Behav Res
DOI 10.3758/s13428-014-0515-6

Conducting real-time multiplayer experiments on the web

Robert X. D. Hawkins

Abstract Group behavior experiments require potentially large numbers of participants to interact in real time with perfect information about one another. In this paper, we address the methodological challenge of developing and conducting such experiments on the web, thereby broadening access to online labor markets as well as allowing for participation through mobile devices. In particular, we combine a set of recent web development technologies, including Node.js with the Socket.io module, HTML5 canvas, and jQuery, to provide a secure platform for pedagogical demonstrations and scalable, unsupervised experiment administration. Template code is provided for an example real-time behavioral game theory experiment which automatically pairs participants into dyads and places them into a virtual world. In total, this treatment is intended to allow those with a background in non-web-based programming to modify the template, which handles the technical server–client networking details, for their own experiments.

Keywords Web experiments · Node.js · Behavioral game theory

Introduction

Many phenomena studied in group behavior research emerge from real-time interactions. In financial or prediction markets, participants can buy or sell shares at any point in time, not just on the hour. Prices are immediately updated and made available for other participants to see and act upon, a condition known as perfect information (Boer, Kaymak, & Spiering, 2007; Deck & Nikiforakis, 2012). Multi-agent systems like crowds, flocks, and social networks update their perceptions of neighbors in real-time and are subject to time-delay constraints forcing individuals to act asynchronously, yet they often remain stable and capable of reaching group-level consensus (Cao, Morse, & Anderson, 2008; Olfati-Saber, Fax, & Murray, 2007). In group foraging tasks, players' knowledge of resource levels and the positions of other players must be kept up-to-date in order to make informed decisions and converge to accurate equilibrium distributions (Goldstone, Ashpole, & Roberts, 2005).

Recent experiments in behavioral game theory have demonstrated that allowing real-time interaction dramatically increases cooperation levels in the prisoner's dilemma and also facilitates cooperation under robust communication protocols in public goods provision (Friedman & Oprea, 2012; Charness, Oprea, & Friedman, 2012). Similar dynamics also play a role in student-teacher interaction, collaborative brainstorming, social influence, group decision-making, and interpersonal communication (Cialdini & Goldstein, 2004; Dale, Fusaroli, Duran, & Richardson, 2014; Friedman & Oprea, 2012; Michinov & Primois, 2005; Miller, Garnier, Hartnett, & Couzin, 2013).

There are two primary 'use cases' that drive the demand for a way to implement such environments on the web and across devices. First, foundational examples of group behavior like those mentioned above are increasingly included in the curriculum of undergraduate courses in psychology, education, and economics. Instructors are looking for ways to increase engagement in the classroom, making interactive experiment demonstrations particularly attractive. With the tools we describe in this paper, instructors can code up an experiment on a web server, post a URL for students to follow on their laptops, smartphones, or tablets during class, and immediately collect the results for discussion. There are no software requirements for the students aside from a browser, and mobile devices are treated no differently than desktop devices as long as keyboard input is avoided.

Second, online labor markets like Amazon Mechanical Turk offer many advantages to researchers in the behavioral sciences but these advantages are currently unavailable for many group behavior applications due to the technical

R. X. D. Hawkins (✉)
Department of Psychological and Brain Sciences, Cognitive Science Program, Indiana University, Bloomington, IN 47405, USA
e-mail: hawkrobe@indiana.edu

Published online: 01 October 2014

difficulty of running real-time, multiplayer experiments online. These advantages include recruitment from a large, stable subject pool, quick deployment of new experimental conditions, and remote administration with little to no time commitment (Mason & Suri, 2012). Furthermore, early studies demonstrated that results from the web version of an experiment were qualitatively indistinguishable from a controlled laboratory version, regardless of compensation level (Mason & Watts, 2010; Paolacci, Chandler, & Ipeirotis, 2010), and the subject pool is much more diverse than the typical sample of American college students (Buhrmester, Kwang, & Gosling, 2011).

Most behavioral experiments currently being developed for the web do not require participants to directly interact with one another. Those that do, like the public goods game reported by Suri and Watts (2011), are staged in discrete blocks. In other words, each turn consists of players simultaneously and independently entering their responses, usually into text boxes, under some time pressure. Once the time window is up, responses are collected and the web page automatically refreshes with updated information for another round of play.

While staged designs are straightforward to implement online with HTTP request methods (e.g. GET and POST), they are insufficient for the full range of group behavior contexts. The real-time examples given above require participants to see what actions others have taken immediately after those actions have taken place and to take actions at any point in time without artificial restrictions. For such tasks, converting the natural real-time design to a staged design for ease of running it online would sacrifice their central mechanism.

Recently, the software development community has coalesced around a set of JavaScript-based tools built specifically to address these challenges, which were originally encountered in building web applications like chat rooms and fast-paced multiplayer games. In this paper, we adapt these preexisting tools to run low-maintenance real-time multiplayer experiments directly in the browser. The resulting system does not require an experimenter to be present to administer the experiment, nor does it cap the number of experiments running simultaneously. Due to extensive coverage elsewhere (e.g. Mason & Suri, 2012; Paolacci, Chandler, & Ipeirotis, 2010), we will not dwell on the details of running experiments on specific crowd-sourcing platforms. Although our code was tested and successfully run on Amazon Mechanical Turk, it is intentionally platform-independent.

