Project Number MBJ 1404

Knecht

A Server for Data Analytics and Multiplayer Games

A Major Qualifying Project
Submitted to the faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted by

Ben Carlson, Patrick Feeney and Ian Fite 2014

Advised by

Gary Police, CS Professor of Practice

Brian Moriarty, IMGD Professor of Practice

Abstract

Knecht is a Node.js client/server application that allows networked http-based applications to store arbitrary data for later retrieval and share data between multiple clients. It is designed to support game analytics and multiplayer game functionality for game engines written in HTML5/Javascript and other browser technologies.

Contents

Project History....1

Annotated Example of Knecht In Use....8

Appendix A: Local Installation Instructions....29

Appendix B: Live Installation Instructions....30

Appendix C: Knecht Client API....31

K.configure (options)....31

K.getUsers (constraints, callback)....32

K.register (username, password, options, callback)....34

K.unregister (callback)....35

K.login (username, password, options, callback)....36

K.logout (callback)....37

K.changePassword (password, callback)....38

K.putData (field, data, options, callback)....38

K.getData (field, options, callback)....40

K.deleteData (field, options, callback)....41

K.getGroups (constraints, callback)....42

K.startGroup (group_name, callback)....43

K.closeGroup (group_name, callback)....44

K.listenInput (group_name, clear, callback)....45

K.submitInput (group_name, input, callback)....46

K.listenUpdate (group_name, limit, clear, callback)....47

K.submitUpdate (group_name, field, data, callback, member, permission)....49

K.stopListening (group_name, permission)....50

K.getGroupData (group_name, field, callback) 51
K.setPermission (group_name, field, permission, callback, member) 52
K.addMember (group_name, member, callback) 53
K.removeMember (group_name, member, callback) 54
Appendix D: Demo Source Code 56
Appendix E: Reference Bibliography 89

Figures

Figure 1: An example of code used for testing. 5
Figure 2: Demo login field. 9
Figure 3: Demo registration conformation menu. 9
Figure 4: Demo main menu. 11
Figure 5: Demo password change field. 12
Figure 6: Single player demo. 13
Figure 7: Single player demo upon saving. 13
Figure 8: Single player demo before 15
Figure 9: Single player demo after reloading previous save. 15
Figure 10: Trying to load a single player save after deleting it. 16
Figure 11: Demo group finding and creating field. 17
Figure 12: Group demo with no members other than the host. 20
Figure 13: Group demo with two members from the client's perspective. 23
Figure 14: a client joins a group as the host sees it 25
Figure 15: A client joins the group as the client sees it. 27

Project History

Our Major Qualifying Project was undertaken over the course of the 2013-2014 academic year. It was originally planned as a 3-term MQP, to take place over A, B, and C terms. Two members of our team, Patrick and Ian, were to claim 2/3 credit in C term to fulfill the 4/3 credit requirement of a double CS/IMGD MQP, while our third member, Ben, claimed the 3/3 needed to complete his IMGD requirement. Due to numerous issues that arose over the course of the year, it was necessary to extend the project into D term.

Ben was the member who brought the team into contact with our project advisor from the IMGD department, Professor Moriarty, and told us about his ideas for potential MQPs. It was significantly more difficult to locate a CS professor to advise for the second half of Ian and Patrick's MQP credit, as many of the professors in that department were either unavailable or felt that their expertise lay outside the focus of our project concepts. Eventually, Professor Pollice was chosen to be our advisor, due to his experience with software engineering.

The goal of our project underwent major changes of direction in its early stages. The initial concept, proposed in D term before the project's official start date, was to create an improved version of an existing chase game (written with Professor Moriarty's Perlenspiel engine) in which the map would be procedurally generated.

An additional challenge to this task was to be that the game entity the player chases would 'tease' the player by remaining close, but always out of reach. This would have necessitated coding an AI for the game capable of navigating any given map that might be generated for it. The generation would additionally have to be such that the game would always take about five minutes to complete.

Other proposed components of the project at this stage included writing ports of the Perlenspiel engine to additional platforms, such as the Ouya. This concept was eventually dropped, due to the realization that, since it is a purely JavaScript- and HTML-based engine and can therefore run in most browsers, there was little practical need for platform-specific versions of Perlenspiel to be created.

In A-term, the project began with its focus still entirely on working with the Perlenspiel engine, with an eye towards eventually implementing Professor Moriarty's idea for a fluid volume-based puzzle game. The major work done during this time was the creation of a robust suite of unit tests, written by Patrick, to aid in debugging and developing future versions of the engine. Notable bugs among those uncovered during testing were one that accidentally swapped the blue and green channels of a color object in the color validation function and numerous small holes and discrepancies in default values and parameter checking.

Meanwhile, Ben and Ian worked together to develop the first rudimentary versions of a client-server application, originally conceived as a means to collect analytics data on Perlenspiel's users, that would eventually become the core product of our MQP. The task was split between them, with Ben in charge of server-side coding, and Ian in charge of the client to communicate with it. By the end of term, a simple proof of concept had been created, demonstrating that the two halves of the application were indeed capable of making contact with each other through the use of AJAX requests.

Progress on both parts of the project was slower than anticipated, and extended into B term. Patrick continued to improve the thoroughness of the Perlenspiel unit tests, while providing some editing for clarity to the client-side JavaScript being developed by Ian.

Ben continued to work on the server side of the project, but near the end of term had to step away from the project due to health issues. At this time Patrick took over for him in writing the server-side portion of the code.

During the first half a C term, Ian and Patrick implemented preliminary versions of all server functionality, with Patrick writing the majority of the code and Ian writing the majority of the documentation. At this point Ben's health had recovered and he worked closely with Professor Moriarty to integrate Knecht with the latest version of Perlenspiel.

D term was spent debugging the code, including several major issues that arose involving our notifications system, refining both the in-code comments and external documentation of our API, producing sample applications to demonstrate Knecht's functionality, and writing this report. A few functions were added to the API late in the project during demo development when it became clear that needed functionality was not yet present.

From the outset of the project, we wanted to work on a project that was portable; a project

that would work independent of the devices we would test our project on. After going through preliminary ideas and some rudimentary testing, we eventually developed server code and tests designed to ensure that our server code and client code would be able to save data and facilitate communication with multiple users in an application- and platform-agnostic manner. Below is our account of the project's inception, planning, development, and testing.

At first the project began as the idea of a portable game engine that could have code be transferred from device to device and still be able to play, because it would run independently of the machine. Our thoughts turned to Professor Brian Moriarty's game engine: Perlenspiel. Perlenspiel utilizes minimalist graphics and the core engine is written completely in JavaScript, designed to be run via web browsers. Due to its simple, lightweight graphics and API, testing would not be as demanding and could easily show our proof of concept.

In the pursuit of efficiency, originally planned to write the engine in C++ using the Simple Direct Media Layer (SDL) Library to handle graphics, since that engine is supported on Windows, MacOS, Linux, Android, using Mozilla's SpiderMonkey to interpret the JavaScript code.

We quickly realized that this initial project idea was redundant. Because many of these devices already have web browsers that are able to play Perlenspiel games accessed from the web, we decided to look at other ideas instead. Professor Moriarty expressed an interest in a program that would facilitate game analytics for a class he hoped to teach in the 2014-15 academic year. This application would require storage of arbitrary data, independent of whatever computer the game was being played on, necessitating a remote server to store that data.

Once that was decided, it was clear that we would be nine tenths of the way to being able to store save game data or high scores as well. It was decided then that the benefits of allowing the same application to retrieve data stored on the server were great, compared to the small amount of additional programming it would require.

As possible implementations were discussed, it became clear that there wasn't any reason different clients couldn't access the same data in real time. We eventually decided to write multiplayer functionality into our application as well. Because Perlenspiel is an engine, not a single standalone game, we decided that this server would be game agnostic, meaning that the server code was not being written around an existing game. Any application would be able to use this server provided it was on a platform capable of sending JSON packets over http requests.

As our platform for building the server-side application component of our project, we decided to use Node.js. Node.js is a C++-based platform built on top of the Google V8 Javascript engine, allowing server applications to be written entirely in JavaScript. Our decision comes from Perlenspiel being written in Javascript, the ubiquity of Javascript as the main scripting language of the web and the growing interest in Node.js being used for server code. Additionally Node.js is modular, meaning that any additional platform functionality we required could easily be found and plugged into the core module.

As our development environment, we considered both Aptana and WebStorm before ultimately choosing WebStorm. This decision was due to WebStorm's customizability, the ease of setting up test servers, and the fact that Aptana at the time was not being as actively developed as WebStorm. We also decided to use MySQL for the database. This decision comes from MySQL being one of the most common databases (meaning there would be plenty of resources to refer to) as well as the fact that Node.js has an easy to use and well-documented module for accessing MySQL databases.

For data transfer, we decided to implement a RESTful API. This decision comes from REST being adopted as a standard for server requests and data transfer. All request types in REST (POST, PUT, GET, DELETE) could be easily applied to both the multiplayer functionality and the data storage and permanence. Additionally, REST is asynchronous, which would let our application handle many users sending requests at one time, and handle them when necessary. Because we were using Javascript, we decided to use AJAX (Asynchronous JavaScript and XML) and JSON (Javascript Object Notation) objects to send requests and transfer data. JSON objects are readily converted via Javascript's built-in stringify function to a format that can be stored in a MySQL database.

Because of the scope of the project and the fact that there were three members working on it, we decided to use Github to store and distribute all of the necessary files for the project. In the main folder of the project is the code and documentation for both its server and client components. In subfolders are all of the files required for testing both data permanence and multiplayer communication, as well as code for the most up to date version of Perlenspiel.

We began with simple server testing. A basic server was made and run locally on our computers to determine how to send data and store it in a database. The first successful test was that of storing and retrieving data from the database. For this, we used Felix Geisendörfer's mysql module for Node.js. All queries are made by calling the query() function which takes the query as a

string and a callback function used to handle any errors and results from the query. All queries return an array of the results in JSON format.

Testing client requests to the server proved to be more difficult. All requests are standard AJAX requests, but the format of the request header and the address URI were unintuitive at first. Our web browsers were preventing us from making these requests because they were cross-domain requests, or requests that were being sent to an outside location. While this feature avoids security risks, it interfered with the operation of our project.

It is possible to get around the cross domain restriction using CORS, Cross Origen Resource Sharing, which allows data to be accepted from machines outside the server. CORS just requires a couple extra headers in the server's response to the request: one detailing how to view an outside request, one detailing who to accept it from, and one detailing what types of requests and what types of data are acceptable to send. As long as the request abides by all of these specifications, the request will be accepted.

Another issue was having Ajax requests recognize the address to which data should be sent. Eventually we recognized that the address has to be formatted in three parts. The first is that the address must begin with the "http://" string to indicate that the request is being made with the Hypertext Transfer Protocol. The second part is the address name or the IP address of the server to which requests need to be sent. The last part of the address must be a colon followed by the port number that the server is listening to. If the address is formatted in this way, then the address will be recognized and the request will be sent.

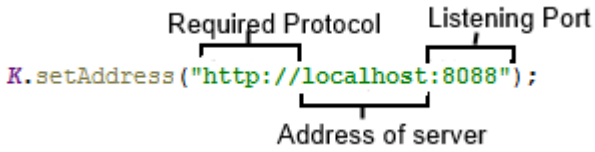

Figure 1: An example of code used for testing.

