Project Number: GTH-EMC1

COMMON COMPONENTS REPOSITORY

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____
Ryan Genato

_____
Jared Razee

_____

Date: September 25, 2011

Approved:

_____
Professor George T. Heineman, Major Advisor

Table of Contents

Table of Figures

1. Introduction

EMC is a computing company that works with businesses to deliver an Information Technology experience as a service. In today's market this translates to network based storage that is fast, cost-effective, secure and stable. Through their products and services EMC delivers real responses to buzzwords and terms such as "cloud computing" and "agile development". Through EMC's 30-year history, the company has focused on providing personalized IT experiences for banks, financial service firms, healthcare and life sciences organizations, airlines and transportation companies, and many other large businesses. The revenue from such services has placed EMC in the top 200 of the Fortune 500, with a record $17 billion profit in 2010 ("Corporate Profile").

Over the past year, EMC has ventured into the Small-to-Medium sized Business (SMB) storage market. EMC introduced the VNXe Storage Processor (SP), a unified storage system "designed for IT professionals with little storage expertise" ("EMC Unveils New VNX"). In order to drive down the cost of VNXe for consumers, EMC designed the software platform of the VNXe to enable end-users to manage the device. This reduces the need for a business to have a dedicated network storage professional and the need for EMC to send out support technicians for every fault scenario.

A convenient dashboard display called Unisphere supports the IT administrators who have to upgrade or replace faulty hardware components by graphically displaying parts and highlighting faulty hardware in red, going as far as placing part numbers into an order form. Unisphere currently allows end-users to identify unhealthy or failed components, but components may exhibit serious faults that will prevent the dashboard from providing any assistance to the end-user. Faults of this nature stop the Customer Replaceable Unit (CRU) Monitor from running, which prevents the ability to access CRU information, leaving the consumer confused and EMC unable to diagnose what is wrong. This project aims to design and implement a persistent repository of CRU information to provide the required information about the component parts even during a serious fault.

This project is being undertaken by EMC to solve an important storage problem in the development of the VNXe. The main goal of this product line is ease of use for both the end-users and for EMC's technical support team. A persistent repository of data reduces costs and is more conveniently applicable for EMC's future plans of integrating its currently existing VNX product line with consumer software. However, this approach may not be perfect due to standard hardware failure rates. Even with a persistent repository, the risk of system failures like hard drive corruption, mismatched data from human error, and intentional external manipulation

still loom large for implementations. This project was designed to bridge the gap between consumers and their data stored at EMC. The CRU Monitor is the most important part of the architecture of the project, and the repository that this project seeks to implement gives the Monitor a reliable persistent foundation for all of its concurrent use cases.

The CRU Monitor is the software EMC uses to communicate between the repository of CRU information and the various plug-ins and interfaces required by the rest of the system. It is responsible for accepting the data from its sources and passing it onto the repository for safe keeping while it continues processing. One of the most notable aspects of the CRU Monitor is its flexibility in processing data of all types, completely oblivious to the data itself yet still aware of how to handle it. It is an important piece of the architecture of the VNXe, and maintaining the repository where it processes information is a big feature to be added to the project to prevent significant data loss in the event of a CRU failure.

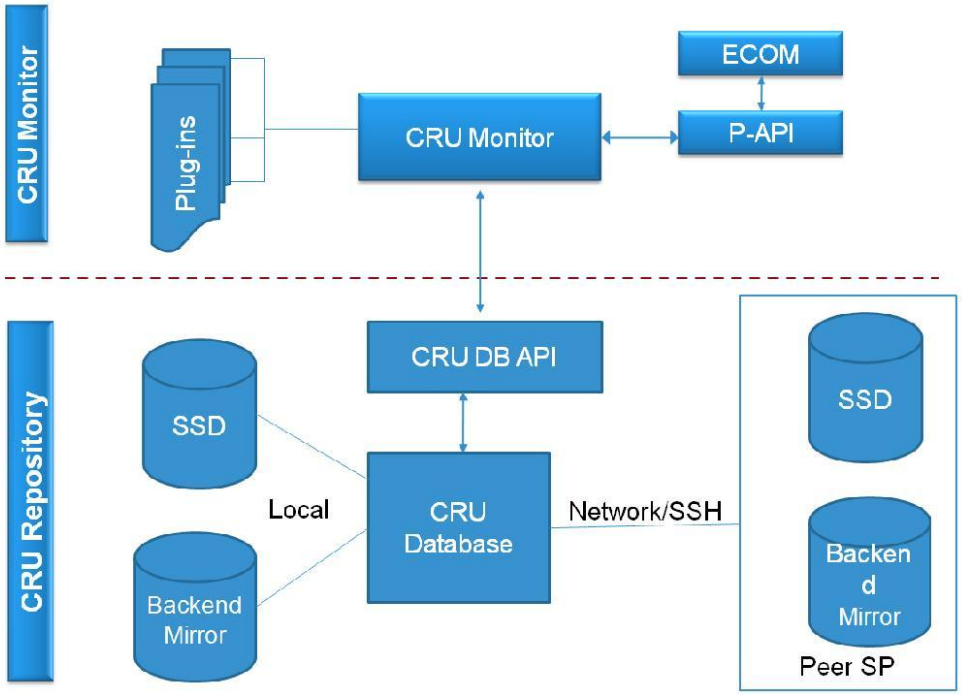

Figure 1 Original High-Level Architecture for CRU Repository

As figure 1 illustrates, the repository is a large aspect of the project and as such requires careful implementation to interact cleanly with many different moving parts.

1.1 CRU Monitor

The data generated from the CRU health monitor is passed to our *Persistence Manager* in data structures known as “chunks”. The implementation of the API does not need to know what

information is contained in these chunks. Its responsibility is simply to handle their storage. This means that the persistence manager will operate in a stateless manner, performing the same way no matter what information it is handling. The CRU Monitor will determine the relevant information to place in these chunks, and will also handle the interpretation of the data in the chunks. Determining where to store this extracted information was an important focal point of the initial design. A primary storage device, such as the solid state drive (SSD) on an SP or the backend mirror (a collection of hard disks attached to the SP) will contain all of the chunks. A secondary storage device such as the bootflash (small capacity flash drive hard-mounted to SP) will contain a select portion of chunks. This area for storing select critical information known as the "cache" is necessary in the event the primary storage device faults. In the final implementation, the API reads from a configuration file, allowing an arbitrary user to dictate the primary, secondary and network storage locations.

The persistent repository is in the form of a plain text file, in order to be scalable for use by other EMC applications such as the VNX platform. It is also designed to be scalable to multiple types of repositories, beyond storing data only relevant to CRUs. Because this implementation is information independent, CRU data can be replaced in the chunks with anything and the storage aspects will still work in the same manner. Using a universally recognized format for the repository makes the project much easier to understand from a programming point of view, and much easier for a new user of the API to handle. The decision to utilize a file instead of a database also protects against wasting time fixing costly database mistakes when more important functionality could be worked on during the limited design cycle. Figure 1 illustrates the relationship between the CRU Monitor and the initially proposed CRU repository, including the different storage methods. It gives a clear image of where this project fits into the large scale of the VNXe project.

The CRU Repository API will be used to facilitate communication between the final version of the CRU Monitor and the Persistence Manager. There is functionality for adding chunks to a heap and then storing them to the physical disk, discarding chunks on the heap, finding out how many chunks are in the persistent repository, and retrieving all the persisted chunks. The CRU Monitor can access any of these functions at any time in any order, depending on the nature of its own requests and requirements.