The architecture of the system is depicted in Fig. 1. We begin by running server-side JavaScript code with Node.js, using the Express module to listen for users to connect. When users access the URL on which our system is listening, the module Socket.io is invoked to establish and maintain a persistent connection between the user (or client) and the server. Upon first connecting, users are automatically placed in a waiting room or trigger a new experiment instance if other

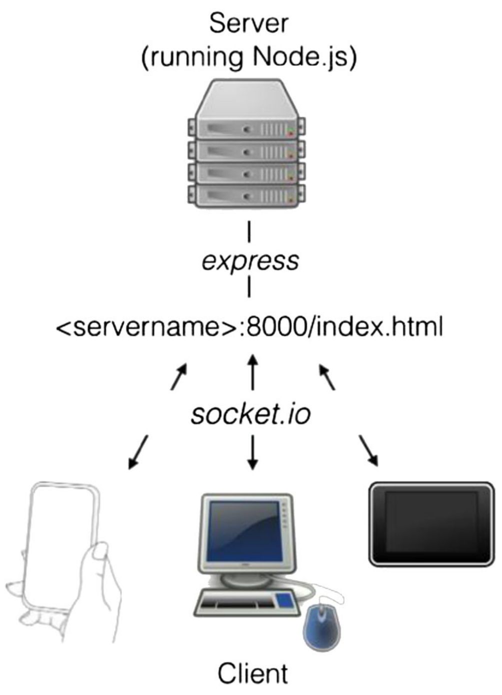


Fig. 1 Diagram indicating how various tools fit together. The server runs Node.js code, which listens on a URL using the express module. When clients on any devices connect to that URL, a Socket.io connection is established. jQuery and the HTML5 canvas are used client-side to collect input and display information to the user (Icons reproduced under Creative Commons CC0, except server and desktop workstation icons, from the RRZE Icon Set, distributed under Creative Commons CC BY-SA)

players are available. Throughout the experiment, this connection is used to update the user's display with new information from the central server (using the HTML5 Canvas for graphics) and asynchronously letting the server know about user input (collected using jQuery) as soon as either becomes available. The experiment proceeds according to the logic encoded in the server-side code, writing the current state of the world to a text file on the server as often as the experimenter would like. When this logic dictates that the experiment has ended, the user is redirected to another URL such as an exit survey or a portal to submit a payment request.

While there exists substantial technical documentation on each of these components individually (Hughes-Croucher & Wilson, 2012; Takada, 2012; Tilkov & Vinoski, 2010), the following sections focus more broadly on the complementary roles they play in the context of online group behavior experiments. Documented code and instructions for getting our demonstration running from scratch are available online.[1] The demonstration itself can be run without any programming at all, along with many variations formed by tweaking the variables set at the top of the `game.core.js` file. However, if the reader has a background in programming, even in non-web-based languages like MATLAB, R, or Python, this framework is flexible enough to implement nearly any real-time,

[1] https://github.com/hawkrobe/MWERT

multiplayer scenario. The goal of the template, then, is to automatically handle the most difficult web-based aspects, therefore largely reducing it to a tractable non-web-based problem (which may still, of course, require substantial creativity and work). The goal of the paper is to guide the reader through steps that may need to be taken to modify the code, including handling some peculiarities of JavaScript, and, at a more conceptual level, to help the reader assess whether these tools will be an appropriate solution for their problem in the first place.

To accurately assess the time investment required to put an experiment online, it is important to note that one additional step must be taken to run experiments securely through an online labor market: a database must be shared across the experiment discussed in this paper and a 'gateway' website (e.g. to post as an 'external HIT' on Mechanical Turk). The task of creating the database and gateway is common to all web experiments, not just those featuring a real-time, multiplayer setting, and is therefore out of the scope of this paper and its accompanying template code (but see Goldin & Darlow, 2013; Mason & Suri, 2012; Reips, 2002). Since the problem of updating a database in real-time from within Node.js is specific to our system, however, we will cover the tools necessary to do so below. For researchers who have already created a database and gateway for previous web experiments, the template code has been designed for easy integration with pre-existing SQL databases, and instructions are included in the GitHub README.

Tools

Node.js

There is a significant engineering challenge behind running real-time multiplayer experiments in the browser. A potentially large number of players must somehow communicate with an ongoing server-side process that listens for input from each player and responds instantaneously to each with customized information about what their screens should display. Node.js is a server-side JavaScript environment built specifically to handle such applications (Hughes-Croucher & Wilson, 2012; Tilkov & Vinoski, 2010; Takada, 2012).

In addition to undergirding a set of modules to easily serve up files and allow clients to interact (including the essential Socket.io), Node.js handles the onslaught of tasks in an intelligent, efficient way. Input/Output (I/O) operations are often a bottleneck for web servers, especially in real-time experiments where actions and updates must be sent and received many times per second. Traditional I/O techniques scale poorly. For instance, synchronous I/O keeps a list of requests and carries them out in sequence, always waiting for the first request to complete before beginning the next. This is simple to implement, but wasteful; the server or client spends most of its time waiting, when it could be working on the next task.

One common way to fix this problem (used by Apache servers) is to spawn a new, independent 'thread' with each new request, so that the new thread can work on the new request while the old thread is still waiting. This creates other problems, though: each thread comes with a substantial memory overhead, and can wreak chaos when the different threads need to access a shared resource. Node.js avoids all of these problems by sticking to a single thread and automatically switching between tasks in an asynchronous, non-blocking manner using "callbacks," a powerful concept in programming languages.

In simple terms, a callback is a piece of code attached to an event, such that the code only runs when the event takes place. The core process in Node.js, then, is an "event loop" which continuously checks whether any events have occurred. If so, the loop takes a break to run the attached callback function. This may be familiar from client-side JavaScript, where some code may run when a user clicks a button or presses a key. Node.js takes the same approach to server-side I/O events. When the server needs data about a user's current progress in the experiment, for instance, it passes a callback function along with the request. It can then forget about the request and proceed to perform other tasks. When the I/O operation has completed and the data is ready, the event loop detects this and runs the function specified by the original callback. This allows a high volume of requests to be processed in a very short amount of time.

