Meerkat: A Power Loss Notification Protocol

Christopher Hunt
California Polytechnic State University
San Luis Obispo
chhunt@calpoly.edu

Vanessa Forney
California Polytechnic State University
San Luis Obispo
vforney@calpoly.edu

Paul Fallon
California Polytechnic State University
San Luis Obispo
pfallon@calpoly.edu

Michael Norris
California Polytechnic State University
San Luis Obispo
mnorris@calpoly.edu

ABSTRACT

Power loss presents a problem for systems, even when uninterruptible power supplies are in place. Frequently, these battery backups fail without warning, and a user can face data loss if his or her remote server dies. However, due to capacitance in normal power supplies, the server can still operate for a short time after power is cut. In this paper we analyze the compute time provided by various capacitance values. We also present the Meerkat protocol, which takes advantage of the time between loss of power and loss of computation ability on a dying machine to notify the "buddy" Meerkat.

Keywords

power loss, notification, protocol, backup, network

1. INTRODUCTION

The concept of backup power has been an integral part of important infrastructure (hospitals, scientific laboratories, naval ships) for decades [1]. When applied to the computing world, backup power supplies can provide an extra layer of protection to reduce the risk of data loss. Even if these backups fail, normal power supplies contain capacitors that hold charge for a short amount of time. Most commercial operating systems will trigger signals when a change in capacitance is detected, signaling an impending loss of power [2]. This allows the OS to theoretically perform certain actions before full shutdown.

In this paper, we document how much a machine can compute after the power has been cut, as it drains the remaining power in capacitors. Specifically, we gauge this by measuring the time the dying machine can keep sending UDP packets over a network to a buddy system. In a real network, the system receiving the dying signal can execute a user-defined callback that handles the partner machine going down. A large scale real-world example of this is a server spinning up additional threads and processes to deal with increased client connections, now that another server has gone down [3]. Individuals can utilize this functionality also. If a user's remote server goes down, it may not have time to send a complex notification to the user's smartphone. With Meerkat, a surviving buddy server can execute a more complex routine when it receives a loss signal.

We begin with an explanation of the protocol implementation in Section 2, discuss the details of the experiment on both the hardware side and the software side in Section 3, and evaluate our findings in Section 4 before concluding.

2. IMPLEMENTATION

The goal of the Meerkat program is to monitor the power status and alert another system when power loss is experienced. This is done by running Meerkat in the background on one system and monitoring the power status through a GPIO pin. When the current Meerkat experiences power loss, it sends an alert to its buddy Meerkat that initiates a callback. That way, the Meerkat that has power can initiate the callback while the dying Meerkat only has to send out a loss packet. The reason for this design was because there is only a limited amount of time a system can run after power loss is experienced, and so we want as few steps as possible for the dying Meerkat. The program consists of two phases: the buddy configuration phase (`WAIT_ON_BUDDY`) and the general data handling phase (`WAIT_ON_DATA`). The configuration of the system is shown in the state diagram in Figure 1.

The buddy is another system running Meerkat that should be alerted when the current system loses power. A Meerkat can assist another Meerkat, provided the other Meerkat has completed the handshake protocol and has been added to the current Meerkat's clan. A Meerkat's clan consists of any Meerkat that has sent a buddy request packet and has received a buddy confirmation packet. Once a Meerkat has a clan, it will be on the alert to assist the Meerkats in the case that one experiences loss.

The program is run on the command line as follows:

```
./meerkat buddy_ipv4_address callback_binary_name
-p my_port -b buddy_port
```

The specified values include the buddy IPv4 address and the callback binary, along with optional values of the current Meerkat's port and the buddy Meerkat's port. The callback binary is initiated in the case that a Meerkat in the clan has sent out a loss packet. The buddy IPv4 and buddy port specify where the current Meerkat will send a loss packet when it experiences loss itself. Optionally, the current Meerkat's port can also be specified, but both the buddy and personal port have a default value of 12345.

