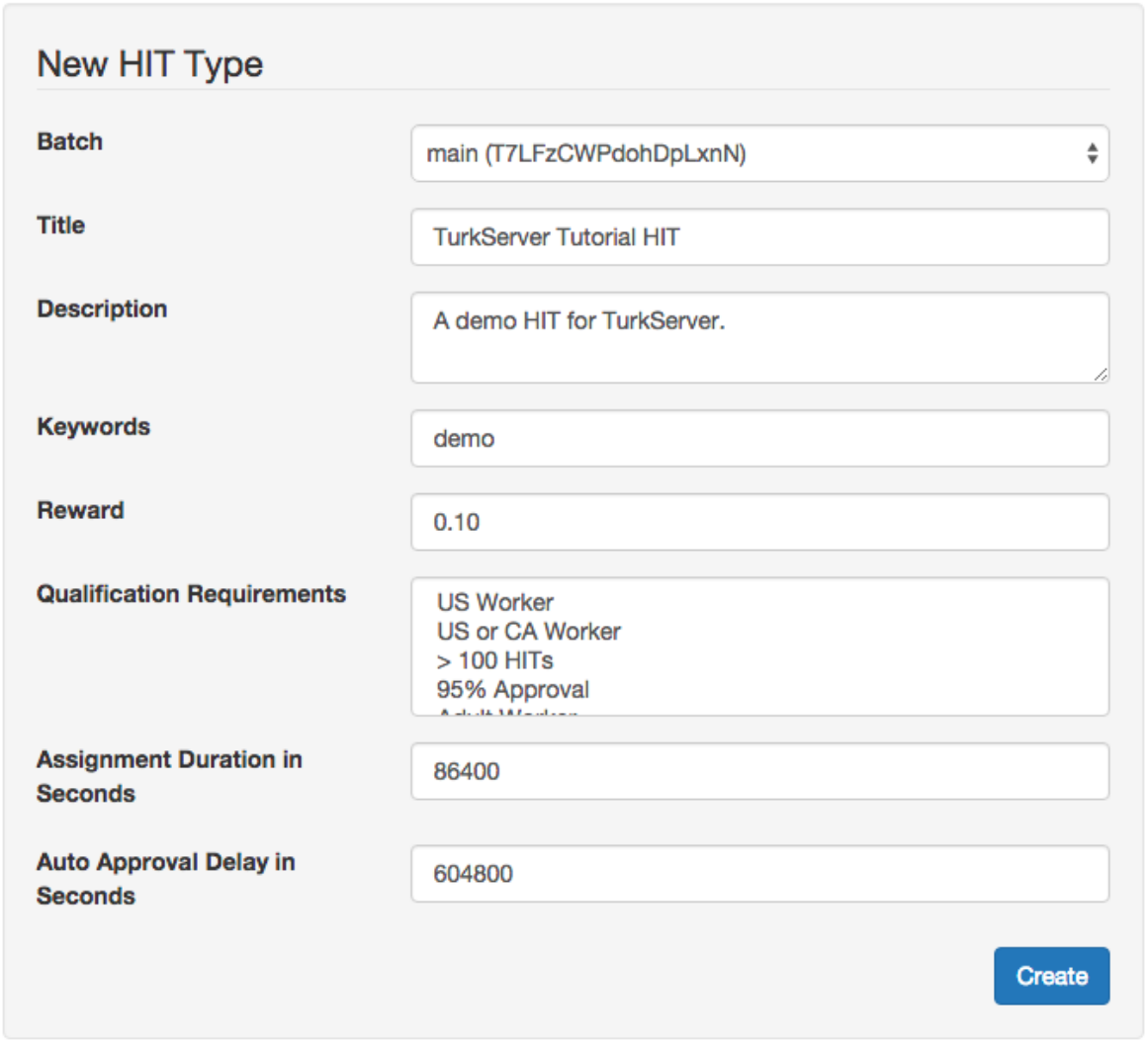
- If you have deployed your app to Azure App Service:(Recommend for beginners)
  - By default, Azure already enables HTTPS for your app with a wildcard certificate for the *.azurewebsites.net domain. If you don't plan to configure a custom domain, then you can benefit from the default HTTPS certificate. Read more here
  - For a better performance and connectivity, you can turn on websockets instead of long-polling in your App Service, simply by one-click in `Application settings` through Azure portal.
- If you have deployed your app to VMs using Mupx:
  - You can add SSL support following here, however, you have to obtain your own SSL certificate from a certificate authority (CA) and a private key first.
- For other deployments, you might set up your SSL manually.

3.6.2 Sandbox Test

If you have deployed the `tutorial` app and enabled SSL successfully in a public server. Then you can follow this step to do a sanbox test in Amazon Mechanical Turk.

1. **Create HIT Type**

   Click on "MTurk" in the navigation pane. Create a new HIT type in the "New HIT Type" form:

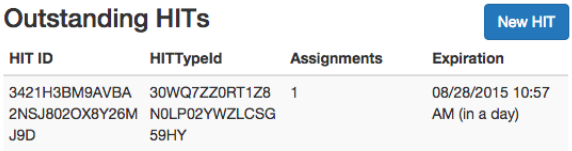


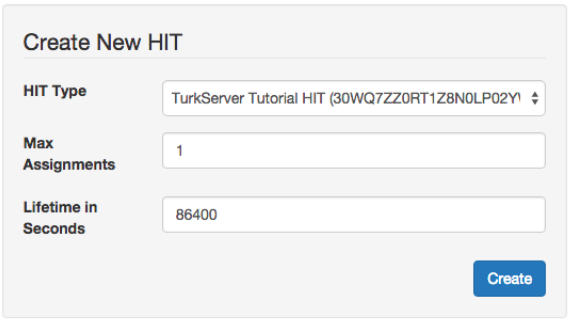
Note: Don't select any qualifications right now, because we're just testing on the sandbox.(Plus, most qualifications have different ids on sandbox than production, so it wouldn't work properly anyway.)

When you're done, click "Create". On the following screen, you should see a button that says "Register." Click that to register your new HIT type with Mechanical Turk.

2. **Create HITs**

Click on "HITs" in the navigation pane, select the HIT type you just created in the "Create New HIT" form:

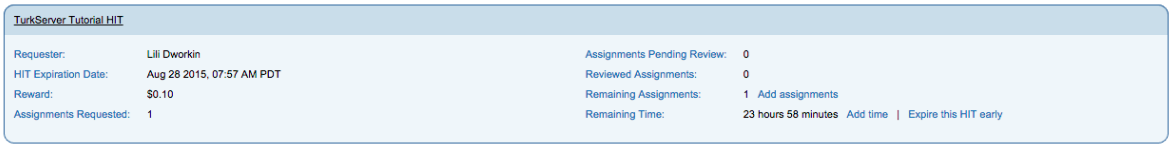




When you're done, click "Create".

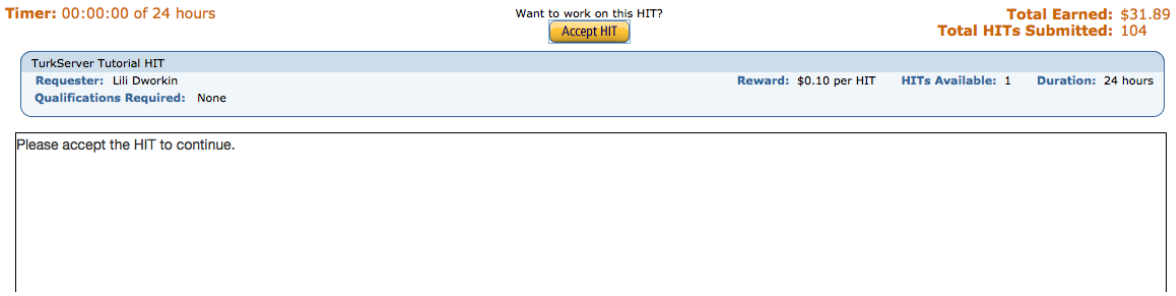
3. **View As Requester**

   Now go to the Amazon MTurk requester sandbox to check that your HIT was posted successfully. Click on "Manage" in the blue toolbar, and then click on "Manage HITs individually" in the top right. You should see your HIT:



4. **View As Worker**

   To test this as a sandbox worker, go to the worker sandbox *in a different browser or Chrome incognito tab*. Find the HIT you just posted, click "View a HIT in this group". If all went well, your app will be loaded into an iFrame at the default route `'/'`, which shows the "home" template and prompts you to accept the HIT:

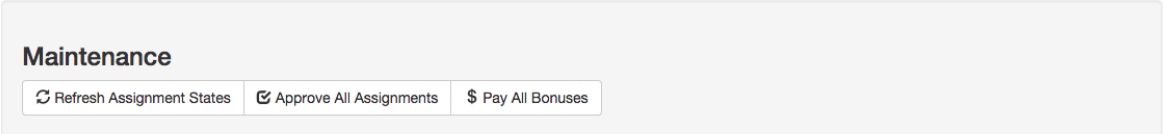


   Click the "Accept HIT" button. You should now be put into an experiment, which shows the content under `'/experiment'` route in your app.

5. **View As Admin**

   As the admin, the next thing you should do is check on the Mechanical Turk status of this assignment – is it "submitted" and/or "approved"?

   Click the "Refresh Assignment States" button in the "Maintenance" well at the top of the page:



   The **Status** field of the assignment should now change from "Unknown" to "Submitted":

This means that Mechanical Turk knows that this assignment has been completed, but it has not yet been approved. We set the "Auto Approval Delay in Seconds" property of our HIT Type to a whole week, so we need to manually approve this assignment if we want it approved before then. Click the "Approve All Assignments" button in the "Maintenance" well to do this. You'll see a popup that asks you to confirm – feel free to leave the message blank, and hit "Ok." The **Status** field of the assignment should now change from "Submitted" to "Approved."

6. **Complete**

   Now that the assignment is approved, we can pay the worker's bonus.

   Click the "Pay All Bonuses" button in the "Maintenance" well. You'll again see a popup that asks you to confirm – you do need to enter a message here, which will included in the email that Mechanical Turk sends to the worker to notify him of the bonus. When you click okay, you'll see that the **Bonus** field of the assignment now has a "Paid" label next to it:

Assuming your worker sandbox account is tied to a real e-mail address, you should soon receive the corresponding notification e-mail.

Congratulations! – You have successfully posted a HIT, completed it, and paid yourself for doing so.

3.7 *Ready to launch*

Congratulations, you made it!

Now you can modify the demo, turn off sandbox in settings and launch your own experiments in the real world.

Enjoy your new research experience using TurkServer!

CHAPTER 4

API Documentation

The TurkServer API is built automatically from the code and can be accessed here.

CHAPTER 5

Examples

5.1 Basic Examples

The basic *tutorial* describes how to build a simple, but full fledged example app (see source code.) Take a look at this app as a working starting point for something you can deploy online.

5.2 Research Projects

Although studies of individual behavior are desirable and easy to build, TurkServer really shines for studying groups, especially for complex interactions or social behavior among many people over time. For ideas, inspiration, or just examples, take a look at some *research projects* that have been built using TurkServer.

CHAPTER 6

Basic Tutorial

This tutorial walks you through building a simple, but fully functional app that you can deploy and that demonstrates many of the features of TurkServer. If you'd prefer to start hacking from a working example, you can start poking around the code directly and follow the *deployment instructions* to get it online.

> Much of this fantastic tutorial is due to the excellent work of Lili Dworkin.

6.1 Getting Started

For the purposes of this tutorial, we will use the default Meteor "counter" app that comes bundled with every installation. To generate the necessary files, simply type `meteor create app`. To run the app, `cd app` and `meteor`. Then open a browser and navigate to `http://localhost:3000/` to verify that all is working properly. You should see the following:

Before moving on, don't forget to `meteor remove insecure` and `meteor remove autopublish`.

For easiest integration with TurkServer, you will need to add some basic routing to your app. This how you connect different urls ("routes") to different templates. We will start with two routes:

1. The default path `/`, which will get shown when a worker is previewing your HIT.

2. A second path `/experiment`, which will get shown after the worker accepts the HIT.

To do this, first add the Meteor package iron-router via `meteor add iron:router`. Then edit the block of HTML at the top of demo.html to look like this:

```
<template name="home">
    Please accept the HIT to continue.
</template>

<template name="experiment">
  <head>
    <title>demo</title>
  </head>

  <body>
  <h1>Welcome to Meteor!</h1>

  {% raw %}{{> hello}}{% endraw %}
  </body>
</template>
```

We have done two things here: we wrapped the existing HTML in a template called "experiment", and we added another template called "home." We want to show the "home" template when a worker is previewing the HIT, and the "experiment" template after they have accepted.

Next create a new file called routes.js and add the following to it:

```
Router.route('/', function() {
  this.render('home');
});

Router.route('/experiment', function() {
  this.render('experiment');
});
```

For a more detailed explanation of this code and a better understanding how routing works, see the iron-router docs. Essentially, we have told the app that when we navigate to the default route `'/'`, we should see the "home" template, and when we navigate to the route `'/experiment'`, we should see the "experiment" template.