The GitHub README explains how to download and install Node.js and accompanying modules locally or on a web server, and we will not cover the details here. Instead, we proceed to give some demonstrations of the use of callbacks through a pair of essential open-source modules developed and maintained by the Node.js developer community: Express and Socket.io. Callbacks are the main difference between writing (and modifying) code in Node.js and writing code in 'synchronous' languages like MATLAB, R, or Python.

Serving files over the network with Express

Express is a highly abstracted web application framework for Node.js, which allows us to start up a minimal web server where participants will be directed when ready to participate in the experiment. This can be done in very few lines of code, which we put inside a file called `app.js`:

```
var app = require('express')();
var server = app.listen(8000);
app.get('/*', function(request, response){
        var file = request.params[0];
        response.sendfile('./' + file);
    });
```

Walking through this code, we first use the `require ()` function to load the Express module. The section line tells the module to begin listening on port 8000 and returns the resulting server for later use. The rest establishes our first callback to handle file requests. The `app.get()` function takes a path to match against (using an asterisk means ours will match all user queries), and also a function to be run when that path is matched.

Inside the callback function, we automatically have access to two parameters: the *request*, which contains information about the client and the content of their request, and also the *response*, which is in charge of sending files back to the client. In this simple example, we simply extract the name of the file that the user is requesting and then give the response object the appropriate path to that file on the server machine. Suppose hypothetically that the web server where this code has been placed is called "research." To begin listening for users with Express, we would enter "node app.js" at the command line. Now, if a user tried to navigate to "http://research:8000/index.html" in their browser, the `app.get()` event would be triggered, with its corresponding callback function. It would extract "index.html" as the requested file name, then look for a file of this name in the same directory as `app.js` to serve to the user. This `index.html` file loads all the client-side JavaScript code we want to run. One major function of this client-side code is to establish a persistent pipeline to the server with Socket.io. Note that the code in `app.js` should not need to be modified to implement new experiments – after it establishes the server, it automatically attaches various callback functions, which will be defined elsewhere, to the appropriate events. This way, the programmer can just worry about what the functions need to do without worrying about the way they're wrapped together.

Maintaining persistent server–client connections with Socket.io

The central problem of client–server networking is synchronizing and communicating across the separate game states maintained on different machines. Socket.io builds upon the framework provided by Express to sustain a persistent connection between experiment participants and the server, allowing parties to send and receive messages at any point in time (Hughes-Croucher & Wilson, 2012). It has a number of nice properties. It can detect which browser a participant is using in order to automatically choose the fastest and most capable communication method the user's system can handle. It will also attempt to reconnect if a user's connection drops suddenly. This module is the core of multiplayer functionality.

In Socket.io, we again attach callback functions to specific events, such as a user connecting, disconnecting, entering input, or sending an update of their state. We use the `on()` construct to define and label an event on one end of the client–server relationship, then use the `emit()` construct with that label to trigger it from the other end of the relationship. The basic setup example is given below. This appears in `app.js` and will not need to be changed:

```
var io = require('socket.io')(server);
io.on('connection', function (client){
  client.emit('onconnected',{ id: "AD5000"} );
  client.on('disconnect', function (){
    endGame(client);
  });
  client.on('message', function(s){
    onMessage(client, s);
  });
});
```

After stacking the Socket.io module on top of the Express server set up in the previous section, we attach a function to the 'connection' event, which is run every time a new participant (client) connects. The first line inside that function uses `emit()` to trigger the client's 'onconnected' event and pass along some data assigning them an id. That 'onconnected' event is defined and labeled in a client-side script (in our case, `game.client.js`) that runs when they load `index.html`. Next, we establish a 'disconnect' event that the client will automatically trigger when leaving the webpage, in order to gracefully alter the experiment to handle their absence. In the `endGame()` function, we could remove their avatar or end the game for others if their presence was essential to the experiment's validity.

The final 'message' event is intended to simplify the task of modifying the template for users without a web programming background. Of all the code in the template, Socket.io functions are the most likely to need to be modified to build original experiments. While Socket.io allows several sophisticated communication semantics, our template code places an extra layer on top which reduces this problem to sending and parsing strings. By attaching an event called 'message' to the client object (and also to the server object, elsewhere), we route all handling of client-side events through a single function (`server_onMessage()` in `game.server.js`) and all server-side events through a single function (`client_onMessage()`, in `game.client.js`).

The client can send a message to the server by calling `game.socket.send(s)` where `s` is a string, and the server can send a message to the client by calling `game.<client_object>.send(s)`. Note that since only a certain number of people are associated with each 'game' object (two, in the example below), this effectively separates participants into separate 'rooms' each with their own communication channel. Inside the opposite `onMessage()` function, the programmer sets up an if statement detailing what to do when that string is received.

For example, suppose each player controls an avatar and the programmer wants to allow the player to set the direction this avatar is moving by clicking on the screen. This requires the client to send a Socket.io message to the server, notifying it of the change. The actual collection of click coordinates is handled by jQuery, described below, but for now suppose the *(x, y)* coordinates are (120, 240). After extracting these numbers, the client might send the string "c.120.240" to the central server by calling `game.socket.send("c.120.240")`, using the key 'c' to distinguish the 'click' type of event and passing along the associated data. Inside the `server_onMessage()` function on the server-side, we can split this string, match on the key, write an instruction to change that client's "destination" variable, and pass along the update to the other client. Any server–client communication a researcher might need to add can be implemented in this fashion, which is familiar from conventional imperative programming languages.