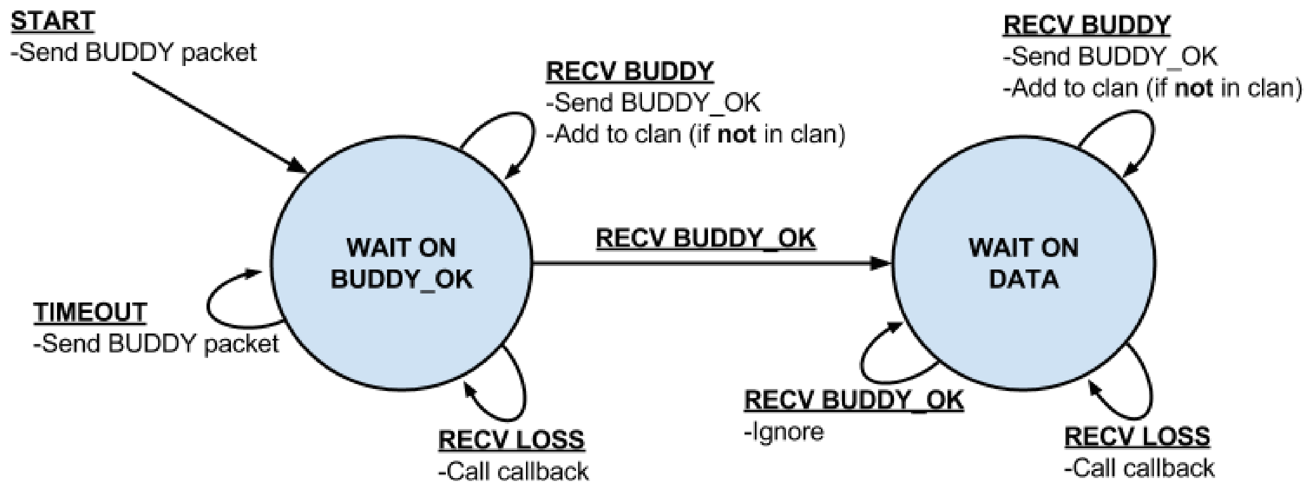

Figure 1: State diagram of the Meerkat program.

The Meerkat program requires there to be two instances of Meerkat running in order for the protocol to run properly. When the first Meerkat is launched, it expects that the buddy Meerkat will be launched in the future. The Meerkat continues to send out buddy request packets until it gets a response, and so if the buddy Meerkat is not launched properly on the expected port, then the current Meerkat will not move onto the `WAIT_ON_DATA` phase. In the same way, the Meerkat system will not run properly if the IP address or port number expected is not the actual IP address or port number of the buddy. The expected use of the Meerkat system is to communicate IP addresses and port numbers (or the use of default ports) between the users setting up both systems with Meerkat. Upon decision of a port and communication of the IP address, the users can launch their instances of Meerkat separately, but the connection to send packets on loss will not be configured until both the current Meerkat and buddy Meerkat systems are launched.

The source code for the Meerkat program is available at `https://github.com/vanessaforney/meerkat`.

2.1 Implementation Terminology

The keywords of the Meerkat protocol implementation are defined below.

current Meerkat The instance of Meerkat that is running on the current system.

buddy The system running Meerkat that the current Meerkat will send a loss packet to.

clan The Meerkats that claim the current Meerkat as their buddy. Meaning, any of these Meerkats may send the current Meerkat a loss packet.

loss packet A packet sent when a Meerkat loses power.

buddy request packet A packet sent when a Meerkat is requesting a buddy. Receiving a buddy request packet results in sending a buddy confirmation packet in return and adding the Meerkat to the current Meerkat's clan.

buddy confirmation packet A packet received by the current Meerkat that confirms its buddy request packet was received. This packet marks the transition from the `WAIT_ON_BUDDY` phase to the `WAIT_ON_DATA` phase and begins querying the GPIO pin. The current Meerkat can now send loss packets successfully when it loses power. Also known as the `BUDDY_OK` packet.

assist The term used to describe what to do when a Meerkat receives a loss packet from a Meerkat in its clan.

`WAIT_ON_BUDDY` The initial phase where the current Meerkat is waiting for a buddy confirmation packet.

`WAIT_ON_DATA` The secondary phase where the current Meerkat is waiting for incoming buddies (through buddy request packets) or loss packets from Meerkats in its clan.

GPIO pin General Purpose Input/Output pin on the Raspberry Pi that the Meerkat reads from to determine if it has experienced loss.

2.2 Phases

The Meerkat system consists of a `WAIT_ON_BUDDY` phase for initialization of the buddy, followed by a `WAIT_ON_DATA` phase. The program begins by verifying the ports, a properly formatted IP address, and a valid binary callback. A Meerkat is created with these specified values, and a `sockaddr_in` is initialized in order to avoid delay when sending packet on power loss. The Meerkat is then configured to send UDP packets after a socket descriptor is bound to the port value. UDP communication between Meerkats was chosen because when power loss occurs it is unimportant to confirm receipt of the packet. By the time a packet would be acknowledged in the event of power loss, it is likely that the current Meerkat would have died. Thus, UDP packets are the ideal form of communication for the Meerkats. After configuration, the Meerkat then enters the `WAIT_ON_BUDDY` phase.

2.2.1 WAIT_ON_BUDDY

The `WAIT_ON_BUDDY` phase continues until the Meerkat successfully configures its buddy, or the other Meerkat to send a packet to when loss is experienced. In this stage, the Meerkat continues to send a buddy request packet every second until it receives a confirmation packet from its buddy.

