# FAWN-DLi: A Data Library for a Fast Array of Wimpy Nodes

Jonathan Gluck

June 12, 2012

**Abstract**

Energy efficient computing is growing in importance as a means to cut costs and conserve power. One recent approach to reducing the energy cost of data management is to use large clusters of energy efficient but slow computers rather than smaller clusters of more conventional high-power machines. For these systems to perform similarly to high-power machines but on slower computers they often support only the simple data interface of a key-value storage system, which can incur high communication costs if sophisticated data operations are needed. In this project we extend one existing low-power system, FAWN-KV, to support more sophisticated sever-side data operations. We added support for lists, sets, and maps and compared the performance of our modifications to the performance of unmodified FAWN-KV on data operations like those used in modern systems. Our modifications improved performance by up to a factor of three and outperformed the basic system in all tests.

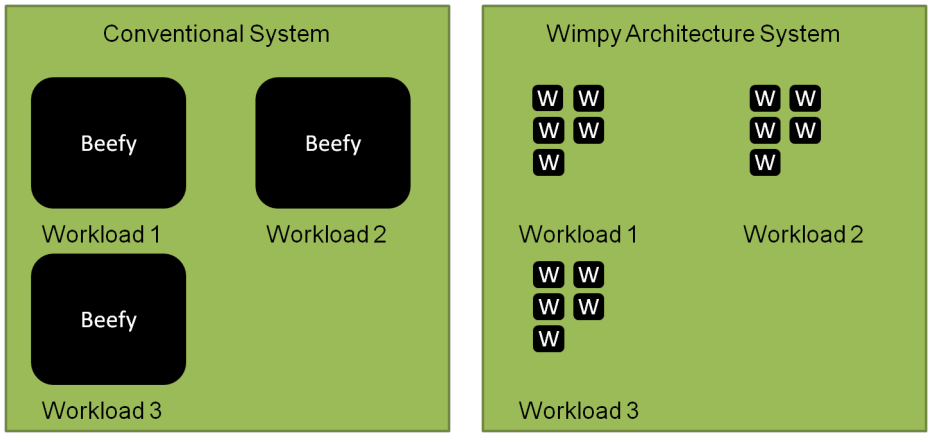
## 1 Introduction

Power efficiency is becoming an issue of great importance to modern computation. Operating costs of computation have been steadily increasing with the growth of data-centers. Much of this increasing cost is due to power consumption, as up to 50% of the three year cost of owning a computer is attributed to power consumption [1]. Recent trends show an increase in power consumption due to computation; in the United States, energy consumption due to operating and cooling data centers increased 56% over 2005 to 2010 [9]. If these trends of growing reliance on computation continue, then solutions addressing power consumption must be found.

One way to address power consumption is to imagine alternative system architectures that are able to consume less power. These alternatives are often simple prototypes that attempt to maximize power efficiency by using novel hardware. One such alternative solution is to distribute workloads over arrays of slower machines.

This low powered distributed architecture (which I will call the "wimpy" architecture) seeks to address increasing power consumption by distributing existing jobs over low powered machines. Currently, computational jobs are run on very fast, very energy intensive

machines. These powerful machines; however, spend much of their time idle, waiting on system resource availability. While these computers are idle, they are still consuming much of the power that they would even when at peak efficiency. Wimpy computation is based on the idea of exploring the possibility of running these same jobs distributed onto clusters of slower, less powerful computers. By utilizing slower processors, wimpy machines spend less time idle waiting for system resources.This model is shown in Figure 1. These slower computers are paired with fast storage mediums so that the time spent waiting on system resources is even lower. By running the same job on less powerful machines in a similar amount of time, it is possible to reduce energy consumption. The wimpy computation architecture proposes a simple solution to reduce power consumption.




(a) (b)

Figure 1: Conventional system (a) vs. wimpy hardware system (b), Note that workloads which were previously one to a machine (a) are now distributed across a small cluster of wimpy nodes marked by "W" (b)

Although wimpy hardware may be able to lower power consumption, it has a few downsides. Wimpy systems are not just slower than their non-wimpy counterparts; they are asymmetrically slower. Certain operations, such as modulo are slower than would be expected, even when the slower overall processing speed is taken into account. This is because, in the design of wimpy processors, much of the machinery found in a modern processor is removed. Because certain operations are slower than others on wimpy machines, software must be written specifically, incurring additional development costs, with wimpy machines in mind. These wimpy software solutions must be balanced for the system, avoiding expensive operations that could overtax the wimpy processor.

We seek to explore the issue of wimpy system complexity. Due to the need for fitted software, current prototype wimpy systems tend to be very simple, providing limited data interfaces. These limited data interfaces leave out many features that users might want. Additionally, due to a lack of sophisticated functions, data flow can become needlessly overwrought. We explore this idea of wimpy system complexity, keeping an eye to creating a more usable data interface for one prototype system.

We also wish to see if more complex workloads are feasible for wimpy systems. In order to accomplish this goal, we take one wimpy prototype, FAWN-KV (Fast Array of Wimpy Nodes - Key Value), and attempt to create a high-performance set of features that extend its current simple data interface. FAWN-KV is a wimpy key-value store. In our work we extend its data interface by implementing basic container interactions. These additional data interaction methods are contained within FAWN-DLi, (Fast Array of Wimpy Nodes - Data Library) a data library built with minor modifications to the existing FAWN-KV framework. With FAWN-DLi we test whether it is possible to take a simple wimpy prototype system and integrate additional complexity in the form of features desirable to programmers. An important aspect of our work with FAWN-DLi is gauging the ability of these wimpy machines to handle the introduction of more complex workloads while remaining power efficient. We implement FAWN-DLi to explore this problem.