One final role of Socket.io may be relevant for modifying the system. Part of what makes web programming confusing is the need to constantly synchronize variables across the different machines in the network. When a researcher adds a new variable to the client-end code, the server-end code cannot initially access it, even if it's global. The server holds one instance of the state of the experiment, but each client is tasked with maintaining its own state. For most experiments, the server can be solely responsible for all moment-to-moment computations, updating its local variables and then broadcasting the results to the clients. This regular update (sent every time the screen is redrawn – multiple times per second) is handled by a function called `server_send_update()` in `game.core.js`. It bundles together all the variables the client may need to know about and triggers a client-side function called `client_onserverupdate_received()` which updates the clients' local variables to match the server. When adding new variables to the code, it will be necessary to include them in this bundle.

Graphics with HTML5 canvas

In the previous sections we used the Node.js infrastructure, along with Express, to monitor and respond when participants arrive on the page and Socket.io to form a connection over which both sides can react to events and pass information. We will address the problem of collecting user input in the next section. Here, we address the problem of updating the players' screens to reflect the current state of the experiment. In a group foraging task, for example, we might want to represent the participants' avatars as triangles on a grid and indicate current resource levels via a set of graphs at the bottom. In a behavioral game theory experiment, we might give the players a choice between two payoffs, represented as circles with accompanying text labels.

There are additional technical requirements for implementing this component of the experiment system. These requirements are unique to the real-time aspect of our experiments, but not necessarily the multiplayer aspect: (a) we need to update the display fast enough to allow real-time interactions with a low memory footprint and (b) we need to draw the image directly in the browser such that it will be compatible across mobile and desktop devices. If one player decides to take an action that changes the state of the game, such as changing the direction that his or her avatar is moving on the grid or buying shares of some stock, the other players' screens must immediately reflect this new state. Both of these technical challenges are met by the `<canvas>` element introduced in the HTML5 protocol. HTML5 is supported by all modern browsers, including mobile browsers, and can work on some older browser versions through vendor prefixes. Essentially, the canvas element gives us access to a set of simple JavaScript functions to draw shapes, colors, and text inside a participant's browser window. We initialize it in the `index.html` code that we served to the participants with Express:

```
<body>
   <canvas id="viewport"> </canvas>
</body>
```

In the client-side JavaScript (called `game.client.js` in our source code), we can then grab this element and extract its 'drawing context', an object containing a wide range of useful methods to display information on the user's screen. For example, to present experiment instructions, item labels, or real-time messages to participants, we can use the `font` property to set what the text will look like, then call the `fillText()` method to draw it at a specific *(x, y)* location:

```
viewport = document.getElementById('viewport');
ctx = viewport.getContext('2d');
ctx.font = "15pt Helvetica";
ctx.fillText("Welcome to the experiment!",250, 250);
```

To draw more complicated items, such as the participant avatars, resource indicators, line graphs, and other visual aids populating the world of the experiment, the appropriate canvas tool is the 'path.' Objects are drawn in a block of code beginning with `beginPath()` and ending with `stroke()`, with the intervening commands describing every line segment making up the larger figure. Straight lines can be drawn by placing the 'pencil' at a coordinate location using `moveTo()`, and then specifying another coordinate location as the line's endpoint using `lineTo()`. Circles can be constructed around the given point using `arc()`, and areas can be filled in using `fill()`. For example, to draw a 5-pixel radius black circle centered at (50, 50) with a thin gray border, we would use:

```
ctx.beginPath();
ctx.arc(50,50, 5, 0, 2* Math.PI, false);
ctx.fillStyle = 'black';
ctx.fill();
ctx.lineWidth = 1;
ctx.strokeStyle = 'gray';
ctx.stroke();
```

Note that all drawing takes place client-side, and for ease of modification, we have collected these functions together in a file called `drawing.js`. As mentioned in the previous section, if a participant's display is not updated correctly even though the server has the correct information, this may be because it has not properly communicated that information to the client in the `server_send_update()` function. For our application, we decided to make the central server the 'expert' in charge of synchronizing the users' states. This means that both players will always draw the same image at the same time. However, more advanced applications may use client-side prediction to smooth out movements, in which case the server allows the different participants to maintain slightly different states and catches them up to one another when movement has stopped.

Collecting user input with jQuery

The other most important piece of information in an experiment comes from participant interactions with the webpage. Again, this is not unique to real-time, multiplayer experiments, but worth mentioning in the context of HTML5 and for readers new to web experiments. To communicate participant input in the form of mouse clicks or keyboard presses, we must bind *event handlers* to HTML elements. jQuery is a popular, powerful JavaScript library with this ability, among many others. Say we want to register the coordinates of a click anywhere in the `<canvas>` element of our window and take some set of actions using those coordinates. Since the canvas is listed in the body of `index.html` with the id "viewport", this is the name of the element we will use in the call to jQuery:

```
$('#viewport').click(function(e){
  e.preventDefault();
  var offset = $(this).offset();
  var relX = e.pageX - offset.left;
  var relY = e.pageY - offset.top;
  client_on_click(relX, relY);
});
```

Referencing an HTML element with `.click(function(e){ …} )` is a shortcut to bind a 'click' event handler, which will execute the given function when a click is detected in the canvas. The variable `e` contains the details of the event, including its coordinates. We always use `preventDefault()` to protect against any unwanted actions the browser or operating system may take automatically, such as selecting or allowing the user to drag-and-drop the element. The rest of the snippet converts absolute coordinates (i.e. location in the entire browser window) to coordinates relative to the boundaries of the "canvas" element, and passes them off to a function called `client_on_click()` which takes the desired actions using current values of global variables.

An event handler only needs to be bound to an element once, at initialization, and then the program will automatically run the specified function each time the event takes place. If we accidentally attach more than one of the same event handler to an element (perhaps at each iteration of a loop), the function will be called once for every event handler, even though the event itself only occurred once.

