Behav Res (2017) 49:1696–1715
DOI 10.3758/s13428-016-0824-z

REVIEW

nodeGame: Real-time, synchronous, online experiments in the browser

Stefano Balietti[1]

Published online: 18 November 2016

Abstract nodeGame is a free, open-source JavaScript/HTML5 framework for conducting synchronous experiments online and in the lab directly in the browser window. It is specifically designed to support behavioral research along three dimensions: (i) larger group sizes, (ii) real-time (but also discrete time) experiments, and (iii) batches of simultaneous experiments. nodeGame has a modular source code, and defines an API (application programming interface) through which experimenters can create new strategic environments and configure the platform. With zero-install, nodeGame can run on a great variety of devices, from desktop computers to laptops, smartphones, and tablets. The current version of the software is 3.0, and extensive documentation is available on the wiki pages at http://nodegame.org.

Keywords Behavioral experiments · Software · Real-time · Browser · Online · Open-source · JavaScript

Introduction

Online research is almost as old as the Internet (Musch & Reips, 2000). However, until just a few years ago, it has mainly focused on surveys or individual decision tasks, whereas synchronous group behavior experiments were defined as "rare commodities" (Horton et al., 2011). More recently, the number of synchronous online experiments has increased (Suri & Watts, 2011; Wang et al., 2012; Ciampaglia, 2014; Mao, 2015; DellaVigna, 2016), but all of them made use of custom implementations, leaving a large number of open methodological challenges yet to be solved, or reimplemented, by each individual researcher (Paolacci et al., 2010; Hawkins, 2014). For example: *How to efficiently synchronize a large number of computers? How to bring together a large number of participants? How to handle disconnections? How to dispatch new games? How to validate the requirements for participation? How to avoid repeated (or multiple) participation? etc.*

Finding common solutions to these challenges can open up a concrete chance of rapid progress in the understanding of human interactions in the coming years. For this to happen, it is necessary that researchers can focus more on *research* and less on implementation issues. This motivates a strong need for standardization and for the establishment of a common platform for performing synchronous online experiments, similar to what Z-tree has represented for the growth of the field of laboratory experiments (Fischbacher, 2007). Improving and standardizing the methodology for conducting synchronous online experiments is expected to generate novel insights along three major conceptual dimensions: (i) size, (ii) time, and (iii) granularity. The next subsection briefly explains the meaning of these three dimensions (see also Fig. 1).

Three dimensions for future online behavioral research: size, time, and granularity

Size

Online research is not bounded by the size of university labs; therefore there is virtually no limit to the number of

✉ Stefano Balietti
s.balietti@neu.edu

[1] Network Science Institute, Northeastern University, 177 Huntington Ave, Boston, MA, USA

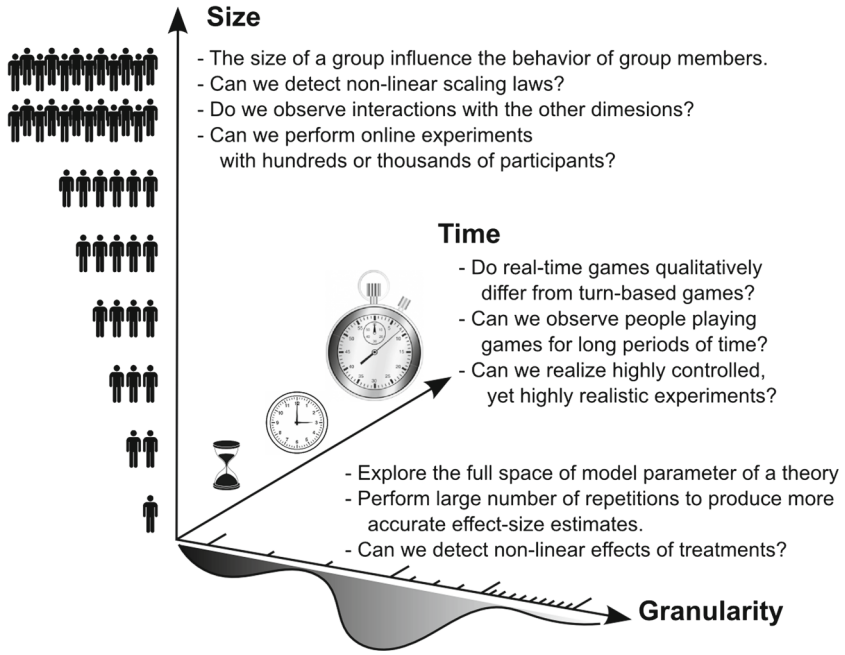

Fig. 1 Three conceptual dimensions for online experimentation's future developments. *Size:* running experiments with group of hundreds or even thousands of participants. *Time:* testing groups for extended period of times, or with games that involve real-time interactions and measurements. *Granularity:* scanning the full parameter space of a theory to detect possible non-linear effects

individuals that can simultaneously interact (Reips & Krantz, 2010). The size of a group is an important variable that can affect the behavior of interacting individuals in a number of ways. For example, Heinrich (2004) argued that group size is the main driver of cultural complexity. More recently, Derex et al. (2013) found experimental evidence of Heinrich's hypothesis, but other scholars are still contending this view (Andersson and Read, 2014). Group size is also known to negatively affect the level of cooperation in collective actions and public-good games (Olson, 1965; Nosenzo et al., 2013), although this negative externality can be mitigated (Chaudhuri, 2011). In sum, group size is an open, hot area of research, especially when online experimentation can shift the meaning of a "large group" from a couple of dozens to hundreds of simultaneous participants.

Time

Real-time gaming is another opportunity offered by online experimentation (Hawkins, 2014). Just like when players are expected to react to changes in group size, moving from discrete to continuous time can "fundamentally alter the character of strategic interaction" (Simon & Stinchcombe, 1989). The shift can be so extreme that decision-makers can even converge to equilibria that are significantly different from those found in experiments in discrete time. For instance, Friedman and Oprea (2012) found that cooperation in continuous time prisoner's dilemmas experiments increased from less than 50 % to over 90 %. Furthermore, real-time interactions can be exploited to perform experiments in more "immersive" environments or on more realistic tasks. For example, Mao (2015, Chap. 4) recently studied team-work coordination efficiency in a task that consisted of localizing tweets about Typhoon Pablo.[1] Furthermore, online research can also "stretch" the time dimension to study human interactions for extended periods of time, such as weeks or months (Butler, 2001; Bos, 2002; Salganik et al., 2006; Centola, 2010), allowing researchers to draw insights on evolutionary or cascading dynamics. In sum, online experiments grant great flexibility to experimenters to study human interactions at different time intervals, from very quick (almost real-time) to very long (weeks and months).

Granularity

Online experiments, due to their lower costs, allow one to test multiple combinations of parameters of behavioral theories (Birnbaum, 2004; Reips & Krantz, 2010). Nowadays, the scanning of a model's parameter space is usually done via computational modeling, but in the future this could increasingly be done via a batch of parallel online experiments. The result would be a so-called "phase diagram," where the value of the outcome variable is plotted

[1] See http://en.wikipedia.org/wiki/Typhoon_Bopha for details

against all the possible combinations of model parameters.[2] This fine-grained analysis will allow researchers to detect the existence of non-linear effects between treatment and outcome variables, and to measure their effect-size much more accurately. In sum, large-scale online experiments will make it possible to conduct stronger tests of the predictions of behavioral theories, an endeavor that has been extremely difficult so far. This, in turn, might facilitate convergence and bring synthesis into the fragmented situation of many social sciences disciplines today (Balietti et al., 2015).

Software for online synchronous experiments

As argued in the previous section, online synchronous experiments are a promising methodology, and a growing area of research. However, the field is still young, and it is missing established software for conducting synchronous online experiments. The experimental software Z-tree (Fischbacher, 2007), which can be considered as a de facto standard for synchronous laboratory experiments, was not originally designed to work on the Web, and neither does it support a very large number of participants or real-time interactions.

A number of projects have made an attempt to fill the gap; however, none of them has yet become a standard. Early examples are: Wextor (Reips & Neuhaus, 2002), FactorWiz (Birnbaum, 2000), and EconPort (Cox & Swarthout, 2005). More recent examples at different stages of development, are: MWERT (Hawkins, 2014), ConG (Pettit & et al. 2014), oTree (Chen et al., 2014), the CLOSE project (Lakkaraju & et al. 2015), the MIT Seaweed project (http://dspace.mit.edu/handle/1721.1/53094), Breadboard (http://breadboard.yale.edu/), and Microsoft's Virtual Lab (https://github.com/VirtualLab). Finally, even if currently limited to single-participant Web experiments, the software jsPsych (Leeuw, 2014) has received a fair amount of adoption.[3]

This paper introduces *nodeGame*,[4] a new free and open-source framework to conduct synchronous online experiments designed to support the three lines of research previously described: size, time, granularity. An experiment—or a game[5]—implemented in nodeGame can run in any device equipped with a browser, be it personal computer, a mobile device, or a computer in the university laboratory. nodeGame can be freely downloaded from http://nodegame.org.

Outline of this paper

The remainder of the paper is organized as follows. Section "nodeGame: Overview" gives an overview of the nodeGame software, the design principles, and the domain of applicability. Section "Creating experiments with nodeGame" describes how to create a new experiment in nodeGame, trying to avoid the most technical details, which are available in the online wiki of the project Website. Section "nodeGame Features" highlights some of the main features that make nodeGame particularly suited to carry out online experimentation. Sections "The Window object" and "Widgets" focus on two specific features of the API: the Window object, and the Widgets, a collection of reusable components that can be loaded dynamically during an experiment. Section "nodeGame's architecture" introduces the nodeGame technical architecture, and might as well be skipped if the reader is not a developer. Finally, "Conclusions" summarizes the advantages and current limitations of nodeGame, concluding the paper with some general considerations about the use of open-source software in scientific research, and about the benefits of accelerating the cycle of hypothesis testing and generation in the social sciences.

nodeGame: Overview

nodeGame is designed and implemented taking into account the features and limitations of current experimental software, aiming to accomplish the following goals:

1. use only open-source and free software technologies,
2. realize a robust and fault-tolerant application,
3. run discrete- and real-time experiments,
4. support hundreds of simultaneous participants,
5. grant flexibility and fine-grained control to the experimenter.