At the writing of this paper, we have implemented FAWN-DLi a data library built on top of FAWN-KV that provides container data interactions. We have tested the effects that FAWN-DLi has on system performance, both on wimpy and on non-wimpy hardware. To test these effects we have implemented a series of micro-benchmarks as well as a more realistic simulated workload. We have found that FAWN-DLi outperforms unmodified FAWN-KV in all of our tests, and is able to perform comparably to modern database systems.

If current trends in energy consumption continue, energy efficient computing is going to become paramount. FAWN-KV is a promising prototype system that adopts a wimpy computation model. As a balanced prototype, FAWN-KV's data interactions are very simple. We seek to expand upon FAWN-KV's existing data interactions to provide a more desirable user experience. Our work with FAWN-DLi shows that it is possible to implement additional complexity on top of these simple systems, creating energy efficient software with desirable feature sets.

In this paper we examine how FAWN-DLi fits into the field of database research. We then discuss the FAWN project and the system designs of both FAWN-KV and FAWN-DLi. We discuss our experimentation methodology for FAWN-DLi and the results of those experiments.

# 2 Related work

In the design of FAWN-DLi we touch upon many fields, as such, the areas from which we draw inspiration are many. The first, and most direct of these sources are the wimpy systems on which FAWN-DLi is based. Such systems include FAWN-KV, SkimpyStash, and SeaMicro [1, 3, 4]. We postpone a more detailed discussion of FAWN-KV until the next section, as it is the system that we directly modify. Because wimpy systems are often very simple, we are interested in the ability of these systems to handle more complex workloads. In Lang 2010 [10] this idea of complex workloads on wimpy machines has been previously explored, albeit with slightly different axes of complexity. When implementing FAWN-DLi we were faced with several design decisions, one of these decisions was on how to store our FAWN-DLi containers. We adopt a document based model for container storage on FAWN-

DLi, similar to that of CouchDB [5] or Redis [12]. In the following section, we discuss these areas from which FAWN-DLi can trace its beginnings.

## 2.1 Wimpy systems

The concept of replacing high powered processors with many less powerful processors has been explored in several previous works, including the FAWN project, SkimpyStash, and SeaMicro [4, 16, 17]. The idea behind these wimpy systems is that the time spent idle by the processor can be reduced by reducing the speed of the processor. Because the throughput of a system may be attributed to its slowest component, and because that slowest component tends to be storage medium, slowing down the processor has little effect on the throughput of the system [16]. Idle cycles are essentially wasted energy, so the ability to handle the same job with fewer idle cycles is appealing. Even with their slower processors, wimpy systems still experience idle cycles spent on the I/O gap (The time spent waiting on a read request to disk). In order to further reduce idle time, The FAWN project's FAWN-KV makes use of SSDs as its storage medium. Because SSD's are faster than disk, this reduces the I/O gap further. Using SSDs to reduce resources spent waiting on the I/O gap has been explored previously [4, 11, 16, 17]. The strengths and weaknesses of the hardware used in Wimpy systems informed the design of FAWN-DLi because FAWN-DLi is intended to run on wimpy platforms.

The software on these wimpy systems is just as important as the hardware. Because of differences in processor speed as well as storage medium, off the shelf software may not run as effectively [8]. Because of the asymmetrical slowdown for certain operations on wimpy machines, wimpy projects must develop software that avoids these costly operations. FAWN-KV was designed with wimpy architectures in mind, and as such is a distributed hash table highly tuned for wimpy systems. In particular, the majority of the computation in FAWN-KV takes place on the non-wimpy nodes in the system. The wimpy nodes are left with less computationally complex jobs so that they are able to operate efficiently [17]. Another example of a system built from the bottom up for wimpy machines is WattDB, an energy efficient wimpy database management system that is able to dynamically scale power usage. In order to prevent any wimpy node from becoming overloaded, query processing and storage / retrieval tasks are delegated to separate nodes [14]. With FAWN-DLi we are particularly interested in the amount of complexity we can coax out of these wimpy nodes, and if that complexity might overload their less powerful processors.

## 2.2 Complexity of interface

We are interested in whether it is possible to expand FAWN-KV's simple data model with added complexity. Relational databases have largely dominated the field of data storage for the past forty years [15]. This is in no small part because of how flexible the relational data model has proven to be. Relational data models opt to store entities in a row-column data structure. Each individual stored comprises a row, while the attributes of those individuals are stored in columns.

In relational systems, such as MySQL or PostgreSQL [2, 7], users are able to modify both the storage model of the system and the client data interface. In both of these systems users may perform arbitrary queries so long as they are stated in their respective query language. This allows for upgrades to the system to be performed with little to no downtime. While this flexibility is attractive, it comes at a cost. In order to efficiently retrieve data, relational systems such as these must create and maintain computationally expensive indexes.

Wimpy systems such as FAWN-KV may not be able to afford the high processor cost of building and maintaining MySQL's or PostgreSQL's indexes. FAWN-KV's adoption of a NoSQL data model, providing only a `Put/Get` data interface, trades flexibility for ease of processing. This is not necessarily a bad trade-off. FAWN-KV seeks to address a set of well defined workloads, allowing quick storage and retrieval of data in a random access pattern [1]. If all that is required for a given workload is the ability to quickly store and retrieve data, then FAWN-KV's limited data interface is more suited to the job than the more flexible interface of MySQL or PostgreSQL.

We seek to extend FAWN-KV's data interface with more flexibility; however, we choose not to implement arbitrary queries like those found in MySQL or PostgreSQL. Instead we implement a framework that allows users to define server side data interaction methods at compile time. In contrast to the data interaction models of relational databases, FAWN-KV's approach is designed to be pre-defined to remove the need for a query parser.

## 2.3 Document based data

FAWN-KV currently stores data in its distributed hash table as untyped blobs. This method of storage is often referred to as "document based storage". One popular system which adopts a document based storage model is Apache's CouchDB [5]. CouchDB allows its users to define queries as arbitrary views through which it then returns its documents. In essence, CouchDB users may store data in any format they like, so long as their client knows how to phrase a request for a given piece of data. This differs from FAWN-KV's strict data model which adheres only to `Put` and `Get` requests. This is because FAWN-KV is blind to the data stored within it. While CouchDB allows its users to interact with its stored data; FAWN-KV requires that users only interact with data locally.