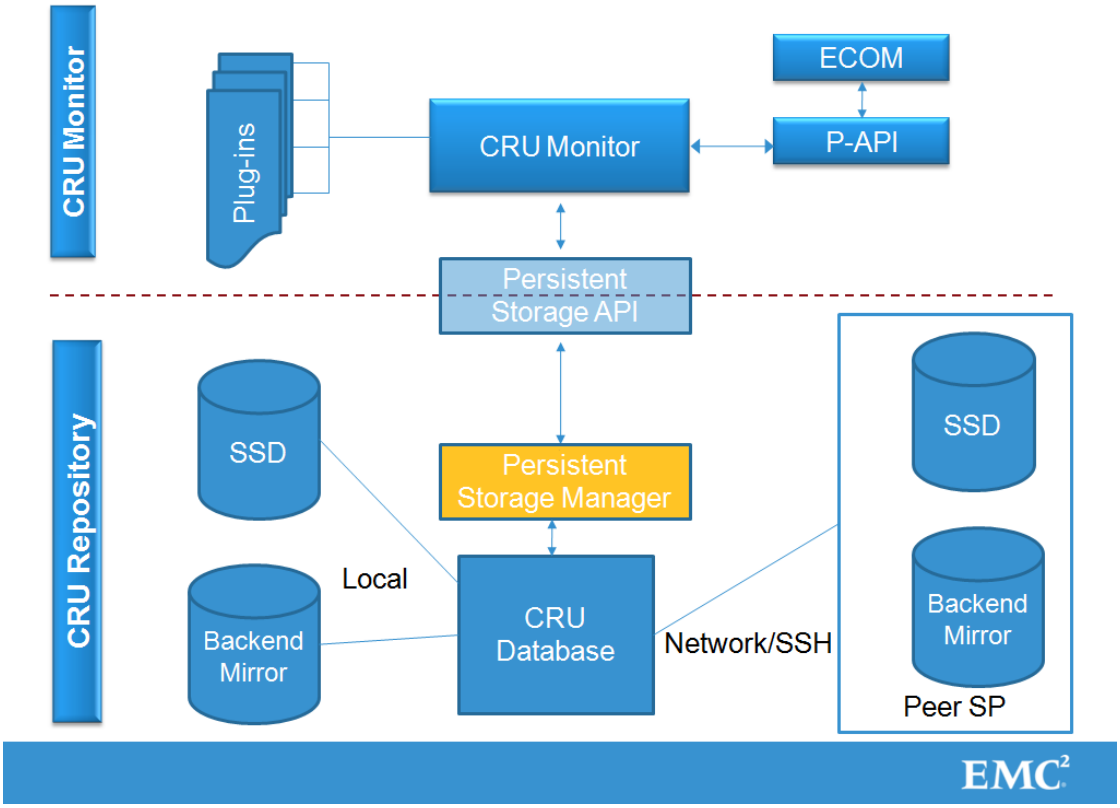

Figure 2 Edited High-level Architecture for CRU Persistent Repository

While the persistent storage will be in the form of a plainfile, there is another element to storage that must also be mentioned: the temporary storage for chunks that are added to the heap and then later stored to persistent storage. We call this area transient storage. The difference between persistent and transient storage is a key aspect of the functionality of our code. The transient data made by calls to `add_chunk()` needs to be kept in a malleable state, unlike the data that will be written to disk by the `store()` calls.

The `add_chunk()` function executes some of the most important steps in the repository implementation. It is responsible for adding a new chunk to transient storage in the area pointed to by a metapointer, and then returning the updated metapointer back to the CRU Monitor. During the first call to this function, or in any other event where an empty metapointer occurs, the function must correctly initialize a metapointer, add the chunk, and return the metapointer to the function caller. To do this, the createMeta() function was created to assist `add_chunk()` in these outlier cases. The createMeta() function allocates memory for a metapointer, with a constant capacity of chunks hard-coded in a way that allows EMC to easily change its size in the future. It is initially set to a value of 64. In the event a 65th chunk is added, the `add_chunk()` function will allocate more memory space for the chunks depending on a similar second hard-coded constant. These two constants together will determine the new size of the transient array,

and the chunks will be added as usual. In other words, if both constants were set to 64, the capacity would double to 128 chunks and the 65th chunk will be added to transient storage with 63 allocated spaces left over. Using a couple of constants was a vast improvement over the initial implementation, which started with 64 spaces for chunks and then automatically doubled each time the limit was reached. This worked well and helped contribute to our requirement of automation, but it was ultimately abandoned because of how it would perform in specific use cases. For example, if exactly 1,025 chunks were added into transient memory, it would have forced the original implementation to double until it allocated a total of 2,048 spaces. That would have left 1,023 spaces open for chunks that would never be used, a ridiculous amount of waste. Making the size of the transient array customizable reduced the impact of situations like that, fixing potential memory leaks and allocating resources much more efficiently.

After adding chunks into transient memory, the CRU Monitor may find itself in a position where it must manipulate or recall one of those chunks. To support this functionality, the get_chunk() function was created to find the requested chunk in persistent storage, copy its data to a new space in memory, and return a pointer to the freshly copied chunk. Once the chunk has been accessed, it can be manipulated by the CRU Monitor, then added back onto a metapointer and stored as usual. An important note about this function is where it brings the chunk after it has been gotten. Chunk data is copied into transient storage rather than allowing a pointer for direct persistent memory access. The reason for this is to protect multiple processes from accessing the same chunk, creating a situation where two different versions of the same chunk exist in memory.

An important secondary feature in the API implementation is the function called `get_num_chunks()`. Its job is to scan transient memory to determine how many chunks exist associated with a given metapointer. An integer value is returned to the function caller with this information. This function makes error checking much easier for all of the functions, especially for the get_chunk() and `discard_chunk()` functions. As repositories are being stored and retrieved on a day-to-day basis in different orders, the indexes in the arrays of chunks become unreliable. The `get_num_chunks()` function eliminates this uncertainty by telling the CRU Monitor exactly how many chunks exist in transient memory, preventing errors and exceptions such trying to access a chunk that is outside of the array boundaries. Even though this function is unaware of the persistent file, it can be used to tell the CRU Monitor how many chunks exist in persistent memory by simply calling it immediately after a `retrieve()`. Since `retrieve()` calls all persisted chunks into transient memory, `get_num_chunks()` can be used right away to inform the user how many chunks are in the file.

The `discard_chunk()` function removes a selected chunk from transient storage. This function is important as it allows the CRU Monitor to have true control over the contents of both transient and persistent memory. In the event of a CRU removal, the appropriate data can be removed without affecting the other transient chunks. The function accesses the chunk to be discarded by its index, frees any memory allocated to it, changes its size to zero and changes the pointer to its data to zero. These steps ensure that the discarded chunk is no longer a part of transient memory, and it is not susceptible to being accessed accidentally. The chunk will still retain its place in memory though it is no longer there, waiting to be overwritten. This is done purposefully in order to avoid any chunk numbering confusion.

An interesting feature of the `discard_chunk()` function can be discovered when attempting to discard a persistent chunk. By calling `retrieve()`, all persisted chunks will be called into transient memory. At that point, a user can call `discard_chunk()` to zero out the data of one or more chunks. When the `store()` function is called, any chunks with a size of zero will not be written to disk. Since `store()` overwrites any previous files associated with the metapointer, it will effectively clear the discarded chunk from the persistent data while properly writing every other chunk. This means that ironically, `store()` is the only function that can delete a chunk from persistent storage.

Similar to the `discard_chunk()` function, the discard() function removes all chunks from transient storage. This is useful for situations where the CRU Monitor would simply like to wipe the transient chunks and start anew. Rather than looping through all the transient chunks and calling `discard_chunk()`, discard() sets the metapointer to NULL and sets all of the metapointer's attributes to zero before freeing the memory associated with the metapointer. Once the metapointer is cleared, every chunk that it pointed to will become inaccessible for the CRU Monitor. This is much more efficient than simply calling `discard_chunk()` for every chunk, saving both time and memory space. The discard() function also made writing test cases much easier and efficient, clearing out data quickly and proving that retrieved data was new rather than simply reusing the old inputs from memory.

An interesting decision we had made regarding the behavior of the `discard_chunk()` function is to retain the spot in the chunk array that the discarded chunk had been placed in, despite clearing the data and pointer attached to it. The reason for this was to keep the numbering scheme of the chunks consistent. If a series of five chunks existed, and the CRU Monitor asked to discard the third chunk, and then the fourth chunk, if the array was collapsed the final chunk in the array (originally the fifth) would be discarded, while if the array was not collapsed the fourth

chunk would be discarded. We made this decision prior to the coding of the implementation and followed this principle in all aspects of our code, in order to consistently handle this issue.