To the best of our knowledge, no other experimental software besides nodeGame has so far achieved all five design goals simultaneously. The following list provides an overview of nodeGame's main features:

[2] See for example (Helbing, 2014) for a phase diagram of a model of social norm emergence in a two population settings.

[3] For more platforms refer to the list of available experimental software maintained by the University of Alaska Anchorage (UAA) Experimental Economics Laboratory at the URL: http://econlab.uaa.alaska.edu/Software.html

[4] The name nodeGame comes from the technology used for its implementation: Node.js. For details about Node.js and architecture of nodeGame, refer to "nodeGame's architecture". The name nodeGame has also affinity with the fact that each client of the game is also a separate computing node.

[5] nodeGame is a flexible environment that permits to implement scientific experiments in the form of games with strategic interactions among players, individual decision tasks, and combinations thereof. In this article, the words game and experiment will be used mostly interchangeably.

- nodeGame provides a programming framework that is easy to extend and customize via the nodeGame API (Application Programming Interface). The current version of the API is 3.0, and the language is JavaScript,[6] both on the client and on the server. Through the nodeGame API, the experimenter obtains fine-grained control over both experimental variables and technical settings. However, most of the complexity regarding the latter is hidden away by nodeGame's preconfigured default options.
- nodeGame's API offers game developers configurable waiting rooms, support for disconnecting and reconnecting players, authorization and technical requirements checks, automated players (bots), reusable widgets components, a highly customizable game sequence, timers, an admin interface, and more.
- nodeGame is entirely based on open Web technologies (HTML5, CSS, JavaScript), therefore, it allows one to create both simple and very complex experiments. For example, nodeGame can go beyond traditional turn-based experiments, and it also allows one to conduct real-time games, by exploiting the power of HTML5 WebSockets.[7] Moreover, nodeGame integrates seamlessly with popular third-party libraries, such as jQuery and D3,[8] to create stunning visualizations and rich user interfaces.
- nodeGame experiments run directly in the browser window. They are completely cross-browser and cross-platform. Desktop devices (Windows, Mac OS X, and Linux) and mobile devices can connect to the nodeGame server and join an experiment at the same time. Every new release of nodeGame is rigorously tested against an array of different browsers to guarantee its correct functioning.
- nodeGame can be used to run experiments on online labor markets.[9] One of those, Amazon Mechanical Turk (AMT), has been extensively used for conducting online psychological, sociological, and economic research (Paolacci et al., 2010; Mason & Suri, 2012). The integration between AMT and nodeGame is based on a shared list of authorization codes, which are assigned to online workers upon accepting a task and are checked by nodeGame as a requirement for participation. However, the upload of the work task on AMT must be done by the experimenter outside of nodeGame.
- nodeGame is under steady development, but it has already been used by several research teams to run experiments in the laboratory and online, such as: prisoner dilemma games, ultimatum games, public-goods games with and without assortative matching (Nax et al., 2016), burden-sharing games (Anderson et al., 2016), art-exhibition games (Balietti et al., 2016), and congestion games (Balietti et al., 2016). Figure 2 contains screenshots of the nodeGame interface for public-goods games with noisy assortative matching.
- Finally, in addition to online and laboratory behavioral experiments, nodeGame can also be used to perform other types of data collection, such as surveys and field experiments, as well as for didactic purposes in the lecture hall.

Creating experiments with nodeGame

The purpose of this section is not to be a thorough user manual for nodeGame, but rather to present the main concepts behind the creation of experiments, and the most important API methods in nodeGame (version 3.0). Additional information and code examples are available on the online wiki of the project: http://nodegame.org/wiki/.

Installation/Quick start

To install nodeGame, just follow the instructions available on the wiki of the project: http://nodegame.org/wiki. After the installation is completed:

- Open a terminal and browse to the folder `nodegame/`
- Start the server with the command: `node launcher.js`
- Access the default ultimatum game at the URL: `localhost:8080/ultimatum`
- Start an automated player (bot) at the URL: `localhost:8080/ultimatum?clientType=autoplay`
- Access the administrator interface at the URL: `localhost:8080/ultimatum/monitor/`

[6] JavaScript is the scripting language of the browser, and it is also an efficient server-side language for dealing with asynchronous I/O (Input/Output) operations, such as accessing a database or the file system.

[7] WebSockets represents a major improvement in the history of Web data communication. For more information see http://en.wikipedia.org/wiki/WebSocket or https://www.websocket.org/.

[8] jQuery (http://jquery.com) is a cross-browser JavaScript library designed to simplify the client-side manipulation of an HTML page. D3 (http://d3js.org) is a JavaScript library for creating data-driven interactive visualizations.

[9] Common online labor markets include Amazon Mechanical Turk, oDesk, Freelancer, Elance, Guru, CrowdFlower, Innocentive, or SurveySampling, etc. If monetary incentives are not required, it is simply possible to upload a link to Web sites like "Psychological Research on the Net" (http://psych.hanover.edu/research/exponnet.html), "The Web Experiment List" (http://www.wexlist.net), or "Online Social Psychology Studies" (http://www.socialpsychology.org/expts.htm).

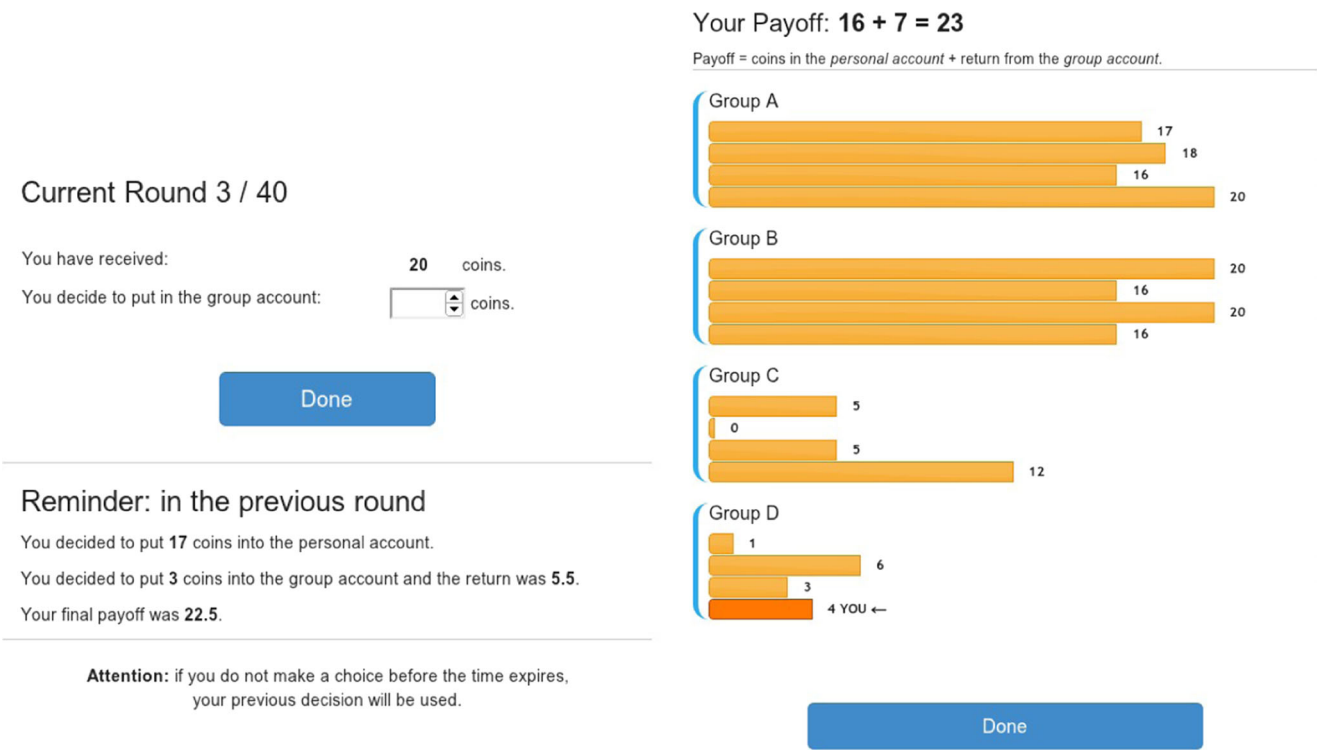

Fig. 2 Screenshot of the nodeGame interface of a public-goods game. *Left:* the interface allows a participant to place a contribution between 0 and the full endowment, and displays the previous contribution and outcome. *Right:* the interface shows the groups formed through a noisy assortative matching based on the subjects' initial contributions. Within each group, payoffs realize

Creating and joining a new experiment

To start the creation of a new experiment, go to the `bin/` folder inside the nodegame installation directory, open a terminal and type:[10]

```
node nodegame create-game myexperiment
```

This command automatically creates a new folder in nodeGame's games directory containing all the necessary files to run a new experiment. After the server is started, participants can join the experiment by appending the name of the experiment, e.g., myexperiment, to the address of the server, viz:

`http://`	`myserver.com`	`:8080`	`/myexperiment`	`?param=XX`
protocol	*server address*	*port*	*game name or alias*	*query string*

All the elements of the experimental URL can be configured. For example, the protocol can be http or https, the port number can be changed or omitted, the name of the experiment can be replaced by an alias, or omitted completely. Query string parameters are optional, and can as well be disabled.

[10]Windows computers might need to manually install the nodegame-generator package. For details, see the wiki page https://github.com/nodeGame/nodegame/wiki/Game-basics-v3

The core components of an experiment

Inside the newly created experimental folder, the following core components should be adapted to suit the desired experimental design:

1. Game variables and treatments,
2. Game sequence,
3. Client types,
4. Waiting room

Each component will be briefly reviewed in the next subsections together with the most important related API commands.

