Power Control Center Applications using Highly Available Distributed RAM (HADRAM)

Damian Cieslicki and Thomas Schwarz, S.J.
Department of Computer Engineering
Santa Clara University
500 El Camino Real, Santa Clara, CA 95053
damian.cieslicki@us.abb.com TJSchwarz@scu.edu

Abstract

With on-going deregulation, with tighter interdependence between energy providers, and with growing demands on reliability, Power Control Centers (PCC) need to both scale up and to allow access to comprehensive real-time data. In this paper, we propose to build PCCs based on geographically distributed and tightly coupled clusters of heterogeneous computer clusters. Our proposal, Highly Available Distributed Random Access Memory (HADRAM), and Distributed Linear Hashing with scalable availability (LH$_{RS}$) uses the distributed memory in the cluster to store all data, with redundancy needed for recovery in case of data loss. This approach allows very fast fail-over, but also lends itself to remote data accesses.*

1 Introduction

Currently, power flow and other power control applications are confined geographically to single Power Control Centers (PCC). Ongoing regulation and the need for greater reliability and security force a basic redesign in the implementation of PCCs. First, PCCs need to be able to scale up, that is, to store and process much larger amounts of data. Second, individually and collectively, PCCs need to become more reliable. In our view, this implies a simpler and more robust design only available in a distributed architecture. Third, PCCs need to provide more services (such as services providing a status digest to neighbors).

Currently, PCCs are implemented on two mirrored systems. When the main system fails, we fail-over to the mirror. This might happen once daily or weekly. A typical time to switch would be two to five minutes. During this time, PCC operators loose some operational control over the system. The literature [LAC00] contains methods to mediate successfully the problems related to failover. PCCs are able to survive a single failure, but cannot tolerate a second one. Some utilities install an additional backup system for emergency purposes. This brings the total number of independent systems to four.

The architecture of today's systems limits the exchange of data. As a consequence, current utilities do not routinely share operational data. The experience of the 2004 Northeastern blackout shows the dangers of this limitation. As the national infrastructure is hardened against terrorist attacks, future regulations might even impose the capacity to control a power grid from a neighboring PCC in case of a successful attack against the local PCC. The same capacity underlies a business model that offloads certain routine applications such as power flow computation for a number of utilities.

At the heart of PCC operations are the acquisition, the storing, and the processing of data. We propose a novel approach for this central piece of PCC architecture, namely using distributed memory instead of disks. Such a solution is only viable if the distributed memory is highly reliable and if it is built on top of commodity systems. This is the core of our proposal.

Compared to the current system architectures, distributed memory has one disadvantage, namely slower access to data in the central database. In order to allow fail-over, the current architecture has to store data on disks. Storing data in distributed memory however, as we propose, is significantly faster. For example, the current bandwidth of disks is somewhat higher than 50 MB/sec [Se04], but lower than the network transfer rate (1Gb/sec = 125 MB/sec. The long-time trend exacerbates the difference. Distributed main memory offers accesses that have less latency

than disk drives by a factor of 10 to 100 and bandwidth limited only by the capacity of the network. In order to combine these attractive properties of distributed memory with high reliability, we store data redundantly on several computer systems. Failure of a few of these computer systems does not lead to data loss; rather, data on lost computers is recalculated and distributed among the remaining machines. We can even geographically separate the machines (within reason, since the distance adds to the latency at the rate of the speed of light, at best), so that sabotage of a single server location does not bring the PCC to its knees.

In the remainder of this article, we first present our views on current and our proposed PCC architecture. We then give then an overview of highly available distributed memory (Section 4). Section 5 explains the principal ways of achieving high redundancy with low storage overhead. Section 6 explains our main technique (HADRAM), which has a flexible usage of distributed memory, whereas Section 7 contains an alternative architecture (LH*$_{RS}$) based on individual records that has recently been implemented. Section 8 presents the research challenges in our proposal and Section 9 concludes.

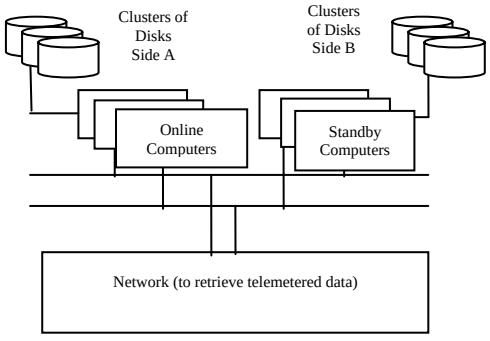

Figure 1: PCC system diagram

2 Current PCC Architecture Overview

Power Control Centers (PCC) have evolved over the last 30 years into sophisticated computational systems that solve such problems as power flow. A typical PCC consists of several high performance computers linked via LAN. These computers are organized into the *online* and the *standby* groups. Should the first group fail, the second will take over operations. The PCC computers are connected via a Local Area Network (LAN), but different PCC do not communicate with each other. Thus, every PCC controls it own small island of automation in the whole US Power Grid. A simplified schematic diagram of a typical EMS system is shown in Figure 1.