The `store()` function handles the conversion of chunks from transient memory into the persistent file. It is undoubtedly the most important feature of the API implementation, storing the CRU data in multiple physical locations in case of a serious fault. When called, the function takes all of the chunks pointed to by the metapointer and writes it out to a temporary file. The file is then copied over the old plainfile in multiple locations. The physical location of the storage file is determined by a configuration file which can be edited by the user of the API. The reason for writing to a temporary file first is to ensure that a whole copy of the repository is maintained at all times. If there is an interruption to the writing process of the temporary file, the old plainfile will still be intact. Detecting whether the repository is outdated was not a design requirement for this project due to the fact that historical records were not kept, but it may be looked at in the future as an extension.

The essential functionality of reading the repository back from storage into memory is handled by the `retrieve()` call. This function looks into three separate locations for a plainfile in order of priority determined by the user, and when a plainfile is found it is mapped into memory using the `mmap()` function. The `mmap()` function provides a direct one-to-one correspondence with the file on disk and a space in physical memory. This allows access to be made directly from the file but through memory addresses. Another advantage to `mmap()` is that its performance scales well with a larger plainfile containing more chunks. The address pointed to the file using `mmap()`is stored in a pointer and passed along in the metapointer structure. After a call to `retrieve()`, the stored repository can now be found in transient memory via the metapointer while it is still stored in physical memory in the file. This operation allows the CRU Monitor to inspect the elements individually in the persistent repository, giving it a chance to discard or add more chunks to the repository as needed.

A distinction that was made regarding the relationship between transient storage and persistent storage is the transparency of persisted chunks to all functions except `store()` and `retrieve()`. The practice that was decided for the persistent repository and CRU Monitor to follow is that persisted chunks should be written to disk and thereafter would be unable to be modified. If the CRU Monitor wanted to delete a chunk in persistent storage, the monitor would first `retrieve()` the persisted chunks and read them into transient storage, call `discard_chunk()` on the appropriate chunk, and then call a `store()` to write the changes to disk. As a result, the `discard_chunk()` function handles only the chunks in transient storage. This keeps consistent with the guideline that the chunks in transient storage should be the only

ones that are modifiable, and any changes to the persistent repository are handled upon a `store()`.

In addition to the storage considerations of the VNXe, an important consideration that needed to be made was the accommodation of two or more SPs in any given VNXe configuration. Once connected to an SP, it is easy to connect to the other SP by using the command "ssh peer". The actual process of copying the database to the peer SP will be handled using scp. SCP is a software tool that allows for secure copying between two locations by using ssh. This allows a file to be copied between two areas using different login credentials and having the information passed securely. For the purposes of this project, the scp command will be used without all of its features as there is no additional authorization necessary to connect to the peer SP.

Memory management is another major concern. Because the ability to swap CRUs at any time without failure is a requirement for this project, the potential for memory and heap fragmentation is high. Our initial choice was to use functions we were most familiar with in order to deal with the issues of memory reallocation. However, constantly calling `malloc()` and `free()` can lead to undesirable situations where the project code runs fine for two months and then suddenly stops because there is no longer enough contiguous memory. Therefore, we chose to store our chunks on the heap to solidify the implementation for the long run. A page file was another option. Although this code is providing a malloc-esque feature to the CRU-monitor to allow storage of the data it processes, our code will shy away from using `malloc()` itself as it provides undesirable effects over the long term.

The structure of the transient storage features an array of all of the added chunks, rather than a linked list or double-linked list. Originally, the plan was to maintain a double-linked list of added chunks. That would have made it possible to collapse the list of chunks in the event of an individual discard without being dependent on an added dirty bit. We felt this was important to protect the `store()` function from storing chunks that no longer exist, which could impact the indexing of persistent data alongside the `retrieve()` function. However, keeping a linked list proved to be stressful since each entry in the list contained pointers. This meant that every entry would have actually been a pointer to a pointer to a struct which contained more pointers, which lead to a lot of confusion when the implementation began. Upon programming the `store()` and `add_chunk()` functions it became apparent that having to skip over discarded chunks was a fair compromise for the overall ease of traversing and manipulating a pointer array.

On the same note as memory management, another integral part of the implementation was thread-safety across all processes. Multiple customers could be calling the `add_chunk()` function at the same time, spawning multiple threads trying to access the same file for an attempted atomic write. If the program locks, it could cause an infinite block where EMC's system is not aware that anything is wrong. Customers would not be able to deal with the block, and EMC would not understand the context of the block, creating a complete standstill. A blocked thread would create a situation where one part of the chunk is either missing or filled in by the first part of the next chunk. When the CRU Monitor attempts to access it, a segmentation fault would occur and crash the program. To prevent this, the implementation features semaphores with critical regions around the write and read commands found in `store()` and `retrieve()` respectively. This prevents the situation where one process is reading the file as another is writing, and vice-versa, although it does not guarantee a lock-free environment.

In order to get as close to lock-free concurrency as possible, the behavior of the API functions will match with respect to their "version" of the repository. Each issue of the `retrieve()` function to pull the persisted chunks into memory will generate a unique copy of the repository for the CRU Monitor to use, directed using a metapointer. A `store()` call will only persist the copy being manipulated by that metapointer. Using the `mmap()` system call we will be able to map the physical data of the repository in a one-to-one correspondence with virtual memory. This will keep a single instance of the repository open at a time, reducing the chance for multiple copies to be in conflict. While there may be multiple instances of the repository open due to multiple `retrieve()` calls, it is the responsibility of the CRU Monitor to make sure the intended changes are saved by the `store()` method using the correct copy. Furthermore, while we are using a Linux specific system call in order to map memory, we are striving to keep the implementation as general as possible. This will make it easier to adapt our persistent repository across other platforms.

During the design of the API, we focused on single-fault scenarios because the VNXe was designed with these situations in mind. Multiple-fault scenarios can be considered an extension of this project but was never a requirement. The worst case scenario as far as this project is concerned is when power is pulled during a `store()` operation. This would result in a partial, incomplete repository being written, and would lead to problems when being accessed by the CRU Monitor the next time the system is started. To accommodate this, we implemented an atomic move operation in the store function to protect ourselves from this. When `store()` is called we will write a new plainfile which has all of the chunks the CRU Monitor would like to persist. When this write is complete, we will use an atomic UNIX move operation to replace the old plainfile. This will ensure that if the power is pulled during the writing of the new plainfile, a

full copy of the repository will still exist, even if it is outdated. Keeping track of whether a repository is outdated or not was going to be handled using either a dirty bit, although the removal of historical record keeping ended this debate.

What was generated alongside the written code is a comprehensive set of test cases for each function. It is important that the implementation is able to handle any request given by the CRU Monitor without drawbacks or failures, so the file "test.c" was constantly updated with new use cases. The reason for this is twofold: we must prove the completeness of the code or expose flaws with the implementation to guide future development. EMC has its own testing framework for code, but as it has not yet been fitted into the architecture, we must work around it. Since the CRU Monitor has not been fully realized, it was our duty to simulate the runtime environment of this code in all of our test cases. It is important to stress that our abstractions of the use cases will be the same across any platform by definition, so integrating our code and test cases with the framework of EMC should be a relatively simple process. Our testing philosophy was to create tests that were easy to run, and easy to analyze. They should allow an EMC employee who is new to the persistent repository to be able to effortlessly test the system after making changes.

Through the entire process of implementing the persistent repository, we were focused on producing code that was not only functional, but clean. We wanted to stop potential problems with the implementation by addressing these problems in the code, where possible, instead of hoping that these issues would not appear in actual practice. One example of this mindset is found in the discard() function. Rather than merely changing the data the metapointer points to NULL, which is a solution that would suffice, we decided to zero out all of the data the metapointer points to as well. This would include any data value for the number of chunks pointed to by the metapointer as well as the chunks themselves and any of their data. In the event the CRU Monitor still retained the location in memory to which the metapointer was pointing to, the data would still be there, and could be misinterpreted as desired data. By zeroing out the data in addition to the metapointer it would be clear that the memory does not contain the desired data.

