FRAICHE: Fast Response And Intelligently Controlled Harvest Environment

Peggy Chi, Jonathan Kummerfeld, Valkyrie Savage
Computer Science Division
University of California, Berkeley
{peggychi, jkk, valkyrie}@cs.berkeley.edu
CS262A Final Project, Fall 2012

1. ABSTRACT

FRAICHE is a complete small-scale sensing and watering system for deployment in community gardens. The goal is to provide gardeners with sophisticated monitoring and analysis of their garden at low cost in an easy to set up system. We achieved this by constructing a central server that receives sensors readings, models the environment to makes predictions about future moisture content, and responds to user queries made through a straightforward web interface. FRAICHE is implemented using a Raspberry Pi, a system with minimal computing power, which poses a challenge since our goal is to provide fresh predictions when queries are received, but with minimal latency. We considered six methods of scheduling model updates, and tested them under three very different use-case scenarios, finding that in each case we can strike an effective balance between freshness and latency.

2. INTRODUCTION

Managing a community garden or small farm is currently a process that requires community members or the farmer to directly monitor their land in an ad-hoc manner. Additionally, beginning gardeners who lack a large base of knowledge sometimes find it difficult to react effectively to the signals they see coming from their plants, watering them to the point of drowning or failing to notice when they are parched. Computing has been leveraged to improve the ease of interaction with and analysis of many other physical processes, and is a natural addition here, where continuous monitoring and effective predictions of soil moisture content can provide crucial information.

FRAICHE is a system with both sensing and watering capabilities. It can deliver information to the owner about a range of factors, such as moisture and temperature status of the soil around different plants, via a web app. The same web app allows them to water plants or schedule their watering, with the actual watering accomplished through a configurable drip irrigation system. The system also provides predictions about future soil moisture content, based on a range of different factors. These predictions are tuned to the particular garden being used, with values predicted based on past observed changes.

We implemented a complete, small scale system to test computation bottleneck issues for our implementation, particularly for how the sensors being used will manage communication with the central controller. To stress test the system, we simulated a scaled up version. This version involves an extremely large number of sensors working in a star network, with a single central controller receiving data from sensors, updating models, and handling queries from users. Constantly receiving sensor updates, using them to update the model, and maintaining low latency responses to clients is a reasonably simple problem with low overhead, but when running such a system on a machine like the Raspberry Pi that has little computational headroom, balancing these tasks becomes a challenge.

In light of this, we tried multiple options for improving response speed. We implemented several different schedulers to balance the loads of serving clients, getting fresh sensor data, and updating the prediction model. We also implemented a system whereby multiple Raspberry Pis functioning in separate gardens can load balance between themselves via a central data repository.

This work focuses on a particular use-case, gardening, but the question we are addressing is more general. When using a low-computation device such as a Raspberry Pi what is the best way to schedule expensive computation? Recent growth in the hobbyist community has seen an enormous range of applications for the device, which often include some form of sensing and a means of communication with users. The issue of latency that we focus on here could be particularly important for cases such as cooking applications and games that include real-world objects.

Our experiments show that while all scheduling approaches are effective in small scale situations, as either the number of sensors or the number of users increases the decision of when to make model updates becomes much more important. Our observations follow expectations, with low latency being easily achievable at the cost of freshness or vice versa, but with more balanced combinations being dependent on the use case. Finally, we show that by taking traffic patterns into consideration we are able to provide fresh information with low latency.

3. RELATED WORK

While the focus of this work is very specific, it is worth taking a broader view when considering related work. The general problem of resource management is well-studied, as is load-balancing, which we consider when demonstrating an implementation with multiple Raspberry Pi boards. For our more specific case of plant monitoring there has been work on various forms of environmental sensing, particularly recently with wireless mesh networks, but we only mention a few specific cases, as generally that work focuses on data collection followed by offline analysis, unlike our setup.

3.1 Resource Management

There has been a bevy of work on resource management for both parallel clusters and multiple virtual machines running on a single physical machine.