Game variables and treatments

Game variables include values such as: how many monetary units are assigned to players, the duration of a timer in a step, the conversion rate from an experimental to a real currency, etc. These variables can be grouped together under a common label to define an experimental treatment.

nodeGame API to define game variables and treatments

All the experimental variables and treatments are defined in file `game/game.settings.js`, which must export

a settings object. Game developers can define as many variables as needed, but some names are reserved for specific options by the nodeGame engine. For example, `WAIT_TIME` controls the default waiting time in case of a disconnection of a player and `TIMER` contains the duration of timers for specific steps.

Treatments are defined as properties of the `TREATMENTS` object. In the example below, two treatments are created: "treatment1," and "treatment2." All the variables defined outside the `TREATMENTS` object are shared by all treatments; variables defined inside the `TREATMENTS` object are available only to a specific treatment, and can also overwrite global variables. In the example below, "treatment1" overwrites the value for "max_offer," and "treatment2" overwrites variable `WAIT_TIME`.

```
var settings = {
    // Game variables shared among all treatments.
    WAIT_TIME: 30000,
    TIMER: {
        offer: 20000,
        respond: 15000
    },
    max_offer: 100,
    // Game variables that are treatment specific.
    TREATMENTS: {
        treatment1: {
             max_offer: 50
        },
        treatment2: {
             WAIT_TIME: 20000
        }
   }
};
```

The game sequence

The game sequence consists of one or more *stages* that are executed sequentially after the experiment begins. Each stage consists of one or more *steps* that are executed sequentially within the stage. For example, the stages of a hypothetical ultimatum game experiment could be: (i) 'instructions,' (ii) 'quiz,' (iii) 'game,' and (iv) 'questionnaire.' The stages could be further subdivided in steps as follows:

- *Instructions*. Three steps: instruction_1, instruction_2, instruction_3. In this way, long texts are broken across multiple pages, and the time spent on each step is automatically measured.
- *Quiz*. One step containing the quiz itself. Optionally, another step displaying the correct answers could be added.
- *Game*. Three steps corresponding to the turns of the ultimatum game: (i) making an offer/waiting for an offer, (ii) accepting or rejecting it, (iii) displaying the results to the players.
- *Questionnaire*. One or more steps, depending on the number of questions in the survey.

Furthermore, stages can be repeated multiple times, each repetition of which constitutes a new *round*. The number of rounds can be predefined before the experiment starts, or it can be determined at run-time, upon fulfillment of certain criteria, e.g., players reaching an agreement upon a bargaining. In our example, the 'game' stage is repeated for three rounds, during which players randomly alternate between the role of bidder and respondent. Figure 3a illustrates the stages and steps discussed so far.

nodeGame API to define the game sequence

The sequence of stages and steps of a game is defined in file `game/game.stages.js` via the `stager` API. The order in which stages and steps are added to the `stager` defines the sequence.

```
stager.stage("instructions")
      .step("instructions_1")
      .step("instructions_2")
      .step("instructions_3");

stager.stage("quiz");

stager.repeat("game", 3)
      .step("offer")
      .step("respond")
      .step("display_results");

stager.stage("questionnaire");
```

Additionally, the `stager` API also defines the following useful methods:

- **`stager.skip(stage, step):`** Skips an entire stage, or a step when executing the game.
- **`stager.loop(stage, function):`** Repeats the stage conditional on the callback function returning true.
- **`stager.doLoop(stage, function):`** Like stager.loop, but the stage is executed at least once.

Client types

Once the game sequence is defined, stages and steps must be "implemented," i.e., the actions taking place in each step must be defined (see Fig. 3b). Each implementation of the

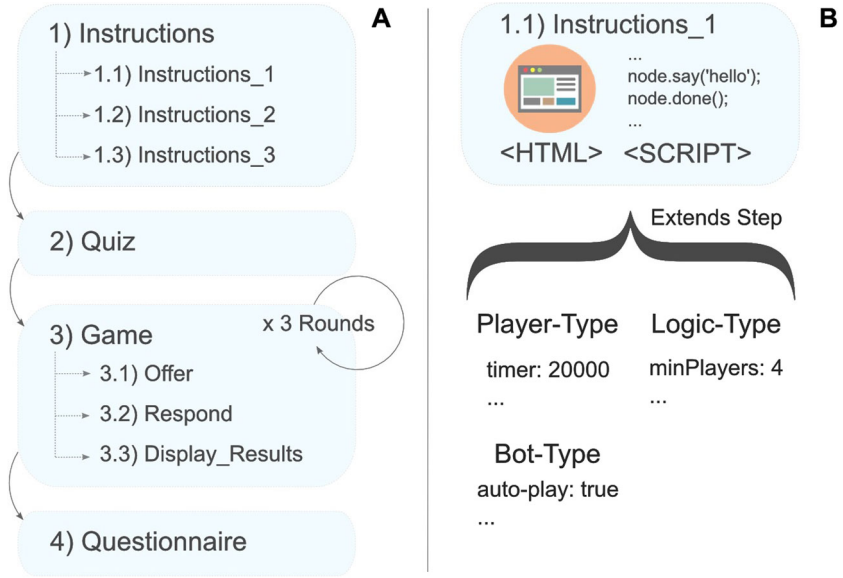

Fig. 3 Creating an experiment with nodeGame. **a** Simplified representation of the stages and steps of an ultimatum game. **b** Simplified representation of a fictitious Instructions_1 step. The step contains a unique name, an execution callback, and an HTML page. In this example, three client types (Player, Logic, and Bot) extend the step by adding further properties to the step object

game sequence is called *client type*, and it can be radically different depending on what its purpose is.

Two client types are mandatory: (i) *player*, and (ii) *logic*. The player type is assigned to participants connecting to the experiment with a browser. It has the purpose to load the pages of the game, to handle users' interaction with the screen (e.g., the movement of the mouse, or the click on a button), and to exchange messages with the server. The logic type is executed on the server and controls the flow of operations of a game room, such as creating sub-groups, accessing external datasets, handling player disconnections, etc. Its implementation obviously varies depending on the experimental design.

Other optional client types are automated players, used to interact with human participants, or to test the correct functioning of the experiment (see Bots and Phantoms in "nodeGame Features").

nodeGame API to define client types

Each client type is defined in a separate file inside the `game/client_types/` directory. Client types are constructed by adding properties to the "empty" stages and steps created in the game sequence, and available through the `stager` object. By default, steps inherit all the properties defined by the stage in which they are included. Moreover, stages inherit all "default" properties defined at the stager level. For example, if we define the property `coins` as follows:

```
stager.setDefaultProperty("coins", 100);
stager.extendStage("game", { coins: 200 });
stager.extendStep("offer", { coins: 50 });
```

then, invoking `node.game.getProperty(''coins'')` would return 50 in step 'offer', 200 in steps 'respond' and 'display_results' inside the 'game' stage, and 100 in any other step.

Game developers can define their own properties, but some names are already reserved by the nodeGame engine for special purposes. The most important reserved properties are:

- **`frame:`** The URL of the page to load.
- **`cb:`** The callback (cb) function to be executed after the frame is loaded, but before the player can interact with it.
- **`done:`** The callback function that is executed when **`node.done`** is executed to terminate a step.
- **`timer:`** The duration of the timer for current step (in milliseconds).
- **`timeup:`** The callback function to execute when the timer for the current step expires.

Inside the `cb` function, the game developer can make use of several methods of the nodeGame API to implement the intended experimental design. On the browser, the methods of the API are exposed through two objects:

- **`node:`** sends and receives messages, starts timers, steps through the game sequence, loads widgets, etc.

– `W` (Window): manipulates information on screen and the behavior of user interface, e.g., disabling right click, locking screen, etc.

At the end of each step, the game developer must call the method `node.done`, which triggers a procedure that advances the client to the next step. For example, the pseudo-code for the "offer" step of the player client type could be as follows:

```
// Player made a choice and clicked the submit button.
submit.onclick = function() {
  // Inform other player about offer.
  node.say("offer", "respondent", offer);
  // Conclude step and store offer in server.
  node.done(offer);
};
```

The respondent would then receive the offer, display it on screen, and terminate the step.

```
// On incoming data labeled "offer", execute the function.
node.on.data("offer", function(msg) {
    // Write offer on screen using the W object.
    W.setInnerHTML("offer", msg.data);
    node.done();
});
```

Waiting room

The waiting room has the purpose of starting a new experiment when certain criteria are met, e.g., a certain number of clients is simultaneously connected, or the maximum waiting time has expired. While waiting for the next game to begin, participants in the waiting room usually receive information about how much time they have been waiting, how many other players are still needed, etc.

The waiting room operates in different execution modes. By default, the waiting room selects a treatment and dispatches a new *game room* as soon as the requested number of players is available. However, it is possible to schedule a date and a time in the future when a new game room will be created, or to manually start the experiment from the monitor interface (useful in the lab environment). For each execution mode, several additional options are available to give the experimenter a fine control over the conditions for dispatching new games. Finally, in the case of more complex experimental setups, it is also possible to use a completely customized waiting room, in place of the standard one.

nodeGame API to define waiting room settings

The criteria for dispatching new games are defined in file: `waitroom/waitroom.settings.js`. The most important options are:

– **`EXECUTION_MODE:`** Available options **`"WAIT_FOR_N_PLAYERS"`** and **`"TIMEOUT"`**.
– **`POOL_SIZE:`** How many players must be connected before dispatching new game rooms.
– **`GROUP_SIZE:`** How many players to assign to every new game room. If the pool size is a multiple of the group size, then multiple game rooms will be created.
– **`N_GAMES:`** Limits the total number of dispatchable games.
– **`MAX_WAIT_TIME:`** Limits the total wait time for participants.
– **`CHOSEN_TREATMENT:`** Decides which treatment is assigned to a given game room (use **`"treatment_rotate"`** for rotating the treatments).
– **`START_DATE:`** Sets the beginning of a game a certain date in the future.
– **`PLAYER_SORTING:`** Sorts players in a specific order before dispatching them.
– **`PLAYER_GROUPING:`** Creates groups of players to dispatch.
– **`PING_BEFORE_DISPATCH:`** Pings all players before a dispatch (useful if the desired group size is large).

nodeGame Features

This section centers around additional features of nodeGame (version 3.0) that have not already been covered in "Creating experiments with nodeGame". NodeGame is under active development, so new features might be added in newer versions, however, those described here are expected to be kept also in future versions.