PCC applications are not modularized and rely extensively on information in the central Supervisory Control And Data Acquisition (SCADA) database. Measurements or process data move bottom-up, while control information moves down. Interconnections to other information systems, such as enterprise information or maintenance management systems have to be handled by specific interfaces [R01].

A PCC needs to continuously acquire, store, and process a large set of network data. With increasing numbers of interconnections between control areas, real-time operation have become quite complex, so complex in fact that adding functionality to the PCC to enable optimization and load prediction has become difficult. We believe that the current and future needs (not in the least to integrate information from neighboring utilities) will make the modularization and commoditization of components essential. We also believe that for the data storage unit an abstraction, that is, provision of a simple interface akin to an abstract data structure is beneficial. These techniques become feasible because increases in computing power and networking outperform latency decreases in magnetic disks. Fifteen years ago, random access of data on a magnetic disk took about 15 msec. Enterprise SCSI drives have cut this to about 5 msec, which corresponds to an annual increase of less than 8%. In this time, actual progress outperformed Moore's law predicting a 60% annual increase in computing power.

3 Proposed PCC Architecture

As we have seen, the current PCC architecture has drawbacks:

- limited scalability
- limited interoperability
- limited reliability and availability
- high operational cost
- significant failover time

In our design we propose a PCC system that is built around a cluster of servers implementing a distributed, but highly available memory store. The system stores data redundantly and is built on top of cheap commodity computers. The basic design is shown in Figure 2.

In our proposal, applications no longer have to reside on the same computer. They only need to be able to communicate reliably with the HADRAM

layer. We can run two copies of the same application on different systems and we can easily add additional applications. One or more Remote Terminal Gateways (RTGs) update the distributed memory with telemetered data that continuously arrives.

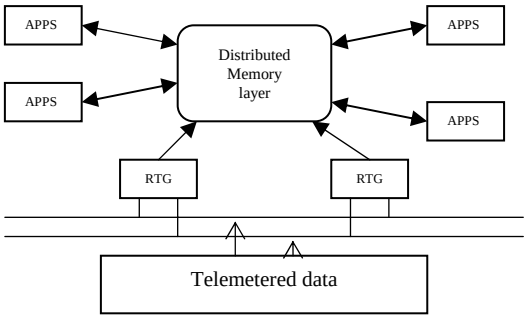

Figure 2: PCC/HADRAM architecture

Any application such as the power flow application accesses telemetered data indirectly, through the HADRAM layer. If it accesses data, it makes a request to distributed memory, which costs time, but it is free to cache the data locally. In the latter case, the data might be slightly stale. The delay is on the order of microseconds in a cluster located in the same room. This is less than the delay to transmit the received telemetered data and much shorter than human reaction time.

On the other hand, the use of distributed memory has significant advantages. The design decouples the SCADA database layer from the application layer. This increases the flexibility of deployment, creates interoperability of products by different vendors, and leads to easier design. Should an application fail, then a replacement application can find the same data in the central store. If a memory component fails, then the data will be reconstructed within seconds. This significantly increases the protection of mission critical applications and data. We can make the distributed memory component *k* –available (so that it can survive *k* failures) for arbitrary, but small values of *k*. The flexibility of the design allows stand-by copies of applications that start running when we detect the failure of the principal application.

The flexible design also makes it easy to exchange data with external entities such as neighboring utilities. For this purpose, we have an *agent* running at a utility. The agent collects data from the distributed memory layer, bundles it, and sends it to its home utility, e.g. through a virtual private network (VPN). Reversely, another agent residing at our PCC receives data from a neighboring utility and places this data into the distributed memory, where it can be processed by whatever application monitors the overall health of our system and that of its neighbors (Figure 3).

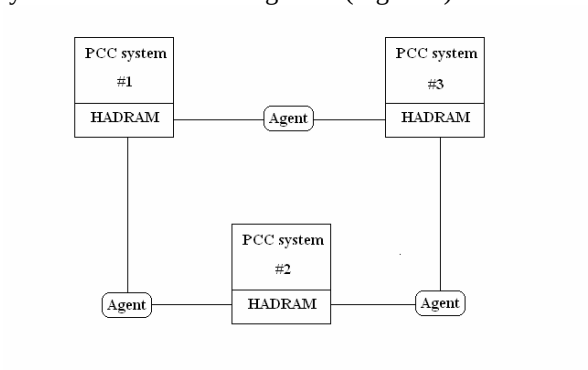

Figure 3: External data exchange using HADRAM

4 Highly Available Distributed Memory

Distributed memory implemented at the Operating System level provides an effective and efficient paradigm for inter-process communication in a cluster. We propose distributed memory at the application level used as a fast, non-volatile storage system.