Now we are ready to add the TurkServer package:

```
meteor add mizzao:turkserver
```

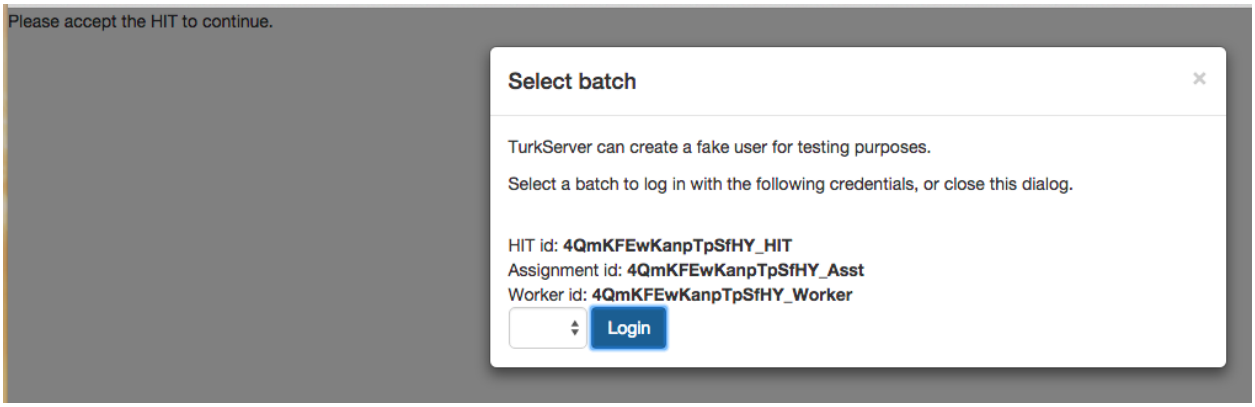
Next create a settings.json file, which needs to contain a password for the TurkServer admin interface (from which you will monitor your experiment) as well as your Amazon key and secret pair. (If you don't have those, or don't remember where to find them, go here). The file should look like this:

```
{
  "turkserver": {
    "adminPassword": "",
    "experiment": {
      "limit": {
        "batch": <int>
      }
    },
    "mturk": {
      "sandbox": <true|false>,
      "accessKeyId": "",
      "secretAccessKey": "",
```

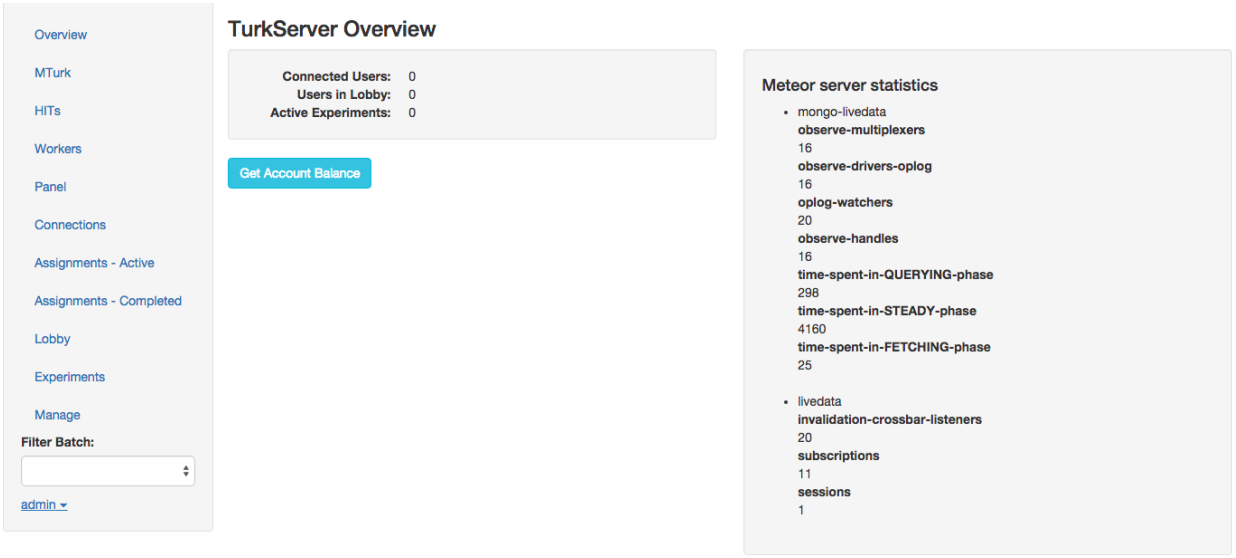
```
      "externalUrl": "",
      "frameHeight": <int>
    }
  }
}
```

For now, fill in "adminPassword", "accessKeyId", and "secretAccessKey". We'll come back to the other parts later.

Finally, run your app with `meteor --settings settings.json`. When you navigate to `http://localhost:3000/` (the default route `'/'`), you should see your "home" template (which prompts you to accept the HIT), and now you should also see a popup that prompts you to select a "batch:"



Don't worry about this for now. Navigate instead to `http://localhost:3000/turkserver` and login with the password you put in your settings file. You should see the following admin interface:



Feel free to explore a bit, but most of what you see won't make sense just yet.

6.2 Building the App

As discussed earlier, TurkServer assumes that every HIT consists of a **lobby**, **experiment(s)**, and **exit survey**. Typically a user will proceed through these stages in the following order: lobby -> experiment -> lobby -> exit survey. You

specify the exact order you want by using an **assigner**. TurkServer comes with a few built-in assigners, but you can also write your own. We'll briefly describe two of the built-in assigners: the "TestAssigner" and the "SimpleAssigner."

First, note that matter what assigner you use, the following two rules are guaranteed:

- When a user accepts the HIT, he goes into the lobby.
- When a user finishes an experiment, he goes back to the lobby.

The **TestAssigner** enforces the following additional rules:

- When a user goes into the lobby for the first time (i.e. after he accepts the HIT), he immediately gets put into a *joint* experiment with all other users who have also accepted the HIT.
- When a user goes to the lobby for the second time (i.e. when he is done with the experiment), he immediately gets put into the exit survey.

The **SimpleAssigner** is similar, except that when a user goes into the lobby for the first time (i.e. after he accepts the HIT), he immediately gets put into his *own* experiment. So this assigner is appropriate for single-user HITs. We will be using the SimpleAssigner is the remainder of the tutorial.

To make this flow work properly, we need to glue each of the two possible states (experiment and exit survey) to a different template in our application. (Note: We don't need to worry about making a template for the lobby, because the SimpleAssigner never lets the user stay there for long – he is always immediately redirected to the experiment or exit survey. If you write an assigner that makes users wait in the lobby, you will need to design a template for it.)

Let's assume that when a user is in the experiment, we want him to see the "Click Me" page. And when the user is in the exit survey, we want him to see a submit button that will allow him to submit the HIT. We already have the "Click Me" page (in the "experiment" template), so let's design the exit survey.

First, we'll create the template. Add a block at the end of demo.html that looks like this:

```
<template name="survey">
  <button>Submit the HIT</button>
</template>
```

To add functionality to this button, add the following block to demo.js:

```
Template.survey.events({
  'click button': function () {
    TurkServer.submitExitSurvey({});
  }
});
```

We have now used the TurkServer API for the first time. The function `submitExitSurvey()` will automatically submit a HIT to Mechanical Turk. The function takes arbitrary JSON as an argument, so you can pass in any data you want. The data will viewable in the TurkServer admin console in real time. A natural thing to do is put a form in your exit survey that asks the worker some follow-up questions about the HIT. For example, modify your template as follows:

```
<template name="survey">
  <form class="survey">
    <p>Was there anything confusing about this HIT?</p>
    <textarea name="confusing" rows="3"></textarea><br/>
    <p>Do you have any feedback for us?</p>
    <textarea name="feedback" rows="3"></textarea><br/>
    <button type="submit">Submit the HIT</button>
  </form>
</template>
```

And modify the events code accordingly:

```
Template.survey.events({
  'submit .survey': function (e) {
    e.preventDefault();
    var results = {confusing: e.target.confusing.value,
                   feedback: e.target.feedback.value}
    TurkServer.submitExitSurvey(results);
  }
});
```

Note that we changed the event from a button click to a form submission, so we need to use `e.preventDefault()` to prevent the browser from automatically handling the event.

Add the following block to routes.js:

```
Router.route('/survey', function() {
  this.render('survey');
});
```

Finally, we need to tell our app that when a user is in the experiment state, we should load the `'/experiment'` route (which is connected to the "experiment" template), and when a user is in the exit survey state, we should load the `'/survey'` route (which is connected to the "survey" template). Add the following inside the `Meteor.isClient` block at the top of demo.js:

```
Tracker.autorun(function() {
  if (TurkServer.inExperiment()) {
    Router.go('/experiment');
  } else if (TurkServer.inExitSurvey()) {
    Router.go('/survey');
  }
});
```

There are a few things worth noting about this code. We have made use of two boolean reactive variables provided by TurkServer: `TurkServer.inExperiment()` and `TurkServer.inExitSurvey()` (unsurprisingly, there is also `TurkServer.inLobby()`). When the state of the user changes, the value of these variables will update accordingly. The `Tracker.autorun` block will re-run whenever the variables change, which means that whenever the state of a user changes, our app will load the appropriate route, and therefore show the appropriate template.

> `Tracker.autorun` is the standard reactive programming approach in Meteor. If you are confused about this, take a minute and read this.

There is still one missing piece. We know that when a user accepts the HIT, he goes to the lobby, and then the SimpleAssigner puts him into an experiment. And we know that when the experiment ends, he'll go back to the lobby, and the SimpleAssigner will put him into the exit survey, where he'll be able to submit the HIT. But how do we let the user end his experiment? Let's add a button on the "Click Me" page that will allow the user to move to the exit survey when ready. Edit the "hello" template in demo.html to look like this:

```
<template name="hello">
  <button id="clickMe">Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>

  <button id="exitSurvey">I'm Done</button>
</template>
```

Note that we added an id attribute to both buttons so that we can distinguish between them in the events code. Now edit the `Template.hello.events` block in demo.js as follows:

```
Template.hello.events({
  'click button#clickMe': function () {
```

```
    // increment the counter when button is clicked
    Session.set('counter', Session.get('counter') + 1);
  },
  'click button#exitSurvey': function () {
    // go to the exit survey
    Meteor.call('goToExitSurvey');
  }
});
```

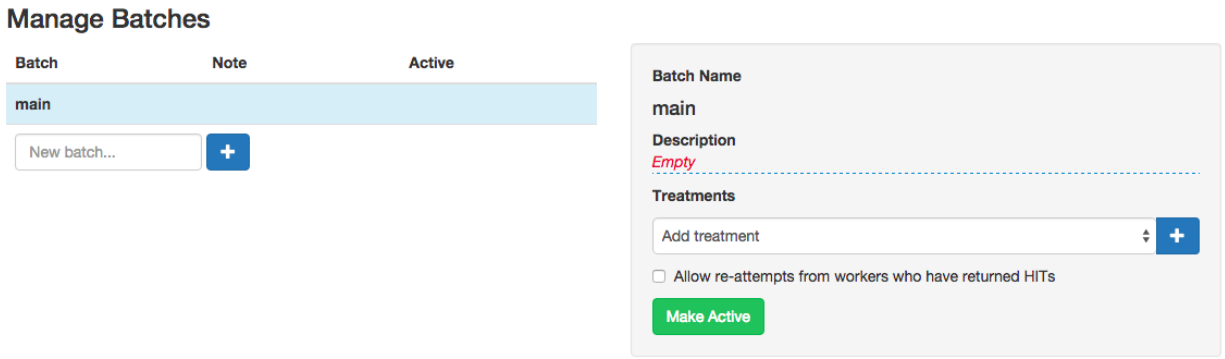
And define the new method `goToExitSurvey()` within the `Meteor.isServer` block (but outside of the `Meteor.startup()` block) at the bottom of demo.js:

```
Meteor.methods({
  goToExitSurvey: function() {
    var exp = TurkServer.Instance.currentInstance();
    exp.teardown();
  }
});
```

In our new method `goToExitSurvey()`, we first grab the JavaScript object corresponding to the user's current experiment. (TurkServer uses the terms "instance" and "experiment" interchangeably; I will try to stick to "experiment.") Then we call the `teardown()` method of this object, which ends the experiment. So this point the user goes back to the lobby, and then the SimpleAssigner sends him to the exit survey, so he can submit the HIT.

We've talked a lot about the SimpleAssigner, but we haven't actually yet told our app that we want to use it. To do so, we need to create a **batch** for our HIT, and then add the SimpleAssigner to this batch. Recall that a batch is a way of defining TurkServer-specific properties of our HIT, and the assigner type is one of these properties.

We can create a batch via the admin interface. Go back to `http://localhost:3000/turkserver`. Click on "Manage" at the bottom of the left sidebar. Under the "Manage Batches" header, type the name "main" into the textbox, and then hit the blue "+" button. You should see a row appear above that says "main." Click on it. In the box that shows up on the right, click the green "Make Active" button.



To add the SimpleAssigner to your new "main" batch, add the following code within the `Meteor.startup` block of the `Meteor.isServer` block at the bottom of demo.js:

```
var batch = TurkServer.Batch.getBatchByName("main");
batch.setAssigner(new TurkServer.Assigners.SimpleAssigner);
```

A few more TurkServer API calls here. First we grab the JavaScript object corresponding to the batch that we created. Then we call its method `setAssigner()` to tell our app that all HITs in this batch should use the SimpleAssigner.

6.3 Running as a Fake User

Now that we have a batch and an assigner, our HIT flow will work properly, so we can finally test it all out. TurkServer allows you to login as a fake Mechanical Turk user, which is very useful for testing without having to create HITs on your MTurk sandbox account. Navigate to `http://localhost:3000/` and you'll see the popup from before. TurkServer is asking us which batch we want to test out. We only have one ("main"), which should be selected by default.

The popup also provides some other information: TurkServer has generated a fake HIT id, assignment id, and worker id for us. What's happening is that TurkServer is "pretending" that:

1. We have created a HIT belonging to the batch "main" and posted it on Mechanical Turk.
2. A worker (you) has accepted this HIT, and therefore a new assignment has been created.

Click "Login" to set this whole process in motion.

After "accepting" the HIT, you go straight into the lobby. The code sees that you are in batch "main", and that the assigner on this batch is a SimpleAssigner. The SimpleAssigner ensures that when you enter the lobby for the first time, you go straight into the experiment stage. So `TurkServer.inExperiment()` becomes true and the `Tracker.autorun` block in demo.js calls `Router.go('/experiment')`. The routing code in routes.js then loads the "experiment" template, so you see the "Click Me" page.