After the initial server testing, both the server code and the client code was developed and tested further and given the name "Knecht" (named after the protagonist in Herman Hesse's novel *Das Glasperlenspiel*, which also inspired the Perlenspiel engine). This time we needed to test both

saving the data and using multiplayer. Both of these tests were written in Perlenspiel, and provided proof of concept while helping us uncover more bugs.

The first of these tests was that of server data storage. This was done by modifying the *PS Paint* demonstration program for Perlenspiel. *PS Paint* is a simple program that allows the user to color squares in a grid. An extra row/menu was added to the program that gave four options: register the account (under a valid email address), log into an existing account, saving data, and retrieving existing data. For this test the data was to be the painted image, which was saved to the database as an array of RGB values. After successfully testing user registration and application registration, we were able to successfully save data, log back into our account, and retrieve the data. The major obstacle debugging revealed was a discrepancy in the documentation which was later corrected.

The second test was a simple multiplayer game that involved two players moving independently of each other. Multiplayer communication was handled by a series of inputs and updates. A host starts the application on their machine where all of the calculation takes place and invites clients to their game session. The client(s) submit updates to the server, which the host polls to update the global game state. These updates are sent back to the game server and, if the clients have permission to access that specific data, update their game instance with the data presented in the update. In the test itself, the client would send input in the form of its XY coordinates. The host would read this data and draw the client's character on the screen. The host would send its coordinates in an update that the client had permission to view, and it too would draw the host's character on the screen. Apart from a few more discrepancies in documentation, the major bugs encountered in this test was that the data was coming back as a string without being parsed into a JSON object, garbage collection on the database, and there were some errors in assigning permissions to the client. This test was run both locally as two game instances in different browsers (Firefox and Chrome) on the same machine as well as over a distance.

Once this was complete, we procured a virtual Internet server on one of WPI's machines, installed and configured Node.js and MySQL, and successfully installed the demo on it, proving that Knecht worked remotely, not just on a virtual server within the same computer. The biggest hurdle at this stage was dealing with inaccurate and/or out-of-date documentation of Node.js.

For the final presentation, we wanted a demo that was an actual game, something that took full advantage of Knecht's capabilities. Several designs were considered, but ultimately we decided that the classic Battleship would best demonstrate the ability to send messages and hide data from unauthorized users. The simple grid-based game was well suited to Perlenspiel's capacities, so we used it for this as we had with previous demos.

Annotated Example of Knecht In Use

Before an application can begin using Knecht, it must first be pointed at a server where Knecht is being hosted, and inform Knecht of its name. This is done by making a call to **K.configure()**. For example, the following code is run within our demo application's init function:

```
K.configure (
     {
          server : "http://perlenspiel.cs.wpi.edu:8080/",
          application : "Knecht Demo",
          error : function ( function_name, error )
          {
               PS.debug( error + " in " + function_name );
          }
     }
);
```

K.configure call

Here we tell Knecht that the server is located at http://perlenspiel.cs.wpi.edu:8080/, and that our application is named Knecht Demo. Now, all subsequent calls to the Knecht API in our demo will use this information to direct requests to the proper server, and to segregate our demo's data from that of other applications.

We also provide an error callback function. Now, any time the server responds to one of our requests saying that an error has occurred, we will be notified by the debug text of the nature of the error and the name of the function it occurred in.

Figure 2: Demo login field.

If for any reason we wanted to change these variables, we would once again call **K.configure()** and provide new values for the server and application members.

K.configure() is also used in our demo to interact with the player's account information. On the login screen, after the user follows the prompt and enters a username into the status bar, the demo stores that name in Knecht's internal variable:

```
K.configure( { username : input } );
```

Next, the demo requires to know whether the player will be registering a new account or logging into an existing one. To find out, it sends a request to the server asking for users who match that name:

```
K.getUsers( { username : input }, promptPassword );
```

As usernames are required to be unique, if any user is found who matches that name then a Login button is shown, and if the retrieved array is empty then the Register button is shown. If an error occurs and there is thus no array at all, or if the player at this point clicks the Cancel button, the buttons disappear and they are once again prompted to enter a username.

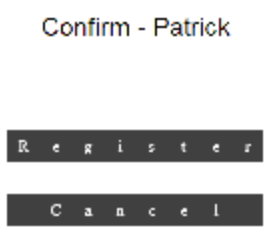

Figure 3: Demo registration conformation menu.

After prompting the user to enter their password, the input variable originally used to configure the internal *username* variable has passed out of scope. However, the stored value can be retrieved at any time through the return value of **K.configure()**. Once a password is entered and the player clicks the appropriate button to continue, one of the following calls is made:

```
var config = K.configure();
K.login( config.username, config.password, initMenu);
```

or

```
var config = K.configure();
K.register( config.username, config.password, initMenu);
```

Both functions take exactly the same arguments, and make exactly the same changes to the Knecht client's local variables. The difference between the two is that **K.register()** creates a new account on the server, while **K.login()** authenticates the player as the owner of an existing one.

In either case, a session ID is generated on the server and passed to the client, which stores it internally under the *session* variable, then passes the response to the callback function provided. In this case, the callback does nothing with the response except check it for errors, in which case the player is prompted to try again to log in.

Because our demo code does not provide a *timeout* option to either authentication function the default value is used, meaning that the player's session ID will last at most 15 minutes of inactivity before becoming expired. The expiration timer is reset each time the ID is used to confirm a user's identity in a request.

Upon receiving an Expired Session error, Knecht automatically attempts to log back in using the internally stored *username* and *password* variables, generating a new session ID if successful. However, if the login attempt fails, or the error received was instead an Invalid Session, the demo

returns the player to the login screen. This might occur because, for example, the player has logged into their account from another computer, thus generating a new session ID and invalidating the previous one. A different application could instead prevent the second login by first executing **K.getUsers()** using the *username* and *online* constraint options, and only attempting to login if the resulting array returns empty.

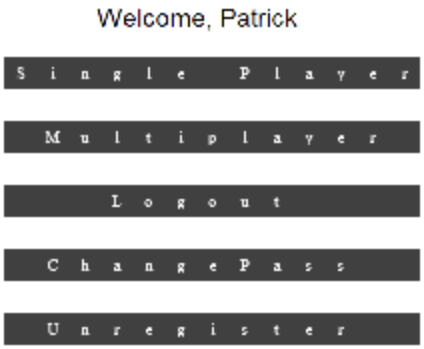

Figure 4: Demo main menu.

There are three account management functions available to Knecht users with a valid login session, each with its own button on the main menu of our demo application. **K.logout()** and **K.unregister()**, executed by clicking on the respectively labeled menu buttons, perform the inverse functions of **K.login()** and **K.register()**, and each only take a callback function all argument since they use Knecht's internal *username* and *session* variables to send their requests.

```
K.logout( initLogin );
```

or

```
K.unregister( initLogin );
```

In either case, the callback that our demo passes returns the player to the login screen.

K.changePassword() does exactly as its name suggests, prompting the user to enter a new

password and altering it without invalidating the current login session.

```
K.changePassword(new_pass, function( response )
{
     PS.statusText(response.error ?
               "Error Changing Password" : "Password
Changed");
});
```

K.changePassword call

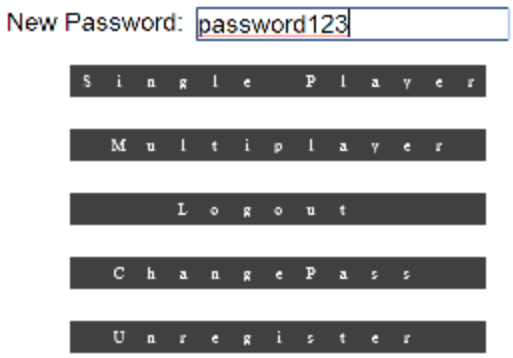

Figure 5: Demo password change field.

The callback given in our demo merely checks the server's response for errors, and updates the status line accordingly.

Selecting the Single Player button on the main menu brings the player into the single-player version of our demo's game screen. The player uses WASD to control the movement of a ghost who has been covered in dripping paint, and is now spreading it all over the floor. The player's current location is marked by a thick gray border, and every board space they enter has its color changed to blue. Pressing the escape key exits back to the main menu.

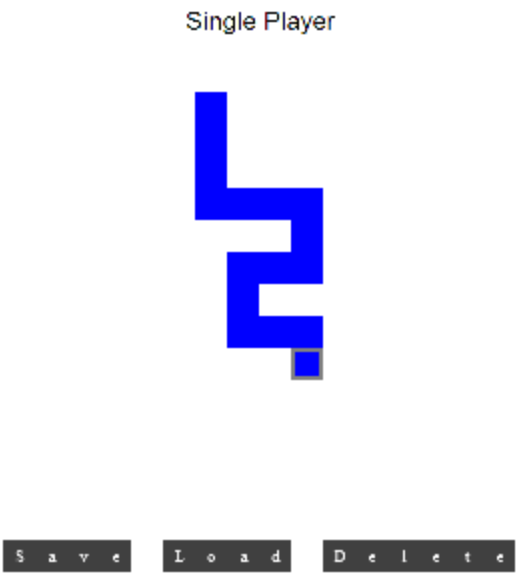

Figure 6: Single player demo.

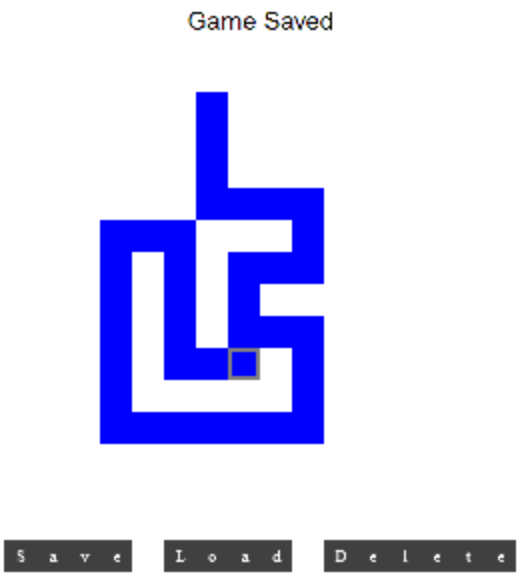

Figure 7: Single player demo upon saving.

There are three buttons at the bottom of the screen, corresponding to the three personal data functions in the Knecht client API. Choosing to Save the player's game causes the following code to execute:

```
K.putData(
     [ 'position' , 'board' ],
     [ players[K.configure().username].position,
PS.imageCapture() ],
     function( response )
     {
          PS.statusText(response.error ?
               "Error Saving Game" : "Game Saved");
     }
);
```

K.putData call

This call to **K.putData()** stores the ghost's current location in the *position* field on the server, and a screenshot of the play area's appearance in the *board* field. Knecht also attaches its internal *username*, *session*, and *application* values to the request it sends to the server, to confirm that the user is authorized to make this request and to prevent the data from colliding with an overwriting the *positio*n or *board* variables used by some other application.