At this time, the Meerkat may receive a buddy request or a loss packet from a Meerkat in its clan. On receipt of a buddy request, the Meerkat will reply with a buddy confirmation packet and add the Meerkat to its clan. On receipt of a loss packet from a Meerkat in its clan, it will assist the Meerkat by initiating the callback binary.

Once the confirmation packet is received, the current Meerkat calls `fork()` to create two processes: one that monitors the GPIO pin status and one that moves onto the `WAIT_ON_DATA` phase. Checking the GPIO pin status includes monitoring the pin to determine the power status of the current Meerkat. When the GPIO pin reads a power loss represented by a non-zero number, the current Meerkat sends a loss packet to its buddy.

2.2.2 WAIT_ON_DATA

The main process moves onto the `WAIT_ON_DATA` phase as soon as the buddy confirmation packet is received. In this phase, the Meerkat may receive buddy request packets or loss packets from Meerkats in its clan. Each of these packets are handled as they are in the `WAIT_ON_BUDDY` phase, where a buddy request causes the current Meerkat to send a buddy confirmation packet and a loss packet causes the current Meerkat to initiate the callback. This time, there is no timeout, and the Meerkat waits until it receives a packet before it takes action.

3. EXPERIMENT

In order to validate that the Meerkat system could successfully monitor and respond to power outages, hardware was needed that would be both inexpensive and resilient to harsh power cycling. With these criteria in mind, two Raspberry Pi model B's were wired together, one acting as the host system who's power would be cut (known as `Death` throughout the experiments) and another to measure the signals emitted by `Death` (this Pi is referred to as `Life` throughout the experiments).

Arch Linux ARM was chosen for both of the Raspberry Pi's due to its low overhead and the team's familiarity with the operating system. All experiments were conducted in the Networks Lab at California Polytechnic State University where the devices could be easily connected to one another. In order to simulate the large capacitors found in server tower power supplies, several different *super capacitors* were used to provide power to `Death`, allowing it to live even after its main power line was cut. Finally, the GPIO pins of both Pi's were used to signal and detect events. As an example of such an event, imagine power being cut to the Raspberry Pi and `Death` finally dying once its capacitors had drained.

3.1 Experiment Terminology

The keywords of the experiment are defined below.

`Death` The name given to the experimental Raspberry Pi meant to simulate a power failure.

`Life` The name given to the monitoring Raspberry Pi meant for measuring the status of `Death`.

super capacitor A type of capacitor characterized by extremely high capacitance and low operating voltage. Operates off of differing physical principles and construction than other types of capacitors.

Power failure The loss of stable power to the Raspberry Pi resulting in the discharging of the capacitors to maintain power. Also known as power loss.

CPU shutdown The shutdown of the Raspberry Pi CPU after a power failure, thus ending all operations.

hardware shutdown The shutdown of the Raspberry Pi board after a power failure. Note that the board stays powered a measurable amount of time after the CPU shuts down.

3.2 Hardware

The design of the hardware was done in several stages to allow for the lessons of past experiments to be considered in future iterations. The first was the stable power GPIO test to prove the experimental design, followed by the Switchable Power GPIO Test to prove the capacitor concept. The final test, Switchable Power Network Test, allowed for accurate measurements and supported the final implementation of Meerkat.

3.2.1 Stable-Power GPIO Test

The first iteration was designed with the goal of simplicity, in order to prove `Death` could be monitored by `Life`. This iteration is shown in Figure 2. The design utilized the GPIO pins for communication between the Pi's and did not feature the capacitors or power switches, with power failure being simulated by software on `Death`.

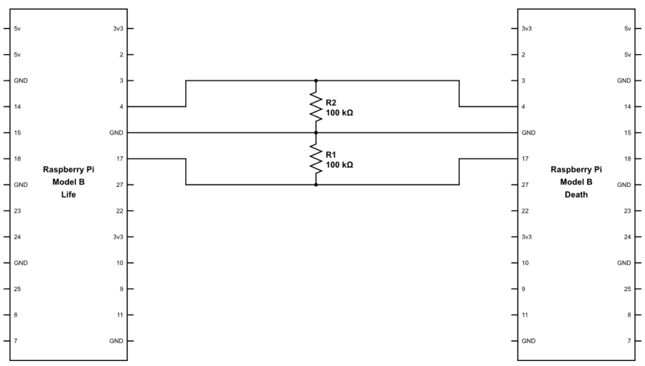

Figure 2: The stable power GPIO test circuit is designed to allow for Life to monitor Death using the GPIO pins, no control of power is implemented in this design.

