Metadata management for EIS

Pierre Karpman

September 5, 2011

Internship report, Magistère Informatique et Télécommunications, ENS Cachan Bretagne and University of Rennes 1

Internship from July 1, 2011 to August 31, 2011 at the Institut für Informatik, Heinrich-Heine-Universität Düsseldorf

Supervised by:

Michael Schöttner, Kim-Thomas Rehmann

Heinrich-Heine-Universität Düsseldorf

Abstract

In this report, we present a design for EIS, a new in-memory distributed file system optimized for intensive parallel file creation and file access. The main use for such a system is as the storage layer for HPC applications that typically need a big number of files to store intermediate data. The main contribution of our design is a way to transform a storage system with a flat namespace into one with a hierarchical namespace, suitable for a file system. We completed the first step of implementing EIS by designing and implementing a DHT to be later used as the flat storage layer. The implementation has been done with the ECRAM framework, which provides applications with a shared object space.

Keywords: In-memory data storage, file system, distributed hash-table, distributed metadata, scalability, ECRAM.

Gestion de metadonnées pour EIS

Pierre Karpman

5 septembre 2011

Rapport de stage, Magistère Informatique et Télécommunications, ENS Cachan Bretagne et Université de Rennes 1

Stage du 1er juillet 2011 au 31 août 2011 à l'Institut für Informatik, Heinriche-Heine-Universität Düsseldorf

Encadré par :

Michael Schöttner, Kim-Thomas Rehmann

Heinrich-Heine-Universität Düsseldorf

Résumé

Ce rapport présente EIS, un système de fichier distribué conçu pour fonctionner en RAM et optimisé pour la création concurrente de fichiers. Le principal intérêt d'un tel système est de servir de moyen de stockage pour des applications de calcul distribué. La principale contribution de ce rapport est un moyen de transformer un système de stockage non hiérarchique en un espace de nom hiérarchique approprié pour un système de fichier. Nous avons réalisé la première étape de l'implémentation de EIS en concevant et implémentant ECHT, une table de hachage distribuée. L'implémentation a été réalisée en utilisant le framework ECRAM qui fournit un espace d'adresse unique à des applications distribuées.

Mot-clefs : Stockage en RAM, système de fichier, table de hachage distribuée, gestion de metadonnées distribuée, évolutivité, ECRAM.

1 Introduction

Data storage and file systems are not new in computer science, and it is a traditional topic in operating systems research. There is a huge amount of papers discussing how to build efficient file systems for all kind of applications: local or distributed file systems, optimized for small or big files, with different operation semantics, etc. Therefore, it might be dubious what relevance can still be found in designing new file systems. And yet, the ever-evolving hardware and changing use-cases call for appropriate new designs.

In this report, we propose a new tentative design for EIS, a distributed file system which characteristics we think may be of some interest.

Our motivations were twofold; first we wanted to design and implement a file system as a proof of concept for the ECRAM framework [2, 4, 7], in order to show how distributed software transactional memory (DTM) could partially be used to build a not-so-trivial, scalable application (that is EIS).

Second, we wanted to experiment with in-memory file systems, which use RAM as the primary storage hardware and not only for caching. It has recently been argued that the steadily decreasing cost of RAM makes the incentive to use it for durable but frequently requested data stronger and stronger [6]. In our case, we are typically interested in storing intermediate data used in high-performance computing applications (e.g. bioinformatics applications), focusing thereby on systems confined to a single geographical area and excluding storage on global worldwide systems like the Internet.

The result is something we would call an expendable in-memory distributed file system, which can be set up quickly for the duration of an application to provide highly available data storage with a traditional file system interface. The design was tuned to enable quick and highly parallel file creation in a same directory and very fast file searching. It is also expected to cope well with a big number of files.

The design of EIS paved the way for the design and implementation of another new component for ECRAM. As it can be seen later, EIS uses an efficient distributed lookup service at the core of it design. It became then necessary to add such a service to ECRAM to allow for a future prototype to be implemented. This resulted in ECHT, a distributed hash-table designed to allow an efficient implementation with ECRAM and with similar scalability requirements than EIS in mind.

The rest of this report is organized as follows. Section 2 briefly presents the ECRAM framework used for the implementation. Section 3 presents the design of EIS and an analysis of its expected performance. Sections 4 and 5 present the design of ECHT, and an experimental evaluation of its performance. We discuss related work in section 6, and section 7 concludes.

2 ECRAM Background

ECRAM is a framework that provides in-memory data storage for distributed applications [2, 4, 7]. Data are stored in the form of binary objects which are

globally identified by the same ID among all participating application nodes. Keeping all objects in RAM allows to achieve low access-latency for arbitrary data access patterns.