Authorization rules

The Web is a great source of anonymity. This is why it permits to do research on more sensitive topics or on individuals with rare conditions of interest that would not be possible to do in the lab (Mangan & Reips, 2007).[11] However, synchronous experiments usually require to identify experimental participants to prevent them from entering the same game room multiple times, or to exclude those who have already played the experiment. nodeGame provides a default authorization system that stores authorization tokens in the browser. Moreover, it is possible to define a custom authorization function that can accept or reject incoming connections based on properties such as IP, browser type, etc.

[11] There also exist studies that can be conducted only in the lab and not online. For example, when some physiological parameters need to be measured directly, or when specialized hardware is required.

nodeGame API to create authorization rules

Authorization rules are specified in the `auth/` directory. File `auth.settings.js` allows one to disable or enable authorization checks. The most important options are:

- **`enabled:`** Boolean flag.
- **`mode:`** Available modes: **`dummy:`** creates dummy ids and passwords in sequential order; **`auto:`** creates random eight-digit alphanumeric ids and passwords; **`local:`** reads the authorization codes from file; **`remote:`** fetches the authorization codes from a remote URI; **`custom:`** uses the 'customCb' field.
- **`nCodes:`** The number of codes to create.
- **`addPwd:`** If TRUE, a password is added to each code created.
- **`codesLength:`** The length of generated codes.
- **`claimId:`** If TRUE, remote clients will be able to claim an id via GET request (useful when posting tasks on online labor markets, such as Amazon Mechanical Turk).
- **`claimIdValidateRequest:`** Authorization callback for incoming 'claimId' requests.

Automated players: bots and phantoms

Bots and phantoms are two computer-controlled client types executed in the server (see "Client types"). The only difference between them consists of the fact that phantoms are executed in a headless browser,[12] while bots are not. A headless browser is a normal Web browser without a graphical user interface. It behaves exactly as a normal computer browser, with the only exception that rendered pages are not shown to anyone. In other words, phantoms are able to load a full HTML page like human players would do with their "headed" browser. This makes phantoms particularly suited for testing and debugging an experiment before launching it. Bots, on the other hand, cannot load HTML pages, and therefore are much more light-weighted processes. This means that an experimenter can create a large number of them without a significant memory and CPU overhead. Bots can be used to replace a disconnected human player during an experiment which requires a minimum number of participants, or to play alongside humans in an synchronous environment.

nodeGame API to handle automated players

Phantoms and bots are defined in dedicated client-type files, each one being a custom implementation of the game sequence. They are usually instantiated by a logic client type in the server-side environments, where the game developer has access to two objects of the nodeGame-server API: `gameRoom` and `channel`. They offer several methods, among which:

- **`gameRoom.getClientType('player')`**: Returns the raw player type that can be modified and adapted to behave in an automated way.
- **`channel.connectBot(options)`**: Connects a bot to channel.
- **`channel.connectPhantom(options)`**: Connects a phantom to channel.

Disconnections and reconnections

In the online world, participants can leave a previously joined experiment for any reason. Dropouts (disconnections) are a major problem for synchronous online experiments, as they increase both the budget and the time necessary to complete the data collection. However, dropouts not only have negative externalities. In fact, as some researchers have pointed out, the voluntary nature of online participation can often lead to the production of better-quality data than laboratory experiments, because in the laboratory subjects feel obliged to stay in the lab even when they have stopped paying attention to the task at hand (Reips, 2002).

nodeGame makes available a number of solutions to handle the dropout problem. For example, it is known that dropouts can be significantly mitigated by making use of apposite measures such as 'warm-up phases', and 'seriousness checks' (Reips, 2002; Reips & Krantz, 2010). Those are easy to implement by just adding an extra stage in the game sequence, or by adding a dedicated game level (see "Game levels"). Alternatively, it is possible to specify a "disconnection handler" associated with a minimum number of players that need to stay connected in order for the experiment to continue. This handler can be global, i.e., throughout the whole experiment, or it can be attached to single stages or steps. In this way, the minimum-players-connected condition is verified only when really needed. In fact, it is common that some parts of an experiment can be executed with a variable number of players, while others have stricter requirements. For example, at the end of a collective behavior experiment, participants are often presented with a final questionnaire. Here, a single disconnection should not affect any other player.

nodeGame has implemented a default behavior for handling disconnections: it immediately pauses the experiment, displaying a notice to all connected clients. Simultaneously, a countdown is started, at the end of which the experimenter can decide whether to continue with less players, connect a bot player, or cancel the experimental session, redirecting the participants to an exit stage.

[12] The headless browser used by nodeGame is Phantom.JS, therefore the name "phantoms." For more information see http://phantomjs.org.

nodeGame API to handle disconnections/reconnections

In the logic client type, three disconnect handlers are available: `minPlayers`, `maxPlayers`, or `exactPlayers`. They can be a number (threshold), or an array containing up to three elements ordered as follows:

1. `threshold`: The desired value of min/max/exact connected players.
2. `threshold_cb`: A callback executed when the specified threshold is passed.
3. `recovery_cb`: A callback executed when the number of connected players is correct again (viz. after the `threshold_cb` was executed once).

Additionally, in the logic client type, it is possible to define a `reconnect` property to fine-tune the reconnection parameters of the reconnecting clients.

Game levels

Game levels divide an experiment into multiple parts, each of which can have a separate waiting room and a different game sequence with distinct synchronization rules. For example, an experiment could start with a preparatory part where participants do not interact with each other, but they only answer some survey questions. Only in a second part would they reach a waiting room and form groups for synchronous play. This setup has the advantage of reducing the number of dropouts in the second part, where synchronous play is happening.

nodeGame API to create game levels

In order to add a new game level, create a new folder with the name of the level (e.g., "part2") inside the `levels/` directory. Then, inside the new game level directory, add a `game/` folder, and optionally a `waitroom/` folder, following exactly the same structure as for the folders with the same name at the top-level of the directory of the experiment.

Matcher

The Matcher API is currently limited to matching players into pairs. Two matching algorithms are implemented: "round robin" (also known as perfect stranger matching), and "random." However, custom matching algorithms can be easily added into the API.

nodeGame API to match players

The following code example illustrates how to create a round robin tournament schedule for four players.

```
// Require and instantiate the Matcher class.
var Matcher = require("nodegame-client").Matcher;
var matcher = new Matcher();

// Select algorithms, and creates schedule.
matcher.generateMatches("roundrobin", 4);
matcher.match([ "a", "b", "c", "d" ]);

// Get next match.
matcher.getMatch(); // [ "a", "b"];
matcher.getMatchObject(); // {a: "b", c: "d" }
```

Monitor interface

The monitor interface includes the list of all games currently available, the list of game rooms dispatched, and the list client connected (see Fig. 4). The state of every client is reported, including the client type and the stage of the game they are currently in. Moreover, the interface allows one to send group or individual game commands to pause, resume, advance, or restart the game. Finally, a chat window can be opened to communicate with participants in need of assistance.

Additionally, for the experimenter's convenience, the configuration of the game is displayed in a separated tab; moreover, the content of the `data/` folder (where the results are saved) is listed and available for download. The state of the server is also displayed in a separated tab.

Multiple games and access points

Self-selection bias (Kraut et al., 2004) is a known problem from which online experiments can suffer. That is, only people who are interested in a certain type of experiment participate or stay in the experiment. However, this issue can be mitigated with the use of an ad hoc countermeasure called the 'multiple-site-access' technique (Reips, 2002; Reips & Krantz, 2010). In nodeGame, the same game can have as many access points as necessary. Each access point is called an *alias* (see "Creating and joining a new experiment") and can be specified in the channel configuration file: `channel/channel.settings.js`.

Furthermore, nodeGame not only supports running multiple access points for the same game but also supports running completely different games at the same time. Each game would then have its own access points.

Requirements

Even if authorized to take part in an experiment, a client might not actually have the technical requirements to go through all the steps. In fact, in the Internet, there are hundreds, or even thousands, of different browsers and browser