Another document based storage system is Redis [12], a structured key value store. Redis is a key value store in which the value may be a string, a list of strings, a hash, a set of strings, or a sorted set of strings. Users are given an array of server side tools, such as pushing to a list or computing set unions. Redis stores its dataset in memory, occasionally pushing to disk. This data model also differs from that of FAWN-KV in that the system is not blind to its data on the serve side. Users may interact with data in meaningful ways on the server.

We attempt with FAWN-DLi to bring FAWN-KV a little closer to CouchDB and much closer to Redis by allowing clients to interact with data on the server side. A key difference between Our work and CouchDB is that users of CouchDB may phrase arbitrary queries. We only support predefined methods implemented in FAWN-DLi, as arbitrary queries might incur too heavy a computational cost. In FAWN-DLi we maintain the document based

storage model by storing the entire container as a value in the key value pair. In contrast to CouchDB's arbitrary queries, Redis provides a set of predefined atomic methods with which data may be accessed and mutated. This is very similar to the model we wish to achieve with FAWN-DLi, as it allows for efficient carrying out of data operations that users might want.

### 2.4 Wimpy systems, less wimpy workloads

It has been shown that small arrays of wimpy nodes are more efficient than a single non-wimpy machine.[18] However, work has been done that suggests these arrays of wimpy nodes show diminishing returns as they scale out [8, 10]. In each of these works, the ability of wimpy clusters to scale up is brought into question. Each paper submits that, due to Amdahl's law, eventually the marginal benefit of adding another wimpy node to a cluster will be overcome by communication overhead. These works are interesting because they both explore whether the wimpy hardware architecture is able to handle large, real world situations. In both of these works the authors eventually come to the conclusion that network overhead causes the diminishing marginal returns of increasing the size of a wimpy cluster, limiting their ability to handle large jobs.

With FAWN-DLi we are also interested in how FAWN-KV scales; however, we are interested in how FAWN-KV accepts additional computational complexity. We seek with FAWN-DLi to explore this scalability and to ascertain whether it is possible to expand these simple wimpy systems with additional complexity without incurring too high a computational cost. Additionally, because of the above works, we attempt with FAWN-DLi to remove any unnecessary transfer of data from the data interactions allowing for larger FAWN clusters to be formed.

## 3 FAWN and FAWN-KV

Our work with FAWN-DLi is based on ideas from the FAWN (Fast Array of Wimpy Nodes) project at Carnegie Melon. The FAWN project is a strong proponent for the adoption of wimpy systems as a means of reducing energy consumption. They propose that instead of using few energy intensive nodes, data centers should spread current workloads across large arrays of less powerful wimpy machines. Distributing the workload in this way could increase availability of system resources, reducing system idle time and, in doing so, increase energy efficiency [17].

The FAWN team have recently demonstrated the viability of the wimpy architecture at JouleSort 2010, winning the 10 GB efficient sorting competition [13, 19].

### 3.1 FAWN-KV

Our work, FAWN-DLi, is built on top of FAWN-KV (Fast Array of Wimpy Nodes Key-Value) a product from the FAWN team. FAWN-KV is a distributed hash table built to fit

within the FAWN model of computation. It divides the workload by distributing the storage and retrieval aspect of the distributed hash table across a ring of wimpy machines.

FAWN-KV is targeted to address the needs of large systems that require fast random access workloads [1]. Such systems include Twitter, Facebook, and Amazon. These jobs involve storing large quantities of data without knowledge of which data will be in demand. Current relational systems are ill suited to this random access model, requiring costly indexes to be created and maintained. FAWN-KV is particularly efficient for handling random access patterns because its key-value storage model dispenses with the need for storage indexes, requiring only a hash function to access the correct data on disk.

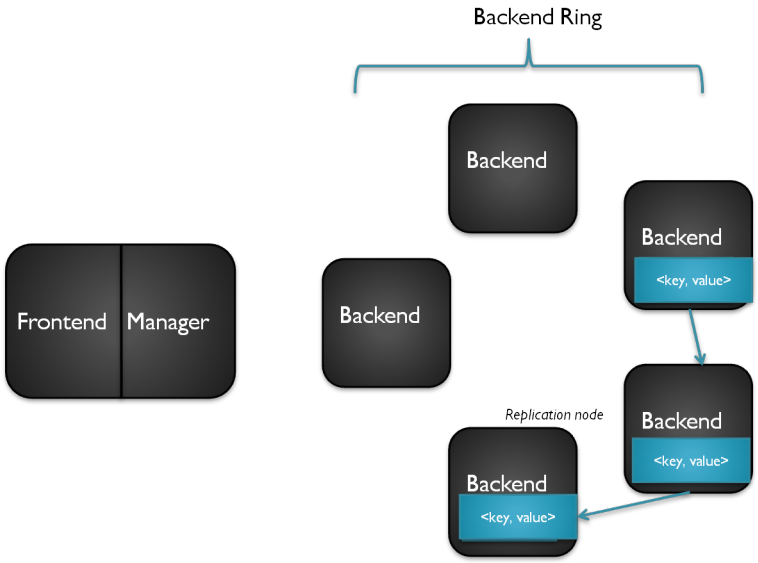



Figure 2: Diagram of FAWN-KV architecture. Note the key value pair replicated on several backend nodes.