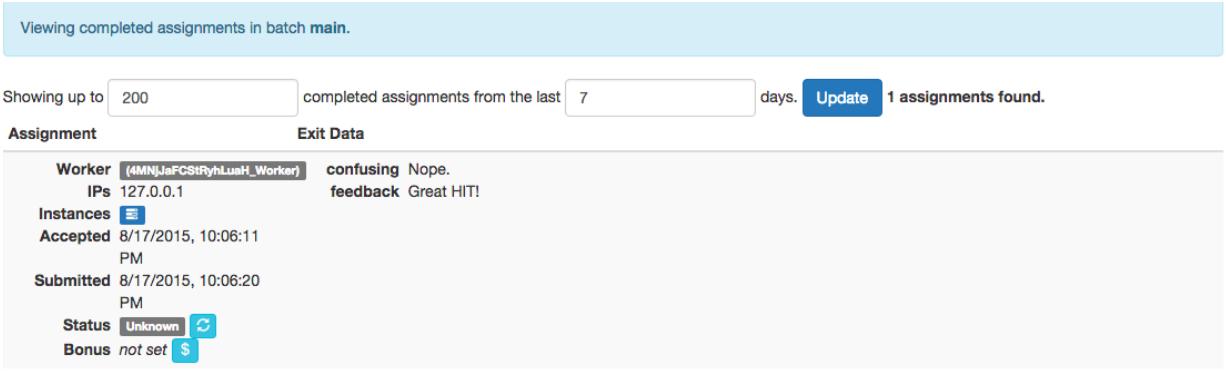
When you are ready to move to the exit survey, click "Exit Survey." Now the client calls the `goToExitSurvey()` method, which ends the experiment, so you go back to the lobby. The SimpleAssigner ensures that when you enter the lobby for the second time, you go straight to the exit survey stage. So `TurkServer.inExitSurvey()` becomes true, which results in our router loading up the `'/survey'` route, and the "survey" template is displayed.

Fill in the survey questions, and then click the submit button. You might see an error, but that's just because we aren't hooked up to Mechanical Turk. Later we'll run this app on the sandbox, and this part will work just fine.

6.4 The Admin Console

Now let's use the admin console to dig a bit deeper into what just happened. Go to `http://localhost:3000/turkserver` and login. Select you "main" batch in the "Filter batch:" dropdown at the bottom of the left navigation pane. You should get in the habit of doing this, as most of the admin console functionality won't work unless a batch is selected.

Now click on "Assignments - Completed" in the left navigation pane. You should see something like the below:

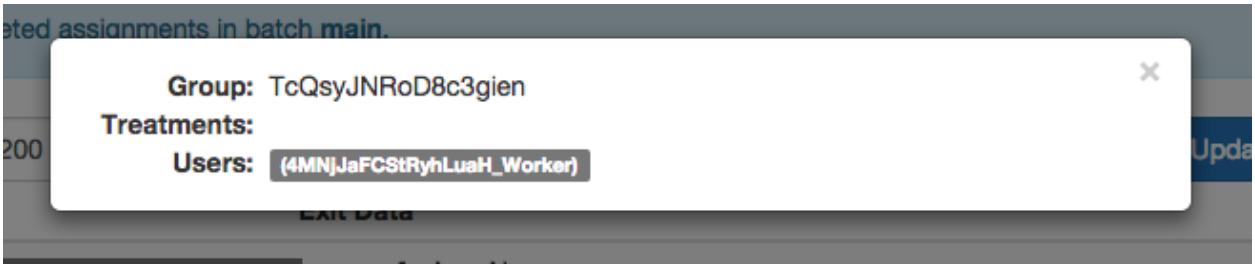


This shows the results of the HIT you just completed as a fake user. More precisely, it shows the details of your **assignment**. Recall the following facts about users, assignments, and experiments:

- A user can accept many different HITs over his lifetime and therefore have many different assignments, but he can only be in one assignment at a time.
- An assignment contains all the experiments that a user did during the HIT. (In our case, each user only does one experiment per HIT, so each assignment will only have one experiment associated with it.)
- An experiment can contain multiple different users. (In our case, each experiment only has one user.)

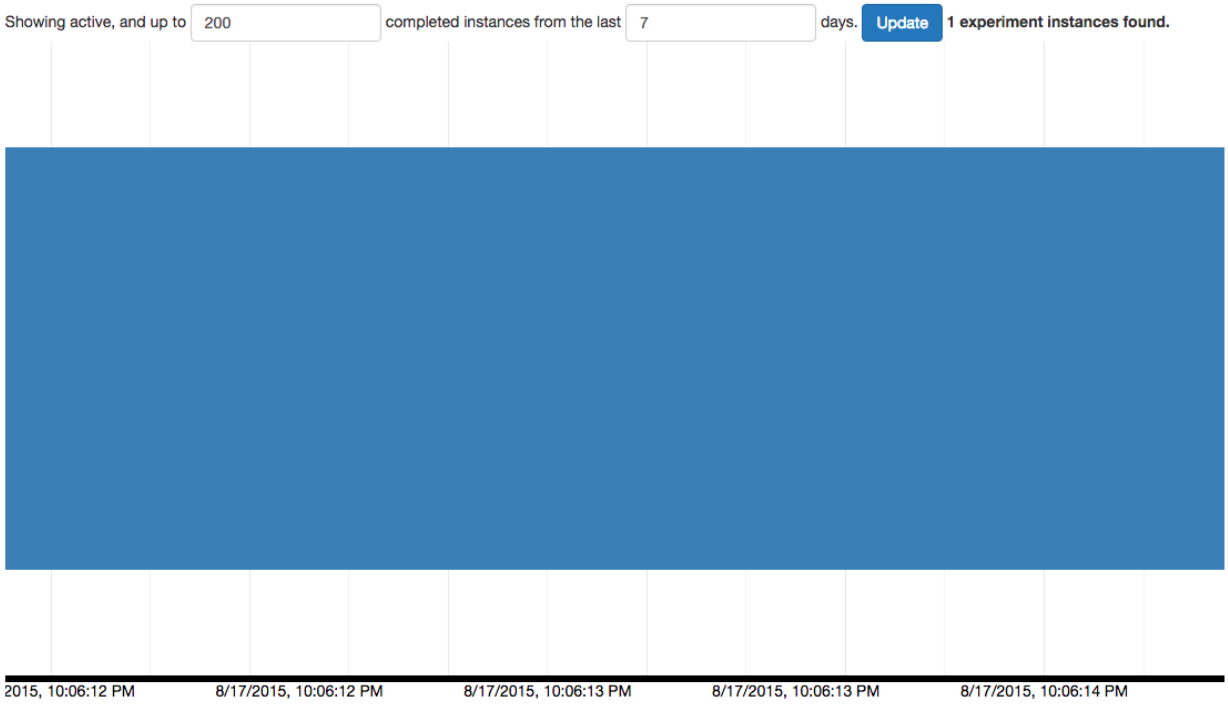
Take a look at some of the data recorded for this assignment:

- You can see your fake Mechanical Turk WorkerId at the top – hover over it to see some more data about your fake user (for instance, he is in a "disconnected" state because he already submitted the HIT).
- You can also see the times you accepted and submitted the HIT.
- The **Instances** field shows one square blue icon, which corresponds to the one experiment you did in this assignment. If you click on that icon, you will see a popup with the id of that experiment, as well as a list of the users who participated:



- There is also a **Bonus** field, which reads as "not set." In addition to paying a user the promised base rate for a HIT, you may want to bonus the user according to the work he did. TurkServer makes it very easy to increment a user's bonus while he completes the experiment, and then pay him later. We'll come back to this shortly.
- The answers you submitted during the exit survey are recorded in the right column.

Now click on "Experiments" in the left navigation pane. You should see something like the below:



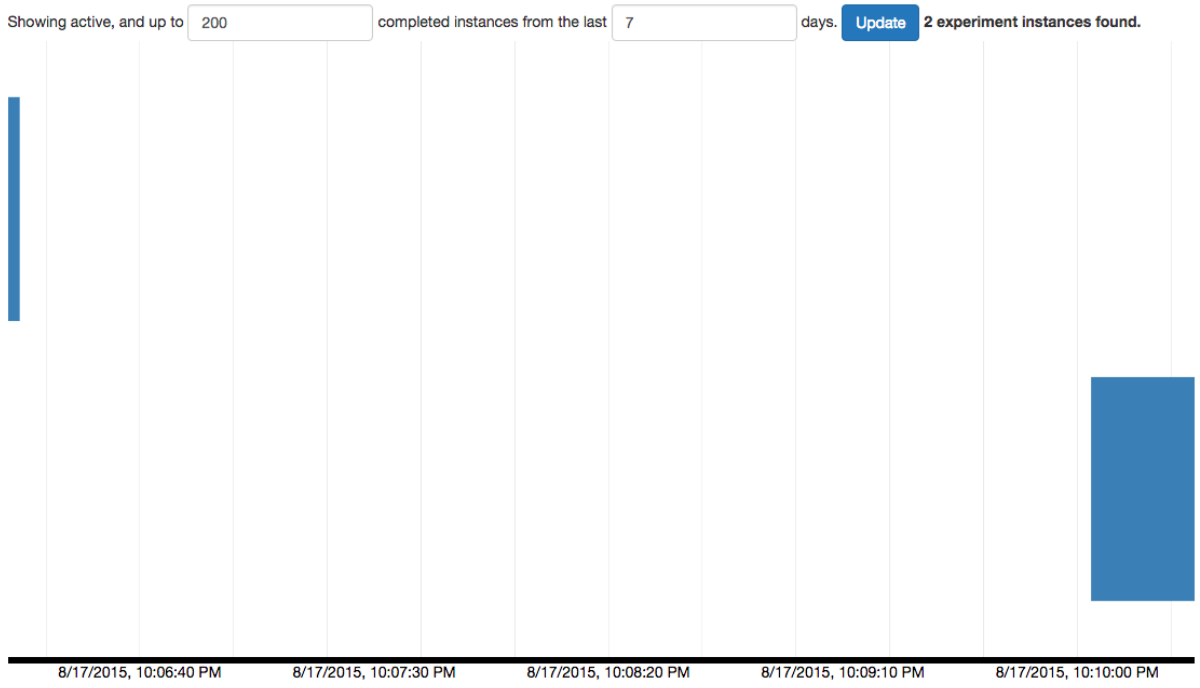
This is a visualization of completed and ongoing experiments. There will be bar per experiment. Since we have only done one experiment, we just see one fat bar. The x-axis is the time spent in this experiment. Click on the bar, and you will see the same pop-up that you saw when you clicked on the little blue square on the previous page.

Scroll down a bit, and you'll see something like:

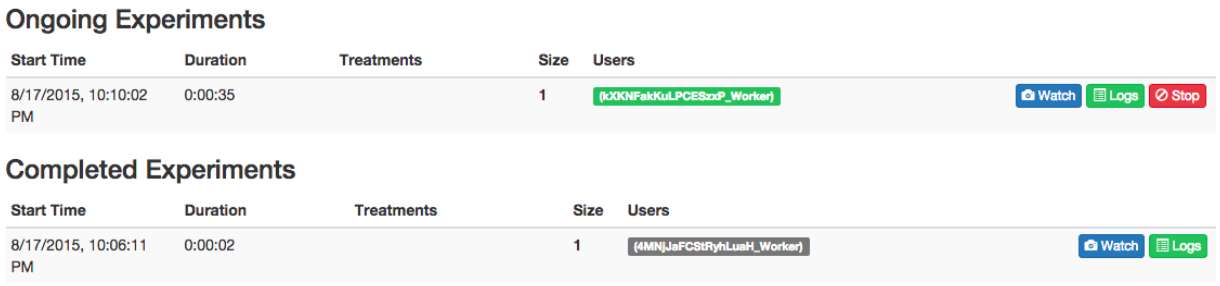
There are no ongoing experiments, but there is one completed experiment. Here we see the start time, duration, and size and user identities. Don't worry about the "treatment" field or the "watch" or "logs" buttons for now.

Now we're going to do another experiment (as another fake user) and this time we'll follow along in the admin console. You will need two browsers for this (or two tabs with one of them in incognito/private mode), because you will be logging in simultaneously as two users: the "admin" user and a fake Mechanical Turk User. In one browser, navigate to `http://localhost:3000/turkserver`, and in the other to `http://localhost:3000/`. Login to both. In one browser, you should see the admin console, and in the other, you should see the "Click Me" page.

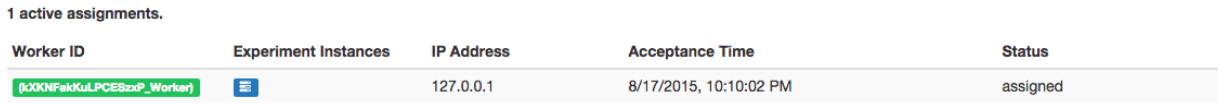
Go back to the admin console and click on the Experiments tab. You should now see two bars in the visualization, and the second one should be "growing" with time, because this second experiment is still going on:



Scroll down, and you'll see that there is one ongoing experiment. The worker ID is highlighted in green, which means that he is currently online. (A grey background means he is disconnected.)

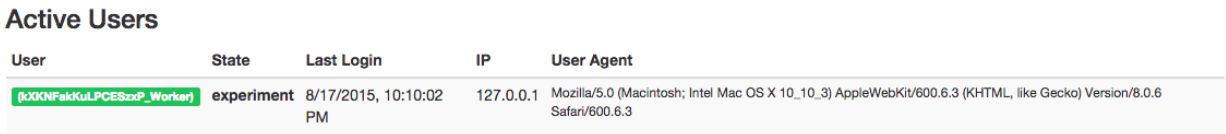


Now click on "Assignments - Active" in the left navigation pane. You should see something like the below:



Pretty self-explanatory. Note that assignments are typically in one of two states: "assigned" or "completed". When a user submits a HIT, the assignment becomes completed and moves from the "Assignments - Active" to the "Assignments - Completed" page in the admin console.

Next click on "Connections" in the left navigation pane. You should see something like:



Here is where you can keep tabs on all currently connected users. A user's state will either be lobby, experiment, or exit survey.

Finally, go back to the other browser window, and submit this HIT. Then in the admin console, verify that all the data has changed accordingly. There should be two completed experiments, two completed assignments, no active assignments, and no connections.

6.5 Partitioned Collections

So this experiment was pretty uninteresting; we didn't even record any data. Let's save the number of times that a user clicks the button, and also update his bonus accordingly. The more times he clicks the button, the more money he will make. (I'll choose to ignore the ridiculousness of this idea.)