We build these systems out of commodity computers for their very attractive performance / price ratio. A cluster can provide a total amount of random access memory that exceeds what any custom memory can provide in a single computer.

However, such a multicomputer contains many more points of failure. We harden the system by storing data redundantly and by providing automatic failure detection and correction. When an application stores a datum in our system, it not only stores the datum directly in the main memory of one node in the cluster, but it also updates one or more pieces of parity data distributed in main memory throughout the cluster. When a node has failed, we use this redundancy to reconstruct the failed data on another node in the cluster.

Using distributed main memory reduces the latency of memory access to essentially the network latency and the processing latency at the nodes and is 50 to 100 times faster than disk accesses. It also allows us to recover the contents stored on a failed node within seconds.

We propose two different types of highly available distributed memory systems, one that stores data in records or objects identified by a single key, and one that allows an application to request, release, and manipulate blocks of memory. An implementation of the first variant, called LH*$_{RS}$, exists at the University of Paris, whereas the second variant, Highly Available Distributed Random Access Memory (HADRAM) is currently under development at Santa Clara University.

The distance between the nodes of a multicomputer obviously determines an upper bound on response time because of the speed of light, which adds about 1 msec round trip delay for every 150 km in distance. This still allows us to geographically disperse the nodes so that no single major event can destroy a sufficient number of nodes for the memory core system to stop functioning.

5 Reliability and Erasure Correcting Codes

We now describe the methods to store data redundantly. The simplest way is mirroring or replication, in which we store the same data item twice or several times. While replication leads to the simplest protocols, its storage overhead makes it a poor choice for distributed main memory. Instead, we use Erasure Correcting Codes (ECC) based on well-known error control codes.

5.1 Parity Code

The simplest ECC is the parity code. It came into the fore with Redundant Arrays of Independent Disks (RAID) Level 5. A number of data blocks on different disks (or in our case nodes) are grouped into a parity group to which we add a parity block. The parity block consists of the bitwise parity (the exclusive or = XOR) of all the data blocks. The algebraic properties of the XOR operation imply that we can retrieve the contents of one data block from the bitwise parity of all other data blocks and of the parity block. In this manner, the parity code provides 1-availability. Instead of placing our blocks on disks as in a RAID, we place data blocks into the distributed main memory of the multicomputer.

5.2 Erasure Correcting Codes

The parity code allows us to recover from a single failure, but often, we need better protection. Many error control codes can be used to efficiently deal with erasures caused by failed storage components. One approach is Rabin's Information Dispersal Algorithm (IDA) [Ra89] that ultimately is based on an ECC [Pr89]. IDA breaks a datum into n different chunks such that any $m < n$ chunks are sufficient to reconstruct the original datum. By storing the n chunks in n different storage locations, we can survive $n - m = k$ failures. Unfortunately, in order to read a datum, all n storage locations need to be accessed.

A better way to use an ECC mimics RAID Level 5. We describe it now. We store data in *data blocks*. We group m data blocks into a *reliability group* and add to this reliability group k *parity blocks*. We calculate the contents of the k-parity blocks from the contents of the m data blocks using the ECC. If there are up to k unavailable blocks, we can reconstruct the contents of all the blocks using the available blocks (necessarily more than

A good ECC has the following properties:

1. Excellent encoding and decoding speed.
2. Minimal overhead: For k-availability we only need k parity buckets of the same size as the data buckets. This is the case if the error control code is "maximum distance separable".
3. Linearity: If we update a data bucket, we only need to send the changes (the XOR of the old and the new value) to all the parity buckets who then calculate their new values independently from their previous contents and the data bucket change.
4. Extensibility: It is possible to add additional parity buckets without having to recalculate existing parity buckets.
5. Flexible design: The ECC should be adjustable to a reasonable set of values of m and n.

Among the many ECC available we prefer a generalized Reed Solomon (RS) code. RS codes use arithmetic in finite fields, which makes them slower than for example Turbo and Tornado codes, but unlike these codes, they are maximum distance separable. Our version [LS00] optimizes updates to the first parity bucket and from the first data bucket, since it uses only bitwise XOR to implement these operations.

6 HADRAM Overview

HADRAM implements redundant, distributed storage in the main memory of a cluster of commodity computers. HADRAM uses a client-server architecture. A HADRAM application (Happ) interacts with the systems through a local HADRAM client (Hclient) who on behalf of the Happ interacts with the HADRAM servers (Hservers) that together implement the distributed memory. The Hservers are themselves applications running on top of the operating system on the nodes of a multicomputer. HADRAM provides an application with storage in the form of HADRAM blocks (Hblocks). The size of the Hblocks is not exactly determined, but a value between 1 MB to 1GB is appropriate.