FAWN-KV is composed of backend nodes (which host the distributed hash table), as well as one manager node (which manages the backend nodes), and frontend nodes (which interact with clients). An example FAWN-KV setup may be seen in Figure 2. The wimpy backend nodes, are arranged in a "ring formation" [1].Given a maximum key size, there are a finite number of possible keys. We refer to this range of keys as the key space. Each backend node in FAWN-KV's ring is responsible for its own division of the key space, as well as the data replication for some number of nodes before it in the key space [17]. The state of the ring is handled by the manager node, which tracks backend join requests as well as backend failure. Clients interact with the backend nodes through the interface provided by frontend nodes. There may be one (as in Figure 2) or many frontend nodes. The frontend and manager nodes receive the most traffic, as such they may be run on non-wimpy machines to mitigate the load.

FAWN-KV adopts a simple data interface with only `put` and `get` commands. This simple interface is perhaps due to an effort by the FAWN team to not overload the wimpy backend

nodes. Data is stored in FAWN-KV as Apache Thrift encoded values at user defined keys. These values are stored redundantly in replication chains. Each node has some arbitrary $K$ replication nodes that follow it in the backend ring. These $K$ nodes are responsible for maintaining an up to date copy of the original node's data. In order to maintain this guarantee, data put on FAWN-KV is only acknowledged as having been successful put after the final node in the replication ring responds to the client. In figure 2 one can see a replication chain where K=2, storing the key value pair put on the first backend node.

FAWN-KV is a highly balanced prototype system that has proven effective in handling workloads for which other systems are poorly suited [1]. Through a series of benchmarks, the FAWN team ascertained that FAWN-KV outperformed traditional architectures for almost all solution spaces [1]. What is remarkable is that these results may be achieved at a tenth of the power consumption of modern high powered machines [1]. The simple put/get interface of FAWN-KV as well as its reliability measures make FAWN-KV an excellent candidate system for modification.

# 4 FAWN-DLi

Our work, FAWN-DLi (FAWN-Data Library), modifies the existing FAWN-KV source code with the purpose of adding additional data interactions. The current implementation of FAWN-DLi provides support for server side interactions with containers (sets, lists, and maps). Instead of requiring users of FAWN-KV to edit containers locally, FAWN-DLi allows server side interactions.

FAWN-DLi is motivated by both a desire to make FAWN-KV a more useful system, as well as a want to remove a potential network communication overhead in FAWN-KV. In addition to the two above goals, we implement FAWN-DLi because we are interested in exploring how well the wimpy hardware architecture is able to scale up its computational complexity.

In this section we discuss FAWN-DLi's features and implementation. We then look at some of the advantages and disadvantages to FAWN-DLi's design.

## 4.1 FAWN-DLi Features

FAWN-DLi provides a suite of functions for interacting with data on the server side of FAWN-KV. In the current implementation of FAWN-KV, users are only able to interact with data by retrieving it and modifying it locally. When they are done with their modifications they must then transfer the edited version of their data back onto the server. FAWN-DLi allows users of FAWN-KV to interact with data directly on the server side using a set of modification commands. In our current implementation of FAWN-DLi we add support for data interactions with simple containers. In particular, FAWN-DLi currently supports interactions with lists, sets, and maps. For each of these containers, FAWN-DLi provides a server side `add`, `remove`, and `contains` method.

Due to how we choose to store containers in FAWN-DLi, which we will discuss in the next section, we are able to make use of FAWN-KV's promise of replication and consistency without additional coding. Whenever a container is modified on the server, it is passed down FAWN-KV's replication chain. Additionally, FAWN-DLi offers consistency for its containers. That is, multiple clients cannot edit a container at the same time, meaning multiple edits cannot overwrite each other. Currently, FAWN-KV cannot make this guarantee.

FAWN-DLi supports three modification methods. Its `add` function allows users to add a value to a container, while its `remove` function allows users to do the opposite. Each of these functions take a key (where the container is located in FAWN-KV) and a value (to either add or remove). FAWN-DLi's `contains` function acts slightly differently for different container types. In the case of lists and sets, it allows users to check for the existence of an element. In the case of maps it will return the value for a given key if that key was found. These three modification methods are implemented in a single function.

## 4.2 Implementation of FAWN-DLi

FAWN-KV has a few interesting design decisions that have implications to its operation and extendability. With FAWN-KV being a simple key value store, we needed to come up with some storage scheme for FAWN-DLi's containers. This decision would affect how FAWN-DLi containers interacted with FAWN-KV's data model. We also needed to define a system for defining modification functions that would be efficient and easily extendible.

When defining how FAWN-DLi's containers were stored it was important to minimize the complexity of their structure. This was important in order to make sure that organizing a container did not overload the wimpy backends of FAWN-KV. Because of this, we chose to store containers simply as encoded values in FAWN-KV's key value pairs. FAWN-DLi's containers are stored as Thrift [6] encoded values. They may contain any Thrift supported data type. This is a side effect of using Thrift as a wire format as other data types are not supported by the Thrift encoder. This choice of storage format allows easy storage of pre-populated containers, as well as retrieval of containers already on the sever. Additionally, it allows for the consistency and redundancy guarantees we discussed in Section 4.1.