Subsequently, the player may choose to restore the game to the state it was in when they saved it, by clicking the Load button, which executes the following:

```
K.getData(
          ['position', 'board'],
          function( response )
          {
               if (response.error)
               {
                    PS.statusText("Error Loading Save");
               }
               else
               {
                    if (response.data.board)
                    {
                         var name = K.configure().username;
                         PS.border(
                              players[name].position.x,
                              players[name].position.y, 0);
                         players[name].position =
     response.data.position;
                         PS.imageBlit(response.data.board, 0, 0);
                         PS.border(
                              players[name].position.x,
                              players[name].position.y, 4);
                         PS.statusText(“Save Loaded”);
                    }
                    else
                    {
                         PS.statusText("No Save To Load");
                    }
               }
          }
     );
```

K.getData call for single player usage

The callback to **K.getData()** defined here first checks that no errors occurred while processing the request. It then checks whether or not the *board* data was included in the response. If it was, then the demo sets the player to the *position* value retrieved, and blits the retrieved *board* image to the screen. If not, it notifies the player that there was no saved game to load, as the requested fields returned undefined.

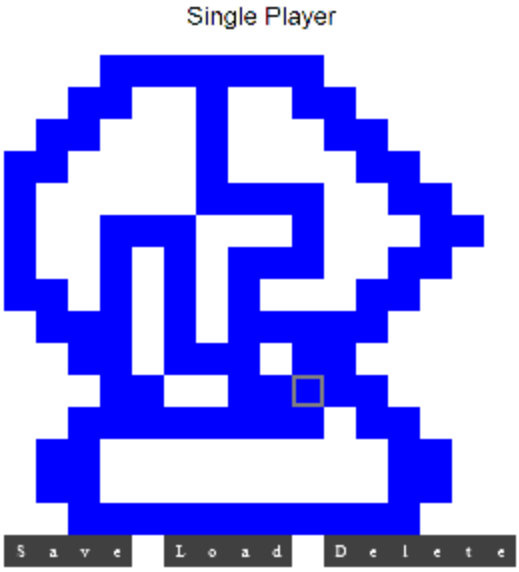

Figure 8: Single player demo before reloading previous save.

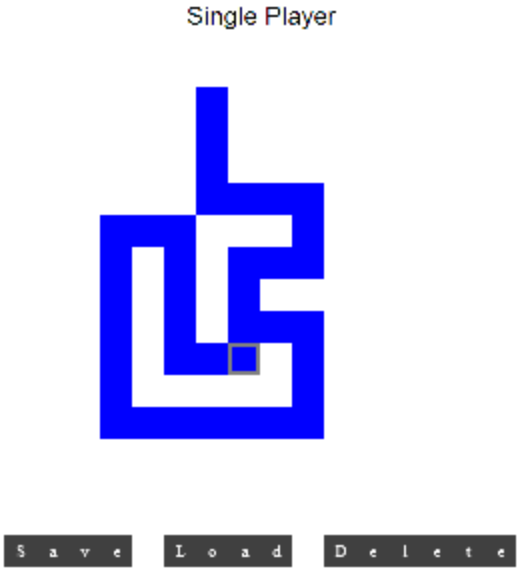

Figure 9: Single player demo after reloading previous save.

Finally, the Delete button executes the following:

```
K.deleteData(
     ['position', 'board'],
     function( response )
     {
          PS.statusText( response.error ?
               "Error Deleting Save" : "Save Deleted");
     }
);
```

K.deleteData call

This call to **K.deleteData()** requests that the server erase the stored game state, if there is any. Note that regardless of whether or not a game had previously been saved, the server returns the same message. It would be possible to check whether there was a save to be deleted first, by checking for its presence in the response to a **K.getData()**, but we do not do so in our demo because it would require additional requests to the server for no additional benefit, beyond being able to display a different status message to differentiate between the two conditions.

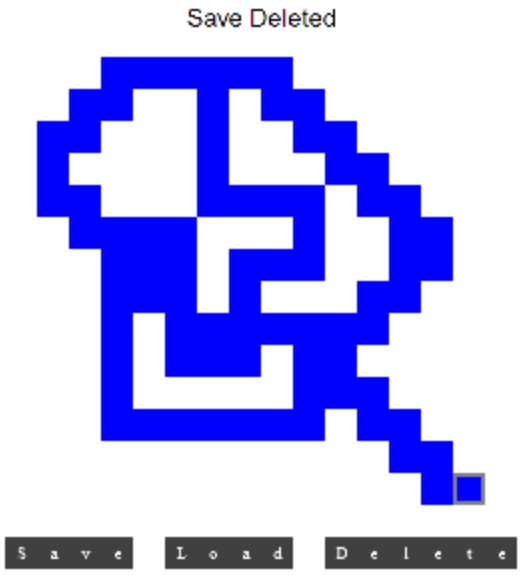

Figure 10: Trying to load a single player save after deleting it.

The true heart of Knecht is its ability to allow multiple users to connect to, access, and manipulate shared data. Upon the player clicking the second button on the menu, Multiplayer, our demo uses **K.getData()** to look for the data field labeled *group* in the player's personal data:

```
K.getData('group', function( result )
{
      if (result.error)
      {
            PS.statusText("Error Finding Group");
      }
      else
      {
            if ( result.data.group === undefined )
            {
                  // prompt to enter group name
      findGroup();
            }
            else // connect to known group
            {
                  K.configure( { group_name : result.data.group } );
                  connectGroup();
            }
      }
});
```

K.getData call for multiplayer usage

If no data is found under that name, then the demo prompts the user to enter the name of the group they wish to join or create, and **K.getGroups()** is used to find if there are any active groups with that name:

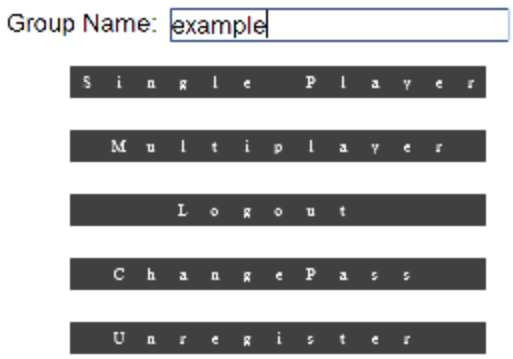

Figure 11: Demo group finding and creating field.

```
K.getGroups (
      {
      group_name : input,
      application : K.configure().application
      },
      function ( result )
      {
            if( result.error )
            {
                  PS.statusText("Error Finding Group");
            }
            else
            {
                  result.groups.length === 0 ?
                        startGroup( input) : joinGroup ( input );
            }
      }
} );
```

K.getGroups call

If there is no such group yet, then the demo calls **K.startGroup()** to create it, with the player as its host. After checking for errors, if then sets the *group* personal field to hold the name of the new group with **K.putData()**, then sets the initial board state with **K.submitUpdate()**, and finally connects to the group just as it would if the player had already been a part of the group.

Note the permission values given to each of the data fields; all are set to false, except for the *board*. As these permissions are set using the host's name, they apply universally. This means that while any user can request and receive the board image for this group, only the hosting player has access to any of the others.

```
K.startGroup( name, function ( result )
{
     if ( result.error )
     {
          PS.statusText("Error Starting Group");
     }
     else
     {
          K.putData('group', name, function ( result )
          {
               if ( result.error )
               {
                    PS.statusText("Error Starting Group");
               }
               else
               {
                    ....
                    K.submitUpdate(
                         ['player1', 'board', 'num_players',
                                   'reject', 'accept'],
                         [player1, PS.imageCapture(), 1, true,
true],
                         function( result )
                        {
                              result.error ?
                                   PS.statusText(
                                        "Error Starting Group") :
                                   connectGroup();
                        },
                        K.configure().username,
                        [false, true, false, false, false]
                    );
               }
          });
     }});
```

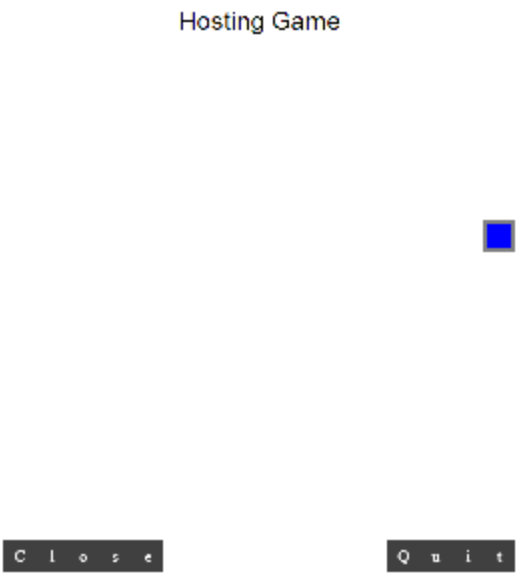

Figure 12: Group demo with no members other than the host.

If instead the named group already exists, but the player is not yet a part of it, the demo uses **K.submitInput()** to send a request to join the group, and **K.putData()** to make a note for itself about it:

```
K.configure( { group_name : name } );
K.submitInput( "JOIN", function( result )
{
     if(result.error)
     {
          PS.statusText("Error Joining Group");
     }
     else
     {
          K.putData('group', name, function ( result )
          {
               result.error ?
                    PS.statusText("Error Joining Group") :
                    connectGroup();
          });
     }
});
```

K.submitInput call

Now that all three branches of our demo's multiplayer program flow have converged on the connectGroup function, we are ready to show the standard procedure for communicating input and application state that makes up the core of our project:

```
K.getGroupData(
     ['player1', 'player2', 'player3', 'player4',
     'board', 'num_players'],
     function ( result )
     {
          if(result.error)
          {
               if( result.error === 'Group Not Found' )
               {
                    K.deleteData('group', function(result)
                    {
                         result.error ?
                              PS.statusText("Group Ended") :
                              initMenu();
                    });
               }
          }
…
```

K.getGroupData call

The first thing a player's client does to connect to the group is to use **K.getGroupData()** to request each of the group's data fields. This function is analogous to the single-user **K.getData()**, with the important distinction that it only returns a value for a field if the requesting user has permission to access that field. In our demo, we use this fact to determine which player slot, if any, our player occupies:

```
var access = false;
var host = false;
if(result.data.player1)
{
     players[result.data.player1.username] =
result.data.player1;
     access = true;
     host = true;
}
if(result.data.player2)
{
     players[result.data.player2.username] =
result.data.player2;
     access = true;
}
if(result.data.player3)
{
     players[result.data.player3.username] =
result.data.player3;
     access = true;
}
if(result.data.player4)
{
     players[result.data.player4.username] =
result.data.player4;
     access = true;
}
...
```

Parsing retrieved data

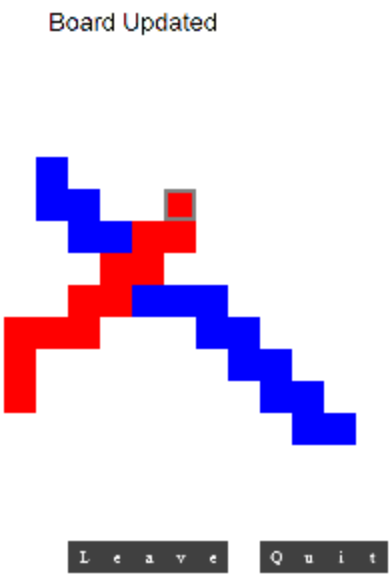

Figure 13: Group demo with two members from the client's perspective.

player1 in the demo is always the host of the group, and any other user who is able to read a player's location, as we will see set in the permissions later, must be that user themself.

If the returned data indicates that a player is host, then the demo executes the function call

```
K.listenInput( processInput );
```

Unlike other functions that send a request to the server, Knecht does not respond to **K.listenInput()** right away. Instead, it holds onto the request until the group being listened to receives at least one input from a user, then responds will all pending inputs in the order they were submitted.