6.1 Client View of HADRAM

A Happ uses a simple application interface (API) to communicate with the local Hclient. We present it in Table 1. The interface is closely related to the *NIX file system abstraction.

HADRAM Operation	Explanation
Allocate	Causes an Hblock to be allocated. Returns handle of Hblock.
Deallocate	Releases Hblock.
Read	Given handle, offset and length (in bytes) returns contents as a byte string.
Write	Given handle, allow users to write data to the Hblock.
Set Lock	Locks given Hblock of memory for exclusive use.
Release Lock	Releases a previous lock of an Hblock.

Table 1: HADRAM API

A Happ that needs to store data first asks for a new block. The Hclient eventually returns with a handle to the allocated Hblock. The Happ then can read and write data by using this handle and by specifying an offset into the Hblock. When the storage needs of the Happ change, it can (via the Hclient) request more blocks or release them. Figure 4 gives a schematic overview of the system.

Happs can share Hblocks by handing out the handle to other Happs. The local Hclients maintain a cache of the Hservers primarily responsible for the Hblocks for the Happs residing on this node. They also have the addresses of the first parity Hblock in the reliability group (Section 6.2) in case the Hserver is unresponsive. Reversely, Hclients register their address with all Hservers that store Hblocks that they have a handle on. This registration needs to be periodically renewed to flush out data on a failed Hserver.

6.2 HADRAM Reliability

Internally, H-blocks are placed into *reliability groups* (the gray boxes in Figure 4) that also contain *parity blocks*. If we cannot access an H-server, we can reconstruct the contents of any Hblock that server stores by collecting a sufficient number of Hblocks in the reliability group and reconstructing the Hblock from there. Our encoding allows us to reconstruct only parts of an Hblock from the corresponding parts of sufficiently enough Hblocks in the reliability group. This capacity is important because we can satisfy an Hclient request without having to move around MBs of Hblock data.

Each Hblock has certain metadata attached to it, more precisely, the addresses of Hclients with handles on the memory. This data is mirrored with all parity Hblocks.

The system declares an unresponsive Hserver eventually to be dead, reconstructs all Hblocks (including parity Hblocks), and places them on alternative Hservers. The system can survive even a string of failures, since block reconstruction is fast and we might have chosen a high availability level *k*.

Because of the properties of our codes, a write operation proceeds in the following way: The Hclient sends the new value to the Hserver with the Hblock. The Hserver calculates the *delta value* of the update, that is, the bytewise XOR of the old and the new data in the region of the Hblock. The Hserver then forwards the update by sending the delta value and the location to all parity Hblocks in the reliability group. The Hservers there calculate the new value of that region in the respective parity Hblock.

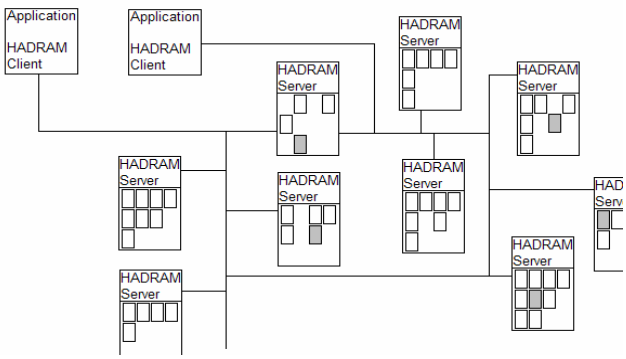

Figure 4: HADRAM architecture

6.3 Hserver Failure Detection

Hservers store data redundantly. The safety of these data depends also on the speed in which they detect failure. Failure detection in a distributed system is impossible [FLP85] if complete reliability is demanded, but in practice possible through so-called *unreliable failure detectors* proposed first by Chandra and Toueg [CHT96a], [CHT96b]. Despite their name, they are highly reliable in practice. After the seminal article, many other authors proposed similar algorithms. These protocols allow a distributed system to achieve consensus on which members are to be considered dead. The semantics of a storage system

increases their reliability. A Hserver that crashes and reboots has lost all data in main memory. Network trouble could make a Hserver temporarily unreachable, but it then retains its data. The worst case scenario is a temporary partition. In one partition, a certain Hserver belonging to the other partition is considered dead, whereas the Hclients in its partition continue to update Hblocks on this Hserver. The majority of the parity Hblocks in the reliability group is going to be in one component of the partition. If at all, the Hblock will be reconstructed on a new Hserver in that component. Once the network problems are over some Hservers and Hclients will find out that they are officially dead and that they have to rejoin the system.