The Dominant Resource Fairness algorithm is designed to help allocate resources fairly in multi-resource systems where processes have heterogeneous demand [4]. It is a port of max-min allocation where instead of considering a process's overall needs they are measured as vectors with entries per resource. However, DRF is designed to work in a system where there are sufficient resources to satisfy all processes' needs, which is different from our situation on the Raspberry Pi.

Jockey is another resource management system, and it is built to hold processes to particular Service Level Objectives (SLOs) [3]. Jockey uses predictive algorithms to simulate each job and estimate the time to completion. FRAICHE differs in that we are aiming to serve models that are as fresh as possible to clients, and the access times of clients are not as well - or strictly defined as SLOs.

Lottery scheduling has been put to use for managing resources: the basic idea is to allot a certain number of tickets to each process requiring a resource based on that process's importance and relative need for the resource [19]. FRAICHE's goals are not to schedule many processes fairly to run in minimal time, but rather to run the model refinement algorithms as close as possible to clients' requests for plant data.

For virtual machines sharing a physical computer, Xen has explored the resource management question [1]. However their main challenges lay in isolation of the various virtual machines from each other in order to ensure security, while our challenges lay mainly in scheduling.

Work on scheduling for parallel clusters has included Lithe [12], use of loosely-synchronized clocks, and others. The goal for scheduling with parallel clusters, however, is based mainly on ordering of events when processes don't share a clock. In contrast, FRAICHE focuses on just one machine on which events take place.

3.2 P2P Network and Load Balancing

Sharing available workstations in a distributed computing environment has been widely studied in the systems community. Nichols discussed a mechanism to run network server programs on idle workstations considering process invocation and machine registry [11]. Condor is a scheduling system that aims to utilize the performance by assigning background jobs to those idle workstations it "hunted" [10]. More recent systems including Pastry [14] and Chord [16] design routing mechanisms to limit hops for a query in a peer-to-peer (P2P) framework. The multiple Pis scenario of FRAICHE consider these system designs to leverage the workload of one web server to other idle machines.

Our work is closely related to load balancing and load sharing designs in distributed networks. Harchol-Balter and Downey proposed a preemptive migration scheme for CPU load balancing using a trace-driven simulation [5]. It required no a priori knowledge about the process behaviors in networks of workstations, which matches our scenario where community clients can freely access the plant data. Harish and Owens [6] designed load balancing models and algorithms for DNS servers, which is a simple but effective mechanism.

3.3 Environmental Monitoring

There are several monitoring systems that share the similar purposes as ours. Macroscope is a wireless sensor network deployed in a 70-meter tall redwood tree [18]. The measured data included air temperature, relative humidity, and solar radiation. Their case study in practice addressed challenges of extracting meaningful information from the collected data and physical installation.

Other sensor network systems targeted on different scenarios: Cardell-Oliver et al. designed an event-driven network to monitor soil moisture to measure water flow [2]. Their environment reactive design consider the rainfall conditions: only when rain falls that causes rapid soil moisture changes, the measurements are collected frequently. PipeProbe is a mobile sensor system that constructs spatial topology of water pipelines by reading sensor capsule data [9]. These projects demonstrated the practical designs deployed in the real world, and therefore inspired us to monitor and provide prediction by modeling the moisture data.

4. PHYSICAL ARCHITECTURE

FRAICHE is composed of two physically separate system components. One is the device co-located with plants in the garden, which we built based on an ATMEL chip. The second component is a Raspberry Pi, which is located outside the garden. The two communicate through an XBee wireless radio.

4.1 In-garden device

The in-garden device contains several components within a 3D printed body and is connected to a drip irrigation system. A servo attached to a valve controls the flow of water in the downstream portions of the irrigation system. This servo is controlled by an ATMEL chip. Attached to the circuitboard with the ATMEL chip is an XBee radio for communication, a battery, a solar panel, and an array of sensors for detecting moisture, temperature, etc., of the soil. On the top of the device there are also four buttons for managing watering configurations while co-located with the plants (i.e. the user need not access the website to adjust watering thresholds: they can push a few buttons while looking at the plant if it seems to be too dry or too wet).

This device awakens from sleep mode approximately once

per hour or when the buttons are pushed. Whenever it is awake, it begins the communication process described below.