2. Methodology

The nature of this corporate-sponsored project lends itself nicely to mimicking a traditional software design cycle as closely as possible. However, due to WPI's 7-week term structure, this cycle had to be designed with this limitation in mind. Our meeting schedule was formed early on in the project and it was difficult to coordinate everyone at first. Balancing our other commitments to the school and our own jobs, together with the responsibilities of the staff at EMC forced the meetings to be short and directly to the point. It was a challenge primarily because as students, most of our free time comes on the weekend while the staff at EMC is only obligated to work during the week. We decided to meet at EMC at least once a week, usually on Wednesdays so that we could go over the design work done over the weekend and prepare ourselves for the next weekend of work. We also decided to meet with our project advisor once a week, usually on Fridays, discussing the plan of attack to make progress every week. Our group met every couple of days in person as needed to meet our mostly self-imposed technical and writing milestones.

Joining the VNXe project team seemed daunting at first due to the sheer size of the architecture laid out by EMC. Figures 1 and 2 illustrated earlier show just how large this undertaking is for EMC. However, after meeting with the team working with us on these features, our role became much clearer. We presented a proposal of our project, outlining our specific roles on the team and exactly what features we would be implementing for the API. After this deadline we were able to get a clear sense of the objectives and expectations the company team was expecting us to meet. Once we solidified our own understanding of the component's role in the architecture, we were able to draft a design proposal for the EMC team. The proposal was made in a corporate setting at EMC's headquarters, and was a great way to introduce us to the environment we would expect in a real development proposal. Any practice presenting for a project like this is incredibly helpful in preparation for the final presentation, and it is a great learning experience as well. The questions that were asked of us by the team focused our thoughts and made the technical aspect of the project much clearer.

Once the proposal was accepted, we were able to set technical milestones for our implementation to complement our consistent writing deadlines. This was an important time for the project because we were not immediately able to access EMC's Virtual Private Network (VPN) and code repositories. The first technical milestone was the most important, developing an understanding of the functionality required by our CRU Repository. At first we knew there must be functions for adding, committing, retrieving and requesting specific pieces of data, but learning the constraints on our program was critical for our background research. The emphasis on thread safety and lock-free algorithms set the tone for every conversation after the fact.

After detailing our project proposal to EMC, we sought to hold a series of design reviews and revise the specification which served as an outline for the rest of our time there. The design reviews would finalize our understanding of the API's functionality and lay the framework for our implementation. Many of our key design decisions were made for this document, so maintaining consistency with what we designed here became an important goal for the future. From this design specification we were able to illustrate a mental model of our repository API functions, which was a critical step in our understanding of the overall requirements.

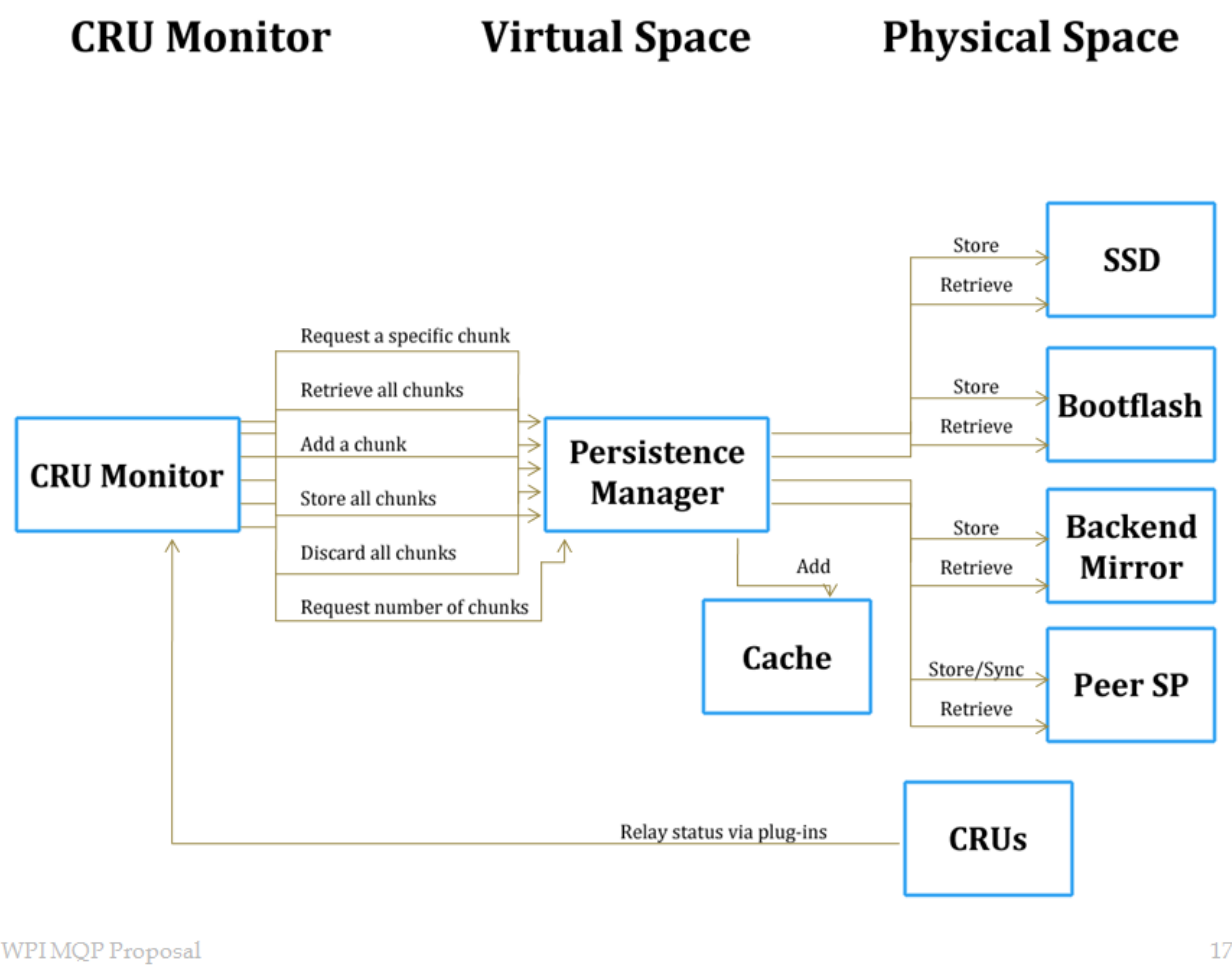

Figure 3 Mental Model of Project Implementation

Figure 3 is the mental model presented during the design phase that shows how all of our functions interact with each other in both physical and virtual space. It is an important abstraction, which is implied by how much the actual implementation reflects this figure. This abstraction was an important presentation to the EMC team, our first real technical achievement on the project. It is important to note that the "cache" is not a cache in the traditional sense of the word, but rather an additional storage location to store critical information found in specific chunks passed in with the appropriate flag by the CRU Monitor.

Over the course of the term, the methods of meeting our original schedule had to be altered in order for our team to adapt to unforeseen problems. The initial plan was to do most of the work on-site at EMC in Hopkinton; however, difficulties on that end altered our plans and moved the majority of our work to WPI. The main reason why the bulk of the project work was completed on our own time on WPI's campus was because we were unable to get access to EMC's VPN for so long. This basic requirement forced us to adjust our meeting schedules and locations, which made asking technical questions a tedious process. Rather than getting instant feedback for our questions, we had to wait on someone replying to an e-mail or phone call. The time between meetings became more difficult because of our relative isolation from the project team. As the term went on, we gradually accepted the VPN situation and proceeded as if we would never gain on-site network access. And as the project continued, communication between our project team and EMC became more frequent and was less of an issue. Our aforementioned weekend scheduling conflicts forced us to plan our time carefully so as to not ever be dependent on one part of the project to keep moving forwards. The difference in resources between the WPI campus and EMC's headquarters was very noticeable and influenced how we approached the entire schedule.