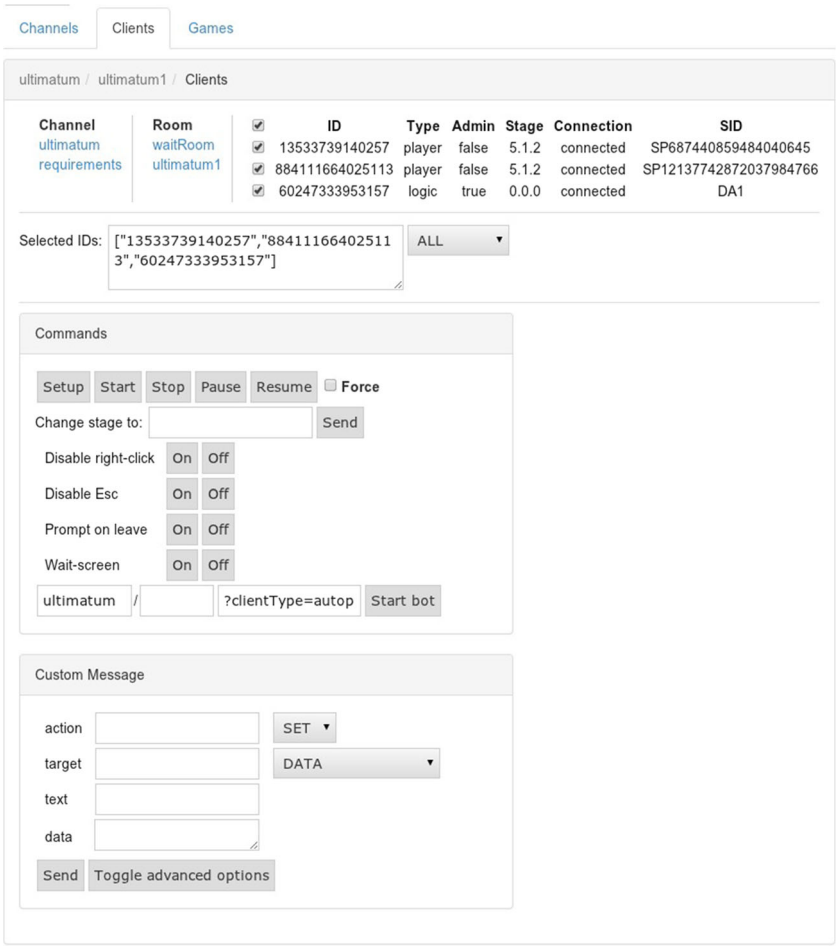

Fig. 4 Screenshot of the monitor interface. At the *top* of the panel, it is possible to browse the game rooms currently active in the channel. The clients connected to the selected room are shown: two players and one logic. The *middle panel* allows one to send a game command to the clients selected in the upper panel; other actions are available, such as manually connecting a computer-controlled client. The *lower panel* permits creating and sending a custom game message to the clients selected in the top panel

versions, each one equipped with a slightly different implementation of the HTML and JavaScript standards. For example, Internet Explorer browsers below version 10 are notoriously famous for not being fully standard-compatible. They can generate glitches in the visualization of a page, or in the communication with the server. Therefore, it is necessary to test the correct functioning of nodeGame on each client before letting it into an experiment.[13]

Clients connecting from mobile devices, such as tables or smartphones, also require special attention. Differences in the size of the displays should be taken into account, and clients with requirements below compliance should not be accepted. Making use of CSS frameworks helps obtaining a consistent cross-device visualization of the experiment.[14] Therefore, they should always be used when performing online experiments. Furthermore, clients connecting via mobile devices might not only experience visualization issues due to the smaller size of the display but also encounter connectivity troubles due to a sudden failure of the mobile network.

By default, in nodeGame, the following requirement checks are performed: whether JavaScript is enabled and cookies supported, if the client can load HTML pages in a dedicated iframe, and if it can communicate with the server without an excessive delay. Additional checks can be added, as seen appropriate. For example, it is possible to

[13] Most people participating in online labor markets usually have multiple browsers installed on their machine, and they can be invited to retry connecting to the experiment with another one.

[14] A CSS framework is a collection of HTML- and CSS-based design templates and JavaScript modules for developing responsive, mobile-compliant Web interfaces. nodeGame makes use of CSS-framework Twitter Bootstrap (http://getbootstrap.com/).

impose restrictions based on the location of the client. Some online labor markets allow to specify similar constraints when uploading the task, but this does not always lead to accurate results. HTML 5 Geolocation API can localize clients even at the level of single streets in dense urban areas.[15] However, clients must allow to be geolocated, and this could be introduced as a requirement for participation in the experiment.

nodeGame API to create requirement rules

Requirements rules are specified in the **`requirements/`** directory. File **`requirements.settings.js`** allows one to disable or enable authorization checks. The most important options are:

- **`enabled`**: Boolean flag.
- **`maxExecTime`**: Limits the maximum execution time for all tests.
- **`speedTest`**: Verifies that the client can exchange a given number of messages in the specified amount of time.
- **`viewportSize`**: Verifies that the client's screen resolution is between the min and max specified.
- **`browserDetect`**: Verifies that the client's user agent has got the required name, type, and version (more checks available).
- **`cookieSupport`**: Verifies that the client supports cookies ('persistent' or 'session').

Templates and internationalization

nodeGame supports the dynamic creation of HTML pages from templates that are rendered upon request from a connected client. Templates allow one to write modular Web pages that can be filled with blocks of content depending on the actual configuration of the experiment. As will be explained below, the use of templates presents multiple advantages, and therefore they should be preferred over static HTML pages whenever possible.

Firstly, templates introduce a clearer separation between changeable and static parts of the user interface. For example, in an ultimatum-game-like experiment, participants would divide a certain budget B of monetary units between a bidder and a respondent. An experimenter could hard code the actual value of B in every page, but this would make the code very hard to maintain whenever the value of B is updated. Instead, by using a template, the value of B would be automatically inserted in every page by the template engine. In this way, updating the value of B in the settings of a treatment does not require further modifications in other parts of the code.

Secondly, templates can create nested layout structures that reduce the complexity of the markup of the single components and the load on the server upon requesting them. A nested layout usually consists of a fixed outer frame and a variable number of interchangeable blocks. With this configuration, instead of requesting a whole new page to the server every time, only the block actually being updated will be downloaded.

Finally, templates allow one to achieve the internationalization of an experiment, e.g., the display of the text of the experiment in different languages. This can be obtained separating the translation files into different directories (contexts) corresponding to the different prefixes of the requested languages, e.g., "en", "de", "it", etc. They will be automatically matched upon receiving a new request, and the properly localized page will be rendered and sent to the client.

nodeGame API to create templates

nodeGame serves all the static files from the `public/` folder. However, it is possible to dynamically create HTML pages by adding files inside the `views/` folder.

The `views/` folder is further divided into two sub-folders: `templates/` and `contexts/`. The `templates/` folder contains Pug[16] files to create and format HTML pages. Templates support dynamic code, and allow the game developer to reuse and extend components across pages. The `contexts/` folder contains JavaScript files named after the corresponding template. Each file must export a function returning the "context," i.e., an object containing all the variables used to fill-in a template.

Timers

Response times can be used as an indicator of the type of internal reasoning process used by decision-makers. Its systematic analysis can reveal a personality trait that makes use of heuristics vs. iterative, rationale thinking (Rubinstein, 2013), or whether a person is expected to play more cooperatively or more selfishly in synchronous games (Rand et al., 2012). To support research investigating these questions, nodeGame has a dedicated API for defining new timers, and measuring time intervals between specific events. Most importantly, such intervals are measured directly on the client machines, so that they network

[15]The HTML 5 Geolocation API is available on all major modern browsers. Since it can compromise user privacy, it cannot be used unless the user approves it. For info see http://en.wikipedia.org/wiki/W3C_Geolocation_API.

[16]The Jade template language is now called Pug. See http://pugjs.org.

latency is excluded from calculations. The precision of JavaScript timers is in the order of hundreds of milliseconds, slightly higher, but still comparable to other technologies, such as Flash, Psychtoolbox, and Java (Neath et al., 2011; Reimers and Stewart, 2015). Such differences are anyway not detectable by human subjects, and the variance in response times is approximately equivalent (Leeuw & Motz, 2016).

nodeGame timers are synchronized with the flow of the game, so that they are automatically paused and resumed as the game goes. Finally, the value of a timer can be easily visualized on the screen in different formats using the appropriate widget (see "Widgets").

nodeGame API for timer operations

Each game exposes a default timer object: `node.game.timer`. This timer is used to measure the time of execution of the current step and contains the default timeup function as defined by the stager. Therefore, under normal conditions, the game developer should never change its properties directly, but rather create new timers when needed using the `timer` API, available via the `node.timer` object.

```
// Creating a new timer.
var mytimer = node.timer.createTimer('mytimer', {
     update: 1000,
     milliseconds: 10000,
      // This is executed at every update.
      // Hooks can also be an array of functions.
     hooks: function(){
    },// This can be an event, or a function
      // to execute when the timer expires.
     timeup: "TIMEUP"
});
// Timer operations.
mytimer.start();
mytimer.pause();
mytimer.resume();
mytimer.stop();
mytimer.isStopped();
mytimer.isRunning();
```

Furthermore, the `node.timer` object allows the game developer to measure time intervals from the beginning of a step, a stage, the whole game, or a custom event. The time interval measured can be absolute or effective, i.e., the time during which a game has been paused is not counted.

```
// Measuring time intervals.
node.timer.getTimeSince("step");
node.timer.getTimeDiff("start", "step");
// Creating a custom timestamp.
node.timer.createTimestamp("myts");
// Measuring effective time, i.e. without paused time.
node.timer.getTimeSince("myts", true);
node.timer.getTimeDiff("start", "step", true);
```

The Window object

The Window object, or `W`, is available in the browser environment to load and cache pages, manipulate the DOM (Document Object Model), and to setup the user interface.

Frames and header

Frames are HTML pages containing the interface of a game step. Unless a widget is used, frames must be generally implemented by the game developer in one or more separate files: as static resources (in `public/`), or as dynamic templates (in `views/`). Frames are loaded into a dedicated iframe,[17] which is separate from the rest of the page.

The header is another special region of the page that can be appositely created to contain elements that need to persist throughout the whole game, e.g., a timer, or a counter with the accumulated earnings. The position of the header relative to the frame can be easily manipulated via API.

Frames and other static resources can be explicitly cached at any time during an experiment. Caching provides a twofold advantage. Firstly, it decreases the load on the server (especially stages are repeated for many rounds). Secondly, it guarantees a smoother game experience by bringing down the transition time between two subsequent steps. However, caching might not be available in older browsers (e.g., IE8), therefore a pre-caching test is performed automatically upon loading nodeGame, and the results are saved in a global variable.

nodeGame Window API to load and manipulate frames

- **`W.generateFrame()`**: Generates the frame.
- **`W.generateHeader()`**: Generates the header.
- **`W.setHeaderPosition(position)`**: Moves the header in one of the four cardinal positions: 'top', 'bottom', 'left', 'right.'
- **`W.loadFrame(uri, cb, options)`**: Loads a page from the given URI, and then executes the callback cb. This function is automatically called whenever the **`frame`** property of a step is set.
- **`W.setUriPrefix(prefix)`**: Adds a prefix to the URIs loaded with **`W.loadFrame`**.
- **`W.preCache([res1, res2, ...], cb)`**: Caches the requested resources, and executes the callback cb when finished.
- **`W.cacheSupported`**: Flags the support for caching resources.