Replication and backup of the objects are dealt with internally, to ensure high data availability and reliability. It is possible to ensure consistency in case of concurrent accesses by using transactions on distributed shared storage, so that objects can in fact be considered as stored in a distributed software transactional memory (DTM), with several consistency models being available.

3 The design of EIS

The core of EIS design is a way to transform a distributed storage system with a flat namespace into a storage system with a fully hierarchical namespace, while retaining the good scalability properties and distributed nature of the former. In particular, a great care is taken to ensure that the transformation will not prevent parallel file creation and file access within a directory.

The design we present here is tailored to the functionalities provided by ECRAM; it is however probably possible to adapt it to other systems. It uses a DHT for its underlying flat storage system, strongly relying on the ability to store key/data pairs with arbitrarily chosen keys.

3.1 Hierarchical namespace construction

The first step in namespace construction is simple, and consists in associating a file's metadata with its path as a key. For instance, a pointer to the metadata for the file */Food/clafoutis* is located under the key **H**(*/Food/clafoutis*), where **H**(.) stands for the DHT hash function, supposed to be virtually conflict-free.

Accessing already existing data is therefore easy and fast, be it a file or a directory, and the challenge lies mostly with the management of directory metadata (which is not so surprising, as this is where we actually keep the information about the hierarchical nature of the namespace).

In order to preserve scalability for parallel file creation, most of the directory metadata (that is the list of files inside the directory) is actually split into local *tables*, separately and concurrently managed by different nodes taking part in EIS and having an interest in the directory. These local tables are then pointed to by the global, easy-to-locate metadata. For instance, in the previous example, a DHT lookup for the key **H**(*/Food*) would yield a series of pointers to local tables.

More precisely, the directories metadata are only useful when one wants to list the content of a directory. They do not need to allow efficient searching for a particular file, as this can be done instead by a lookup in the DHT. The actual information in the local tables therefore contains little more than a list of pointers to other files' metadata, and is typically updated only on file creation and deletion. It also keeps a reference to the global metadata of the directory. It should be noted that since those tables are local, the only way for some node to access the table of another is through remote-procedure calls. The expected performance impact of using local tables is discussed in section 3.3. Finally, the placement policy for the metadata for a new file is easy: it is put in the local

table managed by the node asking for the file to be created. A local table is created and added in the global directory metadata list if it didn't exist yet.

Now for file metadata, additionally to directly useful metadata like everything from pointers to the file's data or access rights, specific information is kept to allow efficient file deletion. It consists in fact of a pointer to the entry for the file in the directory local table to which the file belongs. Then, if one wants to delete the file, this information is used to quickly perform the operation on the metadata, either locally or by the way of a remote call, without any need for further metadata lookups or data structure traversal. Finally, the file metadata can be used as a witness for the file existence, therefore making possible to probe the existence of a file of a specific name with a constant number of operations.

The structure of EIS described so far is summarized by the figure 1, the colour-code bearing the following meaning:

- red is for information located inside the DHT;
- yellow is for information located in ECRAM's single-ID space, with concurrent access managed with transactions;
- blue is for information located in ECRAM's single-ID space that shall not be accessed concurrently.

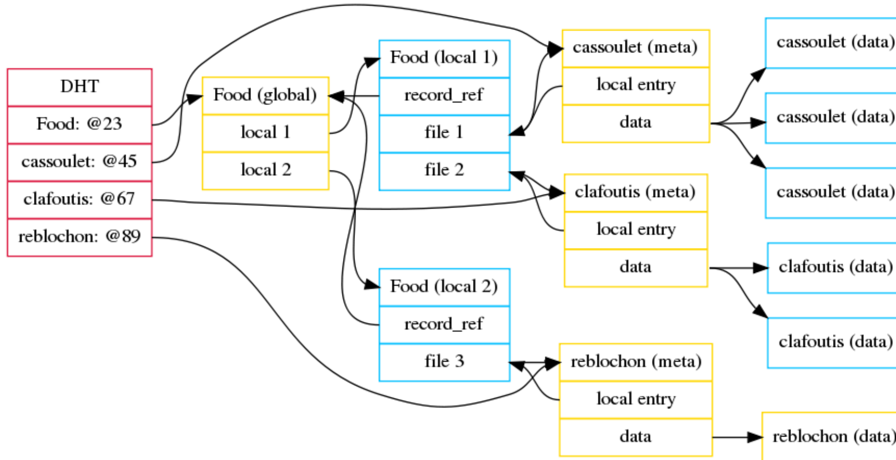

Figure 1: EIS structure overview

It is now easy to see again that all data and metadata can be retrieved quickly from the DHT, while hierarchical information is enforced by the directories metadata. We justify our choices for managing concurrency with transactions or not by discussing a few scenarios.