An important part of our coding process was making sure we applied proper test cases to every function as they were written. A separate file was made for testing how our implementation stacked up against the use cases outlined in our previous presentations to EMC. This practice will help in integrating the project code to EMC's native testing framework, which may be difficult due to the aforementioned access issues. The `add_chunk()` and `store()` functions were most important, since all of the test cases depend on them to work. For example, `retrieve()` cannot be tested unless there are chunks added and subsequently stored to the database. Our testing framework ended up being a standalone program that could be run from any Linux terminal, which will be a useful tool if there are any future extensions to this project.

After our first design review, wherein the repository API and persistent repository structures were discussed at length, there were a few additions and changes to our initial conception of the project. The test cases would have to be as close to automated as possible, providing easy to read feedback that was quickly comprehensible. An option for testing with more verbose output, detailing each test scenario step by step, would also be a necessity. The intention was so when a person had a fresh copy of the code, it could be compiled easily and split into two separate modes, one for the compact set of output for easy analysis and one for the verbose output.

Furthermore, test cases would have to go beyond testing each function. They would have to show the time elapsed for each function based on varied constraints. The reason for this is to demonstrate the performance capabilities of the persistent repository, and fittingly these tests are called performance tests. The performance tests simulate real-world conditions and expected typical behavior of the persistent repository in actual usage. This series of testing would be in a controlled environment and any performance issues that result may be addressed before the repository is launched. Much like our other tests, it had to be portable to different machines, to allow for performance comparisons to be made between different machine configurations. One of the key parts of our results was this performance suite, a separate standalone program that produces tangible data to show how the code performs.

Another result of the design review was the addition of a configuration file, to be used to obtain the locations for the various storage methods employed by the persistent repository. This was something we had to build into our implementation and figure out the correct area to place the configuration read in. Once it was made a requirement, it became a key component of the `store()` and `retrieve()` functions. Rather than hard-coding the storage locations into the implementation file, an easily accessible and malleable file could be edited by any arbitrary EMC employee to fit their needs at run time.

The implementation of the CRU repository is the product of the thoughts and plans detailed within this document. Our first major challenge with the implementation came with understanding the structure of the metapointer and its relation to the chunks. We were given an example of the API, which defined the metapointer as a pointer which pointed to the beginning of the chunks in transient storage. The metapointer would effectively be the area in memory where the chunks would be located. Multiple metapointers should not affect each other; when a metapointer is created it should be pointing to its own set of chunks, and when another is created it should have its own set. Figure 4 outlines the structure of the metapointers and illustrates how they interact with the chunks themselves.

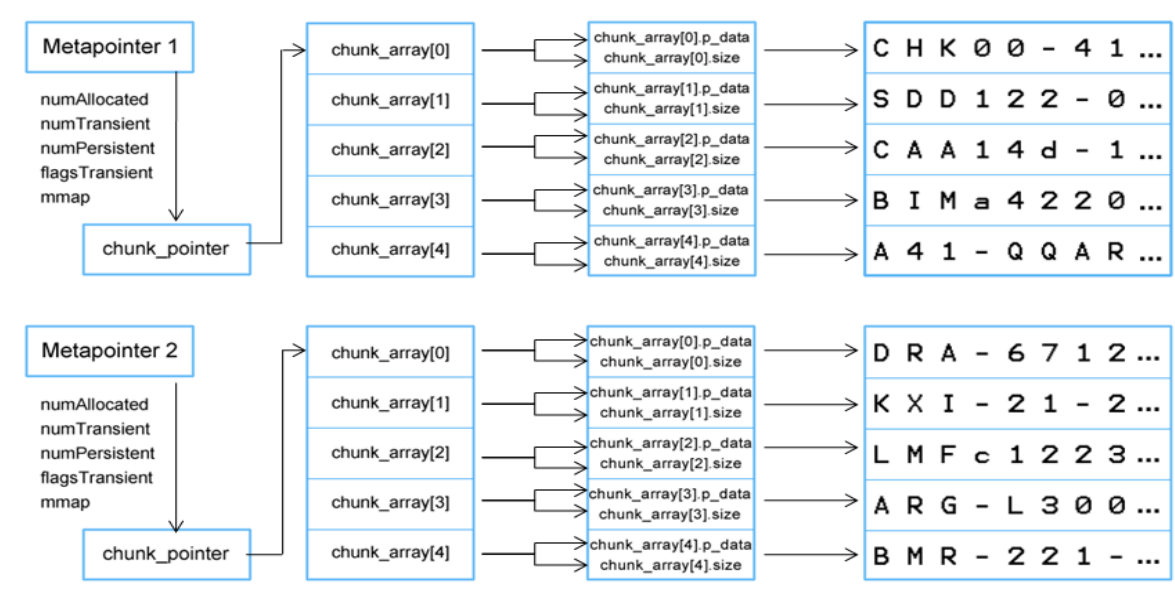

Figure 4 Memory Structure of Transient Storage

To keep track of multiple chunks in transient storage, an array of chunks is used. This allows the chunks to be easily accessible via an index. Each chunk contains a pointer to its chunk data, and a size_t describing the size of the chunk. There are no other fields in the chunk structure, leaving us with no way of distinguishing individual chunks or the messages within. When a chunk is added, the metapointer is passed which tells the function where the chunk array is located in memory. The function traverses the array to the end of the array, and then adds an additional chunk element to the array. The pointer to the chunk information is stored in the p_data field and its size is passed to the size field. Another consideration regarding the persistent file was for the structure of the flags for the chunks. Currently there exists only one flag, to discern between writing the chunk into persistent storage as well as a cache location, or just writing it to storage. But in order to accommodate future additions, we decided to store the flags as a short pointer. This gives us the flexibility for sixteen different flags and should be sufficient for any future growth EMC has for the repository.