4.2 Raspberry Pi(s)

The Rapsberry Pi is a low-cost, portable single-board computer released in 2012 (USD35, about credit-card sized: 3.370 by 2.125 inches). In our system, the Raspberry Pi is connected to the Internet and also has an XBee mounted in order to communicate with the in-garden devices. It serves three purposes: it maintains communication with all in-garden plant systems, it provides a web interface for gardeners to monitor their plants and update their watering configurations, and it maintains models of the typical watering patterns of the plants.

The Raspberry Pi's communication with the in-garden plant systems takes the form of a very brief exchange initiated by those systems. To differentiate between readings, messages sent by each in-garden device are preceded by that device's unique ID. When a reading is received by the Raspberry Pi, it either sends the most recent watering instruction it received from the gardener via the website or a single acknowledgment character.

The Raspberry Pi also runs a simple webserver which allows gardeners to track the measured readings of their plant along with a display of the predictions made by a model of each plant's moisture conditions.

In the scenario we describe with multiple Raspberry Pis, each has a connection to a central database (stored in the cloud using Amazon EC2 service) which serves as a data repository. The system is designed to be scalable: when a new device connects to the system, it registers with its IP address and the information is broadcast to every Raspberry Pi in the current garden network.

5. SOFTWARE IMPLEMENTATION

All systems were implemented in Python 2.7, and all tests were performed on a Raspberry Pi Model B, running Debian Linux. Specs of the Raspberry Pi are as follows: 700MHz ARM 1176JZF-S core, Broadcomp VideoCore IV, 512MB SDRAM shared between CPU and GPU, 32GB SD card. The Raspberry Pi is rated to 700mA at 3.5W. On the client side, the web user interface complies with web standards and runs JavaScript to display the latest data with visual graphs in real-time.

5.1 Webserver

We elected to use a pre-made open-source webserver for our implementation: Tornado [17]. It was created by FriendFeed for their primary web stack, and later open sourced by Facebook after they acquired FriendFeed [13]. Tornado allows for different handlers to be attached to different web addresses. One of these is the plant data page, which is loaded by each client; one is a WebSockets endpoint, which is connected to by the main page upon loading; and one is a sensor update endpoint for testing.

The plant data page contains a graph, powered by the Google Stocks API, of historical plant water data and also a display of the predicted water model. It has a form by which the client can send explicit instructions for desired watering specifications to the in-garden device and toggle automatic watering. The WebSockets connection is created in JavaScript when this main page is loaded: the WebSockets pipe is used for transmission of plant and model data to the client and instructions transmission from the client. The most recent 15 historical data points for plant moisture are transmitted when the page is loaded, afterwards sensor and model data are updated whenever new information is available.

In order to build a test harness which can simulate many sensors and many clients without the need for in-garden devices, our other endpoint is a sensor update endpoint. From here, we log the data from incoming sensors to a file for later access. This endpoint also updates the schedulers with information about when new sensor data are available.

5.2 Schedulers

Six independent schedulers were built: Naïve, Periodic Offline, Sensor-triggered, Hybrid, Low Load, and Predicted Demand. The scheduler API is as follows:

```
gotSensorEvent(plant_num, value)
gotClientRequest(plant_name)
isTimeToRunML() # returns bool
runMLPredict() # returns float
runMLUpdate() # returns float
```

gotSensorEvent, gotClientRequest, and isTimeToRunML are over-riden by subclasses as appropriate (the last returns False by default). All subclasses use the default runMLPredict, which returns a float indicating the predicted soil moisture at the next time step, and the default runMLUpdate, which feeds all not-yet-used sensor date to the model.

The server calls gotSensorEvent and gotClientRequest when it gets appropriate events and before it handles them internally. A periodic callback is implemented to ping the scheduler every 5 seconds, prompting it make an update to the model if isTimeToRunML returns true. This is important for the scheduling algorithms that do not have event-based model updates.

5.2.1 Naïve Scheduler

The naïve scheduler is very simple: when its gotClientRequest function is called, it takes all sensor data not yet incorporated into the model and updates the model with it. It uses default functionality for the remainder of the functions. This approach ensures maximum freshness, as the model is as up-to-date as possible when the prediction is made, but will lead to greater latency as the user is waiting while the model update occurs.