Since the purpose of this circuit was to provide a testbed for the GPIO library software and experiment design, it is meant to be as simple as possible. The two Raspberry Pi's have a tied ground in order to provide a mutual reference point. Two communication lines between the GPIO pins 4 and 17 allow for a signal upon power failure and a signal for hardware shutdown respectively. Pull-down resistors are included in the design in order to ensure that there are no "floating" pin signals. An experiment begins with `Death` setting GPIO pin 17 high to allow `Life` to see it has power. Then `Death` alerts `Life` of a power failure on GPIO pin 4, and sleeps for a user defined amount of time before simulating hardware shutdown by setting GPIO pin 17 low.

3.2.2 Switchable-Power GPIO Test

Having proven the monitoring concept, the second iteration was designed to allow for a manually switched power failure to prove the concept of a temporary capacitor power source. Once again, the GPIO pins were used for communication between the Raspberry Pi's to simplify the design. Several capacitance values were tested with this hardware iteration, allowing for estimates of the capacitance necessary to provide sufficient time for operations on the dying Raspberry Pi. The tests with 0.5 mF, 1 mF, 2 mF and 4 mF capacitance levels failed to power the Raspberry Pi for any measurable amount of time. Thus, the experiments were reproduced with series and parallel combinations of 1 F, 1.5 F, and 2.5 F super capacitors which displayed sufficient energy capabilities.

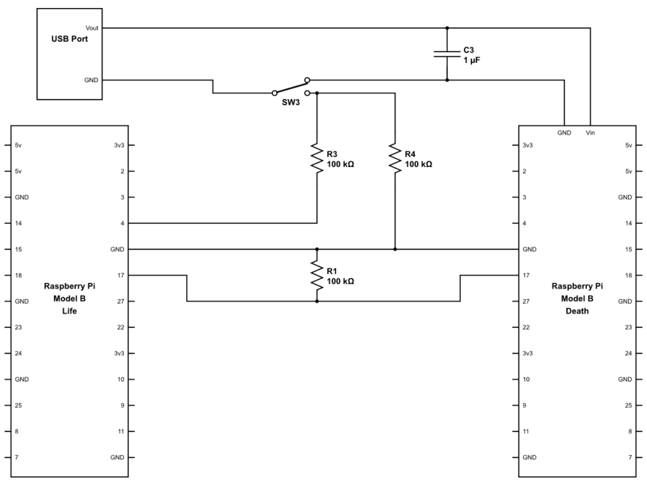

Figure 3: Switchable power GPIO test designed with a circuit that allows for `Life` to monitor `Death` using the GPIO pins, with the power to shut off `Death` by a manual switch.

The circuit to support the experiment can be broken down into two main parts shown in Figure 3. The first part, shown at the top of the figure, is the USB port plugged into the `Death` with a switch and capacitor spliced into the line. This allows for power failure to be simulated with the charged capacitor placed in parallel with the Raspberry Pi. The second part, shown near the center of the figure, is a rerouting of current upon power failure to signal the GPIO pin that `Life` is monitoring. Both a pull-down and a current limiting resistor are included in order to provide faster switching and protection. This system replaces the communication line on GPIO pin 4 from Figure 2. GPIO pin 17 is maintained to signal the hardware shutdown of `Death`. Therefore, the experiment begins, once again, with `Death` setting GPIO pin 17 high. When the user switches off power to `Death`, both `Life` and `Death` are alerted on power failure through GPIO pin 4 going high. `Life` waits for GPIO pin 17 to fall when `Death's` hardware shuts down and can no longer power the pin.

3.2.3 Switchable-Power Network Test

A network was introduced in iteration three to fully replace the GPIO communication system. The circuit shown in the Figure 4 replaces the GPIO pin 17 communication lines with a packet based network (not shown), leaving only GPIO pin 4 for interrupt based power loss notification. Therefore upon power failure GPIO pin 4 alerts both `Life` and `Death` of the event. `Death` immediately responds by continuously sends loss packets to `Life`, which `Life` will monitor and timestamp. The timestamp of the last packet is considered to be the timestamp of `Death's` CPU shutdown and, when compared to the power loss signal, results in the time of life of the Raspberry Pi after power loss. Thus the circuit has three functions: provide a capacitor to power the Raspberry Pi upon power failure, a means of controlled power failure, and hardware support to alert both Raspberry Pi's when power failure occurs.

Projecting the custom hardware designed for this project onto a real system is a straight forward process. The first function of the hardware is done by the capacitance within the systems power supply and therefore already incorporated into all computing systems. The second function of the hardware was only necessary in an experimental setup and would be replaced by the unpredictable and uncontrolled power failures of real systems. The final function of the hardware is the most difficult. However, high end servers and computers often support a computing interface for notifying the system of power instability and failure.

As well as just being experimental, our implementation of Meerkat runs on this hardware design.

3.3 Software

This section details the software used in the experiments, including the optimization and application of the GPIO library and the adaptations to the original Meerkat software for the purpose of these experiments.