Let's create a new Mongo collection for this purpose. Create a new file models.js and add the following:

```
Clicks = new Mongo.Collection('clicks');
TurkServer.partitionCollection(Clicks);
```

You should be familiar with the first line of code. The second is a bit more mysterious, but bear with me for now.

Now go to demo.js, and within the `Meteor.isClient` block, add the following:

```
Tracker.autorun(function() {
  var group = TurkServer.group();
  if (group == null) return;
  Meteor.subscribe('clicks', group);
});
```

Next, within `Meteor.isServer` block, add:

```
Meteor.publish('clicks', function() {
  return Clicks.find();
});
```

It's not crucial to understand all of what's happening here, but the general idea is important. A key concept to Turkserver's functionality is the idea of "partitioned" Mongo collections. A partitioned collection has the property that *every document belongs to exactly one experiment*. When a user is in an experiment, he can only see documents in the collection that belong to that experiment.

To be more specific, normally we know that `Collection.find()` will return all documents in the collection. But if this collection has been partitioned and the user is in an experiment, then `Collection.find()` (regardless of whether it is called on the client or server) will only return the documents in the collection that also belong to the experiment.

> The partitioning of collections is a key abstraction that allows experiments in TurkServer to be designed at the instance level. This in turn allows for many different types of experiment designs, while differing very little from Meteor's standard publish-subscribe model. In many cases, multi-person interactions are as simple to design as single-person interactions.
>
> Advanced users might mix partitioned and non-partitioned collections in Meteor code. Partitioned collections will show different data to participants in each experiment instance, while non-partitioned collections must be separately controlled by your code.
>
> See the docs for partitioner for more details on how this works.

Now we can make more sense of the code. The line `TurkServer.partitionCollection(Clicks)` sets Clicks to be a partitioned collection. So every document will belong to one and exactly one experiment.

Next, look at the subscription code. First note the variable `TurkServer.group()`, which is a reactive variable that returns the id of the experiment that a user is currently in. Because this code is in a `Tracker.autorun` block, every time the value of this variable changes (i.e. whenever the user enters or exits an experiment), the block will re-run. If the value is not null, then the user is indeed in an experiment, so we re-subscribe to the Clicks collection.

A re-subscription triggers the publication code on the server. The `Meteor.publish` code we added looks standard, but because the user is currently in an experiment, the `Clicks.find()` part will *only return documents that also belong to this experiment*. This way, a user only subscribes to the documents that are relevant to his experiment, automatically dividing real-time data between groups. (If this was confusing and you need a refresher on Meteor's publish/subscribe protocol, take a look at the docs.)

Now we are finally ready to use our new partitioned collection. Instead of storing the number of clicks on Meteor's `Session` object, we'll store it on our new Clicks collection. To make this work, we'll need to add a new document to this collection whenever a user enters an experiment, and initialize the "count" field of this document to 0. TurkServer makes it easy to take certain actions upon the start of an experiment – use the `TurkServer.initialize` function, which takes as input another function that will be run upon every experiment initialization. Add the following within the `Meteor.isServer` block:

```
TurkServer.initialize(function() {
  var clickObj = {count: 0};
  Clicks.insert(clickObj);
});
```

We don't need to add any other fields to the Click document besides `count`. This is because Clicks is partitioned, so whenever we add a document, it automatically gets "tied" to the current experiment. More precisely, behind the scenes, a hidden field called `_groupId` is added. The value of this field is the id of the current experiment (i.e. the id of the Mongo document that corresponds to the current experiment – more on this shortly). So later when we analyze

our data, we can search for a Click document with `_groupId: <our experiment Id>`, and we'll find the right document and be able to tell how many times we clicked the button. We'll go through that exercise later and it will be clearer then.

Now go to demo.js and edit the `Template.hello.helpers` and `Template.hello.events` blocks as follows:

```
Template.hello.helpers({
  counter: function () {
    // get our Click document
    var clickObj = Clicks.findOne();
    // if it exists, return the count field
    return clickObj && clickObj.count;
  }
});

Template.hello.events({
  'click button#clickMe': function () {
    // update our Clicks document
    Meteor.call('incClicks');
  }
});
```

Note the use of `Clicks.findOne()` – because Clicks is partitioned, even if other users have inserted their own Click documents in other experiments, we only have access to our own click document, so we can treat the Clicks collection as if it only contains one document.

Finally, define the `incClicks` method in your server-side `Meteor.methods` block:

```
Meteor.methods({
  incClicks: function() {
    Clicks.update({}, {$inc: {count: 1}});
  },
});
```

Again, because Clicks is partitioned, we don't have to specify which document to update – the code automatically knows to update the document corresponding to our experiment.

Now to test that this all worked properly, load up `http://localhost:3000/` again, login as a fake user, and go through the HIT. Click the button a few times and check that the number goes up appropriately. Then submit the HIT.

To double-check that the Clicks document was inserted and updated correctly, we can use Meteor's Mongo console. Type `meteor mongo` at the command line, and then `db.clicks.find()`. You should see something like:

```
{ "_id" : "pZJRGTTgktiAooW8y", "count" : 6, "_groupId" :
"JrYnXMhbqdspadttQ" }
```

As promised, a secret field called `_groupId` was added to the document. The value of this field is the id of the experiment to which the document belongs.

Now say we had a ton of users who all completed the experiment, and we just wanted to find the click count of a particular experiment. Typing `db.clicks.find()` will call up every single document, which will be unhelpful. But if we know the id of the experiment we want, we can search for the corresponding Click document. We can find experiment ids in a few places in the admin console:

- From the Experiments page: Click on any of the blue bars. When the pop-up appears, look at the value of the **Group** field. That's the id of the experiment you clicked on.
- From the Assignments - Completed page: Click the blue box next to **Instances** to select the experiment that was completed in that assignment. The same kind of pop-up will appear.

Let's grab the id of the most recent experiment. So either click on the right-most bar on the Experiments page, or the top-most blue square on the Assignments - Completed page, and then copy the value of the **Group** field on the pop-up that appears. Now go back to the mongo console and type `db.clicks.find({_groupId: '<value you copied]})`. You should see the same Click document you saw earlier.

6.6 Setting Payment

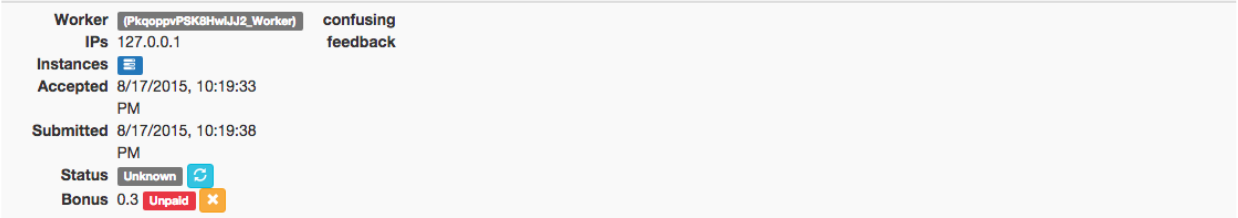
So this is well and good, but we still haven't dealt with bonusing the user according to how many times he clicked the button. Let's edit our `incClicks` method so that whenever we increment the counter, we also add some money to the user's bonus:

```
incClicks: function() {
  Clicks.update({}, {$inc: {count: 1}});
  var asst = TurkServer.Assignment.currentAssignment();
  asst.addPayment(0.1);
},
```

First this code grabs the JavaScript object corresponding to the user's current assignment. Recall that we have an assignment object for every <user, HIT> pair, so this is the natural place to store a bonus. Then we call the method `addPayment()` to increment the bonus by a fixed amount (in this case, 10 cents).

You should start seeing a pattern emerging with these TurkServer API calls. First we grab a particular JavaScript object – e.g. a batch, an experiment, or an assignment. Then we call one of its methods. If you're curious about what methods are available on these objects, see the relevant files in turkserver-meteor/lib, namely batches.coffee, instance.jsx, and assignment.jsx.

To see if this worked properly, go through the HIT yet again. Be sure to click the button a few times. Then go to the admin console and the Assignments - Completed page. On the most recent assignment, you should see a value next to the Bonus field, as well as a red label that says "Unpaid."



So we've set the bonus value, but we haven't yet approved the worker's assignment or paid the bonus. We can't test out this functionality while running as a fake user, so we'll have to move to Mechanical Turk's sandbox environment.

6.7 Deployment

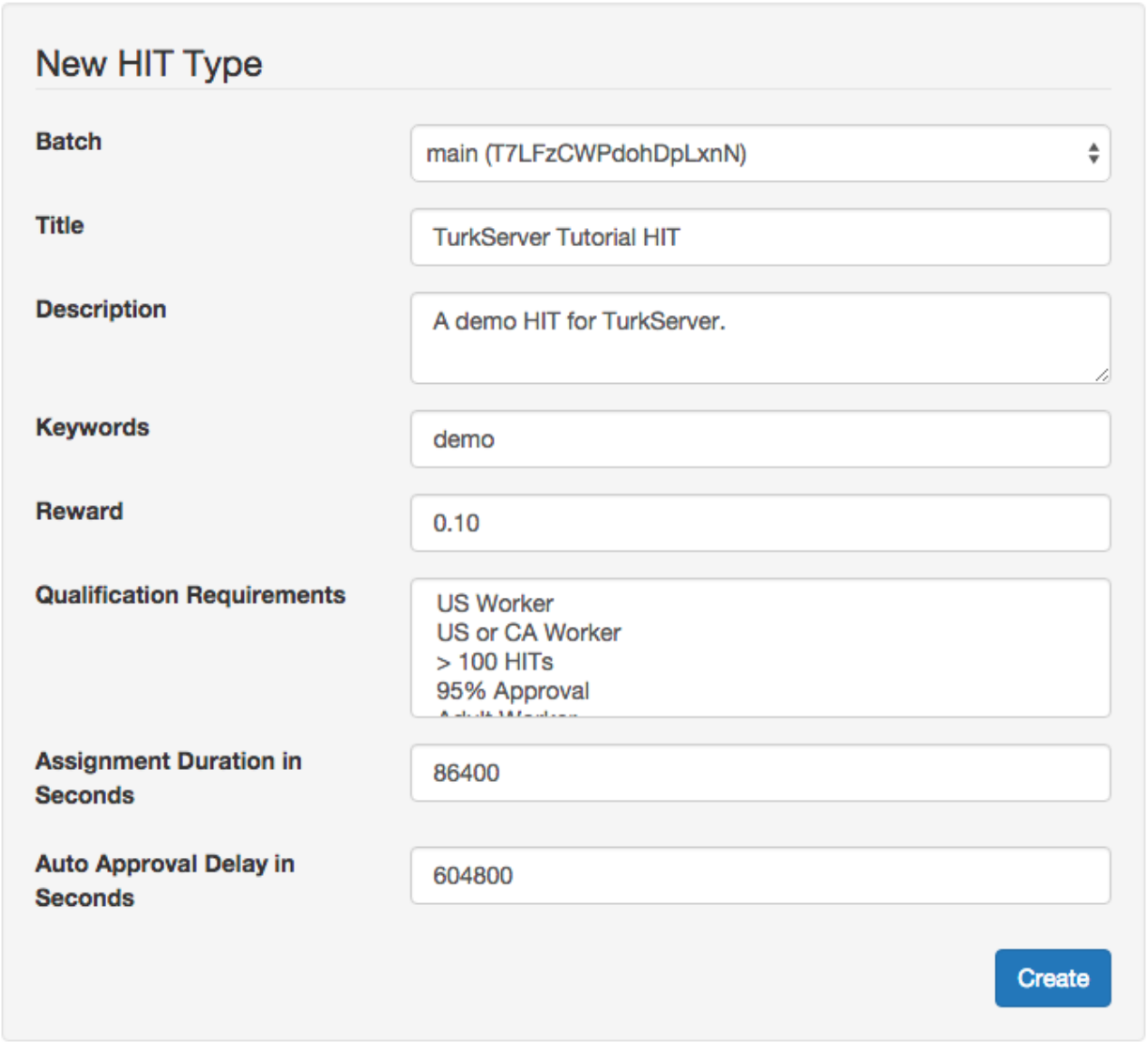
To continue with the tutorial, follow the *deployment instructions* so that you can use the actual MTurk API and get the app running in the MTurk sandbox.

6.8 HITs and HITTypes

The next step is to create a new HIT type. We can do this in the admin console. Click on "MTurk" in the admin console navigation pane and go through the process of filling out the form on the right side. First we have to choose a batch for the HIT type. All HITs with this HIT type will belong to the batch we choose. This important because the

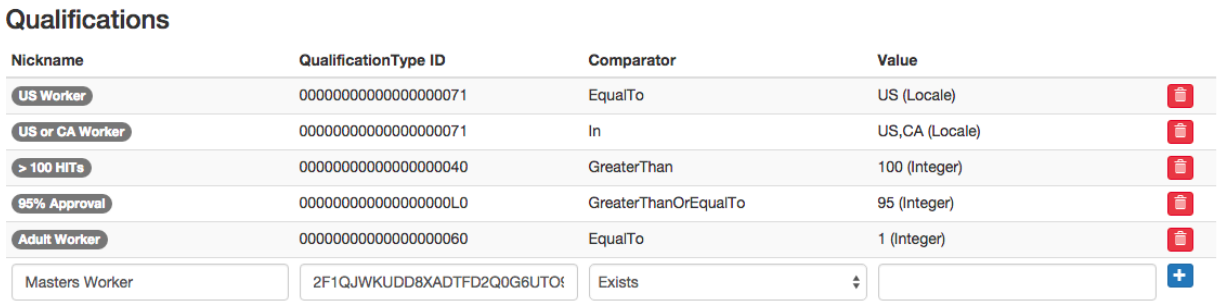
only way TurkServer knows which **assigner** to use on a particular HIT is by checking which **batch** the HIT belongs to. If TurkServer doesn't know what assigner to use, then the app won't know where to send a user after he enters the lobby, and the user will end up in a limbo state.