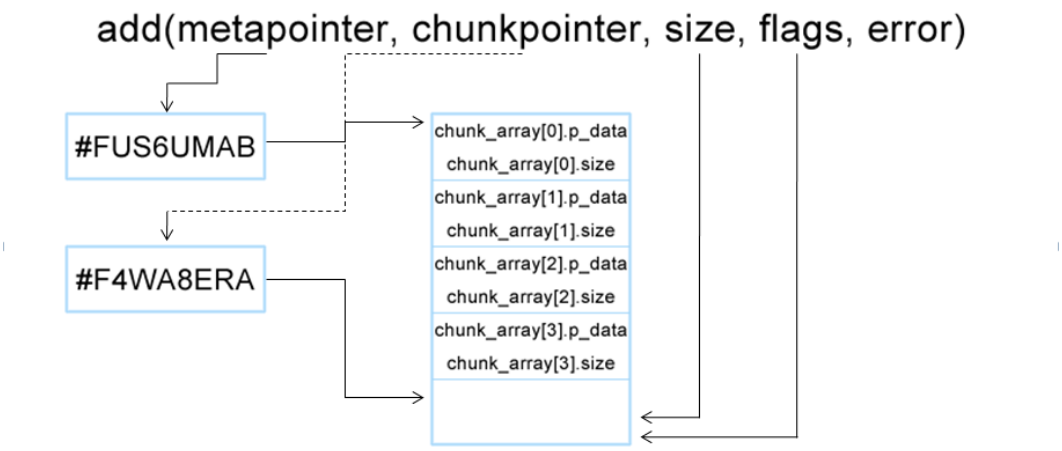

Figure 5 Visual Depiction of the `add_chunk()` Function

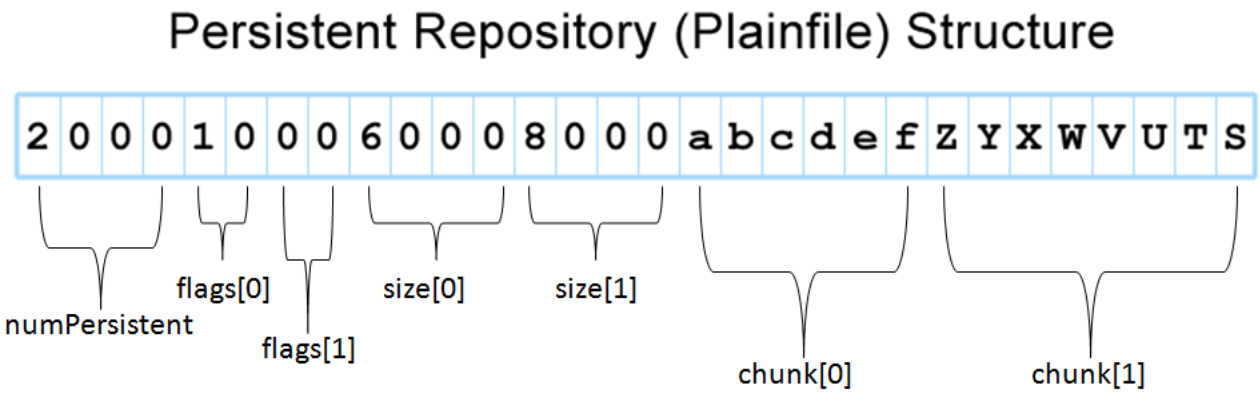

Figure 6 Plainfile Structure By Byte

The physical layout of the persistent repository plainfile was a major decision we had to make in the design process. The structure of the plainfile would be the standard to which our `store()` and `retrieve()` functions would have to be built around, and it had to be properly formed to ensure efficiency. Our decision to store the number of chunks at the beginning of the plainfile enables `retrieve()` to assign the correct amount of memory for the returned metapointer without having to traverse through the entire plainfile first. Another choice we made with traversal in mind is storing the flags for all chunks and the size for all chunks before storing the chunk data itself. We had considered storing a chunk's flag, size, and chunk data one after another before moving to the next chunk, but if the CRU Monitor had requested a specific chunk from persistent storage, the plainfile would have to be traversed chunk by chunk. With our structure, the memory offset to the exact chunk could be calculated by adding the sizes of each previous chunk together, avoiding the need to traverse chunk by chunk. This structure saves

computing time on every single function call that is made, a huge improvement over storing chunk data piece by piece.

Our decision to structure the plainfile as mentioned and its positives for efficiency with retrieval were made less pertinent by a decision which resulted from a design review. The conclusion that resulted was that when a retrieve is called, all chunks should be read into transient memory and they should have no ties to the persistent data whatsoever. This meant that every chunk in the persistent storage file would have to be added and the mapping to a specific chunk was not needed. However this functionality has been preserved and documented in order to allow for future extensibility into this option should it be found necessary.

The relationship between the chunks that existed in transient memory and the chunks that were stored persistently was a major focus of ours through the entire design and implementation process. Figure 7 illustrates the separation between the two places of memory. In the hypothetical situation where the CRU Monitor added three chunks, called to store the chunks, and then added two more chunks, only three chunks would exist in the persisted file. The chunks would remain in transient memory, and the additional two chunks would be placed at the end of the chunk array. The color coding serves to better represent the chunks and how they are stored in persistent memory. But it is important to note that the chunks in transient memory as pointed by the metapointer are in no way linked to their representation in persistent storage. Editing a chunk in transient memory will not affect the persisted file until another call is made to store the chunks in transient memory.

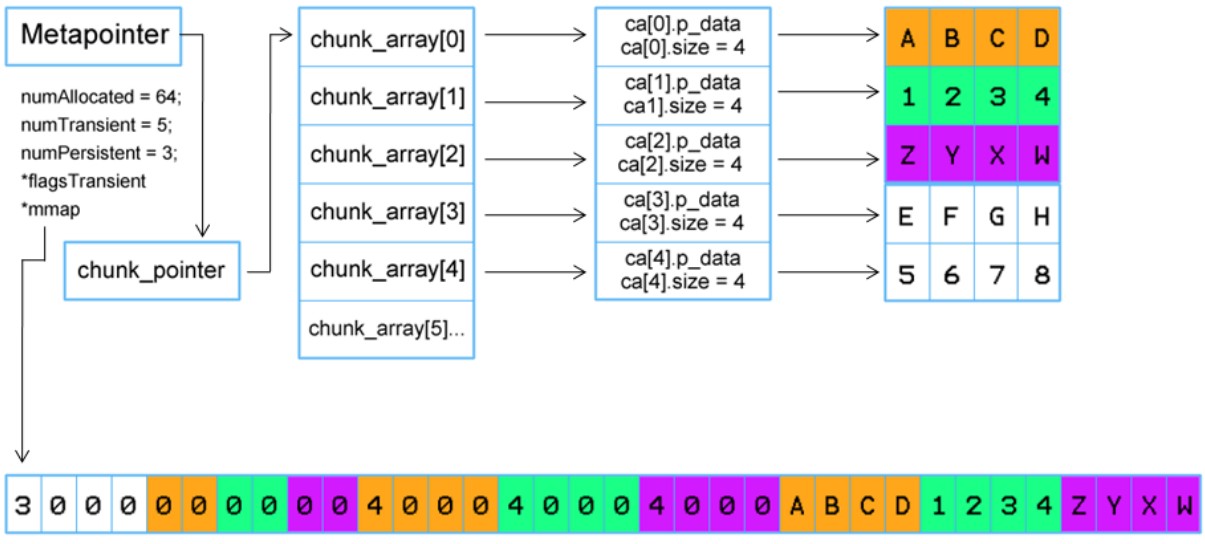

Figure 7 Transient and Persistent Storage Relationship

Because chunks in persistent storage cannot be affected, the method of deleting a chunk in the persistent file is not readily apparent. The `store()` function is the only function capable of writing to the persistent file, and as a result is the function used to effectively "delete" a chunk. In the event the CRU Monitor wanted to remove a chunk from persistent storage, it would first retrieve the current persistent file and read all the chunks into transient memory, call `discard_chunk()` on the chunk it wishes to delete, and then call `store()` once again to write out all the remaining chunks to the plainfile. The `store()` function will handle the discarded chunk appropriately.

Keeping the persistent repository as autonomous as possible was stressed in the development of our implementation. At the enterprise level, our repository must be able to run for a long period of time and require as little interaction with the lower levels of the implementation as possible. For this reason we developed a configuration file read-in that allows the CRU Monitor to define where to store and retrieve the database plainfile without having to edit the code and recompile. A separate text file containing the paths to the primary, secondary, and cache storage locations may be edited as needed and the persistent repository would only have to be restarted for the changes to take effect. This is useful because the code will never have to stop running just because the desired storage location changes. If an SP crashes, it is a simple matter of rewriting one directory name in the configuration file instead of having to examine the code line by line and hope that all everything was changed properly.

In order to ensure we were producing efficient code in addition to functional code, we divided our testing into two categories: functionality and performance. The functional tests are used to verify that our functions were written properly and performed exactly the tasks they were intended to perform. We offer a verbose test which details the actions of the test file, showing step-by-step how each test is working, and a compact test which performs the same tests but focuses on the bottom line - whether tests pass. This is presented to the user using a "." for a pass and an "F" for a fail. The performance tests measure the time taken to perform a series of tasks, similar to the actual behavior of the CRU Monitor to the persistent repository when actually implemented. The tests are tried at different powers of ten - ten, one thousand, one hundred thousand chunks, or chunks of size ten, one thousand, and one hundred thousand bytes - and the amount of time it takes to perform these actions (add and store, retrieve) are then returned to the user.

3. Results

The best way to verify the correctness of the implementation is with a comprehensive set of test cases, scenarios which demonstrate that the code performs exactly as it was intended. To test the code, many different situations were set up to emulate the environment at EMC where the code would actually be running alongside the CRU Monitor. The test cases are incorporated into a program utilizing functions that make various different calls to the API in different orders. The majority of these function calls are designed to prove that the API implementation gives the correct results under a normal series of function calls. However, there are some test cases that were designed to fail intentionally to show that the code has accounted for possible errors. Some of those situations include passing an invalid metapointer as a parameter, and discarding chunks that do not exist.

Our functionality test cases show how our implementation behaves in real-world scenarios, simulating all of the function calls and displaying the corresponding output. Requirements outlined in our design phase dictated that we have a testing framework that was easy to run and extensible to fit in with EMC's own framework. There are tests for many different orderings and groupings of functions, ranging from simple adding to the more complicated use cases of adding, storing, retrieving, adding more to transient memory without storing, and then getting the total number of chunks to validate that the temporary chunk array does not interfere with the persisted file. All of the functions outlined in the API header file were covered, and this was verified by the tool known as "gcov". Using this tool to check which statements were actually executed, we found that 100% of our code had been covered by our set of test cases.