In the *processInput* callback, the host application is able to decide what to do in response to any given input. For example, this block rejects any input other than a join request from users who are not players in the game, and notifies that user by using **K.setPermission()** to give them read access to the *reject* field stored in the group when it was first created.

```
if( !players[user] && result.input[i].input !== "JOIN")
{
     K.setPermission(
          K.configure().group_name,
          'reject',
          true,
          function( result )
          {
               PS.statusText(result.error ?
                    "Input Rejected" : "Error Rejecting
Input");
          },
          user
     );
}
```

K.setPermission call

Whenever a user is granted permission to read a field, an update notification is stored on the database which will be sent to the user the next time they subscribe to the group with **K.listenUpdate()**, which will be explained shortly.

A user who joins properly is given read access on the board and their own position, resulting in that they can see the location of their own ghost and the paint trails left by all players, but the other players' locations are invisible. They are also given permission to the *access* field, which serves as a flag telling the user that their join request has been accepted and that they may proceed to connect as a normal member of the group.

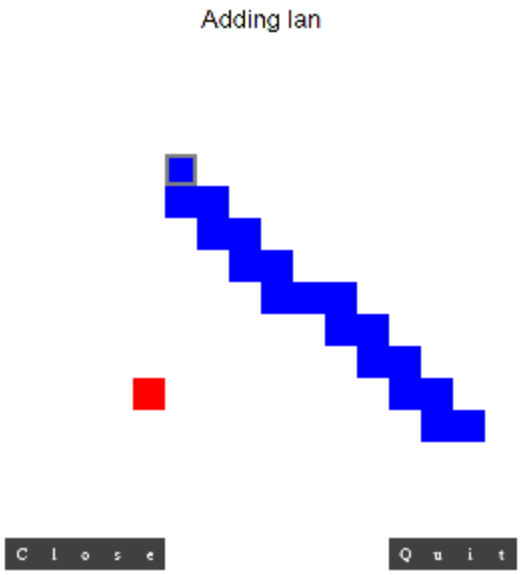

Figure 14: A client joins a group as the host sees it.

```
K.submitUpdate(
     ['player' + players[user].player_num, 'board', 'num_players'],
     [players[user], PS.imageCapture( PS.DEFAULT, { height: 15 } ),
               num_players],
     function( result )
     {
          if (!result.error )
          {
               K.addMember(K.configure().group_name, user,
                    function ( result )
                    {
                         if ( !result.error )
                         {
                              K.setPermission(
                                   K.configure().group_name,
                                   'accept', true,
function( result )
                                   {
                                        PS.statusText(result.error
?                                                 user + " has
Joined" :                                                  "Error
Adding " + user);
                                   } , user);
                         }
                    });
          }
     },
     [user, K.configure().username],
     [ [ true, false, false], [false, true, false] ]
);
```

K.submitUpdate call

The function **K.addMember()** adds a user to the list of users who will receive updates whenever a field with universal permissions is updated. In the case of our demo, this means the *board* field.

Once in the game, each time the player presses a WASD key their coordinates are checked to see if it is a valid move. Unlike in the single-player mode, however, the move is not made immediately. Even if the player's client could see where others were, and thus check to avoid collisions, a modified application could send bad input in an attempt to cheat. Thus, all player action is done through **K.submitInput()**. Example:

```
K.submitInput( "UP", function( result )
{
     if ( result.error )
     {
          PS.statusText( "Error Submitting Input" );
     }
});
```

K.submitInput call

Even the host's moves are submitted this way, with the server echoing them back via the **K.listenInput()** response, both to ensure that the host does not have an unfair advantage from not having to wait for the server to reply, and also ensure that inputs are processed in strictly chronological order.

Each time a move is accepted, the host calls **K.submitUpdate()** to inform all members that the board image has changed. These notifications are received via the server's response to **K.listenUpdate()**, which works on the same principle as **K.listenInput()**. For either of these, notifications are only removed from the server when a request is made including a *clear* parameter, identifying the notifications that the user is acknowledging as seen.

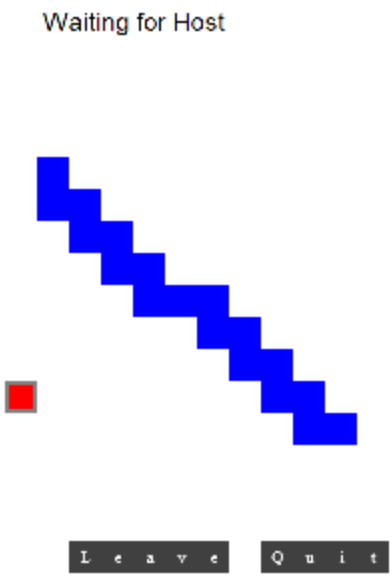

Figure 15: A client joins the group as the client sees it.

If the host is not online when input is submitted, then it will be queued for the next time they call **K.listenInput()**, and no updates to the game state will occur.

A player may Quit a multiplayer session back to the menu at any time by clicking the button labeled as such. If they do so, **K.stopListening()** is called, forcing the server to respond to the player's last listen request immediately without any notifications, which is used in our demo as the signal to stop resubscribing.

A group can be permanently closed by the host, using the **K.closeGroup()** function. In our demo application, the callback to this function also deletes the *group* flag from the host's personal data, allowing them to start or join another group if they wish to do so.

```
K.closeGroup( K.configure().group_name, function()
{
     K.deleteData('group', function ( result )
     {
          confirmStatus( result, "Closing Group", "Error
Closing Group");
          if ( !result.error )
          {
               initMenu();
          }

     }
});
                    K.closeGroup call
```

The flag is also deleted if a group is found to no longer exist when one tries to reconnect to it, or when a player clicks the Leave button in game.

```
K.getData('group', function ( result )
{
     if ( !result.error && result.data.group)
     {
          K.submitInput( result.data.group,
               "LEAVE", function ( result )
               {
                    if ( !result.error )
                    {
                         K.deleteData('group', function ( result )
                         {
                              K.stopListening( function(){});
                         });
                    }
               }
          );
     }
});
        K.getData call that handles the group closing
```

Appendix A: Local Installation Instructions

These Instruction explain how to install Node.js on a developer's Windows machine for testing purposes. It is assumed that the user already has WebStorm 7+ installed on their computer.

To install Node.js

1. go to http://nodejs.org/download/ and click on the appropriate installer to download
2. Run the installer.
    a. IMPORTANT: Pay attention to the location "node.exe" was installed to.

To enable Node.js in WebStorm

1. Open WebStorm
2. Go to File>Settings
3. On the left side of the screen, expand "Javascript" and click on "Node.js"
4. On the line indicated by "Node.js interpreter" Put in the location where "node.exe" is located

To install Modules from WebStorm

1. Open WebStorm
2. Go to File>Settings
3. On the left side of the screen, expand "Javascript" and click on "Node.js"
4. Under the "Packages" area, click "install"
5. Scroll down to "mysql" and select it and install it

To install Modules from the command prompt

1. Open the command prompt window
2. type "npm install mysql" in the command prompt to install the mysql module

Appendix B: Live Installation Instructions

From the Linux or Unix terminal:

Verify that you have a compiler

1. Enter “which gcc”.
2. Something along the lines of “/usr/bin/ggc” should print.

Install Node.js and npm. There are several ways to install them described here:

https://gist.github.com/isaacs/579814

Place a copy of KnechtServer.js in the directory you want to run it from.

To run Knecht on your server, use the terminal command 'nodejs KnechtServer.js' . Note that the command is NOT 'node [JavaScript file]', despite what the documentation says.

To install launchtool from the command prompt

1. Open the command prompt window.
2. Type “sudo apt-get install launchtool”.
3. Enter the superuser’s password to validate the sudo command.

Now the command “launchtool –L –wait-times=1 –tag “nodejs KnechtServer.js” & will launch the Knetch server and relaunch it after a second automatically if it ever crashes.

Appendix C: Knecht Client API

K.configure (options)

K.configure() sets internal client variables and returns their current values.

Parameters:

1. (optional) **options** : object

The **options** parameter contains new values for internal variables:

- (optional) **.server** : string
- (optional) **.application** : string
- (optional) **.username** : string
- (optional) **.password** : string
- (optional) **.timeout** : number
- (optional) **.session** : string
- (optional) **.group_name** : string
- (optional) **.error** : function

  The **.server** property is the http address of the Knecht server that will be used, with port number and trailing slash.

  The **.application** property is the application namespace that will be used.

  The **.username** property is the name of the user account that will be used.

  The **.password** property is the password of the user account that will be used.

  The **.session** property is the login session of the user account that will be used.

  The **.group_name** property is the name of the group that will be used.

  The **.error** property is a callback function that executes when the client receives an error response from the server. It is passed the following arguments:

  1. **function_name** : string
  2. **error** : string
  3. **timestamp** : number
  4. **args** : array of any

The **function_name** argument is the name of the function that received the error response.

The **error** argument is the description provided by the server of what error occurred.

The **timestamp** argument is the time the response was received, in UNIX time.

The **args** argument is an array of the arguments originally passed to the function.

Usage:

Values set with **K.configure()** will be used as the defaults for other client functions if no other value is provided. The default values that these variables begin with are:

- **.server** : 'http://localhost:8080/'
- **.application** : ''''
- **.username** : ''''
- **.password** : ''''
- **.timeout** : 15
- **.session** : ''''
- **.group_name** : ''''
- **error** : null

Note that the following functions may internally change one or more of the above internal variables. These changes are summarized below:

- **K.register()/K.login()** : sets **username** and **password** to the corresponding arguments passed to this function and sets **session** to the login ID string provided by the server in its response.
- **K.unregister()/K.logout()** : sets username, password, and session to ''''.
- **K.changePassword()** : sets **password** to the corresponding argument passed to this function.

K.configure() is the only Knecht client function that makes no requests to the server.

Return Value:

K.configure() returns an object containing the values of each of the client's internal variables after updating the ones that have had new values provided.

K.getUsers (constraints, callback)

This function retrieves a list of users that fit a set of constraints.

Parameters:

1. (optional) **constraints** : object
2. **callback** : function

The **constraints** parameter contains constraints listed users must meet:

- (optional) **.username** : string
- (optional) **.group_name** : string
- (optional) **.application** : string
- (optional) **.online** : boolean

  The **.username** property, if present, requires that a listed user have this exact name.

  The **.group_name** property, if present, requires that a listed user be a member of a group with this exact name.

  The **.application** property, if present, requires that the above group be in this application namespace. It has no effect if no **.group_name** property is set, and defaults to the internal **application** variable set by **K.configure()**.

  The **.online** property, if set to true, requires that a listed user have an unexpired login session.

The **callback** parameter is a function that will be called once the server has responded.

Request:

The request to the server made by this function is **"GET /users"**, with one query variable, **constraints**, whose value is the stringified version of the **constraints** parameter.

Usage:

If no constraints are specified, then this function will retrieve a list of all users registered on the server. No changes are made to data on the server as a result of this function being called.

Response Value:

The object passed to **callback** has the following properties:

- .timestamp : number
- **.error** : string
- **.users** : array of string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The .error member, if present, is the error that occurred processing the request.

The .users member is the list of users meeting the constraints.

K.register (username, password, options, callback)

This function registers a new user account on the server.

Parameters:

1. **username** : string
2. **password** : string
3. (optional) **options** : object
4. **callback** : function

The **username** parameter is what the new account will be named. An error will occur if this username is already in use, or if it exceeds the length of the username column used by the database connected to the server. By default, this limit is 255 characters.