In the case of global directory metadata, we do not expect it to be modified often, as it only keeps track of the nodes which hold a local table for the directory. Even if a lot of nodes were to repeatedly create files inside a directory just to delete all of them shortly after, clever implementations could keep the pointers to the empty local tables entries for a while before actually modifying

the global metadata. Therefore, we think that the cost of using transactional control would be bearable in practice.

In the case of file metadata, there is no particular point in the design that will influence the likeliness of its modification, as this is wholly specific to the file users' behaviours. However, even in case of highly concurrent modifications of a file, we think that the use of transactions is worth trying, especially because only the most recent users of a file are ever notified of its modification. More reflexion on the subject would be interesting, though, and we think that it would be particularly valuable to test how weak consistency models can accommodate a heavy load.

As for files' actual data, as can be seen in figure 1, we think it could be beneficial for users themselves to choose whether the data needs to be accessed under transaction control or not. The use of transactions could allow concurrent and consistent modifications of different parts of a single file, but resorting to exclusive access is of course a possibility.

3.2 File system specifics

The full design of EIS being not finished for the moment, there is no well-defined semantics or consistency models yet for the various file system operations. As suggested just above, we think however that experimenting with different consistency models for transactions can be interesting, but this should be supported by relevant testing and use-cases.

The complete metadata for files has not been defined either, and in particular there is no exact plan yet for how the files' data blocks should be defined and stored. We think that using RAM as the primary storage medium should greatly influence this point of the design, as a bigger variation of block sizes is possible than when a hard drive is used. Therefore, it should be possible to use only a few blocks tuned to the files' sizes, probably pre-allocating space for files to which data is frequently appended.

Finally, we consider that the fault management of the file system should be dealt with by the lower layers of the system, this being the DHT in our case.

3.3 Theoretical performance analysis

In this section, we discuss more precisely the cost of each file system operation in terms of accesses to the DHT, overhead due to the use of transactions, and possibly other kind of communication between EIS nodes.

Reading and writing to a file when its name is known is simple and cheap (most of the time). In case of reading, one access to the DHT is needed to retrieve the file's metadata, and then one memory access is needed for each data block. The actual access to data blocks is dealt with by ECRAM's object management, which ensures that the latest version of the metadata and data is accessed. In the worst case, the metadata and all of the data need to be fetched over the network, but in the case of frequently accessed but seldom modified files, local replicas of the metadata and of the data blocks will hopefully be accessed instead.

In case of writing, the number of accesses is the same, but the data modification implies that replicas of the file's metadata held by other nodes need to

be invalidated. Therefore, one network communication is needed for every node that held a valid replica of the metadata prior to its modification, but nodes that already held obsolete versions of the metadata need not to be contacted. The same holds for the file's data blocks if those are replicated.

In both cases, concurrent access to the file's metadata (and possibly to the file's data) will be detected by ECRAM's DTM system.

In summary EIS is friendly with highly parallel file reading when the data is seldom modified, but frequently updated data will cause network communication proportional to the number of nodes actively interested in it.

Creating or deleting a file or a directory is quite simple too. In case of file creation, a lookup in the DHT is performed both for the file's parent directory and for the file itself, and an appropriate error is returned if the former doesn't exist or if the latter already does. If everything went fine, a new metadata structure is created for the file and put in the DHT. It is then inserted in the creating node's local table for the parent directory, creating it if necessary. The total cost for the successful creation of a file is then two read access and one write access to the DHT, one access to the parent directory's local table, plus an occasional modification of the parent directory's metadata. It should be noted that the latter is the only already existing global information that is potentially modified, and therefore the only potential cause of network communication with other nodes to invalidate locally held replicas. Most of the time, though, nodes would be able to create files in the same directory in a completely parallel fashion, without more overhead than the three access to the DHT and the one access to the local table.

The process is similar in case of file deletion, the only difference and potentially additional cost being that the access to the parent directory's local table may need to be done by the way of a remote call to the node actually holding the table. In any case, finding the proper entry to delete in the local table is easy thanks to it being referenced in the file's metadata. If a local directory table were to be emptied by the deletion, its reference in the global directory metadata may be removed, although it may be interesting to keep it if new files are to be created soon; this is in any case implementation-specific.