To detect failures quickly, we have Hservers with Hblocks in the same reliability group monitor each others heartbeat. Second, Hclients monitor the availability of Hservers through their requests. If a request times out, then the Hclient goes to the Hserver storing the first parity block, which then in turn recovers the relevant portion of the missing Hblock and answers the Hclient's query. This also triggers the "unreliable failure detector".

6.4 HADRAM modes

In the absence of unavailable blocks, HADRAM operates in normal mode. If a Hserver appears to have failed, then the system enters degraded mode. This transition triggers a consensus protocol about which servers are considered dead and eventual reconstruction. If these operations succeed, then the system leaves the degraded mode. However, if there are too many failures to handle, then the system switches to emergency mode where through extensive broadcasting the system tries to find out which Hservers are available and what the data on them is.

We use a distributed consensus mechanism, the famous Paxos protocol invented by L. Lamport [La89] and improved by many other authors [BDFG03], [GM02], etc. to determine whether to rebuild an Hserver's data.

An Hclient contains Hserver addresses for all Hblocks of local Happs as well as for one parity Hblock. Reversely, a Hserver contains registration data for all Hclients with handles of Hblocks that it stores. We store registration data for all the Hblocks in the reliability group with a parity Hblock. If a Hserver is declared dead, its Hblocks are reconstructed on other Hservers. The registration data is then used to inform all registered Hclients about the new location of the Hblocks.

6.5 Load Balancing, Joining, and Leaving the HADRAM cluster

We use Paxos to achieve a distributed consensus on membership of Hservers. A new Hserver broadcasts its availability on the network and thus triggers a consensus protocol to include the new server in the system. Similarly, a different consensus problem allows a server to leave the system. This graceful shut-down first moves all Hblocks to alternate servers. At the core of this process is load balancing, that is also periodically invoked. Load balancing gains a global overview of the system load, in particular the number of Hblocks and then redistributes some of them to equalize the load.

7 LH*$_{RS}$: Record Based Scalable Availability

Scalable Distributed Data Structures (SDDS) harness the agglomerated power of a distributed system in such a way that the speed of operations is independent of the data structure size. The distributed version of linear hashing, LH*, [LNS93], is an SDDS that implements a dictionary data structure, that is, that allows record insertion, record look-up, record deletion, and record update given the record key and in addition also allows a parallel scan of all records for a given byte pattern. LH*$_{RS}$ is a variant with high, scalable availability [LS00], [LMS04], [MS04].

7.1 LH* Overview

LH* supports key-based operations. Each record is identified and addressed by a unique key. Based on this key, the LH* addressing algorithm determines the bucket in which the record is located. Often, though not always, each bucket resides on its own server. When a bucket is full, it sends a distress signal to a central entity, the *coordinator*. The coordinator responds by splitting the next bucket in line. A somewhat counter-intuitive, though highly effective feature of LH* is that the bucket to be split is not the one that is about to overflow, but the one pointed to by the split pointer. When a bucket split, about half of its records are relocated into the new bucket on a new server. The coordinator does not notify the clients of a split. Thus, a client is likely to commit an addressing error from time to time. In this case, the bucket receiving the request forwards the request to the bucket that – according to its view of the system – contains the record. Usually, a single forward operation is necessary, sometime, though rarely, a second one, but

never a third. When the request reaches the correct bucket, the correct bucket sends a message to the client that allows the client to update its address calculation mechanism and prevents it from ever making the same mistake twice. As a consequence, a somewhat active client makes few addressing error.

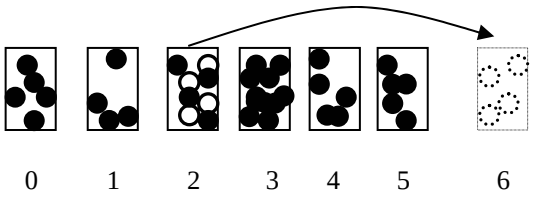

Figure 5: LH* split. Even though bucket 3 is overflowing, bucket 2 is split into a new bucket 2 and a new bucket 6 by moving about half the records (the circles) into bucket 6.

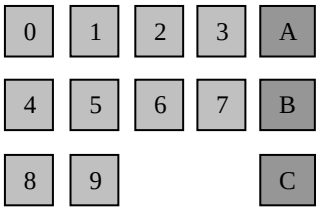

Figure 6: LH*$_{RS}$ Bucket Structure. Shown are three groups with data buckets 0 – 9. Each group has one parity bucket.

The internal organization of an LH* bucket can vary. The best implementations tend to be the ones that organize the bucket using Linear Hashing itself.