Choose the "main" batch, and then fill out all other fields. Most are self-explanatory, but complete descriptions can be found in the Mechanical Turk API docs here. You can leave the "Assignment Duration in Seconds" and "Auto Approval Delay in Seconds" fields set to their default values, which ensure that we have one day to complete our HIT once we accept it, and that our assignment will be automatically approved in a week.



The only non-trivial part of this process has to do with worker qualifications. TurkServer makes it easy to add a few commonly-used qualifications, such as: "US Worker" or "95% approval rate." Simply select the ones you want in the box by holding down the control key (Windows) or command key (Mac) and clicking the names.

If you want to add your own qualification type, use the form at the bottom of the page. For instance, say we wanted to ensure that only workers with a Masters qualification can accept our HIT. Go to the Mechanical Turk API docs page here to look up the corresponding qualification type ID. Note that the docs say to "set the comparator parameter to 'exists' to require that workers have this qualification." Thus, fill out the form as follows:

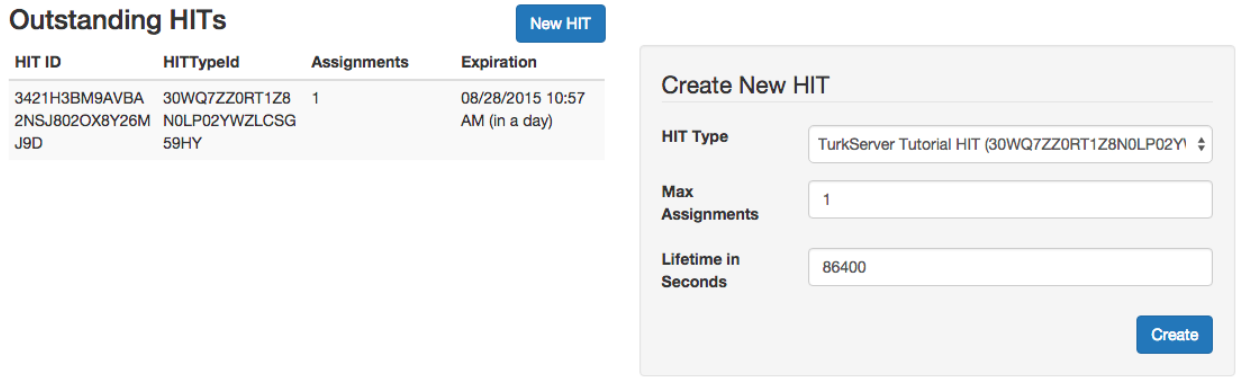


and then hit the blue "+" button. You now should see your new qualification in the dropdown menu in the form above:

Don't select any qualifications right now, because we're just testing on the sandbox (plus, most qualifications have different ids on sandbox than production, so it wouldn't work properly anyway.)

When you're ready, click "Create". On the following screen, you should see a button that says "Register." Click that to register your new HIT type with Mechanical Turk.

At this point we're all set to create our HIT. Click on "HITs" in the navigation pane. In the "Create New HIT" form, select the HIT type we just created from the dropdown menu. Leave "Max Assignments" and "Lifetime in Seconds" at their default values, which ensure that only one worker can accept this HIT, and that the HIT will be available for one day until it expires. When you're done, click "Create." You should see the HIT pop up in the table to the left:



6.9 Using the Mechanical Turk Sandbox

The MTurk Sandbox allows you to test and debug your app without spending real money. See the *deployment instructions* for how to post a HIT and pay your workers.

CHAPTER 7

Research Projects

7.1 Crisis Mapping

> Mao, Andrew, Winter Mason, Siddharth Suri, and Duncan J. Watts. **An experimental study of team size and performance on a complex task.** PloS one 11, no. 4 (2016): e0153048.

paper - short video - code

In this project, we use a real-time Meteor app built on TurkServer to experimentally study and analyze collaboration and coordination with respect to teams of varying size on a highly dynamic crisis mapping task.

7.2 Long-run Cooperation

> Publication forthcoming.

This project allows a hundred people to interact in a prisoners' dilemma experiment, conducted daily over a *month*, collecting an order of magnitude more data than past studies.

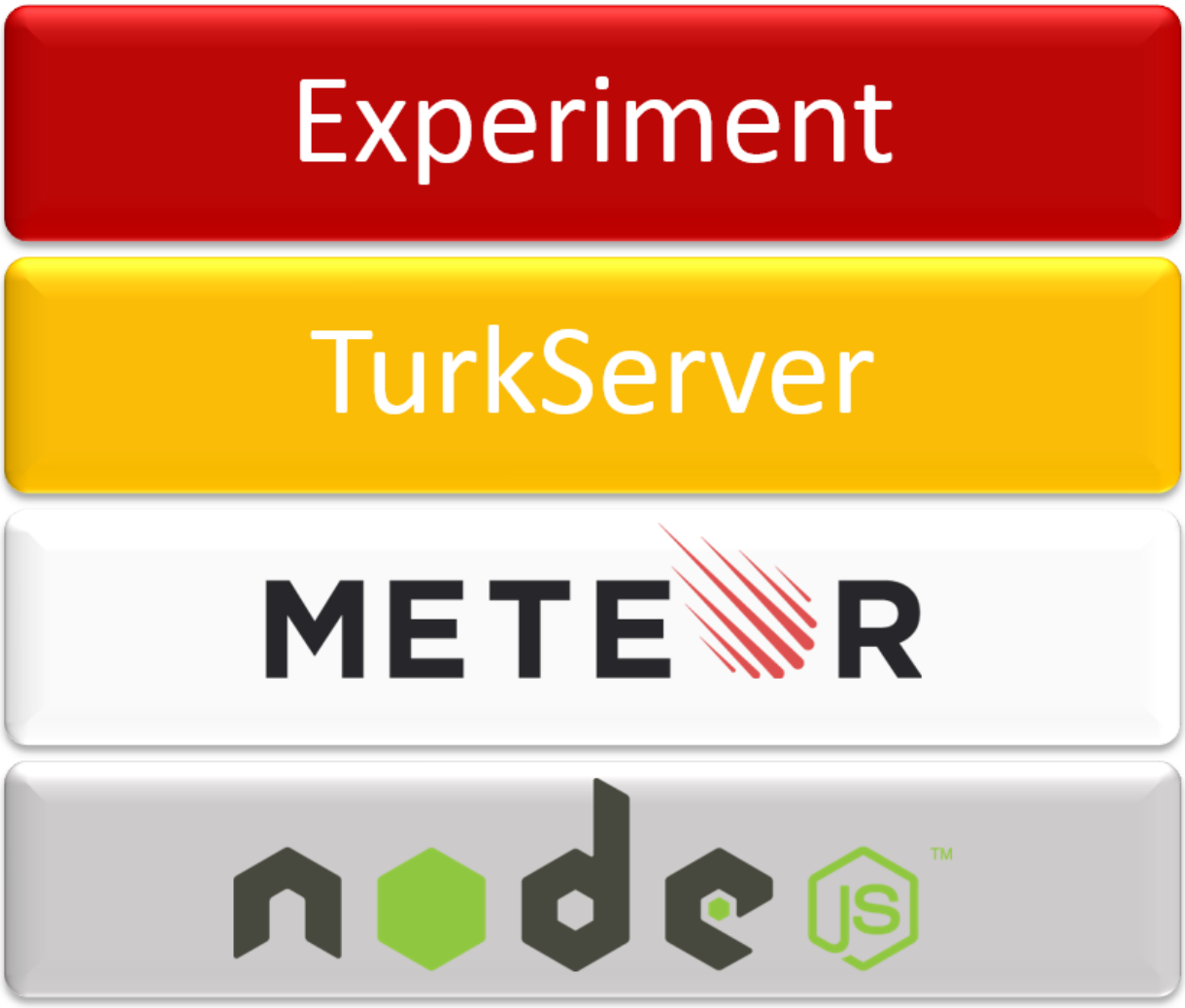
7.3 Your paper here!

Show your work on this page. Add your project here and send us a pull request!

CHAPTER 8

Architecture Overview

TurkServer is a full-stack Javascript framework based on Meteor. This means you do all your programming in Javascript, on both the client-side (user's browser) as well as server-side.



Why is TurkServer designed this way, and what are the core concepts?

- Read *why we chose Meteor* and Javascript as the underlying framework, and why this facilitates social experiments research.
- Understand the *multiple worlds abstraction*, its relation to Meteor's real-time data framework, and how this makes it easier to design apps for group interaction.
- Check out the *live experimenter view* in TurkServer, which allows you to see the status of all connected users and ongoing experiments.
- Take a look at how *treatments* can be assigned to users or worlds in TurkServer.
- Look at *research methods* that guided the development of TurkServer.
- See references to *alternative platforms* and concepts for doing software-based, crowdsourcing-driven experiments.

CHAPTER 9

Why Meteor?

For many years, web programming has been a drag. Developers used Javascript on the client, and a different language on the server. Moreover, when these different languages communicated via AJAX for dynamic pages, good software abstractions usually went out the window, resulting in things like jQuery spaghetti and therefore buggy apps.

What is a researcher to do, then, when most social science experiments actually aren't *social* (since they study individuals), and to actually do social experiments requires continual real-time interaction between large numbers of participants? Meteor is at the forefront of a new generation of web technologies aimed at tidying up the mess describe above, and can simplify building social experiments significantly. As of now (June 2016) it is the most popular web framework on Github, with an active community and extensive documentation, and therefore great staying power.

Meteor's development paradigm is significantly different from almost all other web frameworks, which generally focus on either the front-end or back-end. It is full-stack from the ground up, using one language (Javascript), where the server-side code runs on Node.js and the client-side runs in the browser. UI code is mostly written declaratively rather than imperatively: instead of saying *how* you want to do something, you just write *what* you want to do. Most importantly, the core of Meteor is a distributed data framework (which the declarative style hooks into) that greatly simplifies synchronizing state between multiple clients and a server. This is another level of abstraction over AJAX: instead of worrying about passing messages or calling remote procedures, you can just specify the data that needs to be shared and it will be updated, in real time, transparently. As a result, this can be an order of magnitude reduction in the amount of code (and complexity) for a real-time or collaborative app. As one article put it, Meteor has made MVC obsolete. When leveraging front-end frameworks such as Twitter's Bootstrap that integrate well with Meteor, we can also get started with much less custom CSS.

Meteor has many useful features, but the following are primarily useful for researchers using web apps to study social behavior.

- **Easy development and fast prototyping.** Meteor is a one-stop shop for setting up development for a web app, pulling in dependencies automatically. It runs a local server that updates in real-time as you code. This is very useful for testing new research ideas and designs before committing to a running things at large scale.
- **Real-time data synchronization.** Meteor has powerful primitives for synchronizing data across multiple clients, making social, collaborative apps almost as easy to build as single-user apps. This makes studying social interactions much easier.
- **Fast and easy deployment.** Deploying a Meteor app to a public server is as easy as one command. Getting an app accessible to the general public is often a lot of work for those unfamiliar with devops, and ease of

deployment means you can get on with your research.

- **Active and growing community of developers**. Meteor is one of the top 10 frameworks on GitHub, and is built on the widely popular NodeJS. Building an app with TurkServer leverages these communities as much as possible. Thus, you can find answers to many common issues on sites like StackOverflow and Meteor's forums.

TurkServer is designed as a package (add-on) for Meteor, taking a Meteor app and plugging it into MTurk. The advantage of this approach is that it leverages the many examples of web apps built in Meteor and Javascript, even those not specific to social experiments, without depending on much proprietary functionality. Moreover, being able to contribute full-stack code makes components such as a waiting room, chat, and other common paradigms much easier to share. The future of online experiments is coming, and the next generation of web technologies is a great force multiplier for getting things done!

Find out more about Meteor:

- Getting started
- Meteor Guide
- API Docs
- Meteor on GitHub

CHAPTER 10

Worlds and Assignment

TurkServer uses Meteor's powerful publication-based data model to create multiple *worlds* or *instances* within which data can be synchronized among a group of users in real time. This makes it easy to build and study mechanisms of real-time interaction. However, the real utility of this abstraction is the ability to create new worlds and assign or re-assign users among them, allowing for flexibility of many experiment designs while keeping programming simple.

TurkServer uses the concept of **batches** to logically group instances of experiments together. Each batch allows for some high-level parameters to be set, such as a limit on repeat participation.

Each batch has a **lobby** or waiting area, a state that users are in when they are not part of any world. An ***assigner*** controls the state of the waiting area and therefore can arrange arriving or returning users to different worlds.

A world can return users to the lobby once a task is complete, allowing for users to be assigned to different groups, matched with different partners, and so on. The assigner can also send users to an **exit survey** to complete the task and debrief.

10.1 Examples

To be completed.

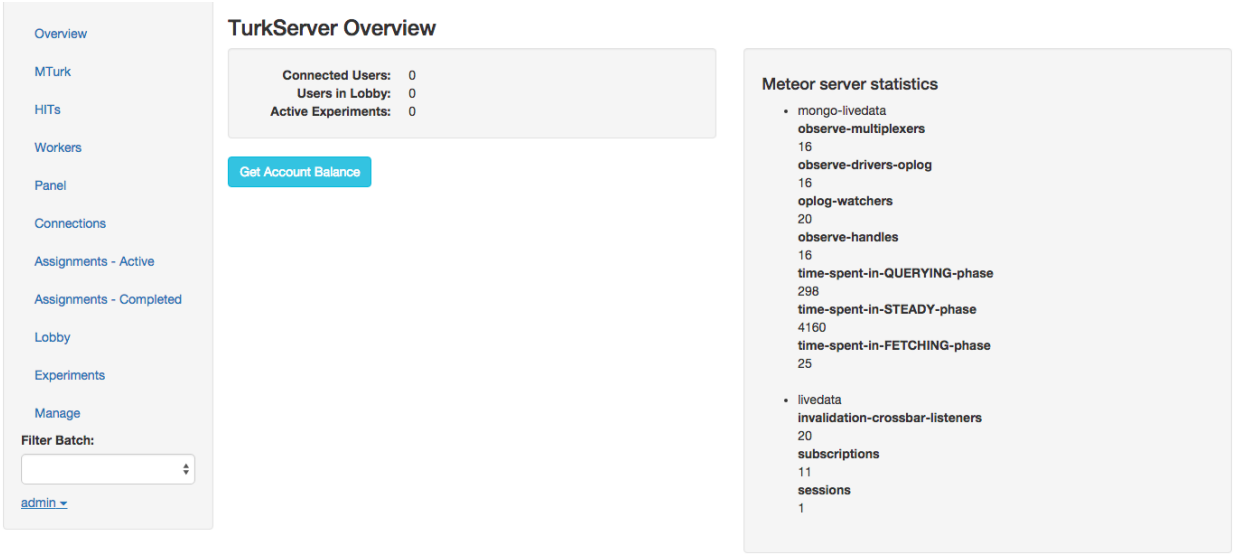
10.2 Additional References

TurkServer's multiple worlds abstraction is implemented by the partitioner package in Meteor, which was written specifically for TurkServer.