Adding to the robustness of the testing framework were a series of error cases, outliers in the set of use cases that the implementation must account for. Situations such as discarding a chunk that does not exist may occur and the behavior of our code must be documented. Since we relied on a configuration file to determine storage locations, we implemented a test case specifically for this functionality. Being able to answer in case of an error is an important task of any coding project, and these automated error cases helped improve the documentation of our code as well as our understanding. Answering the questions of "what" when determining what parameters cannot be handled helped us understand the "why" when discussing why our implementation works. These error cases are included in the test files alongside the functionality test cases, displaying the automated output in the same command.

Performance testing was also an important part of the results. The `gettimeofday()` function was included in a separate test program, enabling the measurement of time taken to run

every function. This will be helpful not only to make judgments for the current implementation, but it will also provide a really useful metric for future project extensions by EMC. Another future extension of the project could include improving the run time of the current implementation. Comparing the run time of different versions of the code can be an important motivating factor in later design work; if a proposed upgrade becomes too weighty it may not be beneficial overall.

Our performance tests attempted to mirror the functionality test cases as much as possible, but some fundamental additions had to be made in order to give a more accurate judgment of the run time at an enterprise level. For example, while the functionality test case file only needed a demonstration that `add_chunk()` worked properly, the performance test file was more interested in testing how well the function holds up under extreme circumstances. It answers questions that must be asked to ensure completeness of the implementation and documentation. For example, if the number of additions or stores increases exponentially, will the run time also increase exponentially? What is the average run time of every function? Is there a performance gap over different machines running the same code? Is there a noticeable lag when different functions are called in a certain order? All of these questions are important to ask and verify the ability of the implementation to perform at a high level. These questions can be answered by running the performance tests on different machines and analyzing the results.

The structure of the performance test cases allowed our functions to generate easily interpretable results almost immediately. Each run of the test case outputs a table of information containing fields dependent on the functions being tested. For example, a test of `add_chunk()` would need fields describing the number of chunks to be added, the size of those chunks, and the time taken to add the various chunks to transient storage. The chunk fields are manipulatable by going into the performance test file and changing them manually, but the time fields are machine dependent. They are determined by calling `gettimeofday()` before and after the desired function takes place, effectively measuring the time an SP or any capable machine would take to perform the operation.

Performance metrics generated by our test cases are very useful tools for analyzing the code, isolating functions which consume the most significant amounts of time. Figure 8 shows how much of a time difference exists between the addition of a chunk to transient memory compared to physical memory. The `add_chunk()` method is demonstrably faster than the `store()` method, giving concrete signs to the interpreter indicating high-risk behavior for running the CRU Monitor. Extending the results of the performance tests shows that every function in our implementation has been optimized for performance except the

`store()` operation, which confirmed our expectations from the design phase of the project. As shown in the figure, the `add_chunk()` function completes its task in the range of 10 – 20 milliseconds. However, the `store()` calls on the same data take upwards of 30,000 milliseconds. It is also worth noting that calls to `store()` have a noticeably large overhead on the initial run time. While running the test multiple times it became clear that the size of the chunks did not affect the trend on the graph. Even when the chunk sizes were entered in the reverse order (1000, then 100, then 10) the trend in the data remained the same.

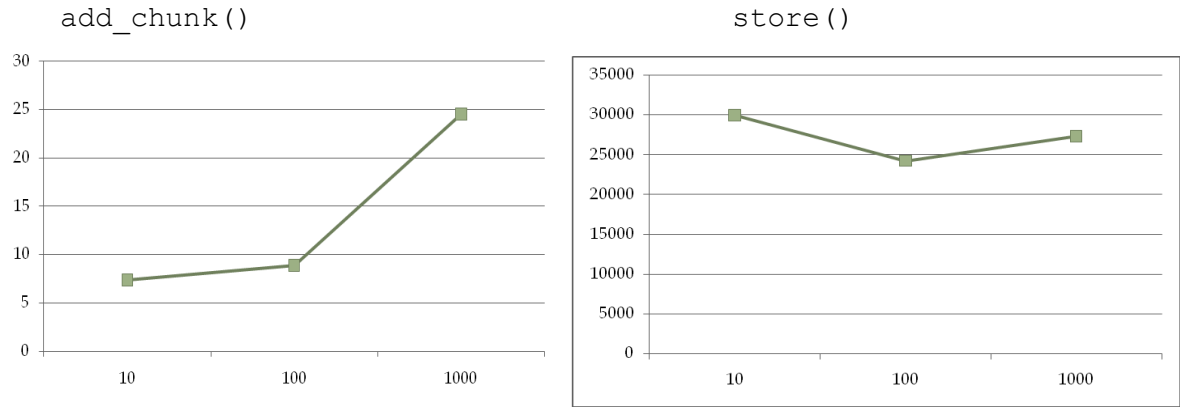

Figure 8 `add_chunk()` vs. `store()` on the same number of different-sized chunks (time in ms)

Similarly, our performance test framework can also be employed to examine how certain functions interact with one another. Again using `store()` and `add_chunk()` as examples, it is possible using our testing framework to give the CRU Monitor substantial data regarding its optimum future behavior. There is an important distinction between those two functions, where `store()` is the only function to write to a file. In the use case where a user wants to add five chunks and store them to a persistent file, which would be the best order of operations: adding them all with `add_chunk()` and then calling `store()`, or taking the safe route by calling `store()` after every individual chunk has been added with `add_chunk()`? As Figure 9 shows, the performance tests dictate that it is significantly faster to store chunks in bulk than it is to do it individually. In fact, even though constant `store()` calls might seem like a safer option because it always maintains an updated repository, that behavior might make the program *less* safe in practice because of the exponentially higher amount of time spent writing files. The repository would become at least ten times more likely to experience a fault during a write operation with no clear benefit. Figure 9 highlights this disparity.

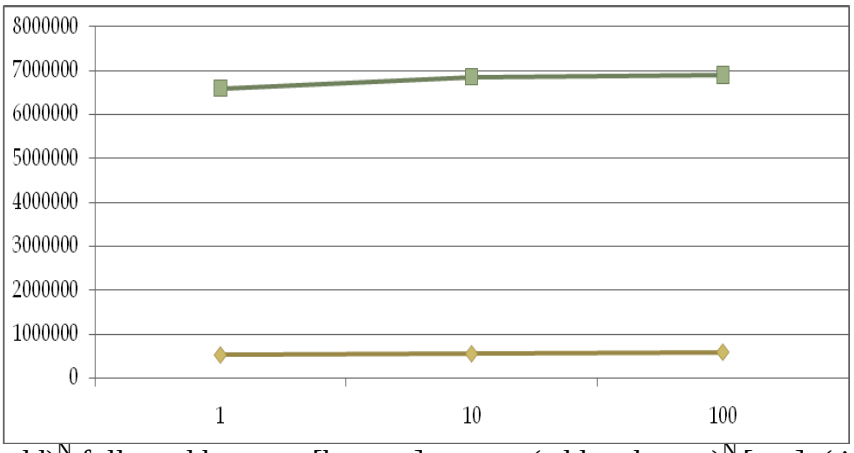

Figure 9 $(add)^N$ followed by store [bottom] versus $(add\ and\ store)^N$ [top] (time in ms)

Another use case that our implementation addresses is storage of the repository file across a network using ssh. Figure 10 demonstrates that the `retrieve()` function is able to perform relatively quickly even while overcoming the network gap. The average local `retrieve()` time is actually lower than the average time for `add_chunk()` on the same number and size of chunks. Even though the local graph on the left appears to show a spike, the scale of the graph is so small that the difference is less than one millisecond between overcoming the overhead and retrieving all of the chunks.

