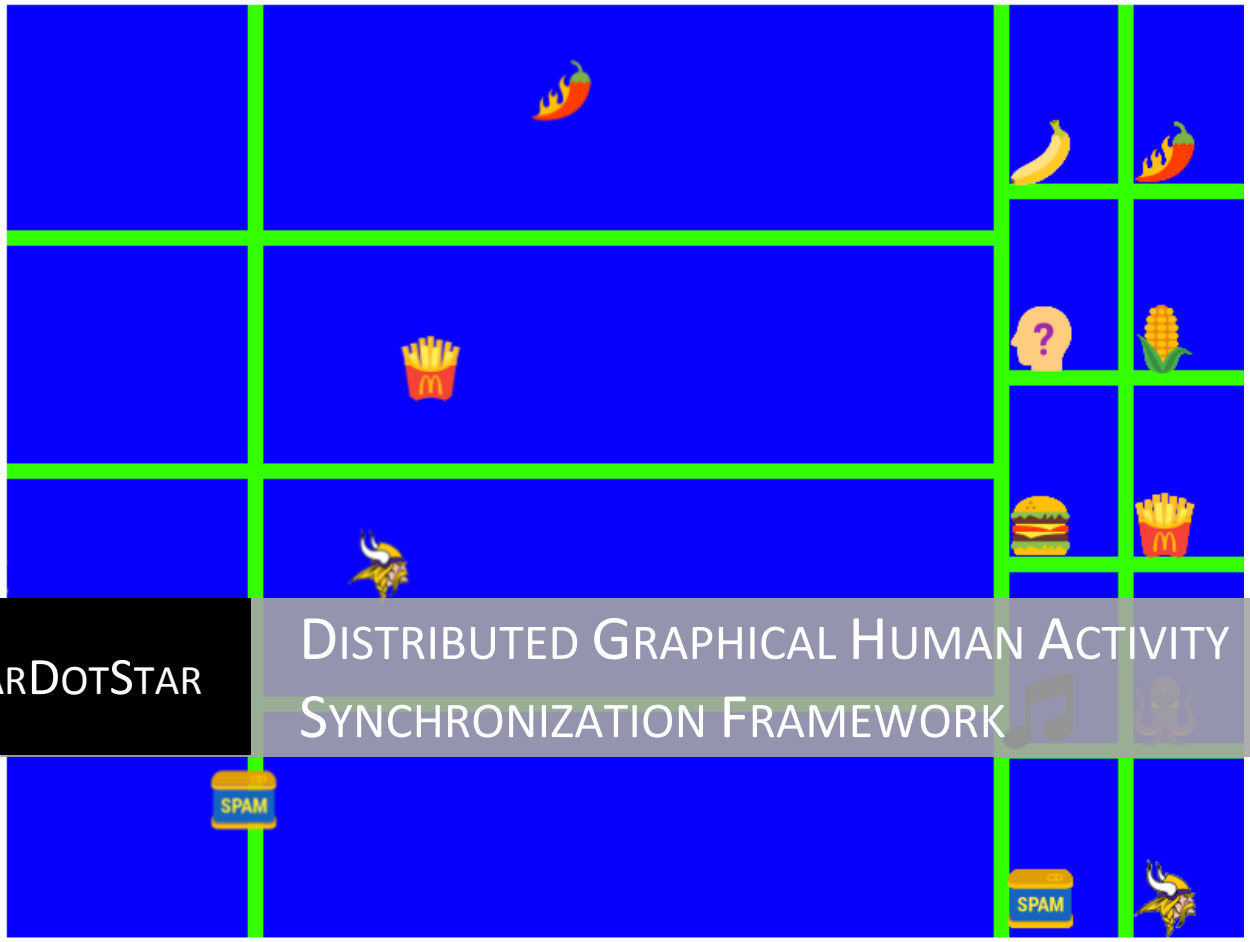
9/1/2018



StarDotStar

Distributed Graphical Human Activity Synchronization Framework

System Design Document | by: Steve Kenny

Executive Summary

A distributed graphical framework for synchronizing human activity

The idea is that geographically separated individuals or groups can be easily directed to perform pre-arranged activities that are designated by specific graphical icons. Each group or individual participant has access to a browser-based user interface which shows the icons appearing and then drifting right to left as they approach a vertical 'now' line on the interface. The activity pre-assigned to the icon is to be performed by the participants when an icon crosses the 'now' line. Slight variations