3.3.1 GPIO Library

GPIO pins were used to monitor all of the state changes throughout these experiments. In general, `Death` had a single GPIO pin which was set to high during the duration of the experiments. `Life` had two wires, one that monitored when `Death's` power was cut, and another to measure when `Death's` GPIO line went low. The low reading signified that `Death` had powered off after draining its capacitors. In all cases, a GPIO library for the Raspberry Pi was needed so that these events could be measured and responded to. Prior to discovering that simple capacitors were not able to supply sufficient amperage, the time scale of all of these events was thought to be on the order of microseconds, and so a fast GPIO library seemed imperative.

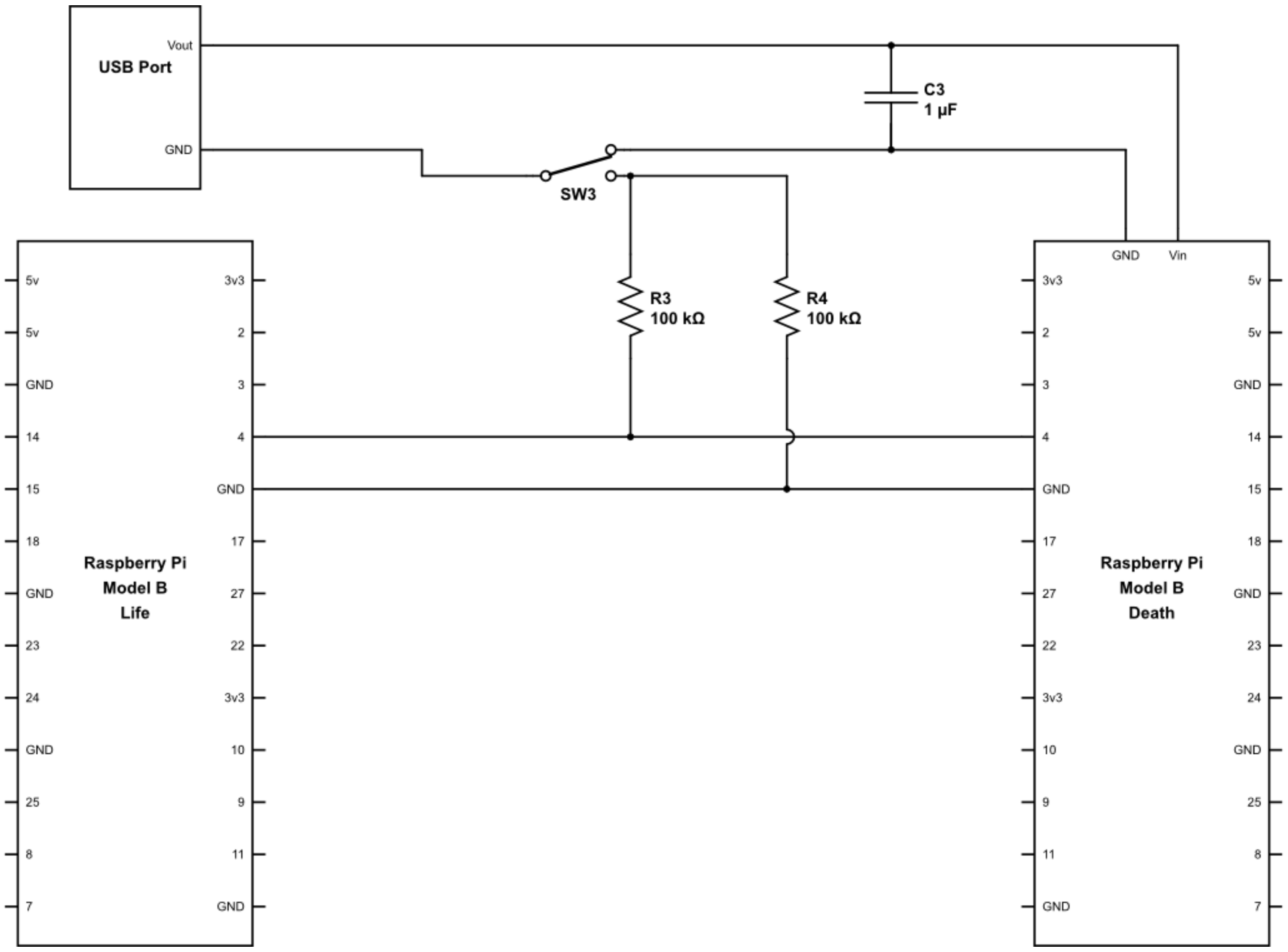

Figure 4: Manually switchable GPIO test designed with a circuit that allows for `Life` to monitor `Death` using the GPIO pins, with the power to shut off `Death` by a manual switch; in addition to this circuit, the Raspberry Pi's are connected through a switched network

The first attempt at creating a library went about enabling, reading and writing GPIO pins by going through the Raspberry Pi's file system, essentially treating the pins as files. This system, while effective, showed a 1 millisecond delay due to the overhead of the file system reads and writes, and was deemed too slow. In its place, a new GPIO library was developed which interacted with the GPIO pins directly using `mmap`. This library was based heavily off of the work of Pieter-Jan Van de Maele [4] with small alterations made along the way that were more specific to this test setup.

