Summary:

We implemented a distributed database server with Raspberry Pi’s and compared the performance per watt and dollar with the Andrew Linux servers. The Raspberry Pis were set up in a master/worker system, where tasks were routed to the workers using consistent hashing techniques. We used the golang RPC package to actually write the distributed system.

Background:

Modern computers are often significantly impeded by issues stemming from memory access. As most systems use disk for large quantities of storage, there is a penalty for both random read latency and power consumption. In a distributed system, this means that accessing data sorted by a hashed key can be a bottleneck for client operations. Additionally, the wattage and price required to power a large server rack can be discouraging to those who want to host their own servers. Using low-power, diskless nodes, like Raspberry Pi’s (running with ARM 700 MHz CPUs with flash memory), many of these problems can be solved. By running a large quantity of small nodes in parallel, a memory bandwidth comparable to a typical server rack can be achieved at only a fraction of the cost.

Setup

There is one master node in the system that clients can connect to and send requests. Upon receiving requests, the master hashes the request and forwards the requests to one of the many workers connected to a local area network based on the hash of the key. By using consistent hashing, all identical keys are routed to the same worker node.

Data Structures

On the master node, a go slice is used for managing the consistent hashing ring and finding the correct workers to forward the requests. Also on the workers, pieces of the database are managed in main memory, also in a map.

Types of Requests

We have three types of requests:

1. Basic GET/POST: These requests update the state of the database or get the current state of the database. Essentially, a client can call GET/POST, which is hashed in the master, and stored in a hashtable data structure on a worker.

2. Hashing: This request takes a value from the database and hashes it multiple times to test the response time when the system cranks CPU intensive jobs.
3. Uploading an image: This task uploads an image to the database and saves the image on one of the worker nodes in the system. Each image is 1MB in size so with various number of clients, we can gain insights about the performance of system when under high bandwidth pressure.

We had a fourth request planned, which would have involved using the Pi’s GPU to actually manipulate the image. However, this was not possible to complete by our final demo. We spent our time working on effectively parallelizing streams of requests, rather than single requests themselves.

**Workload**
In the test benches, we had 10, 15 and 20 clients sending requests simultaneously and recorded the average response time of one single request. We tested on both the raspberry Pis and Andrew machines, using combinations of 3 and 5 workers, for a "pure" GET/POST test, hash test, and image upload test. These workloads tested, respectively, the latency, computation/workload balancing, and bandwidth of the system. Essentially, the workload is designed to reflect how fast the server’s can react to one request and dismissed some of the effects of fluctuation in the network and time to set up the connection.

---

**Approach:**

**Golang, the RPC package, and locking**
As mentioned earlier, we used go, with a focus on the RPC package, to communicate between pis. Go’s RPC package has the advantage of making remote calls seem somewhat transparent, which was very convenient when moving data between clients, masters, and workers. Furthermore, the RPC package can actually transparently spawn new "go" threads, which are well-known for being lightweight, for each RPC call. This is especially beneficial on the master – since the only data stored on the master includes the worker connections, we could have many threads operating on the master node without any concern of race conditions. Unfortunately, on the worker nodes, this is not as simple – we had a new problem of locking the database for key accesses. We intentionally put locks around the worker databases in fine-grained points, only around pulling values out or putting values directly into the database, represented by a hashmap.

**Consistent Hashing**
We used a consistent hashing ring to manage our data, meaning that the master split the range of unsigned integers (0 to 2 ** 31 - 1), and during setup, decided that keys which hash to evenly distributed sections of the ring will always go to a single dedicated worker. In practice, with random enough keys, this worked well, and data was fairly even distributed among workers.

**Local Area Network and Port Forwarding**
Setting up a functional server inside the CMU network is not trivial, so we chose to use our router at home and forwarded any SSH requests to the master node on the Pi. This way, any internal communication is within the local area network and invisible to users.

**VideoCore GPU**
A lot of work was done on the GPU of the Pi, although for our tests, this information was removed, mostly because of time and complexity of the task. We believed that using the GPU on the Pi could provide the user another layer of parallelism, in addition to the distributed nature of the server. On top of that, GPU parallelism would let the servers more fully utilize their resources. However, the API on the VideoCore GPU was just recently released and very poorly documented,
leading to a lot of issues during implementation, which impeded this task. We have preliminary code available for remote GPU access, and will continue progressing on this task throughout the summer.

Results:

A significant challenge for our analysis has been identifying a way to compare the Raspberry Pis to alternative systems, which we would also be able to test. It was especially difficult, because our ideal competitor would be an extremely lightweight existing server, parallelized over multiple blades on the same rack. Since we could not find a competitive server to purchase, our best choice was testing on the Andrew Machines, where many of the implementation details are somewhat abstracted (in particular, communication between Andrew Machines). Also note that it was basically impossible to measure the power usage at runtime, because those machines are locked up somewhere in Gates.

Regardless, we used the Andrew machines as competitors, since our testing task stream would be enough to parallelize as much as possible on these multi-cored machines. Furthermore, the serial speed differences would give us at least a ballpark estimate of the drawbacks and advantages of Raspberry Pi performance. Following is the result from the various test benches run both on the Raspberry Pis and the Andrew machines:

As you can see, figure 1 shows the average response time for a simple GET/POST request test bench. The time on the Pis are indeed slower than the Andrew Machines but this is well expected since we have less advanced serial hardware. However, the point is, for the simple GET/POST requests, the response time is on the same level of magnitude as the Andrew Machines, which would be great for justifying the performance per unit resources used later.