Integrating with a database

To run an experiment created from the tools described in the previous sections using paid participants from online labor markets like Amazon Mechanical Turk, it is necessary to link the system to a database. First, it is our way of securely sharing sensitive information across the experiment URL and the referring website (such as how much to pay the participant). Second, we use it to verify that participants accessing the URL are, in fact, workers from Mechanical Turk who have accepted a HIT, or task. Third, with each "round" of the game, we must update the player's current performance in order to have a way to pay them if they suddenly disconnect.

We used MySQL to handle these basic needs; however, other databases are known to work well with Node.js, including MongoDB and Redis. Services like psiTurk (McDonnell et al., 2012) or TurkGate (Goldin & Darlow, 2013) have been developed to help handle the gateway stage of the experiment before the participant is directed to the URL hosted by Node.js, including managing the database, and we only need to be concerned with being able to access this database inside the Node.js process. To accomplish this, we used a module called *node-mysql*[2] that provided a convenient way to query the database from the Node.js process. More information on using this module is provided in the GitHub README.

Limitations

While the previous section detailed the strengths of several tools for running real-time, multiplayer experiments on the web, several weaknesses must be noted as well. Some are intrinsic to the tools and may be fixed or improved as they

[2] https://github.com/felixge/node-mysql

continue to be refined by the open-source community. Others are intrinsic to the endeavor of running multiplayer experiments, and must be addressed. We begin by discussing the current technical limitations of Node.js and Socket.io. One natural question is the number of simultaneous connections that can be handled – could you have 100 users interacting together in a foraging experiment or are you limited to groups of two or three? How many groups of two or three can be running simultaneously?

Like all software, there are two primary dimensions along which Node.js and Socket.io processing can break down: memory usage and CPU load. If both of these are managed well, developers report being able to handle about 8,000 – 10,000 messages per second without any lag or slow down – far more than necessary for most research purposes. For a point of comparison, the application described below sends regular updates containing game state to each connection once every .66 seconds, with periodic messages sent back to the server as user input is registered. This means we could have approximately 6,000 participants running simultaneously. If recruiting through Amazon Mechanical Turk, it is rare for more than 20 or 30 people to connect at once, even at peak times. This said, it is worth commenting on possible ways these two bottlenecks can be mismanaged.

Because most web servers have a surplus of RAM, the memory bottleneck is essentially the memory limit of the engine Node.js uses to compile and optimize JavaScript code. At the time of writing, this is the V8 engine developed by Google for the Chrome browser, which has a memory limit of 512MB for 32-bit architectures and 1GB for 64-bit architectures (although there are options to increase this up to 1.7GB). The number of users concurrently sending and receiving messages, the frequency of message passing, and the size of individual messages all affect memory usage. Using HTML5 for graphics is a good way to cut down on message size, since no image data needs to be transmitted, only small JSON structures containing the relevant values (e.g. player position and payoff values). Under typical use conditions – small messages, sent only a few times each second – it takes thousands of connections to hit this limit. Of course, Socket.io is a relatively new technology, and from time to time developers report bugs regarding memory leaks, which can rapidly increase the amount of memory used even by very few users. These bugs are becoming less and less common as Socket.io matures – in May 2014, version 1.0 was released, fixing many existing issues – but certainly remain a risk.

The other bottleneck is CPU usage. As detailed in the section on Node.js, the real strength of the system is handling I/O (input/output). If the event loop is tied up with intensive computations, the user will experience lag or dropped connections. As long as the asynchronous programming model is followed, delegating computation-intensive jobs to forked "worker" processes and keeping them out of callbacks, this will not be a problem. The template code provided adheres to this model, and most changes that might need to be made to modify the experiment can take place inside pre-existing callback functions. The only exception is if new Socket.io events need to be added, perhaps to register an additional source of user input or some new step of game logic, in which case the code should be placed in a new function and a new line added to `client_onMessage()` or `server_onMessage()` with the name of the event and a reference to that new function.

Next, we turn to several issues that are unique not to the specific tools, but to multiplayer online experiments in general. Most participants recruited from online labor markets are trustworthy, but when putting an experiment online, several security concerns must be addressed. Many of the most common issues, like SQL injection attacks and the need to encrypt sensitive files, are common to all web applications and are addressed elsewhere (Bishop, 2002; Mason & Suri, 2012). Still, it is easy to imagine players with multiple Mechanical Turk accounts signing up to play themselves for maximal earnings, or for friends to collude through some other communication channel, including Mechanical Turk message boards. There are several ways to address this problem, although a catch-all solution has not yet been found. Most simply, we can catch participants who are attempting to play against themselves by checking IP (Internet Protocol) addresses – if the participant has different accounts up in different browsers, they will register as the same.

We could possibly prevent more elaborate cases of collusion by making a single waiting room where all players pool as a countdown until the next game progresses. Once the countdown finishes, players would be randomly paired so that they would not be able to predict who their partner would be. There remain some situations that can slip through this preventative measure, for instance if there is a large group of colluders to fill the waiting room or if the number of participants who have accepted HITs at a particular time is not large enough to make the waiting room sufficiently anonymous. These questions deserve further consideration elsewhere, and these measures have not been implemented in the accompanying code.

One other issue is the risk of drop-out. One nice feature of Socket.io is that it will automatically seek to reestablish a connection if a client encounters a brief lapse in network connection, so accidental disconnection is unlikely. Sometimes, though, participants may just grow tired of the experiment and decide their time is better spent elsewhere. Like all web experiments, this phenomenon has worrisome implications for motivational confounding (Reips, 2002), but for multiplayer experiments in particular we must also implement a way of handling a dropout gracefully for the remaining participants in a session. For pairs, this involves redirecting the remaining participant to a webpage explaining what happened and assuring them they will be paid for their work up to that point. This is where the database is useful as a way of saving progress throughout the experiment. For larger groups, it may be possible to allow the others to continue, although this attrition should be noted in the data analysis.