By mapping the GPIO pin's physical memory directly into the memory of the program, the latency of GPIO alterations was on the order of a handful of instructions (as opposed to a handful of system calls). Expanding on this idea, macros were used in place of function calls throughout the library as well, resulting in an average delay of 200 microseconds. Ultimately, this speed increase ended up having minimal impact on the final project (due to other hardware issues), but it did allow for higher precision data collection and measurement of events that occurred faster than a millisecond.

3.3.2 Experimental Meerkat

The Meerkat system used the hardware setup described in Section 3.2.3. As described, the Raspberry Pi's monitor GPIO pin 4 to determine their status. The hardware setup forces one Meerkat to be `Life` and one Meerkat to be `Death` for the purpose of our experiment, despite the fact that the Meerkat protocol was designed to act as both `Life` and `Death` on a normal system. Thus, the following adaptations were made to the Meerkat program.

One adaptation was an additional flag to specify if the Meerkat should act as `Life` or `Death`. As `Life`, Meerkat follows a specific course of action upon receipt of the buddy confirmation packet. The `Life` Meerkat splits into two processes similar to the original Meerkat, one which monitors the status pin to determine the exact time of power loss that `Death` experiences, and one to continue onto the `WAIT_ON_DATA` phase to wait for an incoming loss packet from `Death`, a Meerkat in its clan. As `Death`, Meerkat will move into a new phase: the `CHECK_GPIO_STATUS` phase in which it continues to monitor GPIO pin 4 and then continues to send as many loss packets as possible when the pin goes high. The high signals that the Meerkat has lost power, and we measure how many packets were able to go through during this time at varying capacitance levels, described in our Results section.

For the timing, the initial time of loss is recorded on the `Life` Meerkat, representing the time in which power was lost to the `Death` Meerkat. The initial time is outputted, along with the timestamp of each received loss packet. The timestamp of the last loss packet represents the last time of successful communication with `Death`. The difference between these values represents the approximate amount of time `Death` had between initial power loss and CPU shutdown.

4. RESULTS AND DISCUSSIONS

The hardware up time with varying capacitance levels, CPU up time with varying capacitance levels, the effect of concurrent processes on sending packets, and actual Meerkat notification (a single packet) were all tested. The real implementation of Meerkat was altered to record timing data.

4.1 Timing Data

Once the circuitry had been assembled, several tests were conducted to see how the Raspberry Pi's performed under varying conditions. The first test monitored how long `Death` could power a GPIO pin while running only on power supplied by the super capacitors. To do this, one of `Death's` GPIO pins was set high (5V) and would only go low once `Death's` power had dissipated. `Life` was set to monitor this and 4 runs were conducted at each of the capacitance values shown in Figure 5. As the figure shows, there is a linear relationship between capacitance and operating time.

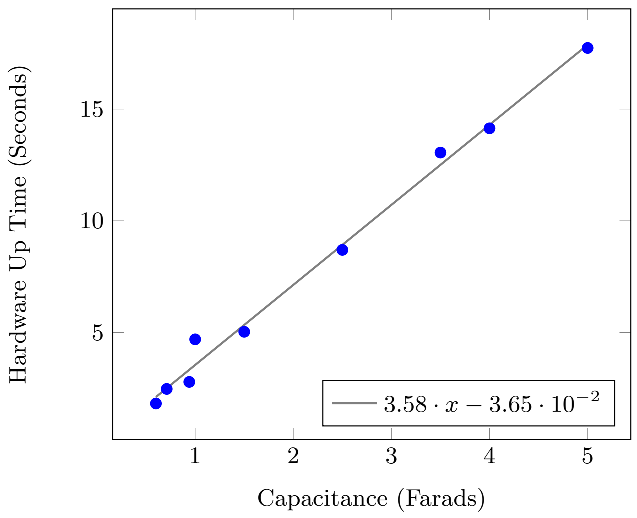

Figure 5: The average time the hardware is alive in seconds compared to the capacitor used in Farads on the Raspberry Pi over a series of four runs

This test was mainly used to verify characteristics of the physical test setup and illustrates some of its key limitations. The fact that there is a linear relationship between capacitance and operating time, even though capacitance decays in an exponential fashion, means that the Raspberry Pi is only being powered for a short duration of the capacitor's total discharge. Taking into account that the Raspberry Pi can only operate if its input voltage is between 4.75V and 5.25V [5], this suggests that the Raspberry Pi's up-time is localized to the initial slope of the capacitance curve, as shown in Figure 6.