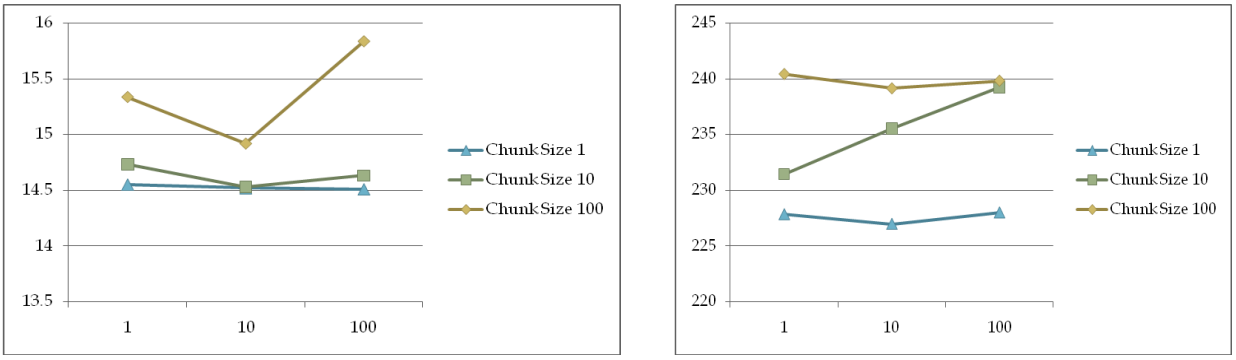

Figure 10 `retrieve()` locally vs. over a network

4. Conclusions and Future Extensions

In conclusion, we believe that this implementation will serve as a real tool to help EMC both in providing convenience to its customers and minimizing costs all around. We accomplished our stated goal of providing a universal repository solution for the VNXe, a component which is also adaptable for other EMC products. Additionally, we included a standalone testing framework to demonstrate the robustness and performance of our code. These tests are designed to run on any machine, further supporting the integration of our project into EMC's currently existing architecture. Adding to that, we included documentation detailing its proper use. These outline our design decisions, why they were made, and suggest areas that may be expanded upon for future use.

At the beginning of the design phase of the project, we asked for a clarification on what would be considered a successful project from the EMC team's point of view. They gave us a detailed list of their expectations for the project, which they held us to consistently even as the project ended. The initial goal to overcome was creating a design specification, and developing an implementation to match it. Although there were issues at first because of how quickly we had to learn the basics of working with EMC and WPI, we were able to complete the implementation as outlined in this document. The second measure of success on the EMC team's list was a portable implementation that could integrate with the CRU Monitor and other EMC products, which was a major motivating factor in all of our design decisions. Our test cases run efficiently on many different machines and certainly satisfy this requirement. The test cases also serve as their own standalone program with interpretable results. They are complete with multiple different options, automated and verbose, to suit EMC's future needs. Performance measurements have also been made available to compare results on different machines. Our test case framework provides complete coverage of the implementation code, as proven by the gcov tool, meeting this requirement. The most pressing requirement was also the most reasonable, they wanted us to make sure that we met our original schedule that we presented with our proposal to complete the project formally. This was taken very seriously, and we planned carefully all term to ensure that our methodology guaranteed completion by the deadline. We did this by always making sure that a working model of the repository existed and building up from there to incorporate any new functionality. By doing so, we were able to design with the future in mind, completing their last measurement of success.

On top of what already exists, there are also some places where this project may be extended into the future for use by EMC. Some technical enhancements may be made to improve the security of the repository, as well as upgrades to provide services that were not

asked for at the time of this implementation. For example, a "magic key" could be added to the repository to detect corrupted or intentionally altered files. A key like this would usually contain a series of bytes at the beginning of the repository's stored plainfiles that must be matched by the calling function in order to grant access. The key could provide a security upgrade by protecting from external manipulation and by protecting EMC's own processors from getting mixed up if they accidentally access the wrong repository. Although we were specifically told to ignore the possibility of storing historical data in our repository API, this is an area we immediately identified as a place where future improvements could be made. Keeping historical copies of the database would be incredibly helpful in the event of a major crash, and a time stamped event log could be utilized here as well to replay system events. Staying on a historical note, version control could be added for backwards compatibility. Adding a couple of extra bytes to the beginning of the plainfiles could let repositories be tagged for use with different generations of EMC products. Another technical addition that could be made is improved command line support, possibly featuring flags for run-time control of storage. It may prove to be easier to determine the proper locations via the command line rather than editing a configuration file, although we did not add this on purpose for automation reasons.

Because this was designed to be a universal storage solution, we hope it will integrate into EMC's future product lines. It works with many other machines besides the VNXe, so it should be easy to integrate with the Unisphere and the CRU Monitor. The implementation can also be used to store data not related to CRUs, because of our information independent solution. To port this code to other machines, an RPM Package Manager could be made which would be simple due to the machine independent nature of our final product. After a term of project work, our implementation meets EMC's requirements for a successful project, and we believe it will serve as a useful storage tool for both the present and the future of the VNXe product line.

5. References

"Corporate Profile." *EMC*. N.p., 2011. Web. 18 Oct. 2011. <http://www.emc.com/about/emc-at-glance/corporate-profile/index.htm>.

McGaughey, Katryn. "EMC Unveils New VNX Unified Storage Family." *EMC*. N.p., Jan. 2011. Web. 18 Oct. 2011. <http://www.emc.com/about/news/press/2011/20110118-05.htm>.

Rochkind, Marc J. *Advanced Unix Programming*. Englewood Cliffs: Prentice-Hall, Inc., 1985.

Troppens, Ulf, Rainer Erkens, and Wolfgang Müller. *Storage Networks Explained*. West Sussex: John Wiley & Sons Ltd, 2004.

Executive Summary

Over the past year, EMC has ventured into the Small-to-Medium sized Business (SMB) storage market, introducing the VNXe: a unified storage system with a software platform accessible to end-users without network storage experience. A convenient dashboard display called Unisphere graphically displays the parts, known as Customer Replaceable Units (CRUs), in the VNXe and highlights faulty hardware in red. Unisphere currently allows end-users to identify unhealthy or failed components, but there are cases where components may exhibit serious faults that will prevent the dashboard from providing any assistance to the end-user. This project aims to design and implement a persistent repository of CRU information to provide the required information about the component parts even during a serious fault.

The persistent repository communicates with the CRU Monitor though a well defined set of functions known as the Database API (DB API). The CRU Monitor obtains and passes "chunks" of information through the DB API and to the persistent repository. The persistent repository adds the chunk information to memory, a location known as transient storage because of its semi-permanent state. The CRU Monitor may add or remove chunks from transient storage as necessary, taking responsibility for interpreting the chunks themselves.

When the chunks are ready to be stored to disk, the CRU Monitor calls the persistent repository to store all chunks in transient storage. The persistent repository writes all the chunk data to a temporary file, and copies the file to three locations. The first and primary is the backend mirror on the VNXe machine. The second location is the solid state drive on the VNXe. This location stores certain chunks, defined by the CRU Monitory using a flag when the chunk is passed to the persistent repository. The third location is the backend mirror located on the networked VNXe processor. This allows the CRU information to be retrieved when a VNXe processor faults.

The CRU Monitor will need to retrieve the chunks in persistent storage, for example after a system restart. When the CRU monitor calls to retrieve the persistent file, the three locations will be tried in order of priority. If there is a fail to access the file on the backend mirror, the solid state drive will be tried, and following that the backend mirror on the networked processor.

This project intends to solve a shortcoming that exists within the VNXe platform. Having a persistent repository for CRU information allows this information to be accessed under situations that the CRU Monitor alone would not be able to do so. It eliminates the need for EMC to send out a technician to service the problem, and saves the end-user the time and stress of having to deal with the issue.

Abstract

EMC is a large-scale computing company focused on data storage and cloud computing. With their VNXe product, they have moved into the small and medium sized business market by offering an affordable storage solution with an intuitive user interface. A persistent repository was implemented to maintain the information for the Customer Replaceable Parts inside a VNXe to enhance sustainability and reduce the cost of serviceability. The repository handles part information that is generated via a monitoring application and stores the information across multiple storage devices, keeping the information in sync between devices. This allows the information to be retrieved despite the occurrence of a serious fault, reducing the need for a technician to be hired to diagnose and service the problem.