The **password** parameter is the new account's password. An error will occur if this exceeds the length of password column used by the database connected to the server. By default, this limit is 64 characters.

The **options** parameter contains the optional arguments to this function. They include:

1. (optional) **.timeout** : number

   The **.timeout** member is the number of minutes the login session should persist without activity. It defaults to the internal **timeout** variable set by **K.configure()**.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function creates a new entry in the server's users table using the arguments provided. The internal variables **username**, **password**, **timeout**, and **session** are set accordingly.

Request:

The request to the server made by this function is **“POST /users”**, with two query variables, **username** and **timeout**, whose values are the stringified versions of the **username** and **timeout** parameters, respectively. The body of the request is the stringified version of the **password** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.users** : array of string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The **.error** member, if present, is the error that occurred processing the request.

  The **.session** member is the session ID generated by the server and stored in the session internal variable.

K.unregister (callback)

This function removes the currently logged in user from the server.

Parameters:

1. **callback** : function

The **callback** parameter is a function that will be called once the server has responded.

Usage:

In addition to the corresponding entry in the server's users table, all data associated with that user will also be deleted. This includes all personal data entries, hosted groups and their associated entries, group memberships and permissions, and pending notifications.

The internal variables **username**, **password**, **timeout**, and **session** are set to their default values.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"DELETE /users"**, with two query variables, **username** and **session**, whose values are the stringified version of the **username** and **session** internal client variables, respectively.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number

- **.error** : string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The .**error** member, if present, is the error that occurred processing the request.

K.login (username, password, options, callback)

This function generates a new session ID for an existing user.

Parameters:

1. **username** : string
2. **password** : string
3. (optional) **options** : object
4. **callback** : function

The **username** parameter is the name of the account to be logged in. An error will occur if this username is not registered on the server.

The **password** parameter is the password used to authenticate the login. An error will occur if this does not match the password for the named user stored on the server.

The **options** parameter contains the optional arguments to this function. They include:

1. (optional) **.timeout** : number

   The **.timeout** member is the number of minutes the login session should persist without activity. It defaults to the internal **timeout** variable set by **K.configure()**.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function updates the session and timeout fields stored in the user's database entry. The internal variables **username**, **password**, **timeout**, and **session** are set accordingly.

Request:

The request to the server made by this function is **"PUT /users/session"**, with two query variables, **username** and **timeout**, whose values are the stringified version of the **username** internal client variable, and the **timeout** parameter, respectively. The body of the request is the stringified version of the **password** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.users** : array of string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The **.error** member, if present, is the error that occurred processing the request.

  The **.session** member is the session ID generated by the server and stored in the session internal variable.

K.logout (callback)

This function expires and invalidates the current login session.

Parameters:

1. **callback** : function

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function updates the session and last ping fields stored in the user's database entry. The internal variables **username**, **password**, **timeout**, and **session** are set to their default values.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"DELETE /users/session"**, with two query variables, **username** and **session**, whose values are the stringified version of the **username** and **session** internal client variables, respectively.

Response Value:

The object passed to **callback** has the following properties:

- .timestamp : number
- **.error** : string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The .error member, if present, is the error that occurred processing the request.

K.changePassword (password, callback)

This function changes the password of the currently logged in user.

Parameters:

1. **password** : string
2. **callback** : function

The **password** parameter is the account's new password.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function changes the values of the **password** internal variable if successful.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"PUT /users/password"**, with two query variables, **username** and **session**, whose values are the stringified version of the **username** and **session** internal client variables, respectively. The body of the request is the stringified version of the **password** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

  The **.error** member, if present, is the error that occurred processing the request.

K.putData (field, data, options, callback)

This function stores a user's personal data on the server.

Parameters:

1. **field** : string or array of string

2. **data**: any
3. (optional) **options** : object
4. **callback** : function

The **field** parameter contains the name(s) of the data fields to be updated. An error will occur if any field name exceeds the length of the field column used by the database connected to the server. By default, this limit is 64 characters.

The **data** parameter contains the value(s) of the data fields to be updated. If the **field** parameter is an array, an error will occur if this is not also an array of the same length. An error will occur if any data value exceeds the length of the data column used by the database connected to the server. By default, this limit is 16 megabytes.

The **options** parameter contains the optional arguments to this function. They include:

1. (optional) **.application** : string

The **.application** member is the application namespace the data should be stored under. It defaults to the internal **application** variable set by **K.configure()**.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

If a specified data field does not yet exist, it will be created on the server. If it already exists, its value will be changed.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"PUT /users/data"**, with four query variables: **username**, **session**, **application**, and **field**, whose values are respectively the stringified version of the **username**, and **session** internal client variables and the **options.application** parameter and the **field** parameter. The body of the request is the stringified version of the **data** parameter.

Response Value:

The object passed to **callback** has the following properties:

- .timestamp : number
- **.error** : string

The **.timestamp** member is the time the server sent its response at, in UNIX time.

The **.error** member, if present, is the error that occurred processing the request.

K.getData (field, options, callback)

This function retrieves a user's personal data from the server.

Parameters:

1. **field** : string or array of string
2. (optional) **options** : object
3. **callback** : function

The **field** parameter contains the name(s) of the data fields whose values are requested.

The options parameter contains the optional arguments to this function. They include:

1. (optional) **.application** : string

The **.application** member is the application namespace the data should be retrieved from. It defaults to the internal **application** variable set by **K.configure()**.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

If a requested data field does not exist, its value will be reported as undefined. No changes are made to data on the server as a result of this function being called.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"GET /users/data"**, with three query variables: **username**, **session**, and **field**, whose values are respectively the stringified version of the **username** internal client variable and the **options.application** and **field** parameters, respectively.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.data** : object

The **.timestamp** member is the time the server sent its response at, in UNIX time.

The **.error** member, if present, is the error that occurred processing the request.

The **.data** member, if present, contains the values of the requested fields in members of the corresponding names.

K.deleteData (field, options, callback)

This function removes a user's personal data from the server.

Parameters:

1. **field** : string or array of string
2. (optional) **options** : object
3. **callback** : function

The **field** parameter contains the name(s) of the data fields to be deleted.

The **options** parameter contains the optional arguments to this function. They include:

1. (optional) **.application** : string

   The **.application** member is the application namespace the data should be deleted from. It defaults to the internal **application** variable set by **K.configure()**.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

If a requested data field does not exist, it is ignored.

An error will occur if this function is called when the **session** and **username** internal variables are invalid.

Request:

The request to the server made by this function is **"DELETE /users/data"**, with three query variables: **username**, **session**, and **field**, whose values are respectively the stringified version of the **username** internal client variable and the **options.application** and **field** parameters, respectively.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

  The **.timestamp** member is the time the server sent its response at, in UNIX time.

The .**error** member, if present, is the error that occurred processing the request.

K.getGroups (constraints, callback)

This function requests a list of registered groups that fit a set of constraints.

Parameters:

1. (optional) **constraints** : object
2. **callback** : function

The **constraints** parameter is an object with the following properties:

- (optional) **.username** : string
- (optional) **.host** : string
- (optional) **.group_name** : string
- (optional) **.application** : string

The **.username** property, if present, requires that a group have the named acount as a member for it to be included in the retrieved list.

The **.host** property, if present, requires that a group have the named acount as its host for it to be included in the retrieved list.

The **.group_name** property, if present, requires a group's name to exactly match this string to be included.

The **.application** property, if present, requires a group to be running the named application to be included.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

If no constraints are specified, then this function will retrieve a list of all open groups.

Request:

The request to the server made by this function is **"GET /groups"**, with one query variable, **constraints**, whose value is the stringified version of the **constraints parameter**.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number

- **.error** : string
- **.groups** : array of object

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

The **.groups** member is the array of groups returned. It is undefined if an error occurred processing the request. Each element contains the following members:

- **.group_name** : string
- **.application** : string

The **.group_name** member is the name of the matching group.

The **.application** member is the application that group is using.

K.startGroup (group_name, callback)

This function registers a new group with the active application and sets the requesting user to host.

Parameters:

1. **group_name** : string
2. **callback** : function

The **group_name** parameter is the name the new group will be given. A Duplicate Group error will occur if this name is already in use for the current application, and no group will be registered.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

As host, the user's application is responsible for all group maintenance and processing of its data and inputs. Group names are application-specific. An Invalid Session error will occur if the user is not logged in before calling this function.

Request:

The request to the server made by this function is **"POST /groups"**, with four query variables: **username**, **session**, **group_name**, and **application**, whose values are respectively the stringified version of the **username**, and **session** internal client variables , the **group_name** parameter, and the **application** internal client variable.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.closeGroup (group_name, callback)

This function deletes a group and its data from the servers, and notifies subscribed users of its closure.

Parameters:

3. (optional) **group_name** : string
4. **callback** : function

The **group_name** parameter is the name of the group to be closed. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

All pending **K.listenUpdate()** requests for this group are answered with no field array to indicate group closure. Group names are application-specific. An Invalid Session error will occur if the user is not logged in before calling this function. A User Not Host error will occur if the calling user is not the host of the group to be closed.

Request:

The request to the server made by this function is **"DELETE /groups"**, with four query variables: **username**, **session**, **group_name**, and **application**, whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.listenInput (group_name, clear, callback)

This function subscribes to input from users directed at a group.

Parameters:

1. (optional) **group_name** : string
2. (optional) **clear** : array of string
3. **callback** : function

The **group_name** parameter is the name of the group to be listened to. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **clear** parameter contains the IDs of inputs received in the previous response and acknowledged by the client. It defaults to the empty array.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

The server does not respond to this request until there is input to be retrieved. Once a response is received, the application should send a new request with the same **group_name** argument and the **.clear** member of the result object as the **clear** argument. An Invalid Session error will occur if the user is not logged in before calling this function. A User Not Host error will occur if the calling user is not the host of the group to be closed.

Request:

The request to the server made by this function is **"GET /groups/input"**, with five query variables: **username**, **session**, **group_name**, **application**, and **clear,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **clear** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.input** : array of object
- **.clear** : array of string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

The **.input** member is an array of objects, sorted in order of ascending timestamps. Each element has the following members:

- .username : string
- .input : any
- .time : number

The **.username** member is the user who submitted the input.

The **.input** member is the input to be processed. If this member is not present, then the group has ended communication with the user.

The **.time** member is the UNIX timestamp of when the server received the input.

The **.clear** member is an array of IDs for the inputs included in this response.

K.submitInput (group_name, input, callback)

This function submits updates to a group for processing by the host.

Parameters:

1. (optional) **group_name** : string
2. **input** : any
3. **callback** : function

The **group_name** parameter is the name of the group the input is for. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's **group_name** internal variable.

The **input** parameter is the input the user wishes to send to the group.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

The input will be processed the next time the host of the group receives a response to their **K.listenInput()** request. Group names are application-specific. An Invalid Session error will occur if the user is not logged in before calling this function.

Request:

The request to the server made by this function is **"POST /groups/input"**, with four query variables: **username**, **session**, **group_name**, **application**, and **clear,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables. The body of the request is the stringified version of the **input** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.listenUpdate (group_name, limit, clear, callback)

This function subscribes to update messages from a group.

Parameters:

1. (optional) **group_name** : string
2. (optional) **limit** : number
3. (optional) **clear** : array of string
4. **callback** : function

The **group_name** parameter is the name of the group to be listened to. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **limit** parameter is the maximum character length of a data value retrieved by this request. If no argument is provided, then no maximum will be set.