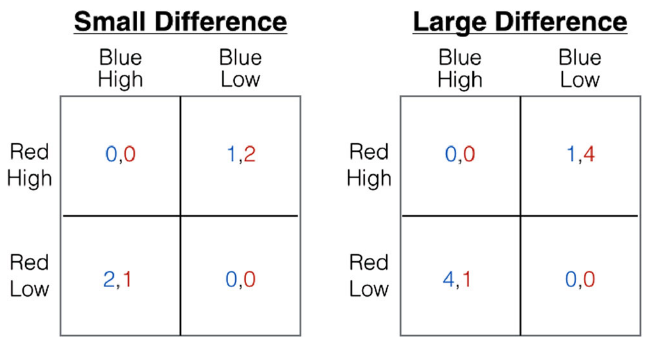
Finally, there is the issue of attention. There's nothing stopping a participant from switching to another browser tab in the middle of the experiment, or while waiting for another player to join. Since the experiments we're interested in run in real-time, this inattention may cause a participant to miss a crucial action taken by another player, thus endangering the assumption of perfect information. Luckily, there is a way to track whether a tab is being viewed at any point in time, using the W3C visibility API. This is implemented in the template code and can be written to the file at each step of the experiment for reference during data analysis. We have also implemented a way to attract a player's attention when another player has connected: if they are not viewing the page, the title blinks off and on in the browser tab, which is a standard convention to notify users that a page has updated.

Application

To demonstrate its utility, we used this JavaScript framework to conduct a real-time group behavior experiment on Mechanical Turk. The source code for running this experiment is publicly available on GitHub with instructions on how to modify it for other uses. The emphasis in this section will be placed on the technological requirements demanded by the experiment setup, and the way that the tools discussed above satisfy those requirements. Like Friedman and Oprea (2012), our experiment tested a distinction in behavioral game theory between a synchronous *staged* design and a real-time, asynchronous *dynamic* design.

We studied a variation on the well studied "Battle of the Sexes" game, which induces some degree of tension between efficiency and fairness. There are two possible 'targets' to choose from, each corresponding to some payoff. One payoff is larger than the other, thereby making the high-payoff target more attractive to both players. However, if both players choose the same target, neither player gets a reward. We manipulated two aspects of this game – the time-course of response and the disparity between the two payoffs – forming a 2 × 2 factorial design. The payoff matrices are given in Fig. 2. This game is interesting from the perspective of group behavior because the players must implicitly organize their behavior into a mutually beneficial strategy in order to be maximally successful: if both players follow the 'greedy' strategy of always choosing the highest payoff, they will come away empty-handed.

When the game is played against the same opponent many times, one possible solution to this problem is to 'alternate,' where each player takes the small payoff on one round in exchange for getting access to the high payoff the next round (Helbing, Schönhof, Stark, & Hołyst, 2005; Lau & Mui, 2008). This way, neither player dominates, and the payoffs come out more or less equal at the end of many games. However, alternation requires the players to agree upon and follow some set of self-imposed rules without explicitly discussing them. This agreement may be more or less difficult to achieve under different conditions. The *real-time* condition is of particular interest in game theory (Friedman & Oprea, 2012) and cognitive science (Spivey & Dale, 2006), since it is closer to the reality found in stock markets, social interaction, and just-in-time production, yet has remained relatively unexplored theoretically and experimentally (Charness et al., 2012).



Small Difference

	Blue High	Blue Low
Red High	0,0	1,2
Red Low	2,1	0,0

Large Difference

	Blue High	Blue Low
Red High	0,0	1,4
Red Low	4,1	0,0

Fig. 2 Payoff matrices for both conditions. Every game has two targets, one of which has a higher payoff than the other. If both players choose the same target, they end up on the diagonal of the matrix and neither player earns any points. Otherwise, each player earns an amount proportional to the payoff of the target

A piece of software called ConG was recently developed to enable researchers in behavioral economics to run such real-time, asynchronous experiments in the laboratory (Pettit, Friedman, Kephart, & Oprea, 2014). It is important to note that while this package is easy to configure and well designed for the particular area of behavioral game theory, it cannot be used over the web. The web framework built from the elements described in this paper potentially require a deeper knowledge of programming, but we believe the flexibility to run any kind of group behavior experiments online and across devices make up for the added complexity.

Methods

After workers accepted the (External) HIT on Amazon Mechanical Turk, they were directed to a gateway website where they filled out a consent form and took a simple pre-test to prove their understanding of the instructions. Then, they were redirected to the URL and port number on which the Express server is listening. Upon connection (via Socket.io, in `app.js`), the worker ID was extracted from the query string and checked against the MySQL database to verify that the user had indeed recently accepted a HIT. Their Socket.io connection was then passed to the `findGame()` function in `game.server.js`. That function creates a new game to serve as a 'waiting room' for the player if there is no game available, or matches the player with an existing game, if one exists.

In the waiting room, the player saw a "waiting for other player…" message and was able to click to navigate their avatar around the environment, although there was nothing of interest to do. Once another player joined, they were placed at opposite ends of the screen, with two targets spaced equidistantly between them (see Fig. 3). In the dynamic condition, players were given a 3-second countdown to secretly place their 'destination' markers, and at the end of the countdown began moving at a constant speed toward the target in small, frequent jumps. At any point in time (even between jumps), they were allowed to change their destination, an event detected with jQuery and sent to the server through the Socket.io connection. With a latency of less than 50 ms, on average, their avatar was changed on the other players' screen to reflect the new trajectory and the next small jump would be in the corresponding direction. Information about the state of the game including both player's positions and destinations was written to a file multiple times a second, at each small jump.