[17] Iframes are special HTML tags that are used to visualize entire HTML documents inside of a nested HTML page. nodeGame uses iframes to render the pages of the different stages of an the experiment. For more information about iframes see: http://en.wikipedia.org/wiki/HTML_element#Frames

Searching and manipulating the DOM

Elements of the frame might not be immediately accessible from the parent page. Therefore, the `W` object offers a number of methods to access and manipulates them easily.

nodeGame Window API to search and manipulate the DOM

- **`W.getElementById(id)`**: Looks up an element with the specified id in the frame of the page.
- **`W.setInnerHTML(search, replace, mod)`**: Sets the content of element/s with matching id or class name or both.
- **`W.searchReplace(elements, mod, prefix)`**: Replaces the content of the element/s with matching id or class name or both.
- **`W.write/writeln(text, root)`**: Appends a text to the root element.
- **`W.sprintf(string, args, root)`**: Appends a formatted string to the root element.
- **`W.hide/show(idOrObj)`**: Hides/shows an HTML element.

User interface

The `W` object also exposes methods to directly restrict or control some types of user action, such as: disabling the back button, detecting when a user switches tab, displaying an animation in the tab's header, etc.

nodeGame Window API to manipulate the user interface

- **`W.disableRightClick()`**: Disable the right click menu.
- **`W.enableRightClick()`**: Enables the right click menu.
- **`W.lockScreen(text)`**: Disables all inputs, overlays a gray layer above the elements of the page and optionally writes a text on top of it.
- **`W.unlockScreen()`**: Undoes **`W.lockScreen`**.
- **`W.promptOnleave(text)`**: Displays a message if the user attempt to close the tab of the experiment.
- **`W.restoreOnleave()`**: Undoes **`W.promptOnleave`**
- **`W.disableBackButton(disable)`**: Disables the back button if parameter 'disable' is true, re-enables it if false.
- **`W.onFocusIn(cb)`**: Executes a callback each time the players switches tab.
- **`W.onFocusOut(cb)`**: Executes a callback each time the players returns to the experiment's tab.
- **`W.blinkTitle()`**: Displays a rotating text on the tab of the experiment.
- **`W.playSound(sound)`**: Plays a sound, if supported by browser.

Widgets

Widgets are reusable user interface components that can be loaded dynamically in a nodeGame experiment. They serve multiple purposes. For example, they can display the values of internal timer objects, the number of remaining rounds in a stage, the amount of monetary rewards gained by a player so far, etc. They can also offer a chat window to communicate between experimenter and participants, or simply display debugging information during the development and testing of the experiment. Figure 5 shows some examples of widgets displaying timers and counting the number of rounds.

Widgets API

Widgets are available via the `node.widgets` object that exposes a number of methods, among which:

- **`node.widgets.get(widget, options)`**: Returns a new instance of the specified widget (no HTML code is created here, so the method is safe to be called also by bots running on the server). Adds a reference of the widget to the collection **`node.widgets.instances`**.
- **`node.widgets.append(widget, root, options)`**: Calls the **`get`** method, and then appends the widget to the specified root.
- **`node.widgets.register(name, prototype)`**: Register a new widgets to the collection.

All widgets implement a set of standard methods, such as:

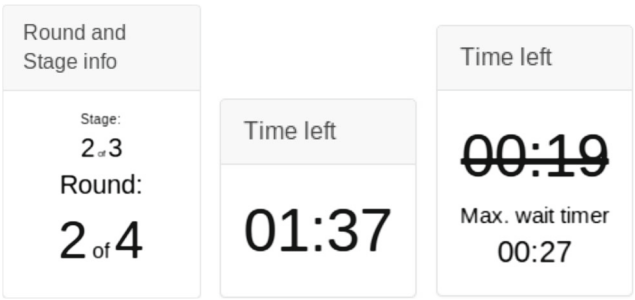

Fig. 5 Illustration of the VisualRound and VisualTimer widgets for nodeGame. On the *left*, the VisualRound widget visualizes the stage and round counts. In the *middle*, the VisualTimer widget shows the default time countdown. On the *right*, a participant completed the stage and this triggered the VisualTimer widget to stop; the timer now shows the maximum time left until all other participants complete the stage

- **hide/show**: Makes the widget visible or invisible.
- **disable/enable**: The widget stays visible, but it is not clickable.
- **unhighlight/highlight**: Highlights the widget in some way, or restores its appearance to default.
- **getValues/setValues**: Get/set the current selection of the widget (when available).
- **destroy**: Removes any listener defined by the widget, removes HTML created by the widget from the DOM, and removes its reference from **node.widgets.instances**.
- **setTitle**: Writes content to the HTML title of the widget.
- **setFooter**: Writes content to the HTML footer of the widget.
- **setContext**: Sets the Twitter-Bootstrap context, i.e., 'primary', 'info', 'success', 'warning', or 'danger.'

Moreover, after a widget has been appended, the following references to the HTML elements of the interface are added as properties of the widget:

- **panelDiv**: The main div containing all sub-elements.
- **headingDiv**: The div containing the title of the widget.
- **bodyDiv**: The div containing the main content of the widget.
- **footerDiv**: The div containing the footer of the widget (hidden by default).

List of selected widgets

- **ChernoffFaces**: Displays parametric images in the form Chernoff faces (Chernoff, 1973). Optionally, an interface to draw customized Chernoff faces can be added.
- **Chat**: Creates a configurable chat element.
- **ChoiceTable**: Creates a row (or column) or selectable choices.
- **ChoiceTableGroup**: Creates a configurable group of `ChoiceTable` widgets.
- **ChoiceManager**: Creates and manages a group of widgets.
- **DebugInfo**: Displays information useful for debugging purposes.
- **DoneButton**: Creates a button that executes **node.done** when pressed. The button can be customized via the **donebutton** property of the stager.
- **LanguageSelector**: Displays a selector box with the languages supported by the game (see "Templates and internationalization"). The selection is saved and used to update the URIs of the loaded frames accordingly.
- **MoneyTalks**: Displays a configurable box counting the accumulated earnings of a participant in the currency of choice.
- **MoodGauge**: Displays a customizable mood measurement interface. Available gauge: Positive and Negative Affect Schedule (PANAS) Short Form (I-PANAS-SF) (Watson et al., 1988; Thompson, 2007).
- **SVOGauge**: Displays a customizable social value orientation (SVO) measurement interface. Available gauge: slider measure (Murphy et al., 2011).
- **VisualTimer**: Creates a box displaying the remaining/elapsed time of a game timer. The box can be customized via the **timer** property of the stager.
- **VisualRound**: Displays a customizable box containing information about the current round, step, and stage.

Widget-steps

Widgets-steps are a particular type of steps where a widget is loaded in the frame and waits for some user input. When `node.done()` is called, the widget checks if the input is correct, and in case it is not, it will not let the player step forward. Correct and incorrect values are stored and sent to the server when the game advances.

To create a widget-step, just add a `widget` property to an existing step:

```
stager.extendStep("mood", { widget: "MoodGauge" });
```

nodeGame's architecture

Even though developed independently, nodeGame follows the same software paradigm proposed by Hawkins (2014). However, nodeGame is not just a proof of concept, but a stable software package ready to be used by behavioral researchers. Compared to Hawkins (2014), nodeGame (version 3.0) does not yet implement a physics engine, nor a collision detection system for the positions of players in virtual 2D or 3D space. For the time being, these features could be implemented by a game developer following the implementation by Hawkins (2014), until they are integrated in successive versions of the software.

The next paragraphs describe nodeGame's architecture and a few technical concepts related with the technologies used: Node.JS, JavaScript, HTML5.

Overview

nodeGame is a modular framework, entirely implemented in JavaScript/Node.js. JavaScript is the standard programming language of the browser, it is completely event-driven

and permits to respond to users' actions, such as the click of a button, and to fully manipulate the content of an HTML page. Node.js is its server side equivalent. Based on the V8 JavaScript engine used by Chromium and Google Chrome, Node.js can execute CPU intensive tasks with performances comparable or even superior to other interpreted programming languages (Bezanson and et al. 2014). However, the greatest advantage of Node.js is that it is entirely event-based, exactly like JavaScript. This means that Node.js deals with I/O (Input/Output) requests asynchronously, making access to resources such as file system and databases "non-blocking." Let us consider the following example to clarify the concept. When a client connects to the server for the first time, its credentials need to be verified. Usually, this implies an access to the database, a task that Node.js delegates to the operating system. In this way, while the credentials are being retrieved from the database, Node.js can serve another request. After a certain amount of time, the operating system returns the results of the database operation to Node.js, which processes and serves them to the requester as soon as possible. This feature makes Node.js particular suited to handle a large number of simultaneous connections, optimizing application's throughput and scalability.[18]

Besides performance and scalability, using Node.js as server side language introduces another important advantage: *code re-use*. In fact, nodeGame follows the Client-Server paradigm, and shares the same classes and data structures between its two main architectural components: (i) nodeGame client and (ii) nodeGame server. This means that an instance of nodeGame client can be executed on the browser to respond to a player's action, or on the server machine to control a game room (logic) or as an automated player (bot). As Fig. 6 shows, nodeGame client itself is a modular application, and nodeGame server imports only those components that are actually needed from it.