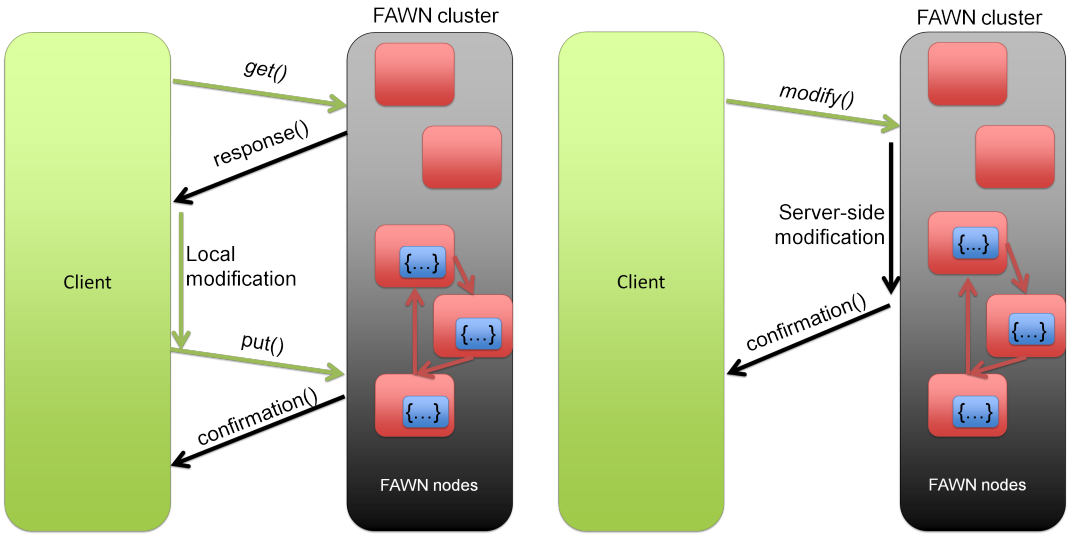
FAWN-DLi is intended to provide a library of data functions to FAWN-KV. We wanted FAWN-DLi to be easily extendible, providing a path to add additional functions with minimal work. In order to achieve this we implemented a single additional method, `modify`. `modify` is a blanket function that is responsible for all forms of server side data manipulation. `modify` takes a key, which corresponds to the distributed hash table entry for the container, and a value which contains the data which will be added, removed, or checked. A single directive byte is prepended to the value. This directive byte corresponds to one of the three container operations. When the backend receives the modify RPC, a helper function checks the directive byte and calls the appropriate backend function. Currently the three supported directives are `"\01"` to invoke `add`, `"\02"` to invoke `remove`, and `"\03"` to invoke `contains`. If a user were to want to add "foo" to the list found at key "bar", then the modify call appear as the following:

```
"modify('bar','\01foo')"
```

We choose to use a blanket function `modify` rather than individual functions for the sake of extendability. With `modify` adding a new container manipulation method is as simple as defining what the next directive would do in the modification helper function.

## 4.3 Advantages of FAWN-DLi

FAWN-DLi's additions to FAWN-KV's data model provide several advantages. The additional data modification methods along with the container consistency promises mentioned in Section 4.1 allow a wider range of workloads, as well as additional functions for users. FAWN-DLi's server side modification methods also allow users to work around a costly network overhead. By allowing users to edit data on the server rather than forcing them to edit it locally, FAWN-DLi prevents the need to transfer the entire value twice. This distinction may be seen in Figure 4.3.




(a) FAWN-KV set modification

(b) FAWN-DLi set modification

Figure 3: Container modification pathway for unmodified FAWN-KV (a) and for FAWN-DLi (b) Note that the set must be retrieved and edited locally in (a) but may be modified remotely in (b)

We can see that in 3(b) the container need not be transferred at all, while in 3(a) the container is transferred twice. This is a more natural way to interact with containers, and it removes a lot of boiler plate code. This change in data transfer has the potential to outperform unmodified FAWN-KV programs.

### 4.4 Disadvantages of FAWN-DLi

Containers in FAWN-DLi are stored as Thrift encoded values in FAWN-KV key value pairs. This choice provides simplicity of implementation, as well as compatibility with FAWN-KV's pre-existing redundancy and consistency promises. However, there is an inefficiency in storing containers in this way.

Thrift represents all containers as lists. Sets are represented as lists with a guarantee that there is only one of each value. Maps are represented as lists of key value pairs with the guarantee that there is only one value for any key. While this encoding maintains all the functionality of the containers, it loses many of the advantages afforded by their data type. As an example, in order to check if a map already contains an object, FAWN-DLi must iterate through each object in the map checking if the key is equivalent. Instead of an O(1) operation, it becomes an O(n) where n is the number of values in the container.

The alternative to storing data as Thrift encoded values in FAWN-KV would be to implement some more complex data arrangement with multiple key value pairs. We choose to store containers as Thrift encoded values partially in order to simplify implementation, and partially because it does not disrupt the existing FAWN-KV's data storage model. If we were to implement some other data storage model, then we would need to implement consistency and redundancy mechanisms aside from those already provided by FAWN-KV. Additionally, FAWN-KV attempts to store its values in sequential order. This makes retrieving containers take the fewest number of page reads as possible. If data were stored in some non-flat format then it might take additional page reads, which would increase query latency. For simplicity we choose to adopt the Thrift encoded container storage model, even with its potential disadvantages.

## 5 Experimentation

In order to test the performance of FAWN-DLi and the impacts it has upon FAWN-KV, we implement a series of benchmarks. Because FAWN-DLi has the potential to unbalance FAWN-KV we pay careful attention to its performance when compared to unmodified FAWN-KV. In our attempt to gauge whether these simple systems can accept additional complexity we move more of the load off of the non-wimpy processor, onto the wimpy backends. This could threaten to overload these less powerful machines. In order to get a better idea of how FAWN-DLi's extra computational complexity affects the system, we run our tests on both wimpy and non-wimpy machines. Additionally, we are interested in comparing these FAWN solutions to a traditional off the shelf relational database.

In this section we discuss our testing systems. We then touch upon each of our micro-benchmarks. Finally, we discuss our more realistic benchmark "Fwitter."

### 5.1 Experimental systems

We perform all of our experiments on two system architectures: wimpy and non-wimpy. We have a cluster of non-wimpy machines, as well as a prototype wimpy node at our disposal.

Additionally, we have access to the FAWN cluster at CMU, which we use to run our larger benchmarks.

The wimpy node that we utilize for these experiments is equipped with an Intel D510 Atom processor. This system is running with the maximum supported 4 GB of memory. As a storage medium this machine uses an 120 GB Intel 510 series SSD. This system is installed with the latest major revision of Ubuntu's desktop version 11.10. This machine is not tuned for research; however, the hardware roughly matches the system specifications for a wimpy node.