CHAPTER 11

Live Experimenter View

TurkServer has a built-in administration interface at `/turkserver`, providing a real-time view of everything that is happening with your users and in your experiments. You can use this to manage batches, manage treatments, view the progress of experiments, and post HITs to recruit subjects.



A brief overview of the different sections:

- **Overview**: provides a summary of traffic and load on the server.
- **MTurk**: manage HIT types and qualifications for Mechanical Turk.
- **HITs**: create, expire, and renew HITs, which will send participants to the server.
- **Workers**: look up information about particular workers who have participated in the past.
- **Panel**: summary of information about all workers known to the system.
- **Connections**: shows all currently connected users and their state.

- **Assignments**: shows currently active and completed assignments, and associated metadata.
- **Lobby**: shows all participants who are currently waiting for experiment instances (see below).
- **Experiments**: shows experiment instances, their participants, and timing information.
- **Manage**: controls batch and treatment data which are used to configure experiments.

The admin interface is mostly self-explanatory. We aim to provide more information about the details of using the interface as its design is finalized.

CHAPTER 12

Treatments

Being able to show users different instructions, interfaces, or information is an important part of conducting controlled experiments.

TurkServer accomplishes this through **treatments**, containing structured data that are made available to the front-end app under `TurkServer.treatment()` as a reactive variable in Meteor.

Treatments contain arbitrary key-value pairs, and can be assigned to both users and worlds. The client code for a user in a particular world will have access to all of the treatment data for both the user and the world, creating a useful way to programming control the display of different parts of the user interface (such as instructions) and mechanisms of interaction. Since treatment data is reactive, it is easy to use with Blaze, Meteor's UI rendering system.

Treatments can also be viewed in the *admin interface*.

Additional details and examples of treatments to be added.

CHAPTER 13

Research Methods

The original design of TurkServer was described in the following paper. The current version (based on Meteor) is not yet published, so please feel free to cite this paper and note that you are using a newer version, or link to this site.

- Andrew Mao, Yiling Chen, Krzysztof Z. Gajos, David Parkes, Ariel D. Procaccia, and Haoqi Zhang. **TurkServer: Enabling Synchronous and Longitudinal Online Experiments.** In the *Fourth Workshop on Human Computation* (HCOMP 2012).

TurkServer also builds heavily off methods for panel recruitment and synchronous experiments pioneered by Winter Mason and Sid Suri in their seminal work on crowdsourced behavioral research.

- Winter Mason, and Siddharth Suri. **Conducting behavioral research on Amazon's Mechanical Turk.** *Behavior research methods* 44.1 (2012): 1-23.

CHAPTER 14

Alternative Platforms

There are many other platforms released by academics for doing online experiments. Here are a list of the ones we know that are currently active.

The main differentiation of TurkServer is relative to the platforms and frameworks below is its focus on facilitating and instrumenting real-time and synchronous interaction. But we've kept the software simple, and you'll find features such as the *live experimenter console* and one-way mirror useful even for single-user experiments.

- Breadboard focuses on network and cooperation experiments.
- CogniLab is a paid service for launching online experiments, no coding required.
- NodeGame is (also) a Javascript library for real-time experiments.
- oTree is a Django (Python) based framework focusing on economic games.
- PsiTurk is a Python-based framework for building and sharing experiments, originating in the psychology community.
- Volunteer Science is a hosted platform that maintains their own volunteer panel.

Unless using volunteer participants, all of the aforementioned software frameworks run on crowdsourcing platforms:

- Amazon Mechanical Turk has been around the longest and is currently the most widely used.
- Prolific Academic is a recently launched framework focusing on academic research.
- See this Quora question for other crowdsourcing systems that may supplement or replace Mechanical Turk in the future.

CHAPTER 15

Design Overview

15.1 Typical Workflow using TurkServer

At a broad level, using TurkServer looks something like the following once you've done a literature review and settled on a research question.

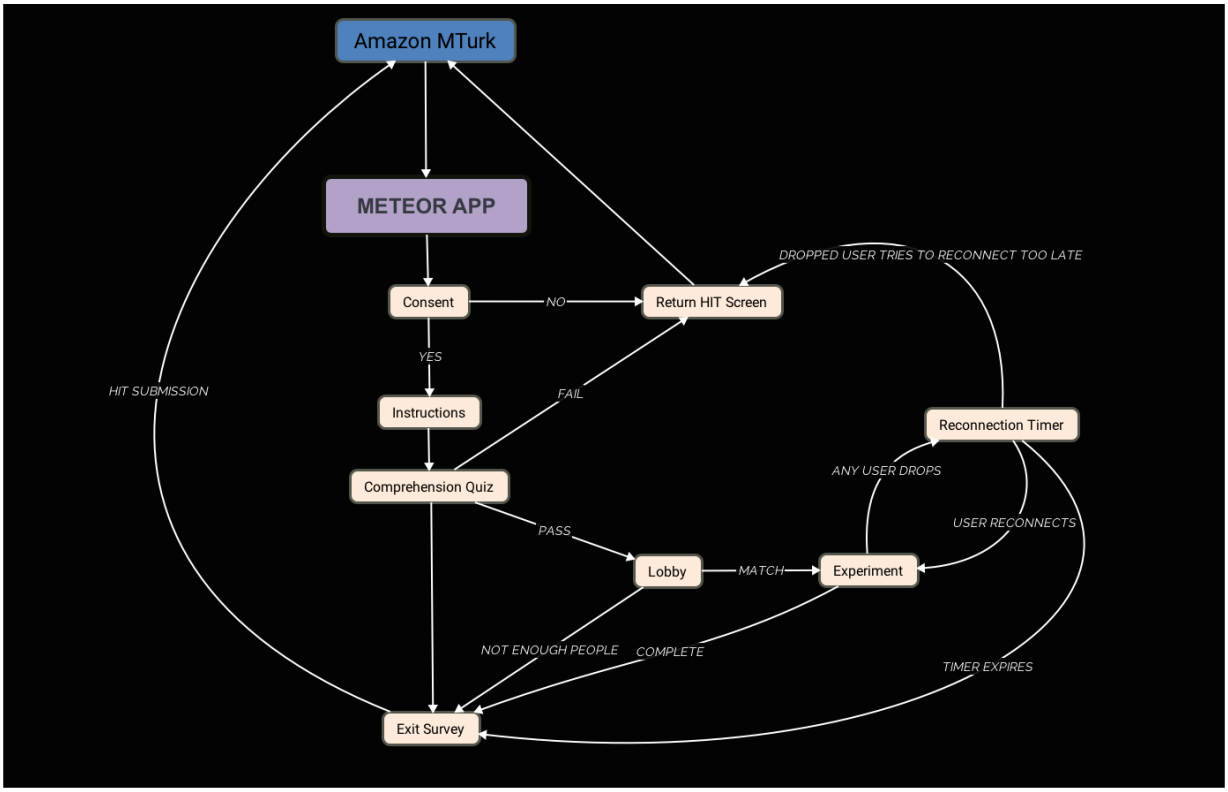
1. Prototype your experiment in a **standalone** Meteor app, designed aroundthe smallest unit of people who will interact. This could even be a single user, but the point is to take advantage of Meteor's ease of development to investigate whether your experiment makes sense. Here are some examples:
    - In an experiment on teamwork, one unit was a team of people working together.
    - In a prisoner's dilemma experiment with random rematching among a hundred people, one unit was two players in a repeated game.
2. If the design is promising, add TurkServer to your app, and integrate with its *assignment mechanism*, which will allow you to make different instances of your prototype world and control how participants enter and leave them.
3. Test, debug, and pilot your experiment extensively, making sure that your *code works as intended* and your *instructions are easy to*. Start with your research team, then others in your lab/department, and finally run a pilot with real crowdsourcing workers. Experiments inevitably involve many redesigns and modifications, and it's better to do this in testing than to shoot from the hip.
4. *Run your experiment*!
5. Share the source code of your app so others can look at your experiment protocol, and replicate or build on it.

15.2 Designing Good Experiments

While TurkServer provides much of the software components necessary for deploying web-based experiments, many elements of your experiment design are important to ensure that your study goes smoothly and workers leave happy.

In particular, when doing experiments that require coordinating 2 or more users at once, there are a number of important considerations for experimental design that affect how well you can collect data when your app is live. Consider

the following "state diagram" (credit: Eshin Jolly) that illustrates possible conditions a user passes through as they participate in a group task:



Among other things, the diagram illustrates:

- A *matching mechanism* to group people together as they are waiting in the lobby.
- Providing *good instructions* and using a comprehension task to check that they are understood.
- Accounting for the possibility of *disconnection and dropout* in your design, both to minimize the amount of data that might be discarded and ensure a good experience for all remaining participants.

See also the following topics:

- Constructing a *one-way mirror* to qualitatively observe activity and diagnose potential issues.
- How to *debug and test your app* before running it live.
- Helpful *packages and software* that you may want to use in your app.
- Other *frequently asked questions* about building experiments.

CHAPTER 16

Assignment and Matching

Experiments with individual users are simple and each participant can proceed independently. Use the `SimpleAssigner` to create this common form of experiment in TurkServer, but gain all the benefits of the live experimenter view.

When matching users together, however, the precise details of the matching mechanism become an important part of the experiment design. This is because users who are waiting an inordinate amount of time for partner(s) to arrive to start an experiment can either get frustrated, or go on to another task and become inactive. This can aggravate problems of *disconnection and dropout* and result in worse quality data.

Depending on how you post your HITs and how attractive they are, this may happen rarely. However, it's still possible, especially if your HIT has been posted for a while and the supply of workers is beginning to dwindle. Here are a few ways to work around this issue:

- Configure your *assigner* to to monitor the amount of the time users are spending in the lobby. If it exceeds some threshold, allow them to submit the task and pay them for their time (e.g. half the HIT amount). Although this can be viewed as paying something for nothing, it's important to *preserve the reputation* of your requester account in order to recruit workers.
- For users that do wait an exceedingly long time and eventually complete your experiment, consider paying them a bonus for their patience.
- When running really large groups, pre-recruit participants with an informational task and schedule a participation session in advance. TurkServer provides functions to e-mail users for this purpose. Then, you can expect all users to arrive at once and be ready to participate.

16.1 Assigner Examples

To be added. For now, take a look at some source code for inspiration.

CHAPTER 17

Providing Good Instructions

When running experiments with crowdsourcing systems, users benefit greatly from good instructions and will be more attentive to the task. Poorly worded or incomprehensible instructions will result in bad quality data. The tutorials package for Meteor was written alongside TurkServer to provide interactive instructions for how to use a Meteor app, with this use case in mind.

When giving instructions, ensure that participants will understand your experiment and can't simply "click through" to get paid. A good way to filter out this kind of behavior is to provide comprehension questions that prevent a user from advancing or moving past a certain point in your meteor app without understanding the task. The tutorials package makes it easier to implement these ideas.

For more information, see the documentation for the tutorials package.

CHAPTER 18

Disconnection and Dropout

Users may disconnect and reconnect to your experiment, or leave without coming back. A good experiment design accounts for this and minimizes the disruption to other users and preserves the quality of the data.

18.1 Examples

18.1.1 Handling disconnections

- Your project code should keep tabs on the connection state of each user and should have some kind of connection timer that prevents one user from getting "stuck" if their partner(s) disconnect
- Likewise if rematching users is important, make sure to build that into your project code and critically set it up in a way that your Assigner knows what to do, for example:
- An Assigner that uses number of instances and a boolean flag to properly route users:
  - When a user hits the TS Lobby the assigner checks if they have been in any experiment instances
  - If they have, it looks for a rematch boolean flag and if it sees it, leaves them in the lobby to get matched again
  - If it doesn't see that flag it assumes they have completed their experiment instance and pushes them to the TS Exit survey
  - If it doesn't see any experiment instances it assumes this is their first time in the lobby and leaves them there to get matched

18.1.2 Handling idle users

- Your project should probably deal with users that take exceedingly long to respond
- One solution is to setup the Assignment duration so that that a user must complete it within a certain time otherwise risk being unable to submit

  - This can be problematic if one user *causes* another user to lose out on the opportunity to submit the HIT because they themselves took too long

- Another solution is to track the idle state of a user through Turkserver and start a forced disconnect if a user has been idling too long
  - This is tricky to setup, but more ideal because users can be handled independently of each other, e.g.
    * If user A idles and is force disconnected, user B can simply be rematched or pushed to the end of the of experiment

CHAPTER 19

One-way Mirror

A one-way mirror is used in some physical lab experiments to allow unobtrusive observation of participants. Using the real-time capabilities of Meteor, this is actually fairly easy to set up on an existing app.

19.1 Creating a digital one-way mirror

In the worlds and assignment section, we explained how TurkServer is designed around multiple worlds that participants can be assigned to. This example explains how to build a one way mirror or view into the experiments you are running, so that you can observe what's happening in real time. Although this may seem like a superfluous feature, in practice you're likely to find it incredibly useful for qualitatively understanding what's going on—even before getting to data analysis.