The rest of the nodeGame infrastructure includes a number of popular software packages as dependencies to handle non-core tasks such as implementing the network transport layer or the logging system. As shown in Fig. 7, nodeGame uses Winston[19] for logging, Express[20] for answering HTTP requests, and Socket.io[21] as a multi-transport messaging library. In particular, Socket.io guarantees fast delivery of messages over the network across a broad range of devices

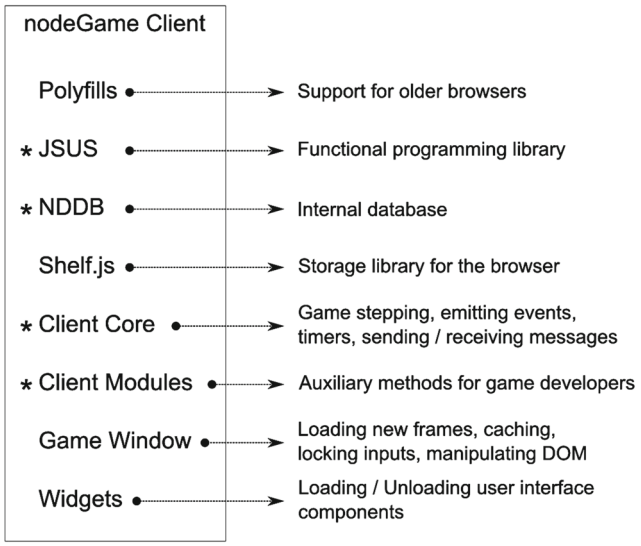

Fig. 6 Schematic representation of the nodeGame client architecture. nodeGame client is composed of a collection of modules serving different purposes. In the browser, all of them are usually loaded, while on the server only those modules marked with an *asterisk* are used by logics and bots

and environments. In fact, it implements a number of different message-transport protocols, and automatically negotiates with incoming clients the one achieving the best results. Socket.io supports HTML5 WebSockets, which represents a major improvement in the history of Web data communication and constitute the royal road for the implementation of large-scale real-time synchronous games in the browser.[22]

Channels

A central component of the nodeGame server architecture is the server *channel*. A server can have as many channels as needed, and generally as many as the number of games currently installed. As illustrated in Fig. 8, a channel is composed of several nested objects: the waiting room, the clients registry, the collection of game levels and game rooms, and two independent game servers, one for the players and one for the administrators. Each of the two internal game servers features a different set of event handlers, and requires different privileges of access to them. Event handlers reply to incoming messages either by returning another message or by triggering an internal operation. The Admin game server has an extended set of event handlers, which includes methods, for example, to redirect and remotely setup any connected client. Such privileged operations are not allowed by the Player game server, whose task is limited to pass along messages. Each of the two

[18] As reported on the Web site of the project (http://nodejs.org): "Node.js building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

[19] See https://github.com/winstonjs/winston.

[20] See http://expressjs.com/.

[21] See http://socket.io/.

[22] For more information see http://en.wikipedia.org/wiki/WebSocket or https://www.websocket.org/.

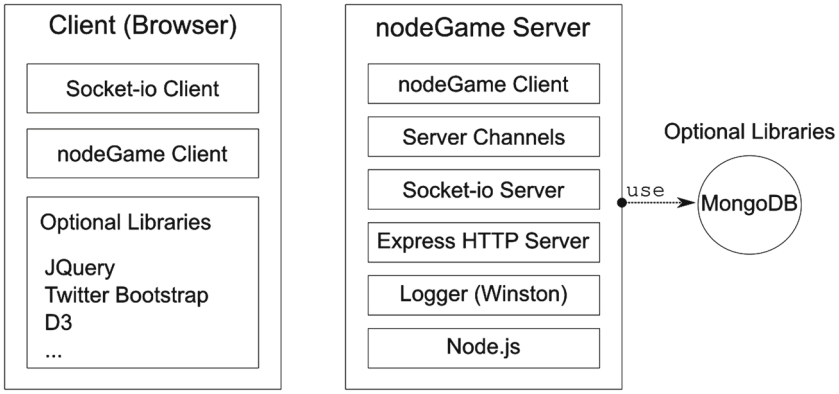

Fig. 7 Schematic representation of the nodeGame architecture. The two main components, server and client, are modularized in nested components. Notice how nodeGame client is shared among client and server. For more details about the internal components of nodeGame client refer to Fig. 6

internal game servers contains a dedicated socket manager, which is an abstraction of the actual type of connection used by the server to communicate with the clients based on their location of execution. For example, Socket.io is the type of socket used to communicate with clients located in the browser of remote machines; Socket Direct, instead, is used to exchange messages with logics and bots running as separated processes within nodeGame server. The channel registry stores information about all connected and disconnected clients, such as their game state and the game room they belong to. The registry can be accessed by the other software components whenever needed. For example, the channel waiting room might consult the registry before starting a new game room. Each game room represents a wrapper around a group of clients, which can exchange messages in a dedicated space. One of those, the *logic*, has the purpose of controlling the advancement of the game. The logic is implemented by the experimenter, as explained in "Creating experiments with nodeGame".

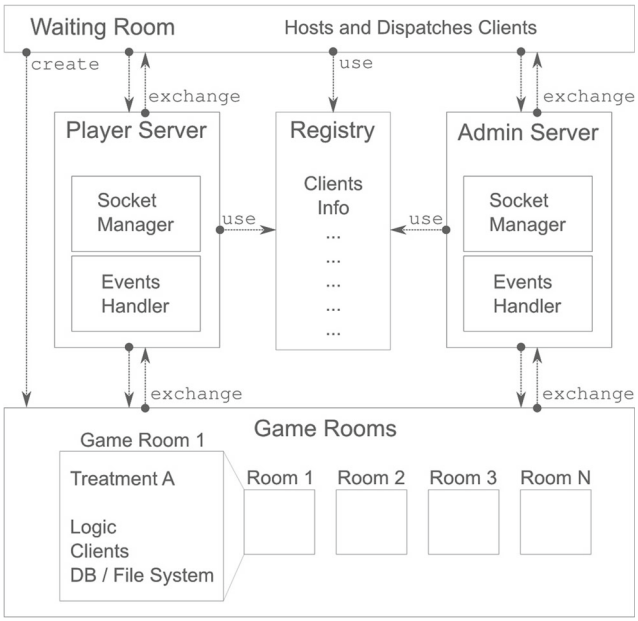

Fig. 8 Schematic representation of the server channel architecture. The server channel is composed of the following components: the waiting room, the clients registry, the collection of game rooms, and two independent game servers, one for the players and one for the administrators

Get involved

nodeGame is an open-source project under active development. A public organization containing its source code is hosted at Github[23] at the address https://github.com/nodeGame/. Therein, it is possible to access the source code of all the individual components of the nodeGame architecture: client, server, and all the supporting libraries. People interested in joining the development of nodeGame are welcome, and are encouraged to follow the guidelines for developers on the project wiki (http://nodegame.org/wiki/), to sign up to the nodeGame mailing list for announcements, help or features requests, or to follow the Twitter channel `@nodegameorg`.

Conclusions

This paper introduced nodeGame, new software for conducting real-time (but also discrete-time), synchronous online experiments. The source code is open, and it can be downloaded for free from http://nodegame.org.

The main motivation behind the software was to support behavioral research along three dimensions: (i) size, (ii) time, and (iii) granularity. As described in "Waiting room",

[23] Github is a collaborative platform for code review and code management of software projects.

nodeGame can support a large group *size* simply by modifying the settings of the waiting room, for example increasing the value of the parameters `POOL_SIZE` and `GROUP_SIZE`. Moreover, the *time* dimension can be easily manipulated by implementing game steps with the desired speed of interaction among players. For example, implementing a continuous dilemma game (Friedman & Oprea, 2012), would simply require: a new call to the method `node.say` each time a player presses a submit button, and registering an event handler with `node.on.data` that updates the screen with the last information received (see "Client types"). Finally, the *granularity* dimension can be addressed by adding multiple treatments with the desired values for the parameters (see "Game variables and treatments"). This is easy to implement because treatments are just normal JavaScript objects that can be created in a standard for-loop.

Summary of advantages of nodeGame

- nodeGame can be executed in different environments, such as laboratory, online, field, and even lecture hall, and makes switching among those easy.
- nodeGame grants great flexibility to the experimenters in the definition of experimental setups.
- nodeGame supports both turn-based and real-time experiments.
- nodeGame's scalable architecture can support large-scale experiments with hundreds of players even with a standard laptop machine.
- nodeGame lowers the barrier for online participation because clients can join an experiment without the need of installing any additional software.
- nodeGame can reduce the costs of maintaining a laboratory pool, since only one computer needs to be updated with future releases of the software.
- nodeGame makes it possible to realize rich and interactive user interfaces with the use of standard Web technologies (HTML, CSS, JavaScript).
- Being open source, it is relatively easy to find good Web developers able to implement experiments at a reasonably low cost.

Current limitations of nodeGame

- Experimenters need some programming skills to implement new experiments in nodeGame. This is a design feature, because programming an own experiment grants more flexibility, and can increase the general performance of the application, a vital goal for achieving large-scale experiments.
- There is only limited support for matching algorithms, and mapping roles to participants. However, with average computer programming skills it is possible to write custom code achieving the same goals in a relatively straightforward way.
- There is only limited support for integration with external databases. Currently, only MongoDB is supported.
- nodeGame does not provide a physics engine or an engine for the movement of sprites in 2D or 3D environments.
- nodeGame does not do the recruitment of participants. This is unlikely to change in the future given the broad ecosystem of online labor markets available.
- nodeGame provides only partial integration with one online labor market: Amazon Mechanical Turk. Experimenters must have their own employer accounts, and upload their tasks outside of nodeGame.
- The online documentation for nodeGame covers most of the use cases, but not yet all of them. 100 % coverage will be reached in the near future, hopefully with the help of a community of users and contributors. As pointed out by Horton et al. (2011), we not only need better tools but also better documentation.

Future developments of nodeGame

Future developments of nodeGame will focus on extending the number of available widgets and predefined game frames, with special attention to multi-player widgets. This will further reduce the implementation load on developers.

Furthermore, more matching algorithms will be provided, including network algorithms and automatic assignments of roles to players.

Finally, even though it cannot be guaranteed that no backward-incompatible change will be introduced, future versions of nodeGame will try to minimize these. To this extent, an apposite migration page on the project of the wiki will keep an up-to-date list of such changes for every new release of nodeGame, see for example: https://github.com/nodeGame/nodegame/wiki/Migrating-To-v3.