The *staged* condition was similar, except instead of a countdown, the round began by instructing each player to "choose a target." Neither player moved until both had chosen valid destinations, at which point the avatars began moving directly in the corresponding direction with the same animation as the other condition. Neither player knew when or what the other player chose before the first movement. However, to mimic the traditional design, players could not alter their trajectory after the round started. Clicking had no effect. This is equivalent to writing one's choice on a scrap of paper in the laboratory and simultaneously handing it to the experimenter for comparison. If one player disconnects in the middle of a game, the remaining player is automatically redirected to a webpage apologizing for the inconvenience and providing a link to an exit survey. Because the amount earned is updated at each round, and 50 or 60 total rounds are played, depending on the condition, we pay each player the amount earned up to the point of disconnection. Mechanical Turk workers were not permitted to participate more than once, using the database to check the worker ID.

Discussion

Full treatment of the results is outside the scope of this paper, but we will sketch some possible analyses to demonstrate the utility of this framework in addressing scientific questions. First and unsurprisingly, we found that efficiency (measured as the sum of both players' total earnings) was substantially higher in the dynamic condition than the staged condition. This indicates that groups can collectively achieve more efficient outcomes when allowed to make decisions in real-time, with access to one another's moment-by-moment actions. In game theory, actions double as signals. The staged version only allowed signals to be sent at the onset of each round, but the dynamic version allowed players the freedom to signal asynchronously at any point in time, thus providing a more information-rich environment for decision making.

Second, the dynamic condition reveals detailed fine-grained data about the moment-by-moment time-course of decision making. For example, the case when both players

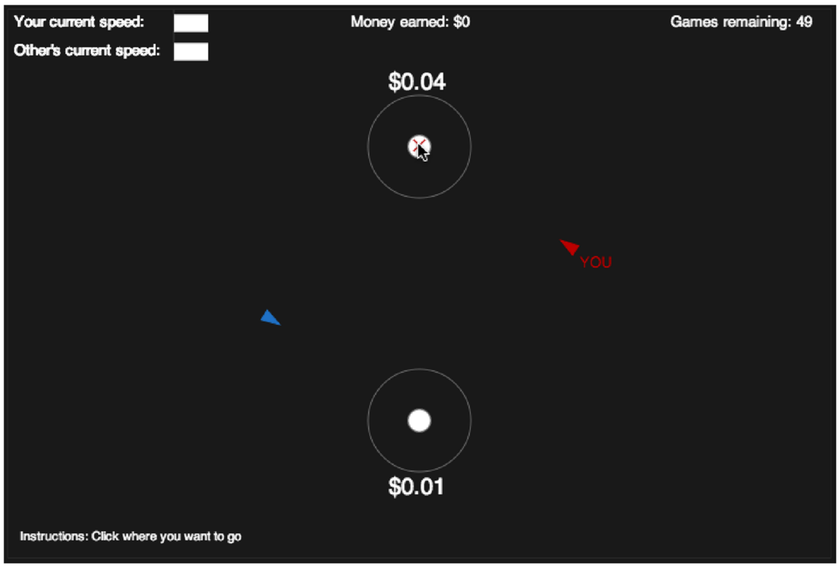


Fig. 3 Screenshot of the first round of the experiment. The colored triangles represent the players' avatars, and the circles represent targets worth a given payoff. Players, targets, and text are drawn on the HTML5 canvas. jQuery is used to collect the coordinates of a mouse click, and the canvas is updated to display that point as a colored cross which remains invisible to the other player. The upper left hand corner shows the relative speeds of each player, which were held constant in the reported experiment

are simultaneously heading toward the high payoff is equivalent to a game of 'chicken' in which each player waits for the other to 'divert' first, but cannot wait too long because the low-payoff is better than nothing. By looking at the distribution of time until divergence, we can test models of players' expectations about one another and their aversions to risk. Finally, we can examine macro-level patterns of learning and shifting equilibria across many rounds.

Conclusion

Although multiplayer experiments and real-time classroom demonstrations pose difficult, technical challenges to researchers and teachers, the open-source community has recently coalesced around a set of tools, notably including Node.js and Socket.io, which have provided a feasible solution. In this paper, we walked through this core set of tools and illustrated how they can be combined to conduct real-time multiplayer experiments on the web. Template code was provided to make many of most difficult elements of client–server networking tractable for those without strong web programming backgrounds.

It may be helpful to give a brief 'roadmap' for successful modification of this code. If the desired scenario still involves two participants being paired and placed into a virtual world, then the primary difference will be in movement dynamics or game logic, both of which are handled in `game.core.js`. The function `server_update_physics()` updates the positions of the players and checks whether there is anything special about the new positions (e.g. is it close enough to a target to warrant awarding a payoff and ending the game?) The function `server_new_game()`, along with its companion function `client_new_game()` in `game.client.js`, determines what happens at the end of a round. The function `client_on_click()` specifies what happens when a player clicks on the virtual world. These are the best starting points for understanding the code.

Modifying these functions will often involve adding new variables that need to be shared across the server and client objects. This is one of the most frequent sources of problems in writing Node.js code, and instructions for handling this step properly are given in the section on Socket.io above. To debug, variable values can be printed out with `console.log()`. Note that when this is called in server-side code, the result will print to the terminal, but when it is called client-side, it will print into the browser developer console. The steps to access this console are different across browsers, but are easy to find online.

If the desired scenario involves more than two participants, or some alteration to the 'waiting room' sequence, then more sweeping changes will need to be made to the `findGame()` and `endGame()` functions in `game.server.js`. One should expect this to involve a larger time commitment, but it should still be tractable with a conventional programming background and easier than starting from scratch. More advanced applications are limited only by imagination and development time. For example, we could insert a set of real cognitive agents into a controlled futures market (Majumder, Diermeier, Rietz, & Amaral, 2009), fish school (Miller et al., 2013), or crowd disaster scenario (Moussaïd et al., 2009), all of which rely crucially on events unfolding in real-time.