5.2.2 Periodic Offline Scheduler

The periodic offline scheduler has a period of 5 minutes. Once five minutes has elapsed since the last time its model was updated, all new data are fed to the model. This approach targets the opposite end of the spectrum from the previous approach, with minimal latency, but potentially poor freshness.

5.2.3 Sensor-triggered Scheduler

The sensor-triggered scheduler overrides gotSensorEvent to update the model after each sensor reading is recorded. This maintains the freshness guarantees of the naïve approach, but

decouples the update cost from the user request. The trade-off here is that there will be many more updates as there is no caching of data.

5.2.4 *Hybrid Scheduler*

The hybrid scheduler has an instance of the periodic offline scheduler and an instance of the sensor-triggered scheduler, which are used to guide the timing of model updates. The rationale behind this approach is that when the user makes a request the update being made involves less data, (since some was processed in the last periodic check), and so latency will be improved.

5.2.5 *Low Load Scheduler*

The low load scheduler defines "low load" as fewer than 15 client requests in the preceding 5 minutes. If this load is seen at one of the periodic callback checkpoints, the model is updated with all fresh sensor data. This approach is primarily targeting the multi-user case, which is expected to suffer in all of the cases above as the extra users lead to delays in query-triggered approaches, and periodic methods could time updates at particularly bad times.

5.2.6 *Predicted Demand Scheduler*

The predicted demand scheduler has a meta-scheduler of the naïve variety. This meta scheduler models client request traffic. When the meta-scheduler reports that client demand is predicted to increase in the near future, the models of plant moisture are updated with all fresh sensor data.

5.3 Prediction Model

A key part of the motivation for this project is the use of a model that makes predictions about future water moisture levels, to allow gardeners to make more informed watering decisions. However, the actual quality of the model used is not one of our metrics. We implemented a linear regression model with forgetful updates (rather than using the standard online updates to the mean values of x and x^2 the value for the count of observed items is assigned a fixed value, which effectively gives older values progressively less weight).

We also considered more complex models, such as a Kalman Filter [8], but found constructing an effective model difficult as our simulated world data was not well represented by a hidden markov model. Rather than changing our simulation to suit our model we decided to stick with the simpler regression model and use time delays to simulate more sophisticated processing, a fixed delay of two milliseconds per measurement per update and an added delay of ten milliseconds per update.

6. SYSTEM EXTENSIONS

The two previous sections described the components of our base system, but in the process of implementing these it became clear that there were drawbacks with our architecture. In particular, the limited computation on the Raspberry Pi was still an issue, and we had concerns about scaling beyond what a single device could handle. Below we describe our investigation of ways to address these concerns.

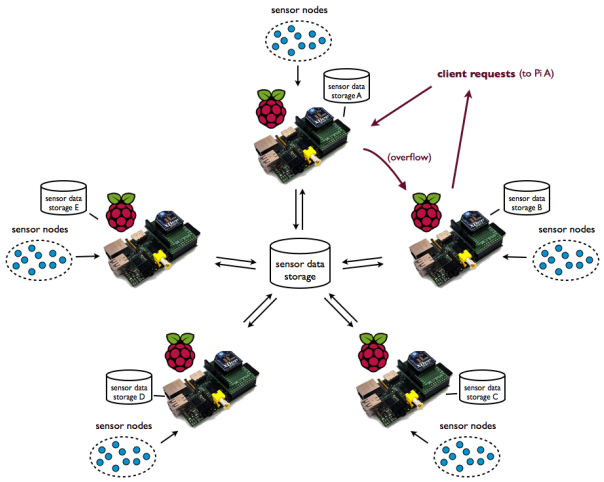

Figure 2: Parallel form of the system architecture.

6.1 Client-side Modeling

One approach we considered using to address the low computation power of the Raspberry Pi was to push computation to the client, as a consumer desktop machine is orders of magnitude more powerful than the Raspberry Pi. We investigated implementations that pushed the prediction modeling to the user via JavaScript and found it was effective for our simple prediction model. However, this approach was not considered worth further investigation as one common use case would be to have client requests coming from mobile devices, which are more powerful than the Raspberry Pi, but not significantly enough to be worth the communication cost.