The following code is built around the partitioner API that TurkServer users to divide a Meteor app into different 'worlds'. Consider the example in the partitioner docs, describing how to use the same `ChatMessages` collection to effectively create a collection of separate chatrooms. To create a one-way mirror allowing you to see all chatrooms simultaneously, you could do the following.

19.2 Set up a new publication on the server

First, set up a new publication on the server, allowing direct access to the data from any particular world.

```
Meteor.publish("oneWayMirror", function (worldId) {
    return [
        ChatMessages.direct.find({ _groupId: worldId }),
        Meteor.users.direct.find({group: worldId}, { fields: { username: 1 } })
    ];
});
```

The above publication returns data about the chat messages and users, partitioned by a given world.

> For those who are curious, `_groupId` is the world identifier in the partitioner package. The `.direct` syntax specifies to override the automatic partitioning or or 'multiple worlds' logic to directly query any

particular world. The "magic" that happens with different groups in TurkServer is that the `_groupId` field is inserted automatically depending on a particular user's group membership.

`Meteor.users` is the standard collection for Meteor's accounts system, which is already defined. This pre-existing collection is partitioned differently from other collections that one creates (such as `ChatMessages`); the second argument is a field specifier so that the query returns just the username instead of other users fields such as the login token, which you would not want other users to see.

19.3 Create a view for the experimenter on the client

TurkServer uses Iron Router to control the display of templates on the client. You can use the API to add a new route for the experimenter to use the one-way mirror:

```
Router.map(function () {
    this.route('expAdmin', {
        path: 'exp/:groupId',
        waitOn: function () {
            return Meteor.subscribe("oneWayMirror", this.params.groupId);
        },
        data: function () {
            return { groupId: this.params.groupId }
        },
        template: 'experiment',
    });
});
```

For the details of the code above, take a look at the Iron Router documentation. It is a pretty well-documented client-side routing package for Meteor, and currently the most commonly used. For example, specifying a `path` of `exp/:groupId` means that browsing to `exp/foo` store `foo` to the `groupId` parameter of the route params. The above function then subscribes to the `oneWayMirror` publication with this `groupId` as well as passing it as an argument to the template in the `data` function. Check out the Iron Router guide to see more examples of how to specify routes.

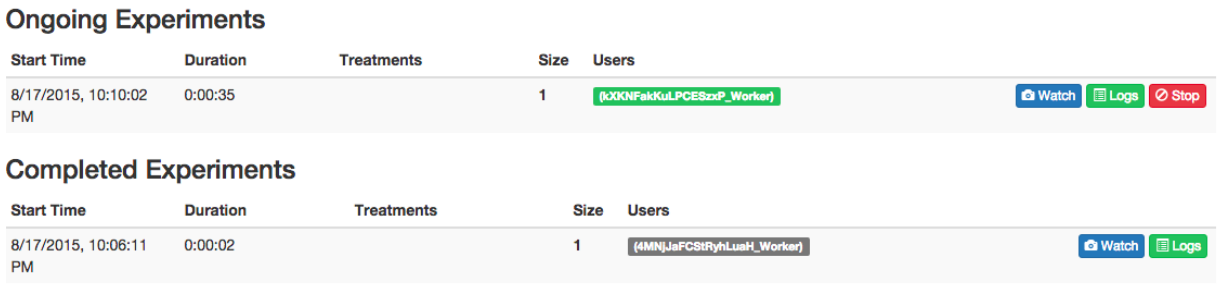
The above code allows the experimenter to visit `https://<server>/exp/<id>` to see what's happening with a particular group of people. The user interface that will be seen is defined by the `experiment` template. Usually you can just use the same interface that you have already built for participants, but for certain instances it may be useful to design a particular experimenter view (see examples below).

19.4 Hook up your mirror to the experimenter console

Since world identifiers are randomly generated by Meteor, it would be inconvenient to have to type them in yourself. Fortunately, you can specify the route for the one-way mirror in Meteor's `settings.json` to view worlds automatically:

```
{
  "public": {
     "turkserver": {
       "dataRoute": "expAdmin"
     }
  }
}
```

Under the **Experiments** view of the admin console, this will create a convenient button that you can click to watch ongoing or completed experiments (worlds) in real time.



19.5 Examples

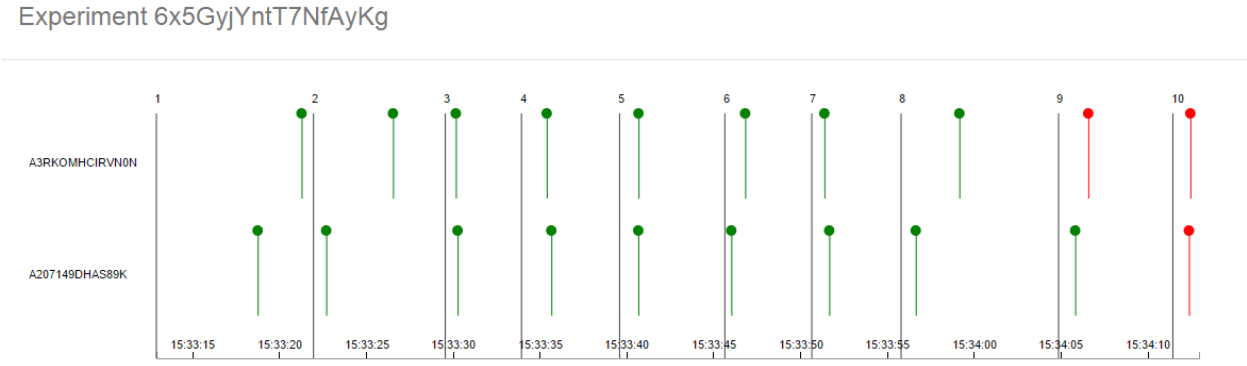
Although the code for a one-way mirror is minimal, it may help to look at the following implementations for inspiration.

19.5.1 Crisis Mapping

This code accomplishes a pretty complex one-way mirror that allows real-time observation over a number of different collections, producing the (fast-forwarded) video above. However, the code is still minimal.

- client code
- server code

19.5.2 Prisoner's Dilemma



This one-way mirror uses much simpler data, but also uses D3 to build a real-time visualization showing the experimenter much more than participants can see—including precise timing information.

- client code
- server code

CHAPTER 20

Debugging and Testing

Debug and test your app thoroughly before launching with real participants. We recommend going through the following steps:

1. Grab some friends and have them play with your app. Are the instructions clear? Is it obvious what a participant should do?
2. Test your app using multiple browsers in a local machine. Try dropping out of your experiment and seeing how your code handles it. Reconnect to your experiment (within a short amount of time) and ensure that data isn't lost.
3. Use the MTurk Sandbox for end-to-end testing, and make sure that the HIT can be submitted correctly and is collecting all the data that you need.

Think everything's ready? The ultimate test of your app is to conduct a *pilot*.

20.1 Code Troubleshooting

Because TurkServer runs alongside your app on both the server and client, strange behavior can occur when writing code without thoughtfulness. While we've tried our best to prevent easily-avoidable problems, some issues might still arise due to these reasons. These are some things to be aware of:

- **CSS conflicts**. TurkServer uses regular Bootstrap classes with no modification. If you use CSS classes that conflict with Bootstrap in your app, or selectors for unqualified tags, the admin backend will likely be messed up.
- **Meteor template name conflicts**. TurkServer templates all have the prefix `ts`.
- **Handlebars helper conflicts**. Internal TurkServer global helpers have the prefix `_ts`.

CHAPTER 21

Helpful Software

Meteor already makes it pretty easy to design a reactive and responsive user interface, but you may find some of the following packages useful. For the most part, Meteor makes the process of installing packages in an app a one-line command. TurkServer itself is a Meteor package.

- Bootstrap, a CSS framework for front-end development.
- Tutorials, a Meteor-specific tutorials package that for providing interactive and concise instructions for web apps. It is very useful for delivering easily digestible experiment instructions.

Add your links to helpful packages here!

CHAPTER 22

Frequently Asked Questions

Where can I find more resources for how to do online experiments?

Andrew Mao has given a tutorial at IC2S2 2016, partially based on the content of this site, and focused on how to run social experiments involving group interaction.

- Andrew Mao. **Experiments of collective social behavior in the online lab.**

At WINE 2013, Sid Suri and Andrew Mao gave a tutorial on the opportunities and practice of online behavioral experiments, targeted at a computer science audience. Studying online behavior is a compelling opportunity for computer scientists, and we believe that experimental work nicely complements existing modeling and data mining approaches by providing a way to verify hypotheses in a controlled setting and allowing for causal claims about interventions and observations, which are important for all types of system and mechanism design. Simply put, the best way to study online behavior experimentally is to do experiments with online participants.

- Andrew Mao and Siddharth Suri. **How, When, and Why to Do Online Behavioral Experiments.**

This tutorial consists of three sections: a review of how the experimental approach fits in and complements other methods for studying behavior; an overview of experimental design concepts from the abstract to the practical, including examples of pitfalls from our own experiments; and a survey of methods and tools that have been used in different fields to conduct experiments online.

I don't want to or can't write code. Can I hire a developer to build my web application (experiment) for me?

In theory, yes. **But there are many issues that can arise when this is process is poorly managed.** Here are some important ones.

A primary issue is the process of taking a research question and operationalizing it. In other words, when you build an experiment, you have a particular question in mind (*or at least you should!*) and you are building an app to collect data to answer that question. This process invariably requires feedback, revision, and modification when you find that the original plan doesn't answer the question as well as you thought.

In contrast, software development often follows specifications that are a priori much more concrete than research projects, and most developers will be much more productive working from a spec rather than the more unstructured research process in continually revising how the implementation of the user interface answers the research question. This leads to potential errors in translation in going from the question at hand to how it is implemnted, and the constant revisions to the original plan could result in the project taking a long time. But the real danger is in this frustration

leading to giving up on doing things correctly, which will result in a poorly designed experiment with questionable data. **As a community, we should ensure that reducing barriers to doing experiments doesn't result in many low-quality experiments.**

The other main aspect to consider is that running an experiment is not simply about having the software, but also the protocol of the experimenter and interaction with the participants. There is much less variation than in traditional brick-and-mortar experiments, as much of this is encapsulated in the software. However, there are still exceptions, a notable one being the importance of *engaging with workers and maintaining your*. As a result, it's very difficult to run a software-based experiment without understanding how the user interface and the rest of the app was built. Either you'll need to be familiar with it anyway, or your developer will also need to be a "lab manager" and run the experiment.

For all these reasons, I strongly suggest having a computer scientist or technically-inclined person as part of the research team—or at the very least, for such a person to work closely with a developer. Otherwise, make sure your developer understands the question you're trying to answer, what you're trying to do in the experiment, and what will be necessary to run the experiment successfully.

CHAPTER 23

Launching Experiments

This section of the guide is rough, and we could use your help to clean it up.

- When you finally launch your meteor experiment rejoice for you have earned a long-rest and a vacation on the beach
- Actually don't because things aren't as "automatic" as they may seem
- You still need to *manage your active experiment* to deal with users in real time
- As of this writing (9/24/15) it seems far more advantageous to launch many "small" HITs instead of several large ones
- Why? Turkers will **not** sit around your lobby waiting to getting matched for very long
- You will likely get angry emails and because they have to return your HIT in order to leave (or simply disconnect) it will be non-trivial/impossible to compensate them for their time (see Matching issues above for some strategies to avoid this)
- Though speculative, this is probably because as other Requesters launch their HITs, your's will fall to the bottom of the list getting lost and missed by Turkers
- To combat this, use the HITs menu on Turkserver to launch new HITs (with the same parameters you set using the MTurk menu) one after another
  - 10-20 seem's fairly reliable and finishes up within 15 minutes
  - 40+ really depends on the time of day and can take anywhere from 30 minutes to never finishing
- *[NEEDS TESTING]* It might be possible to simply add more assignments to an expired HIT and relaunch it, but it's unclear whether this "pushes" your HIT to the top of the page for Turkers to see [UPDATE 9/25/15] it looks like this works!

CHAPTER 24

Working with Workers

- Treat workers like people. They have a community, and talk to each other.
- Workers are active on many forums. They communicate there, and you can communicate with them!
- Monitor your reputation on TurkOpticon. A good reputation means that you will be able to recruit more workers and they will be more diligent.
- WeAreDynamo is a community of workers aiming to improve practices on MTurk.
- Consider best practices for crowdsourcing.

CHAPTER 25

Conducting a Pilot

Conducting a pilot experiment is probably the single most important thing you can do.

Turkers can even give feedback on your experiments. Check that people understand the instructions and know what they are supposed to do.

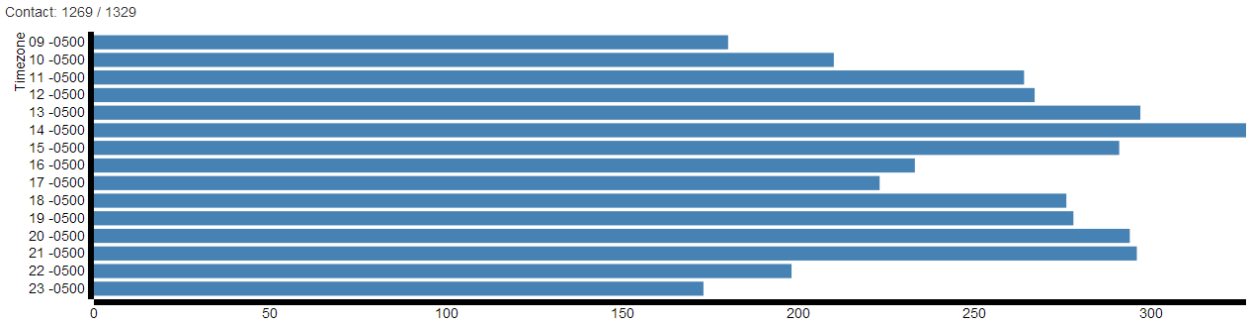
CHAPTER 26

Recruiting Large Groups

Running social experiments often requires many users to participate at the same time. So when are the greatest number of workers simultaneously available on Mechanical Turk?