The **clear** parameter contains the ids of inputs received in the previous response and acknowledged by the client.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

The server does not respond to this request until there are updates to be retrieved. Once a response is received, the application should send a new request with the same **group_name** argument and the **.clear** member of the result object as the **clear** argument. An Invalid Session error will occur if the user is not logged in before calling this function. Data values larger than the **limit** must be requested explicitly with **K.getGroupData()**.

Request:

The request to the server made by this function is **"GET /groups/updates"**, with six query variables: **username**, **session**, **group_name**, **application**, **clear,** and **limit,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **clear** and **limit** parameters.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.field** : array of string
- **.data** : object
- **.clear** : array of string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

The **.field** member is an array of strings corresponding to the names of fields that have been updated. If this member is not present, then the group has ended communication with the user.

The **.data** property is an object containing the values of the above fields. A field whose value

is of greater size than the limit will return as undefined. If this member is not present, then the group has ended communication with the user.

The **.clear** member is an array of IDs for the inputs included in this response.

K.submitUpdate (group_name, field, data, callback, member, permission)

This function stores group data on the Knecht server and notifies group members.

Parameters:

1. (optional) **group_name** : string
2. **field** : string or array of string
3. **data** : any
4. **callback** : function
5. (optional) **member** : string or array of string
6. (optional) **permission** : boolean or array of boolean or array of array of boolean

The **group_name** parameter is the name of the group the input is for. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **field** parameter is the field(s) the user wishes to update.

The **data** parameter is the data the user wishes to store in the above fields. If **field** is an array, this will be interpreted as an array of the same length.

The **callback** parameter is a function that will be called once the server has responded.

The **member** parameter is the user(s) who will have their read permissions on the above fields changed. If this value is set to the host of the group, then all users are granted read permission.

The **permission** parameter is the new read permission value(s) for the above fields and members. If this is a single boolean value, then it will apply to all fields and members. If it is an array of booleans, then it must be of equal length to the array of fields and each element will correspond to all members and the corresponding field. If it it a 2-dimensional array of booleans, then each element corresponds to a single field – member combination. If not argument is provided, permission defaults to true.

Usage:

The update will be received the next time the target members of the group receives a response to their **K.listenUpdate()** requests. Group names are application-specific. An Invalid Session error will occur if the user is not logged in before calling this function.

Request:

The request to the server made by this function is **"PUT /groups/data"**, with seven query variables: **username**, **session**, **group_name**, **application**, **field, member,** and **permisssion,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **field, member, and permission** parameters. The body of the request is the stringified version of the **data** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.stopListening (group_name, permission)

This function cancels a subscription to group notifications.

Parameters:

1. (optional) **group_name** : string
2. **callback** : function

The **group_name** parameter is the name of the group to be unsubscribed. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function should be called whenever a user wishes to voluntarily halt communication with a group while there is a pending request from either the **K.listenInput()** or **K.listenUpdate()** functions. If successful, that request will be immediately responded to with an empty body except

for a timestamp. Group names are application-specific. An Invalid Session error will occur if the user is not logged in before calling this function.

Request:

The request to the server made by this function is **"DELETE /groups/subscription"**, with four query variables: **username**, **session**, **group_name**, and **application**, whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.getGroupData (group_name, field, callback)

This function retrieves group data from the Knecht server.

Parameters:

1. (optional) **group_name** : string
2. **field** : string or array of string
3. **callback** : function

The **group_name** parameter is the name of the group the data belongs to. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **field** parameter contains the name(s) of the data fields requested.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

If a specified data field does not exist for the specified group and application, or if the requesting user does not have read permission on a specified field, its value will be retrieved as undefined. Data fields are group- and application-specific. An Invalid Session error will occur if the

user is not logged in prior to calling this function.

Request:

The request to the server made by this function is **"GET /groups/data"**, with five query variables: **username**, **session**, **group_name**, **application**, and **field,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **field** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string
- **.data** : object

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred

The **.data** member is an object whose members are the fields that were requested. Their values are the values of those fields as stored on the Knecht server. This member is only present if no error occurred.

K.setPermission (group_name, field, permission, callback, member)

This function updates user's read permissions for group data stored on the server.

Parameters:

1. **group_name** : string
2. **field** : string or array of string
3. **permission** : boolean or array of boolean or array of array of boolean
4. **callback** : function
5. (optional) **member** : string or array of string

The **group_name** parameter is the name of the group the input is for. A Group Not Found error will occur if no such group exists.

The **field** parameter is the field(s) the user wishes to update.

The **permission** parameter is the new read permission value(s) for the above fields and members. If this is a single boolean value, then it will apply to all fields and members. If it is an array of booleans, then it must be of equal length to the array of fields and each element will correspond to all members and the corresponding field. If it it a 2-dimensional array of booleans, then each element corresponds to a single field – member combination.

The **callback** parameter is a function that will be called once the server has responded.

The **member** parameter is the user(s) who will have their read permissions on the above fields changed. If this value is set to the host of the group, then all users are granted read permission.

Usage:

Update messages will be posted for all users who have been granted permission, or for all members of the group if the host has been granted permission. An Invalid Session error will occur if the user is not logged in prior to calling this function.

Request:

The request to the server made by this function is **"PUT /groups/data/permissions"**, with seven query variables: **username**, **session**, **group_name**, **application**, **field, member,** and **permisssion,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **field, member, and permission** parameters.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.addMember (group_name, member, callback)

This function registers a user as a member of a group.

Parameters:

1. (optional) **group_name** : string

2. **member** : string
3. **callback** : function

The **group_name** parameter is the name of the group the member will be added to. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **member** parameter is name of the user to be added. A User Not Found error will occur if no such user exists. A Duplicate User error will occur if the user is already a group member.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

Members of a group will receive update notifications for fields that have their permission value set to the group's host. An Invalid Session error will occur if the user is not logged in prior to calling this function. A User Not Host error will occur if the calling user is not the host of the group to be closed.

Request:

The request to the server made by this function is **"POST /groups/members"**, with five query variables: **username**, **session**, **group_name**, **application**, and **member,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **member** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp** : number
- **.error** : string

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

K.removeMember (group_name, member, callback)

This function removes a member from a group.

Parameters:

1. (optional) **group_name** : string
2. **member** : string
3. **callback** : function

The **group_name** parameter is the name of the group the member will be removed from. A Group Not Found error will occur if no such group exists. If no argument is provided, this parameter defaults to the last group assigned to the client's group_name internal variable.

The **member** parameter is name of the user to be removed. A User Not Found error will occur if no such user exists. A Duplicate User error will occur if the user is already a group member.

The **callback** parameter is a function that will be called once the server has responded.

Usage:

This function removes a user's read permissions for the group but does not prevent them from using **K.getGroupData()** on that group's public data. An Invalid Session error will occur if the user is not logged in prior to calling this function. A User Not Host error will occur if the calling user is not the host of the group to be closed.

Request:

The request to the server made by this function is **"DELETE /groups/members"**, with five query variables: **username**, **session**, **group_name**, **application**, and **member,** whose values are respectively the stringified version of the **username**, **session, group_name, and application** internal client variables and the **member** parameter.

Response Value:

The object passed to **callback** has the following properties:

- **.timestamp : number**
- **.error : string**

The **.timestamp** member is the time the response was received, in Unix time.

The **.error** member is the description provided by the server of what error occurred. It is undefined if no error occurred.