7.2 LH*$_{RS}$ High Availability

As the number of buckets and hence the number of sites on which an SDDS file is stored increase, the chances of running into unavailable or failed nodes increases. For this reason, we need to store data redundantly. LH*$_{RS}$ is a variant of LH*, that groups a number of LH* buckets, the *data buckets*, and adds to them *k parity buckets.* The number *k* depends on the size of the file and represents the number of unavailable buckets from which the structure can always recover. Figure 6 shows a small example with three groups each with one parity bucket. Notice that the third group only has two data buckets, but also a parity bucket.

We now describe how to generate the parity bucket contents and how to use it to reconstruct data if a data bucket fails. When a data record is inserted into a data bucket (determined by the LH* addressing algorithm), it receives a unique rank, an integer starting with 0. In case of record deletions, ranks can be re-used. We do not store the rank of a data record explicitly. The data records in a group with the same rank form part of a *record group*. The rest of the record group is made up of *k* parity records. Figure 7 gives an example. The first record group (the diamonds) consists of records of rank 0. Another group, the scrolls, is made up of records of rank 3. Notice that this group only has three data records. The missing data record is treated like a dummy zero record.

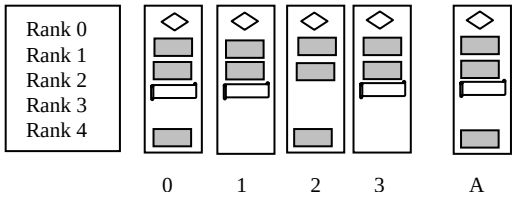

Figure 7: LH*$_{RS}$ record grouping example.

A parity record contains the rank as the key, the keys of all the data records in the group, and a field calculated from the non-key fields with an erasure correcting code (Figure 8).

When a lookup encounters an unavailable data record, it complains to the coordinator who initiates a recovery procedure. First, the coordinator recovers the record by scanning the parity bucket for the key, and then by collecting sufficient records in the record group to use the erasure correcting code to recalculate the non-key field of the record. The coordinator then recovers the missing bucket and places it on a spare server.

Since recovery moves data to a new server, any client accessing this data suffers a time-out, before it goes to the coordinator and updates its mapping of servers to net addresses.

7.3 LH*$_{RS}$ Operations

LH*$_{RS}$ allows key-based access to records and also supports a distributed scan operation. If a client wants to access a record through its key, it first uses the LH address calculation algorithm to find the bucket number of the key.

If the update is a write operation, then the bucket with the record calculates the delta value, that is, the bytewise XOR of the old and the new record data, and forwards this to all its parity buckets. The buckets send the delta values and the record identifier to all parity buckets. The parity buckets update the

respective parity record and acknowledge to the bucket. The bucket then acknowledges to the client.

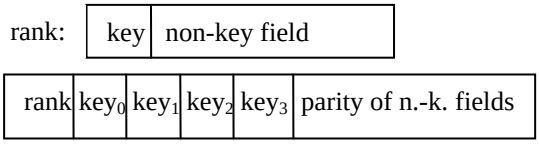

Figure 8: Structure of data record (top) and parity record (bottom)

If a client or another server runs into an unavailable server during an address based operation, it informs the coordinator who triggers a reconstruction. The coordinator is a distributed failure tolerant entity. When the coordinator receives such an alarm, and the client wanted to read a record, it first reconstructs the record from data and parity records in the same record group and sends the result to the client. In any case, the coordinator also causes the bucket to be reconstructed and located on a spare server.

LH*$_{RS}$ allows a scan operation that finds matches for given string in all the non-key fields. This scan is performed at all data buckets in parallel.

7.4 LH*$_{RS}$ Performance

LH*$_{RS}$ has been implemented by Rim Moussa at CERIA, University of Paris 9 on a 1 Gb/sec network with 1.8 GHz Pentium 4 machines. The complete performance numbers are described in [LMS04], and we only give a few highlights here. A record lookup takes 0.2419 ms individually and on average 0.0563 ms with bulk requests. To recover a record takes around 1.3 msec. Bucket recovery takes less than half a second for three out of four data buckets of 31,250 records with 104B each. With other words, LH*$_{RS}$ accesses are about 30 times faster than disk for individual records and can reach a throughput of almost 15 Mb/sec. These numbers support our thesis that distributed, highly available memory systems are quite feasible for PCC applications.

8 Research Problems

We have presented two different mechanisms to implement highly available distributed memory. LH*$_{RS}$ is implemented in a prototype version, whereas HADRAM is currently in implementation. A central data-store obviously allows a completely different PCC architecture that promises to be more scalable, to be much more modular, and therefore much easier to modify.

Distributed memory has been around for a long time, but implemented as part of the operating system. To our knowledge, only Scalable Distributed Data Structures (SDDS) [LNS93] propose to use distributed memory at the application level.