The non-wimpy nodes that we use in these experiments are connected via a local network. These machines are equipped with AMD Opteron 2387 processors. They are equipped with 8GB of memory. As a storage medium these machines use 500 GB 7200 RPM Seagate Barracuda hard drives. They are all installed with a recent release of Ubuntu's desktop, version 11.04. These machines are public and are often in use by multiple users; however, we attempt not to perform experiments while other users are using these machines.

## 5.2 Micro-benchmarks

In order to test the individual performance of our modification methods, we implement two micro-benchmarks. These micro-benchmarks are performed in various hardware and system configurations so we can better understand the effects that these methods have on the system. Because our micro-benchmarks are focused solely on the effect of these methods, they are not based on realistic workloads.

The first of our two micro-benchmarks is concerned with determining how FAWN-DLi's container operations compare to the unmodified FAWN-KV equivalents. This micro-benchmark is meaningful because it compares the relative performance of each method of container manipulation. For both FAWN-DLi and FAWN-KV an empty set is created. Ten objects of length 8,192 bytes are added to the set one at a time. For our micro-benchmarks we choose to use randomized strings as our arbitrarily sized data objects. Each object is then checked with each implementation's respective contains method. Finally, each of the ten objects is removed from the set one at a time. This micro-benchmark is performed both on our wimpy machine as well as our non-wimpy machines.

The second micro-benchmark is performed as the first, but this time varying the size of the objects stored. We are interested in seeing how each of these implementations scales with the size of the objects being stored. FAWN-DLi should show an increasing advantage as data sizes become large because of the inherent network overhead in FAWN-KV. Because FAWN-DLi avoids this overhead, it should scale up with data size more well than does FAWN-KV. Objects sized between 2 bytes and 1 gigabyte are added, checked, and removed from a set. This test is important to tell us how FAWN-DLi affects throughput as the size of objects changes. This micro-benchmark is also performed both on our wimpy machine as well as our non-wimpy machines.

### 5.3 Fwitter

The Fwitter benchmark recreates a realistic workload: the storage and retrieval of Twitter records (or tweets). Large scale systems like Twitter have extremely high traffic, because of this maintaining a reliable high speed storage system is necessary. FAWN-KV is perfectly suited to the task of high throughput key-value storage [18].

The data set for our Fwitter benchmark is scraped from the public Twitter timeline <https://twitter.com/public_timeline>. The purpose of this benchmark is the storage and indexing on FAWN-KV of Twitter records (or tweets) organized by hash tags (topic markers).

As the benchmark stores each Twitter record, it examines them for hash tags. Whenever a hash tag is found, that record's id is associated on the backend with that hash tag. There are two different versions of this benchmark. One stores records on FAWN-KV, with the help of FAWN-DLi. The other stores records on PostgreSQL [7], an open source relational database. There is no third unmodified FAWN-KV version of this benchmark. This is because FAWN-KV makes no consistency promise below the value level. Because containers are stored as values, one user's changes may overwrite another. In FAWN-DLi we extend FAWN-KV's consistency promise to values stored within containers. This guarantees that one user's changes will not immediately overwrite those of another.

In the FAWN-DLi version of this benchmark each Twitter record is stored as a value with its unique record id as its key. Hash tags are stored as keys with values being lists of the record ids that contained them. Whenever a hash tag is found in a Twitter record, its record id is added using `modify` to that hash tag's list on the server side.

On PostgreSQL each record is stored with its associated record id. Additionally, each occurrence of a hashtag creates a relation between that hashtag and the current Twitter record id. With these columns we are able to access all of the same information as from the FAWN-DLi version of the benchmark.

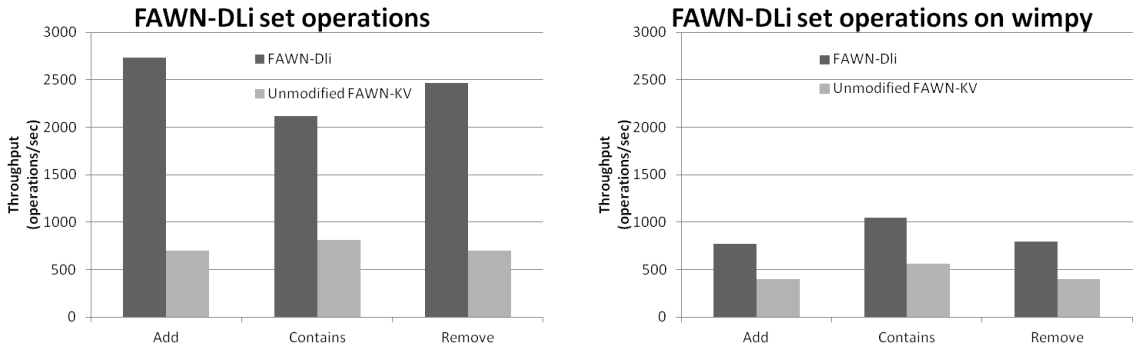
FAWN-DLi and FAWN-KV perform comparably at retrieving containers. The main area in which they differ is modification of containers. Because of this, we only report our findings for the Twitter record storage and not for retrieval. If FAWN-DLi is able to perform similarly to PostgreSQL, it would be promising for the future of these simple energy efficient prototypes.

## 6 Results

We run all of the above experimentation and present our findings in this section. The following results for our micro-benchmarks on both wimpy and non-wimpy tests are obtained by running their respective tests on a single backend storage node. Adding additional backend storage nodes only serves to increase storage space and to offload some of the serving of files to other nodes. This only becomes an issue when stress testing the system with many concurrent requests. For this reason we have declined to include results tested on more than one backend.

## 6.1 Micro-benchmarks

The first of our micro-benchmarks compares the throughput of each of FAWN-DLi's operations, run on a set, to the performence of the corresponding unmodified FAWN-KV methods. Figure 4(a) shows the results of this micro-benchmark running on a non-wimpy machine.