6.2 System Expansion

In a real-world scenario we would expect that user demands would gradually increase, with more sensors and more clients. With this in mind we experimented with easy expansion of our system. Our approach was a simple load-balanced division of sensors and clients, with a broadcast announcement when new devices are added, as illustrated in Figure 2.

Our example implementation involved two additional Raspberry Pi boards and facilities for them to communicate and load balance. Each web server has a connection to a central data repository using Amazon Cloud Services as an intermediary data storage location. The system is designed to be scalable: when a new device connects to the system, it registers with its IP address and the information is broadcast to every Raspberry Pi in the current garden network. Each RPi records this information as a resource list (DeviceList).

Clients connecting to a server (e.g. RPi A) for local plant information will be served with local repository. When the server receives too many client requests over its client capacity, it randomly selects one web server (e.g. RPi B) in the current network from the received resource list DeviceList. To keep the system design simple, it redirects the overflow clients to this selected RPi B without asking for permission in advance. The clients are redirected to a http address with the new IP of RPi B and the requested plant info (RPi A and plant_num). When RPi B receives this request, it pulls the plant informa-

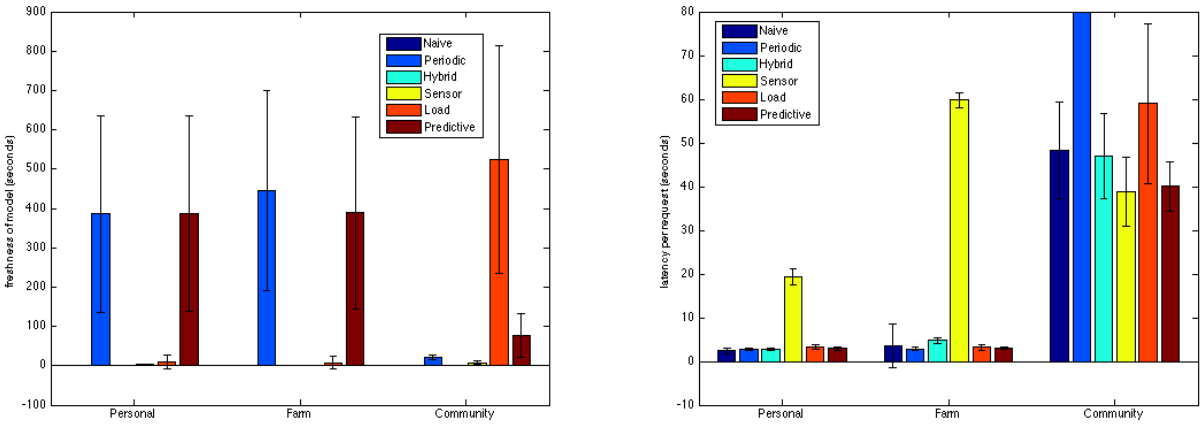

Figure 1: Freshness (left) and latency (right) measurements for the six schedulers under three load types: personal, single user and five sensors, farm, single user and one hundred sensors, and community, fifty clients and one hundred sensors.

tion from the central storage and responds to the client. If RPi B happens to be overloaded, it then randomly selects another server RPi C and attaches its own IP into the ignored list, to prevent to be redirected again for the same request.

Below shows these two major components in our system:

A central storage on Amazon EC2

- receive registration and broadcast the list to every RPi
- update the sensor log from RPi to storage
- retrieve the sensor log for a RPi from the storage

Web server on Pi

- register and record the device list
- serve local clients with local storage
- serve redirected clients by retrieving log from the central storage
- redirect clients to other pis if overloading

This system is scalable and self-balancing. Report to the central storage is done via the free tier of the Amazon EC2 (Elastic Compute Cloud) service that provides flexible and scalable capacity in the cloud. Inter-Pi communication occurs using http requests over web sockets, as an addition to the current server infrastructure.

7. EVALUATION

It was not feasible to set up multiple sensors and run a live test over weeks, so we implemented a simulation harness to emulate the sensors and clients. We ran all simulations with the server on the Raspberry Pi, and the mock sensors and clients on our own laptops.