Some final considerations

nodeGame is entirely open-source and free software. As science increasingly relies on computational technologies to achieve its goals, it is tremendously important that scientific research is performed making use of open-source and free software as much as possible (Sonnenburg et al., 2007; Schwarz and Takhteyev, 2010). Choosing open and free software ensures full replicability to scientific results obtained with computer-mediated methods. Moreover, it guarantees that errors in the source code can easily be spotted and quickly fixed by the same community of users (Von Krogh et al., 2003).

nodeGame is specifically designed to support behavioral research conducted along three dimensions: (i) experiments

with larger group size, (ii) real-time experiments, and (iii) batches of parallel experiments. The simultaneous exploration of these three research dimensions in the laboratory and online is expected to put under severe *test* the predictions of current theories, and to lead to the *generation* of new ones of greater explanatory power. Accelerating the cycle of hypothesis production and testing is key to make rapid scientific progress by iteratively generating and refining evidence-based knowledge about human behavior. nodeGame aims at being a valuable instrument freely available to the community of social scientists in such a process.

Acknowledgments The author is grateful to all the persons who have contributed to the development of nodeGame over the years: Philipp Küng, Benedek Vartok, Sebastien Arnold, Lionel Miserez, Jan Wilken Dörrie, and Nicole Barbara Lipsky. The author is also indebted for insightful and strategic discussions with Oliver Brägger, Stefan Wehrli, Michael Mäs, Dirk Helbing, Ryan O. Murphy, Christoph Riedl, and Luke Horgan. The author's work on nodeGame was gratified over the years by the financial support from the group of Computational Social Science (COSS) at ETH Zürich, the ETH Decision Science Laboratory (DeSciL), and the seed grant PEER by the Institute for Science Technology and Policy (ISTP).

References

Anderson, B., Bernauer, T., & Balietti, S. (2016). Effects of fairness principles on willingness to pay for climate change mitigation, submitted.

Andersson, C., & Read, D. (2014). Group size and cultural complexity. *Nature*, *511*(7507), E1. doi:10.1038/nature13411.

Balietti, S., Goldstone, R.L., & Helbing, D. (2016). Peer review and competition in the Art Exhibition Game. *Proceedings of the National Academy of Sciences (PNAS)*, *113*(30), 8414–8419.

Balietti, S., Jäggi, B., & Axhausen, K. (2016). Efficiency gains in coordination in information-poor environments. Available at SSRN 2802049.

Balietti, S., Mäs, M., & Helbing, D. (2015). On disciplinary fragmentation and scientific progress. *PLoS ONE*, *10*(3). e0118747.

Bezanson, J., et al. (2014). Julia: a fresh approach to numerical computing. arXiv:1411.1607 [cs.MS].

Birnbaum, M.H. (2000). SurveyWiz and FactorWiz: JavaScript Web pages that make HTML forms for research on the Internet. *Behavior Research Methods, Instruments, & Computers*, *32*(2), 339–346.

Birnbaum, M.H. (2004). Human research and data collection via the Internet. *Annual Review of Psychology*, *55*, 803–832.

Bos, N. (2002). Effects of Four Computer-Mediated Communications Channels on Trust Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, (pp. 135–140).

Butler, B.S. (2001). Membership size, communication activity, and sustainability: a resource-based model of online social structures. *Information Systems Research*, *12*(4), 346–362.

Centola, D. (2010). The spread of behavior in an online social network experiment. *Science*, *329*(5996), 1194–1197.

Chaudhuri, A. (2011). Sustaining cooperation in laboratory public goods experiments: a selective survey of the literature. *Experimental Economics*, *14*, 47–83.

Chen, C., Schonger, M., & Wickens, C. (2014). oTree: an open-source platform for laboratory, online, and field experiments. http://www.otree.org/.

Chernoff, H. (1973). The use of faces to represent points in K-dimensional space graphically. *Journal of the American Statistical Association*, *68*(342), 361–368.

Ciampaglia, G.L. (2014). Power and fairness in a generalized ultimatum game. *PLoS ONE*, *9*(6). e99039.

Cox, J.C., & Swarthout, J.T. (2005). EconPort: creating and maintaining a knowledge commons. *Andrew Young School of Policy Studies Research Paper*, 6–38.

DellaVigna, S. (2016). What motivates effort? Evidence and expert forecasts NBER Working Paper No. 22193.

Derex, M., et al. (2013). Experimental evidence for the influence of group size on cultural complexity. *Nature*, *503*(7476), 389–391.

Fischbacher, U. (2007). z-Tree: Zurich toolbox for ready-made economic experiments. *Experimental Economics*, *10*(2), 171–178.

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

Hawkins, R.X.D. (2014). Conducting real-time multiplayer experiments on the Web. *Behavior Research Methods*, 1–11.

Helbing, D. (2014). Conditions for the emergence of shared norms in populations with incompatible preferences. *PLoS ONE*, *9*(8). e104207.

Heinrich, J. (2004). Demography and cultural evolution: How adaptive cultural processes can produce maladaptive losses: the Tasmanian case. *American Antiquity*, 197–214.

Horton, J.J., Rand, D.G., & Zeckhauser, R.J. (2011). The online laboratory: conducting experiments in a real labor market. *Experimental Economics*, *14*(3), 399–425.

Kraut, R., et al. (2004). Psychological research online: Report of Board of Scientific Affairs' Advisory Group on the Conduct of Research on the Internet. *American Psychologist*, *59*(2), 105–107.

Lakkaraju, K., et al. (2015). The Controlled, Large Online Social Experimentation Platform (CLOSE). In *International Conference on Social Computing, Behavioral-Cultural Modeling, and Prediction*, (pp. 339–344).

Leeuw, J.R.d.e. (2014). jsPsych: A JavaScript library for creating behavioral experiments in a Web browser. *Behavior Research Methods*, *47*(1), 1–12.

Leeuw, J.R.d.e., & Motz, B.A. (2016). Psychophysics in a Web browser? Comparing response times collected with JavaScript and psychophysics toolbox in a visual search task. *Behavior Research Methods*, *48*(1), 1–12.

Mangan, M.A., & Reips, U.-D. (2007). Sleep, sex, and the Web: surveying the difficult-to-reach clinical population suffering from sexsomnia. *Behavior Research Methods*, *39*(2), 233–236.

Mao, Q. (2015). Experimental studies of human behavior in social computing systems. PhD thesis. Harvard.

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

Murphy, R.O., Ackermann, K.A., & Handgraaf, M. (2011). Measuring social value orientation. *Judgment and Decision Making*, *6*(8), 771–781.

Musch, J., & Reips, U.-D. (2000). A Brief History of Web Experimenting. Birnbaum, M.H. (Ed.) *Psychological experiments on the Internet*: Academic Press.

Nax, H.H., et al. (2016). A welfare investigation of generalized contribution-based competitive grouping. Available at SSRN 2604140.

Neath, I., et al. (2011). Response time accuracy in Apple Macintosh computers. *Behavior Research Methods*, *43*(2), 353–362.

Nosenzo, D., Quercia, S., & Sefton, M. (2013). Cooperation in small groups: the effect of group size. *Experimental Economics*, *18*(1), 4–14.

Olson, M. (1965). *The logic of collective action: public goods and the theory of groups*. Cambridge: Harvard University Press.

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

Pettit, J., et al. (2014). Software for continuous game experiments. *Experimental Economics*, *17*(4), 631–648.

Rand, D.G., Greene, J.D., & Nowak, M.A. (2012). Spontaneous giving and calculated greed. *Nature*, *489*(7416), 427–430.

Reimers, S., & Stewart, N. (2015). Presentation and response timing accuracy in Adobe Flash and HTML5/JavaScript Web experiments. *Behavior Research Methods*, *47*(2), 309–327.

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

Reips, U.-D., & Krantz, J.H. (2010). Conducting True Experiments on the Web. Gosling, S.D., & Johnson, J.A. (Eds.) *Advanced methods for conducting online behavioral research*: American Psychological Association.

Reips, U.-D., & Neuhaus, C. (2002). WEXTOR: A Web-based tool for generating and visualizing experimental designs and procedures. *Behavior Research Methods, Instruments, & Computers*, *34*(2), 234–240.

Rubinstein, A. (2013). Response time and decision making: an experimental study. *Judgment and Decision Making*, *8*(5), 540–551.

Salganik, M.J., Dodds, P.S., & Watts, D.J. (2006). Experimental study of inequality and unpredictability in an artificial cultural market. *Science*, *311*(5762), 854–856.

Schwarz, M., & Takhteyev, Y. (2010). Half a century of public software institutions: open source as a solution to hold-up problem. *Journal of Public Economic Theory*, *12*(4), 609–639.

Simon, L.K., & Stinchcombe, M.B. (1989). Extensive form games in continuous time: Pure strategies. *Econometrica: Journal of the Econometric Society*, 1171–1214.

Sonnenburg, S., et al. (2007). The need for open-source software in machine learning. *Journal of Machine Learning Research*, *8*, 2443–2466.

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

Thompson, E.R. (2007). Development and validation of an internationally reliable short-form of the Positive and Negative Affect Schedule (PANAS). *Journal of Cross-Cultural Psychology*, *38*(2), 227–242.

Von Krogh, G., Spaeth, S., & Lakhani, K.R. (2003). Community, joining, and specialization in open-source software innovation: a case study. *Research Policy*, *32*(7), 1217–1241.

Wang, J., Suri, S., & Watts, D.J. (2012). Cooperation and assortativity with dynamic partner updating. In *Proceedings of the National Academy of Sciences (PNAS)*, (Vol. 109, pp. 14363–14368).

Watson, D., Clark, L.A., & Tellegen, A. (1988). Development and validation of brief measures of Positive and Negative Affect: the PANAS Scales. *Journal of Personality and Social Psychology*, *54*(6), 1063–1070.