(a) (b)

Figure 4: FAWN-DLi set operations vs unmodified FAWN-KV equivalents. Running on a non-wimpy machine (a) and a wimpy machine (b). Note that FAWN-DLi outperforms the unmodified FAWN-KV functions in all tests.

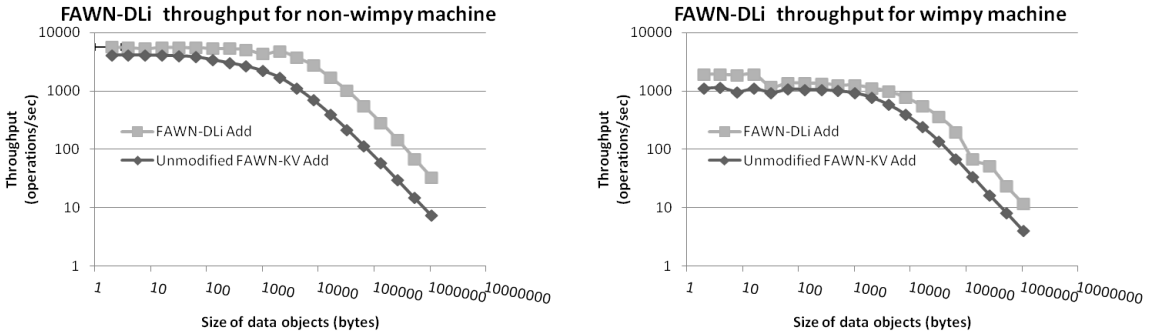
From the results in Figure 4(a) we may see that, in the same configuration running the same micro-benchmarks, FAWN-DLi's data manipulation methods perform the same tasks with twice the throughput of the older unmodified FAWN-KV implementation. It is important to note that the comparison here is between the throughputs of FAWN-DLis methods and the equivalent FAWN-KV functions. As was mentioned in section 5.2, this micro-benchmark builds up a set, then checks its contents, then removes items from the set. This leads to contains checking sets that are always of the largest size, whereas add and remove are operating on sets of varying sizes which are on average smaller.

Weaalso ran this micro-benchmark on a wimpy node, the results of which may be seen in Figure 4(b). The performance of FAWN-DLi on wimpy nodes is of particular interest because that is its proposed running environment. It is also the environment in which FAWN-DLis focus on storage side processing can incur the highest cost.

From the results in Figure 4(b) we can see that FAWN-DLis data manipulation methods also out-perform the corresponding unmodified FAWN-KV implementation on a wimpy node. This result is promising, as it appears that the wimpy backend nodes are able to handle the additional processing load incurred by the added complexity. We may also see that FAWN-DLi performs less well on the test wimpy node than on the test non-wimpy architecture. This is unsurprising, as the wimpy nodes have less processing power. The important aspect is that the wimpy nodes running FAWN-DLi outperform the corresponding wimpy FAWN-KV system. These tests were run on data objects of size 8192 bytes.

In the second of our micro-benchmarks we were interested in seeing how FAWN-DLi functions scaled with the size of objects when compared to the unmodified FAWN-KV system.

In order to see how these throughputs compare when scaling up the size of the objects being stored, we perform the same test many times with varying sizes of objects. The results of this test may be seen in Figure 5(a).




(a) (b)

Figure 5: Throughput of FAWN-DLi add vs unmodified FAWN-KV add while scaling up the size of data stored. Running on a non-wimpy machine (a) and a wimpy machine (b). Note that FAWN-DLi add outperforms the unmodified add at all string sizes.

On non-wimpy nodes we may see that for small data objects FAWN-DLi performs better than the unmodified FAWN-KV equivalent. For these small values the throughput of FAWN-DLi is 1.5 times that of the unmodified FAWN-KV system. This matches the results for objects of size 8192 bytes. As the size of the objects increase, this trend continues. At the largest tested object size, 1 GB, FAWN-DLi achieves throughputs 4.5 times that of the unmodified FAWN-KV system. This difference in performance is probably due to of the high communication overhead incurred by transferring large objects for editing.

When run on the wimpy backend node this test returned similar results. These results may be seen in Figure 5(b).For every object size, the FAWN-DLi data methods achieve higher throughput than the pre-existing FAWN-KV implementation. The results of this micro-benchmark on the wimpy backend appear more erratic than the non-wimpy equivalent. This is perhaps due to the higher variability that background processes have on the wimpy processor. At small object sizes on the wimpy backend FAWN-DLi achieves throughputs of 1.7 times that of the FAWN-KV implementation. As the size of the objects increases, FAWN-DLi continues to achieve higher throughputs. At 1GB FAWN-DLi achieves throughputs of 2.9 times the FAWN-KV implementation. On both the wimpy and the non-wimpy backends we may see that the overhead due to transferring larger objects is higher than the slowdown due to processing those same objects.

## 6.2 Fwitter

In addition to the above micro-benchmarks we implemented a more realistic workload, Fwitter. We ran this benchmark on both FAWN-KV and PostgreSQL, an open source relational

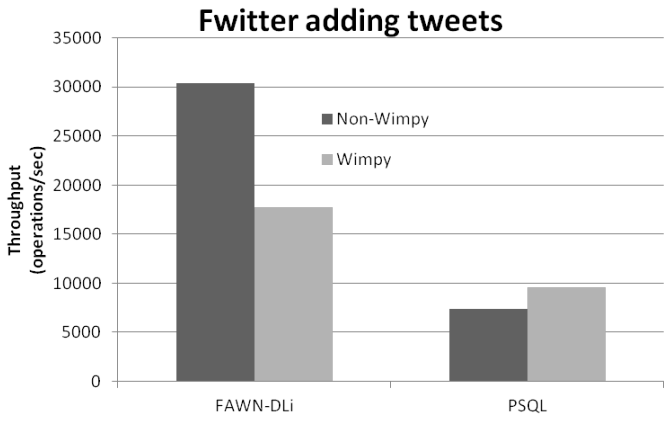



Figure 6: Throughput of adding tweets to "Fwitter" benchmark.

database. We wanted to see how FAWN-KV, written specifically with FAWNs in mind, compares to more traditional off the shelf solutions. Figure 6 shows the results of the Fwitter benchmark run on both wimpy and non-wimpy machines.