The harness was written in Python, using Tornado to manage queries. We experimented with two methods of simulating clients, first with splinter [15] running actual firefox instances, to maximise realism, then with phantomjs [7], which made large scale simulations much easier. It was necessary to use a system capable of running JavaScript as the client's page is only considered to be "loaded" when the 15 most recent historical data points (transmitted via WebSockets, as described earlier) are delivered via JavaScript.

Client processes are forked from the main process: each client has a configurable list of plants from which list one plant is chosen randomly to query at each time step.

8. RESULTS

We simulated three scenarios for FRAICHE: personal garden, community garden, and small farm. Each of these scenarios has a distinctive balance between clients and plants. All simulations were performed on the Raspberry Pi with client requests and sensor updates issued from a MacBook Pro.

Client request batches were issued in 15 second time steps. The number of clients making fresh requests was set to follow a sinusoidal pattern with some added noise. Sensor update batches were also issued in 15 second time steps. All sensors reported new data in every time period, though the updates were randomly spread across the period, to model the fact that the sensors would all be reporting periodically, but not be in sync.

Figure 1 shows average results for measurements over fifteen minutes for each condition. Freshness was tracked server side by the scheduler and logged every time a request is made. Latency was measured on the client side and logged for each request.

8.1 Personal Garden

A personal garden is characterized by a single person caring for a small number of plants. For the purposes of our simulation, we used queries from a single client and sensor readings from five sensors. The results are shown in the first sections of each of the two plots in Figure 1.

The naïve, hybrid, and load schedulers all perform very similarly, and any of them would be acceptable. These conditions simply aren't taxing enough for the cost of updates to impact performance. The poorer results for sensor-based scheduling reflect the fact that there is no batching of readings, and so the effective computation cost of the updates is greater, leading to worse latency. Meanwhile, the freshness results follow the trends we had anticipated, with periodic and predictive having much worse freshness. Since there is no latency gain to be had by switching to these schedulers they provide no benefit.

8.2 Small Farm

One direction in which the system could be scaled is the number of sensors. In our second set of experiments we considered the setting of a small farm, where there is still a single client (the farmer), but now there are one hundred sensors reporting back to the Raspberry Pi.

Comparing the results in Figure 1 for these conditions (in the middle) and the personal garden (on the left), it is clear that the addition of many sensors has a limited impact on performance. The worse result for freshness of the periodic model is within the bounds of error. At first glance the results for latency also seem comparable, with the exception of the sensor-triggered updates, which of course has a worse results because there are now more updates occurring, and so the chance of a collision in timing between an update and the client request is higher. The interesting result here is the increased variance for the naïve approach, which also influences the hybrid result. This is reflecting the same factors that negatively impact the sensor-based update approach, though in a more limited manner.

8.3 Community Garden

Finally, we considered expansion along the number of clients axis, running an experiment with fifty clients simulated by PhantomJs, and one hundred sensors. These results are significantly different, showing several interesting variations.

Focusing first on the freshness results, we see that all except load and predictive are extremely good. The worse result for load based scheduling reflects the fact that there are fewer times at which load is deemed low. The predictive result is not good, but better than the results for the previous experiments, which is caused by the greater information available about user behaviour, leading to better predictions regarding timing of user requests. The most surprising result is for the periodic scheduler, though this may be partially attributed to the timing patterns of requests, which may have conveniently peaked at the right time, an effect that is amplified by the number of clients.

The latency results are a much less positive story, with significant costs for all cases. This shows a source of computation cost that we had not anticipated – the network communication with clients. In this case the clear winner is the sensor based approach, which effectively spreads the load of updates across time, rather than concentrating it, as cached updates do. This leads to a decreased impact overall on clients, and a lower variance as no single client is making a query at the time of a large update.