8.1 HADRAM Scalability

The proposed HADRAM design maintains explicit tables linking the Hclients to the Hservers and vice versa. In a sense, the tables at an Hclient are like the page tables and reverse page tables for virtual memory. While this simple design promises to be very efficient for small systems with a few thousands Hblocks at most, maintaining the tables will become hard for systems with hundreds of nodes offering main memory. Similarly, the number of client applications running in a control center is small. However, HADRAM in this proposed form will not scale to very large systems. While the lack of scalability of HADRAM is not an issue for systems the size of a PCC, it limits its usability. To use the HADRAM mechanism of reliability, a scalable design needs to find a way to access data quickly, to resolve group membership locally, and to distribute load through a dispersed algorithm involving only few nodes of the computer. Continued research in P2P systems and SDDS will contribute to finding solutions here.

8.2 Concurrency of Updates

A distributed memory system like HADRAM or LH*$_{RS}$ can receive updates from many competing applications. We deal with this problem at the data bucket through locking. Frequently however, any type of ordering will do at the data bucket. But even in this case, it appears that we might have a problem with competing updates to the different parity records. More in detail, application 1 might be updating Hblock 1 and application 2 might be updating Hblock 2. Hblocks 1 and 2 happen to be placed in the same reliability group. Any parity Hblock in that reliability group will receive a forwarded update from Hblock 1 and from Hblock 2. Most of the time, the updates will be to different regions of the parity Hblock. However, what happens if the two updates go to the same region? It would seem that the two updates at the parity buckets need to be made in the same order at all parity buckets. Luckily for us, the semantics of the parity update operation and the fact that we are using a linear erasure correcting code imply that the parity

updates commute. Thus, the order of the updates at the parity Hblock updates does not matter.

We still have to guarantee though that we perform exactly the same parity updates at all parity Hblocks and that if we update a data Hblock, we also update all parity Hblocks. If we fail to do so, then it will be impossible to reconstruct data. In [LMS04], we propose a method yielding a guarantee based on acknowledgements for LH*$_{RS}$. We could use the same technique for HADRAM, but it would be less wieldy because in HADRAM, we can update any portion of the Hblock.

Instead, we are currently implementing and measuring a simple log based mechanism that allows all us to bring all data and parity Hblocks to the same view before recovery begins. This implementation uses cumulative acknowledgments and should prove to be more efficient.

8.3 LH*$_{RS}$ Coordinator Design and Implementation

SDDS coordinators like the LH*$_{RS}$ coordinator have a small, but essential role to play. While they probably do not pose a performance problem even for large SDDS, the system cannot function without them. Their basic design is simple using known algorithms such as Paxos. Unfortunately, no one has yet implemented and measured a failure tolerant SDDS coordinator.

8.4 LH*$_{RS}$ Load Balancing

The LH* address algorithm extends a LH* file adds more and more buckets, as the file grows. The basic scheme places a single bucket on a new server. In an applications like the central store of a PCC, the number of servers changes very little. We therefore store LH* buckets on *virtual servers*. Each actual server contains a number of these virtual servers. When a server fails, all the virtual servers located on it are lost and reconstructed on "spare virtual servers" located on other actual servers. We can control the allocation through a distributed address resolution protocol that maps the address of a virtual server to a physical server. Thus, to calculate the address of a record, we first calculate the bucket address, i.e. the virtual server address, with the LH* algorithm and from that the actual solution. For a small to medium system, we can implement the bucket to actual server addressing with tables and table look-up, i.e. an explicit mapping. This mapping should not map buckets belonging to the same LH*$_{RS}$ reliability group to the same server, for the failure of that server destroys the availability of more than a single bucket and thus causes data loss for sure or with higher probability than designed. The mapping needs to be updated every time when a server leaves the system gracefully or enters the system. The location of the spare buckets is part of the mapping algorithm so that the mapping does not need to be updated when a server fails.

As we have seen, the bucket to server mapping needs to avoid placing buckets in the same reliability group on the same physical server. In addition, it should balance the load of all servers, which amounts to allocating about the same number of buckets to each server even as the number of buckets varies. A result by Choy, Fagin, and Stockmeyer shows that this is difficult to solve exactly [CFM96], but hopefully, an approximate solution is not so difficult.

If the number of physical servers is large, then good approximately load-balancing, and efficient assignments of buckets to servers exists [HM03], [HM04], but even these algorithms could be improved.

8.5 Geographically Dispersed PCCs

Distributed memory decouples many PCC applications from each other. We could take advantage of this modularization to geographically disperse the PCCs so that it would remain functioning despite total loss of one or more of the dispersion sites. The PCC as a whole could then continue to function despite a local catastrophe such as a fire or a limited terrorist attack.