Appendix D: Demo Source Code

```
/*
 Perlenspiel is a scheme by Professor Moriarty (bmoriarty@wpi.edu).
 Perlenspiel is Copyright © 2009-14 Worcester Polytechnic Institute.
 This file is part of Perlenspiel.

 Perlenspiel is free software: you can redistribute it and/or modify
 it under the terms of the GNU Lesser General Public License as published
 by the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 Perlenspiel is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU Lesser General Public License for more details.

 You may have received a copy of the GNU Lesser General Public License
 along with Perlenspiel. If not, see <http://www.gnu.org/licenses/>.
 */

PS.init = function( system, options ) {
   "use strict";

   PS.gridSize( 16, 16 );

   PS.glyphColor(PS.ALL, PS.ALL, PS.COLOR_WHITE);

   K.configure (
      {
```

```
        //server: 'http://perlenspiel.cs.wpi.edu:8080/',
        application: 'Knecht Demo',
        error: function ( function_name, error )
        {
           PS.debug( error + " in " + function_name + '\n');
        }
     }
  );

  initLogin();
};

PS.touch = function( x, y, data, options ) {
  "use strict";

  switch(game_state)
  {
     case "login":
        switch(y)
        {
           case 0:
              if (PS.color(x, y) === PS.COLOR_GRAY_DARK)
              {
                 K.login(K.configure().username, K.configure().password, initMenu);
              }
              break;
           case 2:
              if (PS.color(x, y) === PS.COLOR_GRAY_DARK)
              {
                 K.register(K.configure().username, K.configure().password, initMenu);
```

```
            }
            break;
        case 4:
            if (PS.color(x, y) === PS.COLOR_GRAY_DARK)
            {
                promptUsername();
            }
            break;
    }
    break;
case "pending":
    if(PS.color(x, y) === PS.COLOR_GRAY_DARK)
    {
        switch(y)
        {
            case 0:
                leaveGroup (initSingle );
                break;
            case 2:
                leaveGroup ( initMulti );
                break;
            case 4:
                K.logout( initLogin );
                break;
            case 6:
                PS.statusInput("New Password:", function (new_pass)
                {
                    K.changePassword(new_pass, confirmChangePassword);
                });
                break;
```

```
            case 8:
               leaveGroup( function()
               {
                  K.unregister( initLogin );
               });
               break;
         }
      }
      break;
   case "menu":
      if(PS.color(x, y) === PS.COLOR_GRAY_DARK)
      {
         switch(y)
         {
            case 0:
               initSingle();
               break;
            case 2:
               initMulti();
               break;
            case 4:
               K.logout( initLogin );
               break;
            case 6:
               PS.statusInput("New Password:", function (new_pass)
               {
                  K.changePassword(new_pass, confirmChangePassword);
               });
               break;
            case 8:
```

```
            K.unregister( initLogin );
            break;
        }
    }
    break;
case "single":
    if(y === 15)
    {
        if( x >= 0 && x < 4)
        {
            K.putData(
                ['position', 'board'],
                [ players[K.configure().username ].position, PS.imageCapture() ],
                function( response )
                {
                    PS.statusText(response.error ? "Error Saving Game" : "Game Saved");
                }
            );
        }
        else if( x >= 5 && x < 9)
        {
            K.getData(['position', 'board'],
                function( response )
                {
                    if (response.error)
                    {
                        PS.statusText("Error Loading Save");
                    }
                    else
                    {
```

```
                if(response.data.board)
                {
                    PS.statusText("Save Loaded");
                    var name = K.configure().username;
                    PS.border(players[ name ].position.x, players[ name ].position.y, 0);
                    players[ name ].position = response.data.position;
                    PS.imageBlit(response.data.board, 0, 0);
                    PS.border(players[ name ].position.x, players[ name ].position.y, 4);
                }
                else
                {
                    PS.statusText("No Save To Load");
                }
            }
        }
    );
  }
  else if( x >= 10)
  {
    K.deleteData(
      ['position', 'board'],
      function( response )
      {
        PS.statusText(response.error ? "Error Deleting Save" : "Save Deleted");
      }
    );
  }
 }
 break;
case "multi":
```

```
        if(y === 15 && PS.color(x, y) === PS.COLOR_GRAY_DARK)
        {
           if( x > 0 && x < 6)
           {
              K.closeGroup( closeGroup );
           }
           else if( x > 6 && x < 11)
           {
              leaveGroup( initMenu );
           }
           else if( x > 11 && x < 16)
           {
              K.stopListening( initMenu );
           }
        }
        break;
   }
};

PS.release = function( x, y, data, options ) {
   "use strict";
};

PS.enter = function( x, y, data, options ) {
   "use strict";
};

PS.exit = function( x, y, data, options ) {
   "use strict";
};
```

```
PS.exitGrid = function( options ) {
   "use strict";
};

PS.keyDown = function( key, shift, ctrl, options ) {
   "use strict";

   switch( game_state )
   {
      case "single":
         var name = K.configure().username;
         switch(key)
         {
            case 119: //w
               movePlayer(name, 0, -1);
               break;
            case 97: //a
               movePlayer(name, -1, 0);
               break;
            case 115: //s
               movePlayer(name, 0, 1);
               break;
            case 100: //d
               movePlayer(name, 1, 0);
               break;
            case PS.KEY_ESCAPE:
               initMenu();
               break;
         }
```

```
    break;
case "multi":
    switch(key)
    {
        case 119: //w
            if(players[K.configure().username].position.y > 0)
            {
                K.submitInput( "UP", function( result )
                {
                    if ( result.error )
                    {
                        PS.statusText( "Error Submitting Input" );
                    }
                });
            }
            break;
        case 97: //a
            if(players[K.configure().username].position.x > 0)
            {
                K.submitInput( "LEFT", function( result )
                {
                    if ( result.error )
                    {
                        PS.statusText( "Error Submitting Input" );
                    }
                } );
            }
            break;
        case 115: //s
            if(players[K.configure().username].position.y < 14)
```

```
            {
               K.submitInput( "DOWN", function( result )
               {
                  if ( result.error )
                  {
                     PS.statusText( "Error Submitting Input" );
                  }
               } );
            }
            break;
         case 100: //d
            if(players[K.configure().username].position.x < 15)
            {
               K.submitInput( "RIGHT", function( result )
               {
                  if ( result.error )
                  {
                     PS.statusText( "Error Submitting Input" );
                  }
               } );
            }
            break;
      }
   }
};

PS.keyUp = function( key, shift, ctrl, options ) {
   "use strict";
};
```

```
PS.input = function( sensors, options ) {
   "use strict";
};

var game_state;

var num_players = 1;
var colors = [ PS.COLOR_BLUE, PS.COLOR_RED, PS.COLOR_YELLOW, PS.COLOR_GREEN];
var players = {};

function clear_screen()
{
   PS.color(PS.ALL, PS.ALL, PS.COLOR_WHITE);
   PS.border(PS.ALL, PS.ALL, 0);
   PS.glyph(PS.ALL, PS.ALL, 0);
}

function confirmStatus ( result, success, error )
{
   if( result.error )
   {
      PS.statusText( error );
      return;
   }
   PS.statusText( success );
}

function initLogin()
{
   PS.statusText("Knecht Demo");
```

```
    game_state = "login";

    clear_screen();

    PS.glyph(5, 0, 'L');
    PS.glyph(6, 0, 'o');
    PS.glyph(7, 0, 'g');
    PS.glyph(8, 0, 'i');
    PS.glyph(9, 0, 'n');

    PS.glyph(4, 2, 'R');
    PS.glyph(5, 2, 'e');
    PS.glyph(6, 2, 'g');
    PS.glyph(7, 2, 'i');
    PS.glyph(8, 2, 's');
    PS.glyph(9, 2, 't');
    PS.glyph(10, 2, 'e');
    PS.glyph(11, 2, 'r');

    PS.glyph(5, 4, 'C');
    PS.glyph(6, 4, 'a');
    PS.glyph(7, 4, 'n');
    PS.glyph(8, 4, 'c');
    PS.glyph(9, 4, 'e');
    PS.glyph(10, 4, 'l');

    promptUsername();
}
```

```
function initMenu( result )
{
   if(result && result.error)
   {
      initLogin();
      return;
   }

   PS.statusText("Welcome, " + K.configure().username);

   game_state = "menu";

   clear_screen();

   PS.applyRect(2, 0, 13, 1, PS.color, PS.COLOR_GRAY_DARK);
   PS.glyph(2, 0, 'S');
   PS.glyph(3, 0, 'i');
   PS.glyph(4, 0, 'n');
   PS.glyph(5, 0, 'g');
   PS.glyph(6, 0, 'l');
   PS.glyph(7, 0, 'e');
   PS.glyph(8, 0, ' ');
   PS.glyph(9, 0, 'P');
   PS.glyph(10, 0, 'l');
   PS.glyph(11, 0, 'a');
   PS.glyph(12, 0, 'y');
   PS.glyph(13, 0, 'e');
   PS.glyph(14, 0, 'r');

   PS.applyRect(2, 2, 13, 1, PS.color, PS.COLOR_GRAY_DARK);
```

```
PS.glyph(3, 2, 'M');
PS.glyph(4, 2, 'u');
PS.glyph(5, 2, 'l');
PS.glyph(6, 2, 't');
PS.glyph(7, 2, 'i');
PS.glyph(8, 2, 'p');
PS.glyph(9, 2, 'l');
PS.glyph(10, 2, 'a');
PS.glyph(11, 2, 'y');
PS.glyph(12, 2, 'e');
PS.glyph(13, 2, 'r');

PS.applyRect(2, 4, 13, 1, PS.color, PS.COLOR_GRAY_DARK);
PS.glyph(5, 4, 'L');
PS.glyph(6, 4, 'o');
PS.glyph(7, 4, 'g');
PS.glyph(8, 4, 'o');
PS.glyph(9, 4, 'u');
PS.glyph(10, 4, 't');

PS.applyRect(2, 6, 13, 1, PS.color, PS.COLOR_GRAY_DARK);
PS.glyph(3, 6, 'C');
PS.glyph(4, 6, 'h');
PS.glyph(5, 6, 'a');
PS.glyph(6, 6, 'n');
PS.glyph(7, 6, 'g');
PS.glyph(8, 6, 'e');
PS.glyph(9, 6, 'P');
PS.glyph(10, 6, 'a');
PS.glyph(11, 6, 's');
```

```
    PS.glyph(12, 6, 's');

    PS.applyRect(2, 8, 13, 1, PS.color, PS.COLOR_GRAY_DARK);
    PS.glyph(3, 8, 'U');
    PS.glyph(4, 8, 'n');
    PS.glyph(5, 8, 'r');
    PS.glyph(6, 8, 'e');
    PS.glyph(7, 8, 'g');
    PS.glyph(8, 8, 'i');
    PS.glyph(9, 8, 's');
    PS.glyph(10, 8, 't');
    PS.glyph(11, 8, 'e');
    PS.glyph(12, 8, 'r');
}

function initSingle()
{
    PS.statusText("Single Player");

    game_state = "single";

    clear_screen();

    PS.applyRect(0, 15, 4, 1, PS.color, PS.COLOR_GRAY_DARK);
    PS.glyph(0, 15, 'S');
    PS.glyph(1, 15, 'a');
    PS.glyph(2, 15, 'v');
    PS.glyph(3, 15, 'e');

    PS.applyRect(5, 15, 4, 1, PS.color, PS.COLOR_GRAY_DARK);
```

```
    PS.glyph(5, 15, 'L');
    PS.glyph(6, 15, 'o');
    PS.glyph(7, 15, 'a');
    PS.glyph(8, 15, 'd');

    PS.applyRect(10, 15, 6, 1, PS.color, PS.COLOR_GRAY_DARK);
    PS.glyph(10, 15, 'D');
    PS.glyph(11, 15, 'e');
    PS.glyph(12, 15, 'l');
    PS.glyph(13, 15, 'e');
    PS.glyph(14, 15, 't');
    PS.glyph(15, 15, 'e');

    players[ K.configure().username ] =
    {
        position : { x: Math.floor(Math.random() * 16), y : Math.floor(Math.random() * 15)},
        color : colors[0]
    }
    PS.color(players[ K.configure().username ].position.x,
                players[ K.configure().username ].position.y, colors[0]);
    PS.border(players[ K.configure().username ].position.x,
                  players[ K.configure().username ].position.y, 4);
}

function initMulti()
{
    K.getData('group', function( result )
    {
        if(result.error)
        {
```

```
            PS.statusText("Error Finding Group");
        }
        else
        {
            if ( result.data.group === undefined )
            {//prompt to enter group name
                findGroup();

            }
            else
            {//connect to known group
                K.configure( { group_name : result.data.group } );
                connectGroup()
            }
        }
    });
}

function promptUsername()
{
    K.configure( {username : null, password : null } );
    PS.applyRect(4, 0, 8, 1, PS.color, PS.COLOR_WHITE);
    PS.applyRect(4, 2, 8, 1, PS.color, PS.COLOR_WHITE);
    PS.applyRect(4, 4, 8, 1, PS.color, PS.COLOR_WHITE);
    PS.statusInput("Login - Username:", function( input )
    {
        K.configure( {username : input} );
        K.getUsers( {username : input }, promptPassword );
    });
}
```

```
function promptPassword ( result )
{
   if ( result.error )
   {
      PS.statusText("Error Checking Username");
      return;
   }
   PS.applyRect(4, (result.users.length !== 0) ? 0 : 2, 8, 1, PS.color, PS.COLOR_GRAY_LIGHT);
   PS.applyRect(4, 4, 8, 1, PS.color, PS.COLOR_GRAY_DARK);
   PS.statusInput("Login - Password:", function( input )
   {
      K.configure( {password : input } );
      PS.applyRect(4, (PS.color(4, 0) === PS.COLOR_GRAY_LIGHT) ? 0 : 2,
                        8, 1, PS.color, PS.COLOR_GRAY_DARK);
      PS.statusText("Confirm - " + K.configure().username );
   });
}

function confirmChangePassword (result)
{
   confirmStatus( result, "Password Changed", "Error Changing Password");
}

function movePlayer ( name, dx, dy )
{
   var x = players[name].position.x + dx;
   var y = players[name].position.y + dy;
   if( x >= 0 && x <= 15 && y >= 0 && y <= 14 && !PS.data(x, y))
   {
```

```
        PS.data(x - dx, y - dy, 0);
        PS.data(x, y, 1);
        PS.color(x, y, players[name].color);
        if(name === K.configure().username)
        {
           PS.border(players[K.configure().username].position.x,
                        players[K.configure().username].position.y, 0);
           PS.border(x, y, 4);
        }
        players[name].position = {x: x, y: y};
        if( game_state === 'multi')
        {
           K.submitUpdate(
              ['player' + players[name].player_num, 'board'],
              [players[name], PS.imageCapture( PS.DEFAULT, { height: 15 } )],
              function( result )
              {
                 confirmStatus( result,
                                 "Update Submitted at " + result.timestamp, "Error Submitting Update");
              },
              [name, K.configure().username],
              [ [ name !== K.configure().username, false ], [false, true] ]
           );
        }
        else
        {
           PS.statusText("Single Player");
        }
     }
}
```

```
function findGroup ()
{
   PS.statusInput("Group Name: ", function( input )
   {
      K.getGroups( { group_name: input, application : K.configure().application }, function ( result )
      {
         if( result.error)
         {
            PS.statusText("Error Finding Group");
         }
         else
         {
            if( result.groups.length === 0 )
            {
               startGroup( input );
            }
            else
            {//group exists, join it
               joinGroup( input );
            }
         }
      });
   });
}

function startGroup( name )
{
   K.startGroup( name, function ( result )
   {
```

```
if ( result.error )
{
   PS.statusText("Error Starting Group");
}
else
{
   K.putData('group', name, function ( result )
   {
      if ( result.error )
      {
         PS.statusText("Error Starting Group");
      }
      else
      {
         var player1 =
         {
            username : K.configure().username,
            player_num : 1,
            position : { x: Math.floor(Math.random() * 16), y : Math.floor(Math.random() * 15)},
            color : colors[0]
         };
         clear_screen();
         PS.color(player1.position.x, player1.position.y, player1.color);
         K.submitUpdate(
            ['player1', 'board', 'num_players', 'reject', 'accept'],
            [player1, PS.imageCapture(), 1, true, true],
            function( result )
            {
               result.error ? PS.statusText("Error Starting Group") : connectGroup();
            },
```

```
                K.configure().username,
                [false, true, false, false, false]
            );
        }
    });
  }
});
}

function joinGroup ( name )
{
  K.configure( { group_name : name } );
  K.submitInput( "JOIN", function( result )
  {
    confirmStatus( result, "Joining Group", "Error Joining Group");
    K.putData('group', name, function ( result )
    {
      confirmStatus( result, "Joining Group", "Error Joining Group");
      if(!result.error)
      {
        connectGroup();
      }
    });
  });
}

function closeGroup ( result )
{
  confirmStatus( result, "Closing Group", "Error Closing Group");
  if ( !result.error )
```

```
    {
        K.deleteData('group', function ( result )
        {
            confirmStatus( result, "Closing Group", "Error Closing Group");
            if ( !result.error )
            {
                initMenu();
            }

        });
    }
}

function connectGroup( clear )
{
    K.getGroupData(['player1', 'player2', 'player3', 'player4', 'board', 'num_players'], function ( result )
    {
        if(result.error)
        {
            if( result.error === 'Group Not Found' )
            {
                K.deleteData('group', function(result)
                {
                    result.error ? PS.statusText("Group Ended") : initMenu();
                });
            }
        }
        else
        {
            var access = false;
```

```
var host = false;
if(result.data.player1)
{
   players[result.data.player1.username] = result.data.player1;
   access = true;
   host = true;
}
if(result.data.player2)
{
   players[result.data.player2.username] = result.data.player2;
   access = true;
}
if(result.data.player3)
{
   players[result.data.player3.username] = result.data.player3;
   access = true;
}
if(result.data.player4)
{
   players[result.data.player4.username] = result.data.player4;
   access = true;
}
if(access)
{
   clear_screen();
   PS.imageBlit(result.data.board, 0, 0);
   PS.border(players[ K.configure().username ].position.x,
                players[ K.configure().username ].position.y, 4);
   num_players = result.data.num_players;
```

```
if(host)
{
   PS.applyRect(0, 15, 5, 1, PS.color, PS.COLOR_GRAY_DARK);
   PS.glyph(0, 15, 'C');
   PS.glyph(1, 15, 'l');
   PS.glyph(2, 15, 'o');
   PS.glyph(3, 15, 's');
   PS.glyph(4, 15, 'e');

   PS.statusText("Hosting Game");
   game_state = 'multi';
   K.listenInput( processInput );
}
else
{
   PS.applyRect(6, 15, 5, 1, PS.color, PS.COLOR_GRAY_DARK);
   PS.glyph(6, 15, 'L');
   PS.glyph(7, 15, 'e');
   PS.glyph(8, 15, 'a');
   PS.glyph(9, 15, 'v');
   PS.glyph(10, 15, 'e');

   PS.statusText("Waiting for Host");
   game_state = 'multi';
   clear ? K.listenUpdate( clear, processUpdate ) : K.listenUpdate( processUpdate );
}
PS.applyRect(12, 15, 4, 1, PS.color, PS.COLOR_GRAY_DARK);
PS.glyph(12, 15, 'Q');
PS.glyph(13, 15, 'u');
PS.glyph(14, 15, 'i');
```

```
            PS.glyph(15, 15, 't');
        }
        else
        {
            PS.statusText("Waiting for Host");
            game_state = 'pending';
            clear ? K.listenUpdate( clear, processUpdate ) : K.listenUpdate( processUpdate );
        }
    }
  });
}

function joinMember( user )
{
  if ( num_players < 5 && !players[ user ] )
  {
    players[ user ]= {
        username : user,
        player_num : num_players + 1,
        position:
        {
            x: Math.floor(Math.random() * 16),
            y : Math.floor(Math.random() * 15)
        },
        color: colors[num_players++]
    };

    PS.data(players[user].position.x,players[user].position.y, 1);
    PS.color(players[user].position.x,players[user].position.y,players[user].color);
```

```
   K.submitUpdate(
      ['player' + players[user].player_num, 'board', 'num_players'],
      [players[user], PS.imageCapture( PS.DEFAULT, { height: 15 } ), num_players],
      function( result )
      {
         confirmStatus( result, "Adding " + user, "Error Adding " + user);
         if (!result.error )
         {
            K.addMember(K.configure().group_name, user, function ( result )
            {
               confirmStatus( result, "Adding " + user, "Error Adding " + user);
               if ( !result.error )
               {
                  K.setPermission(K.configure().group_name, 'accept', true, function( result )
                  {
                     if ( result.error )
                     {
                        confirmStatus( result, user + " has Joined", "Error Adding " + user);
                     }
                  } , user);
               }
            });
         }

      },
      [user, K.configure().username],
      [ [ true, false, false], [false, true, false] ]
   );
}
else
```

```
    {
        K.setPermission(
            K.configure().group_name,
            'reject',
            true,
            function( result )
            {
                confirmStatus ( result, "Input Rejected", "Error Rejecting Input");
            },
            [user, K.configure().username],
            [true, false]
        );
    }
}

function kickMember( user )
{
    console.log(user);
    K.removeMember(K.configure().group_name, user, function ( result )
    {
        confirmStatus( result, "Removing " + user, "Error Removing " + user);
        if(!result.error)
        {
            K.submitUpdate(
                ["player" + players[user].player_num, 'num_players'],
                [null, num_players - 1],
                function( result)
                {
                    if(!result.error)
                    {
```

```
                PS.data(players[user].position.x, players[user].position.y, 0);
                players[user] = null;
                num_players -= 1;
            }
        }
    );
  }
});
}

function leaveGroup( callback )
{
  K.getData('group', function ( result )
  {
    confirmStatus( result, "Leaving Group", "Error Leaving Group");
    if ( !result.error && result.data.group)
    {
      K.submitInput( result.data.group, "LEAVE", function ( result )
      {
        confirmStatus( result, "Leaving Group", "Error Leaving Group");
        if ( !result.error )
        {
          K.deleteData('group', function ( result )
          {
            confirmStatus( result, "Group Left", "Error Leaving Group");
            callback();
          });
        }

      });
```

```
        }
    });
}

function processInput ( result )
{
    if( game_state !== 'multi')
    {
        return;
    }
    confirmStatus( result, "Processing Input", "Error Processing Input");
    if ( result.error )
    {
        initMenu();
        return;
    }
    for (var i = 0; i < result.input.length; i++)
    {
        var user = result.input[i].username;
        if( !players[user] && result.input[i].input !== "JOIN")
        {
            K.setPermission(
                K.configure().group_name,
                'reject',
                true,
                function( result )
                {
                    PS.statusText(result.error ?  "Input Rejected" : "Error Rejecting Input");
                },
                user
```

```
        );
    }
    else
    {
        switch( result.input[i].input )
        {
            case "JOIN":
                joinMember( user );
                break;
            case "LEAVE":
                kickMember( user );
                break;
            case "UP":
                movePlayer(user, 0, -1);
                break;
            case "LEFT":
                movePlayer(user, -1, 0);
                break;
            case "DOWN":
                movePlayer(user, 0, 1);
                break;
            case "RIGHT":
                movePlayer(user, 1, 0);
                break;
        }
    }
}
if ( result.input )
{
    K.listenInput( result.clear, processInput );
```

```
    }
}

function processUpdate ( result )
{
    if( game_state !== 'multi' && game_state !== 'pending')
    {
        return;
    }
    if (result.error || !result.field )
    {
        PS.statusText("Disconnected from Group");
        initMenu ();
        return;
    }
    if( result.data.board)
    {
        PS.statusText("Board Updated");
        PS.imageBlit(result.data.board, 0, 0);
    }
    for ( var i = 0; i < result.field.length; i++ )
    {
        switch(result.field[i])
        {
            case 'reject':
                PS.statusText("Input Rejected");
                leaveGroup( initMenu );
                return;
            case 'accept':
                PS.statusText("Connected to Group");
```

```
            connectGroup ( result.clear );
            return;
        case 'player1':
            PS.border(PS.ALL, PS.ALL, 0);
            PS.border(result.data.player1.position.x, result.data.player1.position.y, 4);
            players[result.data.player1.username] = result.data.player1;
            break;
        case 'player2':
            PS.border(PS.ALL, PS.ALL, 0);
            PS.border(result.data.player2.position.x, result.data.player2.position.y, 4)
            players[result.data.player2.username] = result.data.player2;
            break;
        case 'player3':
            PS.border(PS.ALL, PS.ALL, 0);
            PS.border(result.data.player3.position.x, result.data.player3.position.y, 4);
            players[result.data.player3.username] = result.data.player3;
            break;
        case 'player4':
            PS.border(PS.ALL, PS.ALL, 0);
            PS.border(result.data.player4.position.x, result.data.player4.position.y, 4);
            players[result.data.player4.username] = result.data.player4;
            break;
      }
    }
    K.listenUpdate( result.clear, processUpdate );
}
```