8.4 Client-side Modeling

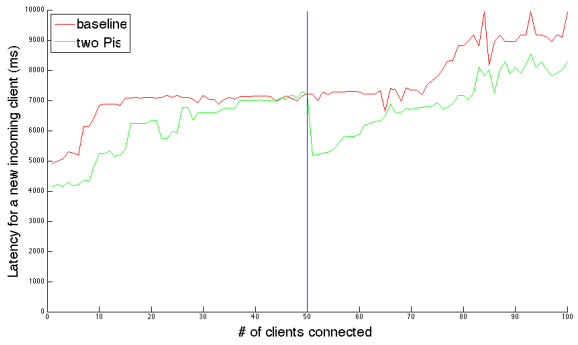

Figure 3: Client request latency with load balancing between two web servers.

To evaluate the effectiveness of pushing the prediction modeling to the client browser, we ran an experiment that simulated a personal garden of five plants using the naive scheduler. However, we did not find the latency significantly improved. It is possible that the performance would increase when the conditions are more heavy loading.

8.5 Load Balancing Between Pis

Finally, we evaluated how the load balancing mechanism performed between multiple web servers. To simplify the scenario, we ran the experiment with two Raspberry Pis. We simulated a heavy loaded community garden condition with 100 clients, each making one request for a unique plant to one RPi. The web server on this RPi then redirects the overflow client requests to another RPi that has low workload. In our experiment, we assume this second Pi as totally idle without performing other work. It pulls out the latest information from the central storage to respond to these redirected clients.

To stress test the system, new client requests were issued in 0.5 second. The clients never leave the system so the servers would have to keep updating the latest sensor data when available. We examined if the latency decreases after the targeted web server started forwarding the requests.

Figure 3 shows the result of our experiment. The baseline in red shows that the latency increases as more clients are connected, especially after serving 80 clients. The load balancing result in green shows how the latency drops when the clients are redirected to an idle server when over 50 clients are connected. We suspect that the overhead of accessing data from the central storage increases the response time, and therefore results in insignificant improvement in general. However, this design is scalable and would be promising when introducing more devices in the network.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we presented FRAICHE, a complete small-scale sensing and watering system for deployment in community gardens. We designed six methods of scheduling model updates, and tested them under three very different use-case scenarios, finding that in each case we can strike an effective balance between freshness and latency. We also expanded our system to a multiple device scenario for load balancing.

In the future, it would be valuable to explore the space of very small commodity computers further. There are many open questions relating to their performance for tasks. Furthermore, we are interested in acquiring real-time weather conditions to improve prediction, such as real-time data from Weather.com. We'd also like to conduct a more thorough evaluation to compare our low-cost, scalable system design with other existing watering systems on the market.

10. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, Oct. 2003.

[2] R. Cardell-Oliver, K. Smettem, and M. Kranz. Field testing a wireless sensor network for reactive environmental monitoring. *International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, 2004.

[3] A. D. Ferguson, P. Bodik, S. Kandula, and E. Boutin. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *European Professional Society on Computer Systems (EuroSys)*, 2012.

[4] A. Ghodsi, M. Zaharia, B. Hindman, and A. Konwinski. Dominant resource fairness: fair allocation of multiple resource types. *the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[5] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[6] V. C. Harish and B. Owens. Dynamic Load Balancing DNS. *Linux Journal*, 1999(64es), Aug. 1999.

[7] A. Hidayat. Phantomjs, 2012.

[8] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

[9] T.-t. T. Lai, Y.-h. T. Chen, P. Huang, and H.-h. Chu. PipeProbe. In *the 8th ACM Conference*, pages 113–126, New York, New York, USA, 2010. ACM Press.

[10] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. *the 8th International Conference on Distributed Computing Systems*, 1988.

[11] D. Nichols. Using idle workstations in a shared computing environment. *ACM SIGOPS Operating Systems Review*, 1987.

[12] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. *ACM Sigplan Notices*, 2010.

[13] D. Recordon. Tornado, September 2009.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware 2001*, pages 329–350. Middleware '01: the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Berlin, Heidelberg, Oct. 2001.

[15] F. Souza. Splinter, 2012.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, Aug. 2001.

[17] B. Taylor. Tornado, 2009.

[18] G. Tolle, J. Polastre, R. Szewczyk, and D. Culler. A macroscope in the redwoods. In *SenSys '05: the 3rd international conference on Embedded networked sensor systems*, 2005.

[19] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.