Because multiplayer experiments require multiple people to be in the same place at the same time, they can be difficult and expensive to conduct for researchers in small labs. We hope that through using this framework, the benefits of online labor markets can be extended to more sophisticated research questions involving real-time, asynchronous behavior. Additionally, this set of tools facilitates a move toward 'big data,' scaling up the number of participants and allowing more sophisticated analysis of moment-by-moment decision processes, not just outcomes.

Author Note This research was in part supported by National Science Foundation REESE grant 0910218 to the Percepts and Concepts Laboratory. We thank Shoshana Berleant and the members of the Santa Fe Institute mailing list for participating in usability tests of the framework and also Rob Goldstone and Johan Bollen for comments on early drafts.

References

Bishop, M. (2002). *Computer security: Art and science.* Addison-Wesley Professional.

Boer, K., Kaymak, U., & Spiering, J. (2007). From discrete-time models to continuous-time, asynchronous modeling of financial markets. *Computational Intelligence, 23*(2), 142–161.

Buhrmester, M., Kwang, T., & Gosling, S. D. (2011). Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science, 6*(1), 3–5.

Cao, M., Morse, A. S., & Anderson, B. D. (2008). Agreeing asynchronously. *IEEE Transactions on Automatic Control, 53*(8), 1826–1838.

Charness, G., Oprea, R., & Friedman, D. (2012, April). *Continuous time and communication in a public-goods experiment* (University of California at Santa Barbara, Economics Working Paper Series No. qt5404914p). Department of Economics, UC Santa Barbara. Retrieved from http://ideas.repec.org/p/cdl/ucsbec/qt5404914p.html

Cialdini, R. B., & Goldstein, N. J. (2004). Social influence: Compliance and conformity. *Annual Review of Psychology, 55,* 591–621.

Dale, R., Fusaroli, R., Duran, N., & Richardson, D. C. (2014). The self-organization of human interaction. *Psychology of Learning and Motivation, 59,* 43–96.

Deck, C., & Nikiforakis, N. (2012). Perfect and imperfect real-time monitoring in a minimum-effort game. *Experimental Economics, 15*(1), 71–88.

Friedman, D., & Oprea, R. (2012). A continuous dilemma. *The American Economic Review, 102*(1), 337–363.

Goldin, G., & Darlow, A. (2013). TurkGate (version 0.4.0) [Computer software manual]. Providence, RI.

Goldstone, R. L., Ashpole, B. C., & Roberts, M. E. (2005). Knowledge of resources and competitors in human foraging. *Psychonomic Bulletin & Review, 12*(1), 81–87.

Helbing, D., Schönhof, M., Stark, H.-U., & Hołyst, J. A. (2005). How individuals learn to take turns: Emergence of alternating cooperation in a congestion game and the prisoner's dilemma. *Advances in Complex Systems, 8*(1), 87–116.

Hughes-Croucher, T., & Wilson, M. (2012). *Node: Up and running: Scalable server-side code with JavaScript*. O'Reilly Media, Incorporated.

Lau, S.-H. P., & Mui, V.-L. (2008). Using turn taking to mitigate coordination and conflict problems in the repeated battle of the sexes game. *Theory and Decision, 65*(2), 153–183.

Majumder, S. R., Diermeier, D., Rietz, T. A., & Amaral, L. A. N. (2009). Price dynamics in political prediction markets. *Proceedings of the National Academy of Sciences, 106*(3), 679–684.

Mason, W., & Suri, S. (2012). Conducting behavioral research on Amazon's Mechanical Turk. *Behavior Research Methods, 44*(1), 1–23.

Mason, W., & Watts, D. J. (2010). Financial incentives and the performance of crowds. *ACM SigKDD Explorations Newsletter, 11*(2), 100–108.

McDonnell, J., Martin, J., Markant, D., Coenen, A., Rich, A., & Gureckis, T. (2012). psiturk (version 1.02) [Computer software manual]. New York, NY. Retrieved from https://github.com/NYUCCL/psiTurk

Michinov, N., & Primois, C. (2005). Improving productivity and creativity in online groups through social comparison process: New evidence for asynchronous electronic brainstorming. *Computers in Human Behavior, 21*(1), 11–28.

Miller, N., Garnier, S., Hartnett, A. T., & Couzin, I. D. (2013). Both information and social cohesion determine collective decisions in animal groups. *Proceedings of the National Academy of Sciences, 110*(13), 5263–5268.

Moussaïd, M., Helbing, D., Garnier, S., Johansson, A., Combe, M., & Theraulaz, G. (2009). Experimental study of the behavioural mechanisms underlying self-organization in human crowds. *Proceedings of the Royal Society B: Biological Sciences, 276*(1668), 2755–2762.

Olfati-Saber, R., Fax, J. A., & Murray, R. M. (2007). Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE, 95*(1), 215–233.

Paolacci, G., Chandler, J., & Ipeirotis, P. (2010). Running experiments on Amazon Mechanical Turk. *Judgment and Decision Making, 5*(5), 411–419.

Pettit, J., Friedman, D., Kephart, C., & Oprea, R. (2014). Software for continuous game experiments. *Experimental Economics*. doi:10.1007/s10683-013-9387-3

Reips, U.-D. (2002). Standards for internet-based experimenting. *Experimental Psychology, 49*(4), 243–256.

Spivey, M., & Dale, R. (2006). Continuous dynamics in real-time cognition. *Current Directions in Psychological Science, 15*(5), 207–211.

Suri, S., & Watts, D. J. (2011). Cooperation and contagion in web-based, networked public goods experiments. *PLoS One, 6*(3), e16836.

Takada, M. (2012). *Mixu's Node book: A book about using Node.js*. Available at mixu.net.

Tilkov, S., & Vinoski, S. (2010). Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing, 14*(6), 80–83.