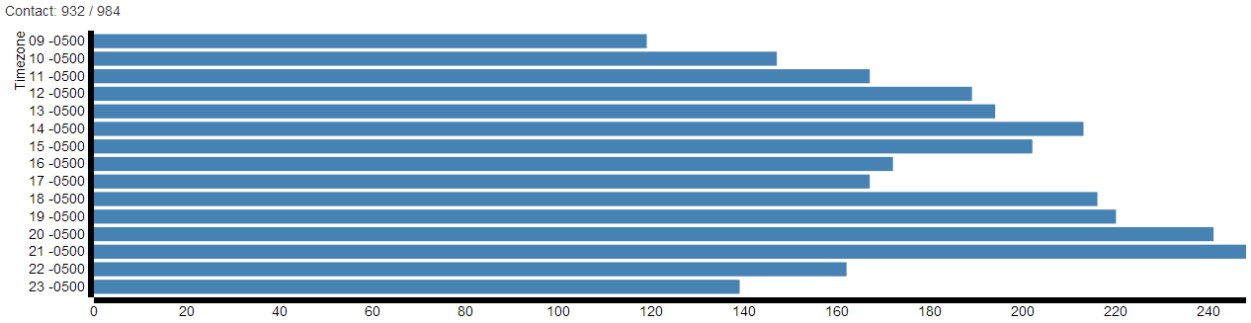
The answer to this question can help in general to guide when to post HITs, However, social experiments often use tasks where many workers must be online at the same time, such as for studying simultaneous interaction between many participants and sometimes this number must be pretty large. In order to do this, we typically schedule a time in advance and notify users to show up at that time.

This scheduling has generally been ad hoc, but in a few cases we've collected some extra data from workers about their availability, normalized by timezone. The following graph shows the distribution of over 1,200 workers available in each hour, per their reports:



The buckets are shown from 9AM to 11PM GMT -5, but are computed from the users' original timezones. A few caveats: the graph is for a few hundred US workers only, and the method of collection could be biased by time of day effects (the time of day that we collected the data will affect the time preferences of users.) However, the pattern squares with previous anecdotal observations that either the mid-afternoon or late evening are the best time to post group tasks, and that people don't tend to be online as much in the morning or at dinner time.

What happens when one starts using this panel of workers? After running a few dozen synchronous experiments all at 2PM EDT, the distribution of the remaining times now looks like the following, with just over 900 workers available. (For now, we are using each worker only once.)



As you can see, we've drastically reduced the number of workers available at 2PM, while not really affecting the number of workers available at 9PM. In order to get the maximum number of workers online simultaneously for our next synchronous batch, we'd do best to shoot for 9PM instead.

Keep in mind that there can be strong time-of-day effects here, as there are dissimilar populations likely to be online at certain times. Because of this, it's best to randomize over all of our possible experimental treatments simultaneously, so that the effect hits all of them. Collecting this time-of-day data is almost essential for overcoming the significant challenges in scheduling a large number of unique users all online at the same time.

The graphs above were generated by the panel recruiting section of TurkServer's admin interface.

26.1 Example of simultaneous recruiting

The following screenshot shows a recent study we deployed using this method, pulling out all the stops for this one, resulting in sessions with 100 participants arriving within 2 minutes of each other. They all participated for over one hour.

In our case, we randomized them into different-sized groups for this study. The simultaneous recruitment was necessary for all of our treatments to experience the same population, and the biggest group of users had 32 participants.

26.2 The experiment design triangle

Perhaps you've heard of the project management triangle: it's often been encapsulated as **"fast, good, or cheap: pick two"** because it's been invariably impossible to satisfy all three in executing projects.

In designing web-based behavioral experiments, there is a generally a similar triangle of three desirable properties that are very difficult to satisfy simultaneously:

- **Large sample size**: desirable to increase experiment power
- **Large number of simultaneous arrivals**: whether for synchronous experiments or to minimize time-of-day effects across treatments (as above)
- **Unique participants**:i.e. those who haven't seen the study before

When running large experiments that are synchronous or require extensive randomization, and that are constrained to unique participants, one will naturally run into the sample size ceiling. It is feasible to recruit about 1500 to 2000 active workers on MTurk on any given week, but fewer and fewer of them are available at the same time as you schedule your experiments.

Alternatively, this means that if you can design your experiments to be less sensitive to repeat participation, it's possible to have a lot of people participate and also gather a lot of data. This is possible for some experiments, but it can be challenging to ensure that the design is answering the right question, and that participants aren't being primed from the past or sensitive to experience.

Finally, the vast majority of online studies that are done don't require simultaneous arrivals because either they are for single users or they are not scheduled consistently at the same time of day. Without this constraint, it's possible to have many unique participants with a sample size of a couple thousand or more, but one should be careful that region and time-of-day effects are being controlled for.

CHAPTER 27

Pre-Launch Checklist

If you've done this before, you know that launching a live app to hundreds of people can be unnerving and stressful, because of the potential of things going wrong. The bad news is that stress can cause you to forget things and even more things can go wrong! The good news is that the best way to combat this is by making a checklist—just like NASA does before space shuttle launches.

27.1 Example Checklist

This is a checklist of common things we've done before running experiments. You should copy it somewhere and augment it with things that are specific to your experiment.

27.1.1 General Tasks

- Make sure any post-experiment cleanup from your last session is all done
- Make sure the code on your deployment server is up to date
- Make sure there is enough money in your account

27.1.2 Email forwarding setup:

- Make sure that email forwarding on the account used to make the HIT (more than likely lab dartmouth email) because Turkers *will* contact you during a HIT if something goes wrong
- It's a good idea to have an email signature to appear more professional

27.1.3 Auto Assignment approval setup:

- Make sure that you have some reasonable timer Assignment auto-approval (e.g. 1hr - 24hrs is probably reasonable)

- This is important in case you need to wipe a database or debug an issue and no longer have access to completed assignments in Turkserver

27.1.4 HIT expiration setup (called Lifetime in TurkServer):

- Make sure that you have a reasonable timer for when a HIT expires (this entirely depends on the nature of your study), i.e. how long it's visible for workers to be able to accept it
- Simple things might be just 24hrs or less (it's probably a good idea to run small batches at different times rather than a large batch all at once for multi-turker matching purposes and to more uniformly sample the Turker pool)
- This is also important in case you run into issues as noted above

27.1.5 Assignment duration

- Make sure to set this to something a little more than the length of your study
- This way if there are problems the a Turker's assignment will automatically end when this time is reached

CHAPTER 28

Managing an Active Experiment

28.1 While you're live:

Server status:

- On your local machine, in your Meteor project folder use mup logs -f to pull recent server logs and check for errors
- Turk Server status
  - Have a window open (or check occasionally) displaying experimentURL/turkserver so you can see what's going on in realtime

Keep tabs on a few places:

- Connections:
  - shows who's connected and in what state of Turkserver they're in
  - use this to see if something weird is happening (e.g. matching is breaking, people keep disconnecting, etc)
- Assignments Completed:
  - shows submitted HITs
  - use this to look at feedback people are providing, and a quick glimpse at experiment outcomes if people's behaviors are linked to their bonus payoffs
- Experiments:
  - shows current and completed experiment instances
  - use this to see who's doing your task, who's connected, who needed to be rematched, how long experiments are taking, number of completed experiments
- Workers:
  - search for a worker by their worker ID

  - If someone emails you with an issue you can see if they submitted a HIT, or returned it, or even if they're actively connected by searching their worker ID here

- Amazon HIT availability
  - You don't usually need to actively keep tabs on this but in case no one is signing up make sure your HIT is visible through Amazon or that there are Assignments remaining so new Turkers can sign up, see paying turkers below for how

Multi-Turk matching monitoring

- Often for multi-agent experiments (i.e. 2 or more people) some kind of matching mechanism exists in your meteor application
- Even if it works flawlessly there's no guarantee that Turkers will be connected *quickly* to other Turkers
- Less some kind of timeout feature within your app, another strategy is to increase their bonus payment through the TurkServer interface and send them a message apologizing for the extended time
  - Assignments Completed > Click bonus under the worker and type in the amount and message to send them

28.2 After you finish a batch

Approve assignments and Pay Bonuses (if any)

- Sometimes approving all assignments or paying all bonuses returns an error in the Turkserver admin interface. Nothing to worry about either way a few minutes and try again or first go to a specific user, refresh their status and then try again. If the error persists see this solution
- Try to be as prompt as possible as this helps the lab reptuation, speaking of which...

Check feedback on Turkopticon

- This the principle feedback forum where the lab gets reviewed by Turkers
- It's **critical** to maintain a good reputation, key to which is providing prompt service

Paying Turkers (alternative)

- If everything is going according to plan you should be able to pay folks directly through the Turkserver admin interface (including bonuses)
- Alternatively you can pay folks through Amazon's mTurk interface (e.g. in case you wiped your Turkserver database prematurely and don't know who completed your experiment!)
  - Login to requester.mturk.com
  - Click Manage > Manage HITs individually (on the right) > Assignments Pending Review/Download Results
  - Scroll down to see who needs to be approved, who was approved, who wasn't
  - Click any WorkerID to accept their HIT and/or pay them a bonus

Communicating with Turkers

- If a Turker has completed a HIT for you previously and they are in your Turkserver database you should be able to contact their through the Turkserver admin interface:
  - Assignments (Active or Completed) > hover over Worker ID > click mail glyphicon
  - You should be able see see sent messages under Panel

        * WARNING: Messages might appear here as if they were successfully sent, but this may not be true! Turkserver currently (as of 9/22/15) requires Turkers to have completed as least 1 HIT with you previously, i.e. Turkserver needs to "know" who this worker is *before* they complete a HIT

- Otherwise you can email workers directly through Amazon's interface:
  - Follow directions above for paying folks through Amazon's interface
  - Clicking any WorkerID will also give you the option to email them

Checking available Assignment(s) information

- Depending on how your Meteor experiment is setup you can get a sense of how many Assignments are left simply based on a combination of the following:
  - HITs > Assignments - Assignments Completed - Assignments Active
- This may not always be reliable if you have special rules setup for rematching Turkers, HIT comprehension, disconnection etc
- In this case navigate to Amazon's request mTurk interface by following the directions above
  - Under Manage HITs individually you should see the "Remaining Assignments" value along with other info about your HIT
  - This is the "gold standard" regardless of what Turkserver says because it's what Turkers will be able to see as well

CHAPTER 29

Troubleshooting

- If something is going wrong you should immediately freak out! (ask Eshin he know's all about this)
- Actually don't that. Just figure out what's up by checking the following:

Is the hosting server online?

- PANIC LEVEL: Low
  - Not the worst thing in the world because if the server is restarted quickly enough, Turkserver will automagically reconnect users in the middle of experiment to the exact point they were at!
  - If not, you risk losing active assignments, getting a few angry emails, but no potentially new users will be affected
- HOW TO CHECK:
  - Try to ssh to the webserver where your meteor project is hosted
  - If you can't or if experimentURL/turkserver (the Turkserver admin interface) is unreachable your experiment is no longer being served to Turkers
- WHAT THIS MEANS:
  - Users currently doing your experiment will be stuck
  - New users trying to view your HIT they won't see anything in the mTurk window or they'll see a message saying the website doesn't exist/isn't reachable; they should not be able to click Accept HIT at this point but this is unclear (plus it would be kind of stupid for them to do so anyway)
- SOLUTION:
  - Try and restart your web server as quick as possible

Is Turkserver working properly? (i.e. is your Assigner behaving?)

- PANIC LEVEL: High
  - This can be pretty bad depending on how your experiment is setup
  - You risk losing active assignments, and new assignments can get stuck in the lobby

- HOW TO CHECK:
  - Pull server logs (mup logs -f) from your local Meteor project folder to see if there are application errors
  - You should be able to see them if you're currently ssh'd into the hosting server as well
- WHAT THIS MEANS:
  - Users currently doing your experiment *could* be ok, but might have problems submitting their HITs
  - New users will likely get stuck in the lobby and never be pushed to the experiment
  - You can have a huge list of stuck Turkers very quickly!
- SOLUTION:
  - Stop all experiments (your call):
    - If you don't think that current users will be affected by this problem don't do this
    - Otherwise: Turkserver admin interface > Experiments > Stop All Experiments
  - Immediately force expire your HIT:
    - Turkserver admin interface > HITs > Click on your HIT > Force Expire
    - This way no new users will be affected
    - Current users may not be able to submit their HITs after this
  - Try and contact Turkers and pay them for accepting the HIT through Amazon's mTurk interface, or through Turkserver
  - Debug your code at your leisure

Is your Meteor app not working?

- PANIC LEVEL: ?
  - This is the most ambiguous type of issue because depending on how your experiment is setup, here are a few possible situations ordered from low to high panic level:
    - New users may still be able to accept the HIT, current users may still be able to submit the HIT, but your experiment may be rendering improperly or data may not be saving properly or saving at all = Turker thinks everything is ok
    - New users can't see the HIT and current users are stuck = Turker will be confused and email you if they're in the experiment
    - New users can still accept the HIT but current users can never submit = Turker will email you angrily
      - THIS IS BAD because you run the same type of risk as Turkserver crashing
- HOW TO CHECK:
  - Pull server logs (mup logs -f) from your local Meteor project folder to see if there are application errors
  - You should be able to see them if you're currently ssh'd into the hosting server as well
  - Check your database:
    - ssh to your web server and navigate to /opt/appName
    - mongo appName
    - `db.yourdatabase.find().pretty()`
  - Make liberal use of console.logs() when things work as expected or don't during your meteor app
- WHAT THIS MEANS:

  - Again this is pretty ambiguous depending on your code see above for a few possible scenarios
  - Users currently doing your experiment *could* be ok, but might have problems submitting their HITs
  - New users will likely get stuck in the lobby and never be pushed to the experiment
  - You can have a huge list of stuck Turkers very quickly!
- SOLUTION:
  - You probably want to stop all current experiments and force expire your HIT before debugging your code, see above for how

CHAPTER 30

Post-Experiment Cleanup

- clear disconnected instances
- cancel disconnected assignments
- pay people
- update qualifications if necessary
- save your data!

CHAPTER 31

Indices and tables

- genindex
- modindex
- search