Existing technology such as Virtual Private Networks (VPN) allow using existing internet connections while protecting the contents of communication cryptographically. Unfortunately, the availability of such a VPN network can be hard to estimate, since several VPN links might pass through the same physical cable or through the same router location. In other words, seemingly high availability at a higher level can hide single points of failure in the underlying infrastructure.

Network delay is another problem for geographical dispersed VPNs and depends on the ability of applications to cache records locally in order to avoid multiple repetitive fetch operations. Experimental confirmation of the viability of this structure is needed before we can implement PCC in this manner.

8.6 PCC Architecture

A central, highly available, but slower repository for all data records in a PCC opens up more possibility

for PCC design than merely allowing outside status assessment. A discussion of these possibilities is beyond the scope of this paper.

9 Conclusions

In this paper, we proposed the use of highly available distributed memory as the central component of a new architecture for power control centers. This architecture allows cheaper, more robust, and more modular designs for power control centers. We have identified a number of important research questions. We are currently implementing and measuring central components of this design.

References

[BDFG03] R. Boichat, P. Dutta, S. Frøland, R. Guerraoui: Deconstructing Paxos. In *ACM SIGACT News*, vol. 34(1), 47-67, 2003.

[CHT96a] T. D. Chandra, V. Hazilacos, and S. Toueg: The Weakest Failure Detector for Solving Consensus. In *Journal of the ACM*, vol. 43(4), p. 685-722, 1996.

[CHT96b] T. D. Chandra, S. Toueg: Unreliable Failure Detectors for Reliable Distributed Systems. In *Journal of the ACM*, vol. 43(2), p. 225-267, 1996.

[CFS] D. Choy, R. Fagin, L. Stockmeyer: Efficiently Extendible Mappings for Balanced Data Distribution. In *Algorithmica* 16, 1996, pp. 215-232.

[FLP85] M. Fischer, N. Lynch, M. Paterson: Impossibility of Distributed Consensus with one Faulty Process. In *Journal of the ACM*, vol. 32(2), p. 225-267, 1985.

[GM02] G. Chockler, D. Malhki: Active disk paxos with infinitely many processes. In *Proceedings of the 21st ACM Symp. on Principles of Distributed Computing (PODC-21)*, 2002.

[HM03] R. Honicky, E. Miller: A fast algorithm for online placement and reorganization of replicated data**.** In *Proceedings of the 17th International Parallel & Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.

[HM04] R. Honicky, E. Miller: Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution**.** In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

[LAC00] Q. Li, M. Allam, D. Cieslicki: Improving the availability and fault-tolerance of network-attached drives storage system. In *Proc. Intern. Conf. on Parallel and Distributed Processing Techniques and Applications*, PDPTA 2000, Las Vegas, USA.

[La98] L. Lamport: The part-time parliament, *ACM Transactions on Computer Systems (TOCS)*, v.16 n.2, p.133-169, May 1998.

[LNS93] W. Litwin, M.-A. Neimat, D. Schneider: LH*: Linear Hashing for Distributed Files. *ACM-SIGMOD Intl. Conf. on Management of Data*, 1993.

[LS00] W. Litwin, T. Schwarz, S.J.: LH*$_{RS}$: A High Availability Scalable Distributed Data Structure using Reed Solomon Codes. *Proc. of the 2000 ACM SIGMOD Intern. Conf. on Management of Data, May 16-18, 2000, Dallas, Texas*, also *SIGMOD Records*, vol. 29 (2), June 2000, p. 237-248.

[LMS04] W. Litwin, R. Moussa, T. Schwarz: LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure, *Transactions on Database Systems (TODS)*, (submitted May 2004.) Also: CERIA Research Report, April 2004, http://ceria.dauphine.fr/rev-thomas-rim-wl-june.pdf

[MS04] R. Moussa, T. Schwarz: Design and Implementation of LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure, *WDAS 2004*, Lausanne. To appear in: *Distributed Data & Structures 6*, Carleton Scientific Proceedings in Informatics.

[Pl97] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. In *Software Practice and Experience.* Vol. 27(9). 1997. p. 995-1012.

[Pr89] F. Preparata: Holographic dispersal and recovery of information. In *IEEE Transactions on Information Theory*, 35(5), p. 1123-1124, September 1989.

[Ra89] M. Rabin: Efficient dispersal of information for security, load balancing, and fault tolerance. In *Journal of the ACM*. Vol. 36(2). p. 335 – 348, 1989.

[Re01] J. Rehtanz: Autonomous Systems and Intelligent Agents in Power System Control and Operation. Springer. New York 2003.

[Se04] Seagate Cheetah 10K.6 datasheet: (www.seagate.com).