Appendix E: Reference Bibliography

- "JavaScript Tutorial." (n.d.). *JavaScript Tutorial*. Retrieved April 30, 2014, from http://www.w3schools.com/js/default.asp
    - Documentation for essential JavaScript methods.
- Richardson, L., & Amundsen, M. (). *RESTful Web APIs*. : .
    - Documentation for proper structure of a RESTful system.
- node.js. (n.d.). *node.js*. Retrieved April 30, 2014, from http://nodejs.org/
    - Documentation for Node.js and essential http module.
- MySQL 5.0 Reference Manual. (n.d.). *MySQL ::*. Retrieved April 30, 2014, from http://dev.mysql.com/doc/refman/5.0/en/
    - Documentation for MySQL database engine used in Knecht.
- Geisendörfer, F. (2012, January 1). node-mysql. *GitHub*. Retrieved April 30, 2014, from https://github.com/felixge/node-mysql
    - Documentation for Node.js module used in Knecht.
- Schlueter, I. (2010, October 3). isaacs/node-and-npm-in-30-seconds.sh. *GitHub Gists*. Retrieved April 30, 2014, from https://gist.github.com/isaacs/579814
    - Documentation for Node.js installation instructions.
- Zini, E. (n.d.). Launchtool release page. *launchtool release page*. Retrieved April 30, 2014, from http://people.debian.org/~enrico/launchtool.html
    - Documentation for Launchtool, used to restart the server automatically.