in the URL when invoking the interface allow for viewing single or multiple 'parts', or for enabling features for the placement of icons onto the cooperative surface via drag-and-drop. All participants see a live, moving portrayal of the icons being placed and their progression towards the 'now' line. The system works with any known browser on any known browser-enabled device connected to the internet. A central server in the Amazon EC2 cloud manages the distribution of events and icons for each running 'performance' of the system.

Major Features

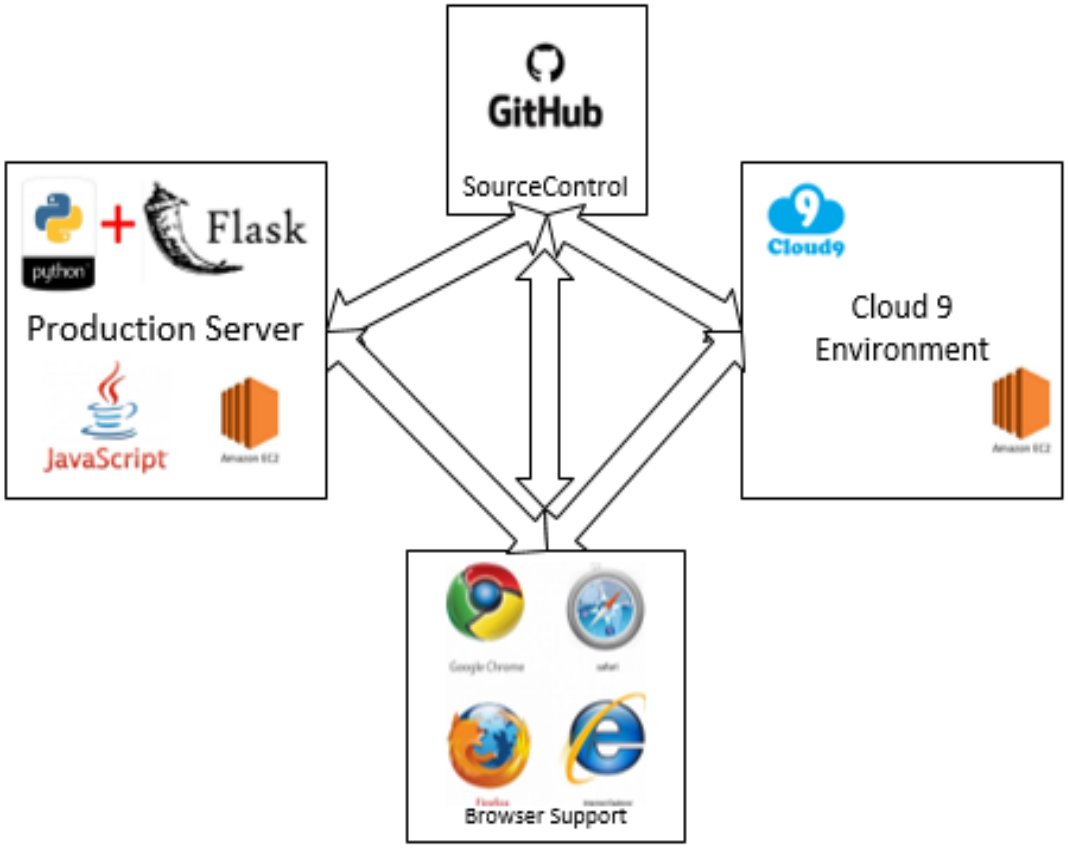
- ***The Client runs in Composer or Performer mode***.
    a. Drag-and-drop interface for Composer nodes.
    b. Performer nodes can show one part or all parts