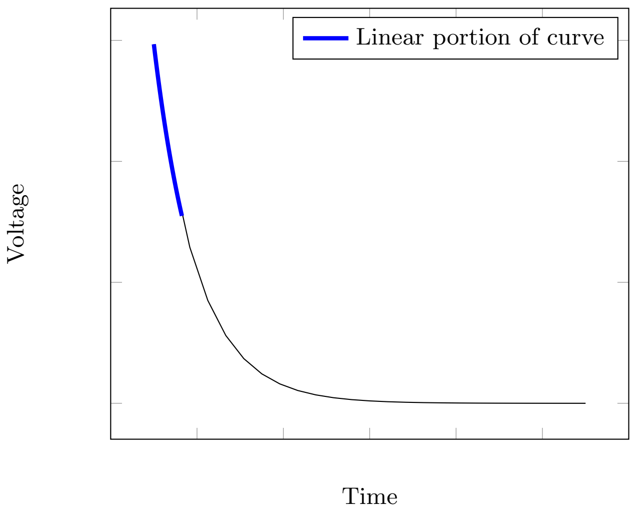

Figure 6: Linear portion of capacitance curve

Also noteworthy is that while the Raspberry Pi's CPU will cease operation around 4.75V, it will continue to output voltage to the GPIO pins until it completely looses power. In this sense, *completely* refers to the minimum voltage that a pin can sense and register as *high*; in the Raspberry Pi's case this voltage is roughly 1.3V [6]. As such, the measured up-time from simple voltage readings will be much greater than the actual up-time of the CPU itself.

In an attempt to measure the up-time of the CPU, a second test was run using the Meerkat software suite. In this test, `Death` would notice that it had lost main power via a GPIO pin connected to the power switch. When power was cut, `Death` would start sending packets to `Life` as fast as the network/processor would allow. `Life` would measure the time between the power being switched and the last packet received, and would consider the difference to be the effective up-time of `Death`. The results of this experiment are shown in Figure 7.

Note that the up-time for these trials is roughly half that of the previous trials which only measured the GPIO pin voltages, which supports the previous assertion that the CPU up-time would be less than the GPIO powered time. One possible pitfall of this experiment is that it relies completely on the Meerkat program, a program which is being run as a user application on the Raspberry Pi and as such is subject to context switches and preemption by the OS scheduler. In all of the experiments up to this point, only a single user application was being run on each Raspberry Pi, and simple timing tests showed that there was little appreciable delay in response under this setup.

To explore the idea of the scheduler's impact on performance further, an experiment was conducted which kept capacitance constant and varied the number of concurrent processes running on the processor. The program chosen to

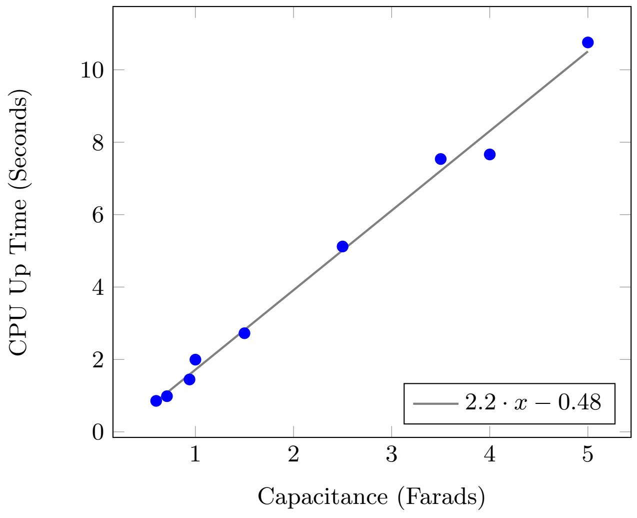

Figure 7: The average time the processor is alive in seconds compared to the capacitor used in Farads on the Raspberry Pi over a series of four runs

run alongside Meerkat was a simple executable that incremented a counter and wrote it to a file. This was chosen because it was both simple and it was also thought to be unlikely that the compiler would optimize it away (as it can do when there are seemingly senseless loops in a program). The results of this experiment are shown in Figure 8 for a capacitance value of 0.6F.