Also, it is worth mentioning that the test benches are run on multiple machines of the Andrew
servers, compared to the same number of Pi's we have in our system. We choose this configuration
over running each worker as a separate thread on the Andrew Machines because the latter choice
would result in very different networking scenario: interconnecting on chip (if we use threads) vs.
using a networking switch that is connected by an ethernet cable.

(Figure 2: Hash1)

(Figure 3: GETPOST2)
In figure 2, similar metrics are plotted for the computational hashing tasks, each of which takes a value in the database and hashes it multiple times to cause the same effect as any CPU bounded job. Since we have a much slower CPU on the Raspberry Pis, the performance of the Pis are expected to be a lot worse than the Andrew Machines. Specifically, the Pis each have a 700MHz ARM core while the Andrew Machines have Intel Six Core Intel Xeon, 3.20GHz. As you can see, the average response time for the hashing task is about 10 times slower for each test cases. It is also interesting to note that as we increase the number of clients (rate of incoming requests), the response time of Raspberry Pi increases close to linearly. This suggests that the CPU is indeed saturated on the workers and as more requests come in, the requests just have to be queued on the workers and handled one by one.

After analyzing raw hashing performance, we observed energy consumption in figures 3 and 4. The y-axis, performance per watt is defined as 1/ave.response time * power consumed. To make the numbers fit nicely on the scale, we multiplied the result by $10^6$. To measure the power usage of the Pis, we went to an ECE lab and measured the current going through the Raspberry Pis under peak stress (Picture of multimeter included at the end). We discovered that there is about 0.3 amps of current flowing through each Pi during this time. Although we were not able to do the same for the Andrew Machines, we looked up the power consumption online and it’s 525W peak time. We did not consider any power saving schemes that the Andrew Machines could be deploying because that would complicate the problem enormously, and given our current resources, it has been essentially impossible to measure true wattage without physical access. This means that we are assuming that each Pi is essentially using $1/500$ times the power compared to the Andrew Machines. Even if the Andrew machines are drawing only a fifth of their power supply, which is probably more realistic, the Pis are still more efficient on a measurement of performance/watt in all tests. Thus, we used this overestimate of power on Andrew machines as an upper bound benchmark – it is definitely an area on which we plan to improve when testing more realistic benchmark comparisons.

We can clearly see that the performance per watt on the Pis is magnitudes higher than the Andrew
Machines, in both tests displayed. One might argue that the Andrew machines might have better abilities to handle multi-threaded workload more efficiently since it has more cores. However, if we look into the specification of the Dell T3500 machines (which is used in the lab on Gates 5000), the actually CPU consumes a minimal amount of power compared to disk and fan. With the SOC nature of our system, as well as flash memory, we are essentially saving power on transporting data back and forth over long interconnects. To summarize, even if we replace the CPU of the Andrew Machines with the same CPU as on the Pi’s, we will still be saving a lot of energy because of the smaller machine sizes and more energy efficient (albeit smaller) memory.

(Figure 5: bandwidth)

There is one major constraint for the Raspberry Pis: bandwidth limitation. With large parallel systems, bandwidth can often become a serious bottleneck, and this is especially true for the Raspberry Pis. Just to see how bad this effect is, we implemented a bandwidth test bench that will upload a 1MB picture onto the workers with every request. We produced figure 5: the average response time per request when we a varied number of clients sending these request streams. We discovered that there is a ‘super linear’ jump in response time going from 15 clients and 20 clients, meaning that the increase in response time is not exactly proportional to the increase in the request rate. This can be explained by the fact that the maximum bandwidth of access to the flash memory (which is where we save the pictures) is about 17 MB/s. Thus, the system works acceptably when we have 15 clients uploading in parallel but breaks down when there are 20 clients, given our current test bench.

System bandwidth is an area on which we hope to improve - the main bottleneck in the system is currently uploading large images to the master and distributing them to the workers as quickly as possible. The master, however, is the single point of access to the system. In future iterations of the distributed pi system, we hope to actually have many masters, allowing for more varied entries to the worker nodes, relieving this bandwidth limitation.
References:

Raspberry Pi GPU spec: http://elinux.org/Raspberry_Pi_VideoCore_APIs
Golang Documentation: http://golang.org/doc/
Raspbian Documentation: http://www.raspbian.org/RaspbianDocumentation
LAN setup: http://pihw.wordpress.com/guides/direct-network-connection/

List of Work By Each Student:

**Sean Klein**: Setting up the RPC connection and the Golang framework to support the basic GET/POST requests; Implementing consistent hashing and load balanced tests; Automating testing with Bash and Python scripts;

**Xiaofan Li**: Setting up hardware and network connections; Implementing Compute and Bandwidth functionality; Implementing GPU compute

**Both**: Testing; Analysis of results; Presentation

Lessons Learned and Future Development:

1. Finding the appropriate data to measure performance was tricky. At first we focused on average time for one client to finish a stream of requests rather than the average response time of one request. However, this metric was misleading since it also took the connection time and various conditions in the network into account. Over the summer, we will develop metrics and tests that are more suited for our purpose and improve the results.

2. The VideoCore GPU was also not the best GPU to work with since it does not support CUDA or a lot of the modern OpenGL libraries. After some research online and multiple experiments, we were able to save a GPU rendered texture in the file and ready to transmit back to the clients. However, the file format is .raw, which we were not sure how to display. For both demo and test purposes, this was an issue - we are plan to have proper GPU image encoding/decoding available in the next couple weeks.
Appendix: Some Pictures:

(hardware setup)
(peak time current)