- ***The System is configurable***
    a. Number of parts
    b. Lambda
    c. Set of icons

- ***The system is applicable to a range of targeted uses***

- ***No limits on geographical separation of participating nodes, or on number of nodes***

1. Technologies used for system and development.



The System was developed entirely within the Amazon AWS Cloud9 (C9) environment and the production server is deployed using GitHub onto an Amazon AWS EC2 compute instance. The Server would also work on almost any Ubuntu platform including the Raspberry Pi or as a Docker instance, although the Pi and Docker versions haven't been fully tested or deployed (yet). The server is a web service endpoint implemented with Python / Flask. The Clients are browser based and run on any platform and any browser that it has been invoked on so far. The system configuration is a simple Json description of a few parameters and a mapping from specific server-resident .png files to the internal friendly name used by the server and client internals. The Clients are implemented as Phaser 2.x JavaScript games that use the RESTful endpoints to give and receive global information about the configuration and the running performance. The server also serves the Phaser JavaScript code to the clients as static html page content. These clients then use an HTML script tag to load the Phaser 2.x infrastructure as a single file.

Here is a short list of the major technology aspects of the system:

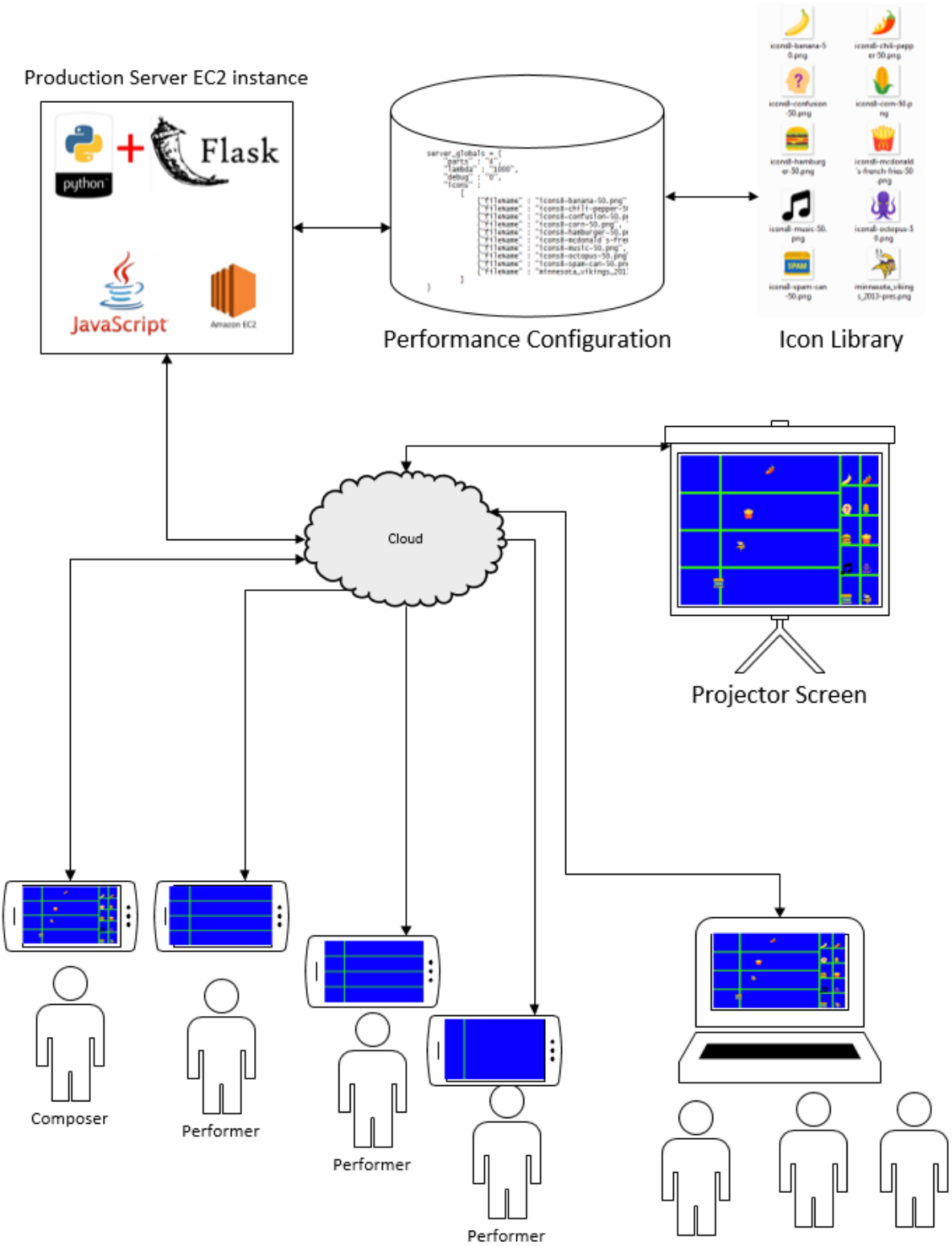
- Amazon EC2
- Ubuntu Linux
- Python
- Flask
- Html
- JavaScript
- Phaser 2.x
- Json
- Git / Github
- Amazon Cloud 9 (C9)