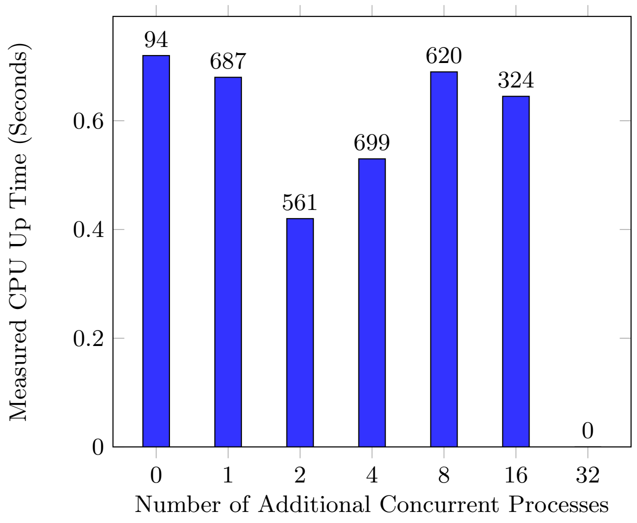

Figure 8: Raspberry Pi CPU up time for 0.6F capacitor, along with the number of packets received (shown as values atop each bar

Of note, Figure 8 plots the number of extra concurrent processes running alongside the Meerkat program (which is why the axis starts counting from 0 instead of 1) against *effective* CPU up time, along with the number of packets received (shown as values atop each bar). The term *effective* is used here because the measured CPU up time is contingent on the Meerkat program being scheduled for the duration of the power outage event.

Figure 8 shows that there is a drop off of CPU up time from 0.7 seconds to 0.4 seconds when more than 2 additional processes are run alongside Meerkat. At first it was thought that this was due to the scheduler preempting the Meerkat program in favor of one of the other running processes (and so Meerkat would only get to run for a small portion of the actual up time). However, this theory was befuddled by the increase in effective up time when more processes are introduced (up to the cutoff at 32 processes where Meerkat often does not get scheduled in time to run). Another notable data point is that when Meerkat is running alone it is able to send out 94 packages on average, however, when a single extra process is introduced, the effective packet output jumps to 687 packets and does not drop beneath 100 until 32 processes are run. This is most likely due to the buffering of UDP packets on `Death`; if Meerkat is the only process fighting for the CPU time then it is never preempted, and as such its UDP packets fill a buffer prior to being sent. However, when multiple processes are introduced, the Raspberry Pi scheduler context switches between them at regular intervals. During these context switches, all of the buffered UDP packets must be flushed (or lost). We believe that in the 0 additional processes case, `Death` takes roughly 0.7s to both fill the UDP buffer and then send 94 additional packets, whereas in the multiple additional processes cases, `Death` is context switched multiple times, resulting in several buffer flushes and ultimately a higher packet throughput.

4.2 Software Results

The final version of Meerkat was the same software on both `Life` and `Death`, and sent a single UDP packet to the other Raspberry Pi. The receiving Pi executed the user-defined callback executable successfully. This succeeded at the lowest capacitance (0.6F) just as measuring the time of Life in the previous section did. However, as mentioned previously, this did require a super capacitor to work at all.

5. CONCLUSION AND FUTURE WORK

We described and implemented a protocol for power loss handling with a pair of machines. This protocol allows a user-defined action to be performed by the surviving system in direct response to the other system going down. We also simulated power loss conditions in an experiment by triggering a GPIO pin and shutting off the power to one test machine. Our results show how Raspberry Pi's perform when augmented with a range of capacitors with the hopes that this data can be extrapolated to more power-intensive setups.

Future work can be done to improve the design of both the experiment and the Meerkat system. For the former, using a MOSFET to control power to `Death` in the place of a manual switch would allow for `Life` to electronically contorl power to `Death`, thus completely automating the experiments (allowing for the collection of more data). Furthermore, soldering the test setup would allow for greater accuracy and stability in the test hardware. For the latter, handling kernel level power loss signals from the operating system instead of from a GPIO pin is a key goal. Testing on desktop machines for exact capacitance values would give more useful data for a Meerkat user to implement. Lastly, including useful callback binaries such as email and/or smartphone notification is desirable.

Acknowledgements

We would like to thank Dr. Phillip L. Nico for inspiration and guidance during this project.

6. REFERENCES

[1] Department of Energy. Fuel cells in backup power applications, 2005. Accessed: 2015-05-26.

[2] Microsoft Corporation Phoenix Technologies Ltd Hewlett-Packard Corporation, Intel Corporation and Toshiba Corporation. Advanced configuration and power interface specification 5.0a, 2013. Accessed: 2015-05-26.

[3] Benjamin Erb. Concurrent programming for scalable web architectures, 2012. Accessed: 2015-05-26.

[4] Pieter-Jan Van de Malae. Low level programming of the raspberry pi in c, 2013. Accessed: 2015-04-26.

[5] Acceptable input voltages for power?, 2013. Accessed: 2015-05-26.

[6] Gaven MacDonald. What are the input voltage thresholds for the raspberry pi's gpio pins?, 2013. Accessed: 2015-05-26.