As may be seen in Figure 6, FAWN-DLi is able to outperform PSQL when run on both wimpy and non-wimpy machines. We may also see that FAWN-DLi on a non-wimpy node outperforms FAWN-DLi on a wimpy node. This is expected as this test is being run on only a single wimpy backend storage node. There is a chance that additional storage nodes could help distribute the load when run on wimpy machines. One interesting result is that PostgreSQL achieves a higher throughput on the wimpy machine than the non-wimpy. This could be attributed to the faster storage medium on the wimpy machine.

In our results for the Fwitter benchmark we see the FAWN-DLi outperforms PSQL on both wimpy and non-wimpy machines. If we examine these results we can see that FAWN-DLi is able to compete with the PSQL implementation on both wimpy and non-wimpy architectures. The results of our Fwitter benchmark suggest that wimpy systems such as FAWN-KV could function as energy efficient options in real world situations.

# 7 Conclusion

Energy efficient computing is growing in importance, with increasing demand and power usage. The wimpy hardware architecture addresses these issues of power usage, with the idea of distributing workloads across small clusters of energy efficient "wimpy" machines. These wimpy systems often adopt simple data models so as not to overload their processing capabilities.

We modify one wimpy system, FAWN-KV, in order to see if its possible to extend these simple prototype systems with additional complexity. In modifying FAWN-KV we are careful to gauge the effect that this additional complexity has on FAWN-KV's balanced system. Our modifications to this system take the form of FAWN-DLi a data library that adds more

sophisticated server-side data operations to FAWN-KV. FAWN-DLi supports server-side container operations allowing users to shift computation to the server side of the system.

We evaluate the effects that FAWN-DLi has on the throughput of FAWN-KV on both wimpy and non-wimpy machines. We implement two micro-benchmarks to gauge this effect, as well as a more realistic benchmark "Fwitter". Additionally, we compare the throughput of FAWN-DLi aided FAWN-KV with a popular open source relational database PostgreSQL.

We find in our evaluation that FAWN-DLi outperforms unmodified FAWN-KV in all tests, and is able to perform comparably to PostgreSQL on both wimpy and non-wimpy machines. Our work with FAWN-DLi suggests that simple wimpy systems like FAWN-KV are not only well suited to their target workloads, but open to extension. We were able to add additional complexity, in the form of server-side functions, to FAWN-KV's data interface without overloading the wimpy architecture of the backend nodes. As an additional bonus, FAWN-DLi was able to remove a significant network overhead by allowing users to edit data without transferring it locally. FAWN-DLi opens a range of workloads that before would have been impossible for FAWN-KV to handle.

FAWN-DLi is an example of what may be achieved by modifying simple prototypes like FAWN-KV, which address issues that are becoming ever more pertinent to the field of computation. These simple systems can be tailored to provide functionality useful to target workloads while maintaining their energy efficiency. We feel that client side processing on these simple key value systems is not a sophisticated enough data interaction. By moving processing to the server side data interactions can take place more naturally. We believe that a trend towards more server side processing will appear in these simple key value systems. Our work with FAWN-DLi shows that these wimpy systems benefit from more sophisticated data interactions.

# References


[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. *Commun. ACM*, 54(7):101–109, July 2011.

[2] Oracle Co. MySQL: The world's most popular open source database. `http://www.mysql.com`, 2011.

[3] Sea Micro Co. Sea Micro Corporation. `http://www.seamicro.com`, 2011.

[4] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

[5] Apache Software Foundation. Apache CouchDB. `http://couchdb.apache.org`, 2011.

[6] Apache Software Foundation. Apache Thrift Project. `http://thrift.apache.org`, 2011.

[7] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org`, 2011.

[8] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[9] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *Analytics Press*, August 2011.

[10] Willis Lang, Jignesh M. Patel, and Srinath Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 47–55, New York, NY, USA, 2010. ACM.

[11] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1075–1086, New York, NY, USA, 2008. ACM.

[12] VMWare Redis Project. Redis. `http://www.redis.io`, 2011.

[13] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 365–376, New York, NY, USA, 2007. ACM.

[14] D. Schall and V. Hudlet. WattDB: an energy-proportional cluster of wimpy nodes. In *SIGMOD Conference–Demo Track*, 2011.

[15] Michael Stonebraker and Joseph M. Hellerstein. What goes around comes around. In *In Readings in Database Systems*, 2005.

[16] Alexander S. Szalay, Gordon C. Bell, H. Howie Huang, Andreas Terzis, and Alainna White. Low-power Amdahl-balanced blades for data intensive computing. *SIGOPS Oper. Syst. Rev.*, 44:71–75, March 2010.

[17] V. Vasudevan, J. Franklin, D. Andersen, A. Phanishayee, L. Tan, M. Kaminsky, and I. Moraru. Fawndamentally power-efficient clusters. In *Proceedings of the 12th conference on Hot topics in operating systems*, pages 22–22. USENIX Association, 2009.

[18] Vijay Vasudevan, David Andersen, Michael Kaminsky, Lawrence Tan, Jason Franklin, and Iulian Moraru. Energy-efficient cluster computing with FAWN: workloads and implications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 195–204, New York, NY, USA, 2010. ACM.

[19] Vijay Vasudevan, Lawrence Tan, David Andersen, Michael Kaminsky, Michael A. Kozuch, and Padmanabhan Pillai. FAWNSort: Energy-efficient sorting of 10GB. 2010 JouleSort Competition `http://sortbenchmark.org`., July 2010.