Phaser can be investigated at www.phaser.io. It is a general purpose javascript framework for supporting the implementation of Browser based 2D and some very limited 3D game development. Sprites defined in Phaser 2 are used to represent the graphical event icons. Here is a fairly recent article comparing 15 JavaScript game development platforms that ranked Phaser as #1 out of 15:
https://ourcodeworld.com/articles/read/308/top-15-best-open-source-javascript-game-engines

The StarDotStar repository is on GitHub. The codebase can be checked out at
https://github.com/skenny47/StarDotStar.git

The server is a Flask/Python Web service running on an AWS EC2 instance that implements the RESTful endpoints used by the clients. The server will also serves the main html page implementing the client user experiences as a static page with embedded script.

2. System Diagram



3. Client Algorithm

The Client follows this basic pseudo code:

```
Initialize the Phaser game internals;
Contact the Server to get Client Configuration;
Fetch individual icons described in configuration.
Consult the URL to determine mode
InitializeUserInterface(mode);

Whenever an Icon is dropped onto the surface
{
     Send an event describing this to server;
};

Meanwhile: Repeat{
                    Every Lambda:
                    Ask Server for latest event;
                    if (event hasn't been seen here)
                    {
                       Post Icon onto screen;
                       Give Icon Game Physics;
                       Give Icon horizontal velocity;
                    }
                }
```

4. Invoking the Client and basic usage

The Client runs in Composer or Performer mode.

The Client user experience has three major forms. These are Composer mode, and two forms of Performer mode. A visual metaphor of a musical score is used to represent these differing viewpoints on the same performance. Variations in the Server URL are used to indicate which Client user experience is being invoked.

- Composer Mode
  - http://ec2-52-36-206-120.us-west-2.compute.amazonaws.com:4747/composer
- Performer Mode showing 'Score'
  - http://ec2-52-36-206-120.us-west-2.compute.amazonaws.com:4747/performer
- Performer Mode showing one 'part'
  - http://ec2-52-36-206-120.us-west-2.compute.amazonaws.com:4747/performer?part=1

5. Client Configuration

The System is configurable

The following is an example of the JSON for an example configuration of the system

```
server_globals = {
   "parts" : "4",
   "lambda" : "1000",
   "debug" : "0",
   "icons" :
      [
         {"fileName" : "icons8-banana-50.png", "name" : "banana"},
         {"fileName" : "icons8-chili-pepper-50.png", "name" : "pepper"},
         {"fileName" : "icons8-confusion-50.png", "name" : "confusion"},
         {"fileName" : "icons8-corn-50.png", "name" : "corn"},
         {"fileName" : "icons8-hamburger-50.png", "name" : "hamburger"},
         {"fileName" : "icons8-mcdonald`s-french-fries-50.png", "name" : "fries"},
         {"fileName" : "icons8-music-50.png", "name" : "notes"},
         {"fileName" : "icons8-octopus-50.png", "name" : "octopus"},
         {"fileName" : "icons8-spam-can-50.png", "name" : "spam"},
         {"fileName" : "minnesota_vikings_2013-pres.png", "name" : "vikings"},
      ]
}
```

"parts" is the number of possible individual 'performer' views contained in the performance. The user interface for a 'composer' U/I instance and for a 'performer' U/I instance that's in 'score' mode will try to show all the parts as horizontal regions, logically numbered from top to bottom as 1 through n, where n is the number of parts in this configuration parameter.

"lambda" is the number of milliseconds between instances of the Client U/I contacting the server for the latest posted event. This value has not been rigorously experimented with to determine the best value for system performances when there are high numbers of nodes. The smaller the lambda, the greater the need for the server to have fast throughput in handling these RESTful invocations.

"debug" is either 0 or 1, and is a flag for turning on or off the verbose debugging output displayed to the background of the Client User Interfaces. This output was useful for helping to fine tune the implementation of the JavaScript in the clients, but has no place showing in the running, final version, so these values are off by default.

"icons" are a list of entries, where each one maps a Filename to a friendly name. The list gives the set of icons and the size of the list determines the number of icons. The user interface best supports icon sets that have an even number of icons. The user interfaces display the icon palette dynamically depending on the number of icons and depending on the existence of specific icons. The filenames must match actual .png files on the server in the Images folder.

6. Potential Applications

The system is applicable to a range of targeted uses.

One of the principal intended uses of this framework is as a means of managing a form of conducted improvisation involving sets of groups of musical improvisers. These musicians would read off of the dynamic score implemented in StarDotStar. Mixed in with the musicians for a given performance of the framework, there would be any number of 'composers' that could add to the dynamic score as it is being performed. The audience could also tune in on their mobile devices or by witnessing a large-scale video projection of the score while it is being simultaneously created and performed in real time.

This is the original intended usage of StarDotStar.

Having said that, a number of open-minded alternative usages for this framework come to mind.

1) Organizing geographically separated groups of people to perform social medial viral tasks like posting about something, or starting live social media feeds.

2) Organizing, starting and ending instances of flash mobs.

3) Crowd management. Orderly dismissal of very large groups of people.

4) Order fulfilment management in fast-food or other on-the-spot preparation situations.

5) Military troop movement and directions.

6) Dynamic Dance Choreography management.

7) Manufacturing / Assembly Line organization

7. Future Directions – Additional Features

Here is a start of a list for new features that will be next to be added to the software

1) More Icons. Locating and uploading many more possible icons.
2) Icon Sets. Creating another hierarchical level for Icon organization to define named sets of icons. These prefabricated sets could be included in a performance configuration with a single named reference. That single 'name' in the performance configuration would link to a separate JSON file with the mapping of the whole set of specific icons.
3) Background image to identify the 'part number' for when the U/I is in performer mode with a ?part=3 in the URL, the background image would show simply that this running U/I is for 'Part 3'
    a. Other possible background images would handle overall dynamics or directives for a specific part or for all parts. 'Play Louder'… 'Stop everything'… 'Resume playing'
4) Handling resizing and rotating of the U/I. Right now the size of the Phaser game surface that implements the U/I is a fixed number of pixels. It should expand to the size of the browser window, and handle device rotation re-orientations.
5) Performance Manager Web service.
   Create a secondary web service on the production EC2 instance with simple endpoints that can start-and-stop a running performance. And that can reboot the production server and a few other administrative functions.
6) Add a new configuration item for the velocity of the icons to support performance situations when a placed icon won't be performed for several minutes or even hours.
7) Add U/I ability for having repeating Icons, or even random icons.
8) Ability to allow performer to double click on an icon to get a pop up window of additional information about the meaning or purpose or attached information for the icon.