The process for directory creation and deletion is completely similar, with the additional check that a directory must be empty to be deleted (checking for emptiness is cheap, unless local tables with no entries can be kept in the global metadata; in this case, it becomes necessary to ask every node with a registered local table if this one is empty or not).

In summary, EIS is friendly with both file and directory creation and deletion, and should be able to withstand highly parallel file creation within a single directory.

Finally, listing the content of a directory is quite simple, but can be expensive. What is basically needed in this case is to retrieve the directory's global metadata at the cost of one DHT lookup, and then to scan the list of local tables and to send a remote request to every corresponding node, asking for the list of the files they locally know of. Therefore, the number of nodes that need to be contacted is equal to the number of nodes owning a local table for this directory (that is the number of nodes that created files in this directory). It

should be noted that the amount of data fetched over the network is not bigger than when using a remote centralized metadata for the directory, only the number of network interrupts will increase. In any way, we think that listing a directory's content wouldn't be needed often in practice, as the targeted users of EIS are most of the time expected to know the names of the files they want to access (when those are intermediate data). Therefore, the case of a big number of nodes writing in the same directory and frequently listing its content is considered pathological, and is deemed acceptable not to perform so well with.

The expected absolute cost of EIS operations are summarized in table 1.

Access type	Freq. access on few nodes	Freq. access on a lot of nodes
Read	😁	😁
Write (w. conflicts)	😐	😣
Search	😁	😁
Create/Delete	😁	😁
List	😐	😣

Table 1: EIS expected performance

If correct, we think that these results justify our design, by fulfilling the scalability objectives we set to ourselves. We hope that a future implementation will validate those claims.

4 The design of ECHT

We now discuss the design of ECHT, a simple distributed hash-table which purpose is to serve as the flat storage mechanism in a future experimental implementation of EIS with ECRAM.

Unlike some DHT designs [11, 9], ECHT needs limited global information sharing. This simplifies the partitioning of the keyspace and message routing, at the cost of scalability.

Yet we believe that sharing some information is relevant because the number of nodes is relatively small (up to a thousand, confined to a single geographical location) and the rate of dynamic nodes arrival or departure is expected to be quite low. Therefore, we do not expect the amount of shared data to be more than a few KB and to be updated more than maybe once every few minutes while in steady state.

Another difference is that we target private local networks rather than wide area networks, which implies first than the maximum round-trip time is very low (typically a fraction of a millisecond), and second that we need not to care about malicious nodes or authentication.

The following sections describe the design choices made for ECHT, and in some cases how something would be implemented when using the functionalities provided by ECRAM.

4.1 Hash function

We currently use MD4 [8] as a hash function. The rationale for this choice is that we do not realistically expect collisions from 128 bits keys generated by a cryptographic hash function (even from a broken one), and uniqueness of the keys is a key requirement for ECHT. The advantage of MD4 is that it is easy to implement and faster than its secure counterparts like SHA224. Fast modern functions like the BLAKE family could be meaningful alternatives when security is needed.

4.2 Synchronisation and globally shared information

The only shared information between the ECHT nodes is a global counter of how many nodes are currently part of the system, as well as a routing table to map each node's unique ID and IP address. The first information is used to compute the ID of the node responsible for a given key, and the second to actually send messages to other nodes.

The node IDs are determined at node start-up, and is the only information that needs global synchronisation, in order to ensure that they remain unique and contiguous. A node may change its ID during its life as part of the node departure protocol.

As we use offline computation to determine the destination of a message, the globally shared information is the only information needed by the nodes. Sections 4.3 and 4.4 detail how it is used to perform a lookup, and how it is maintained when nodes are joining or leaving. Section 4.6 addresses the potential impact and benefits of using weak consistency for the global information.

4.3 Keyspace partitioning

We take advantage of the fact that the number of nodes in the system is known to use a very simple keyspace partitioning scheme. The node responsible for a given key is simply the one which ID matches the longest possible binary prefix of the key. As all the node IDs are contiguous by design, this can be computed without any other kind of communication.

This is considered to be efficient as it splits the keyspace evenly; when a node is joining or leaving, a single other node needs to modify its key-space, effectively changing it by a factor of two.

In effect, this scheme virtually splits the keyspace along a balanced binary-tree, as can be seen in figure 2. Therefore, the number of keys managed by a given node should not differ from the number of keys managed by any other node by more than a factor of two on average.

4.4 Node arrival and departure

When a node is joining the system, it retrieves a copy of the global information and allocates itself a new node ID, equal to the new number of nodes in the

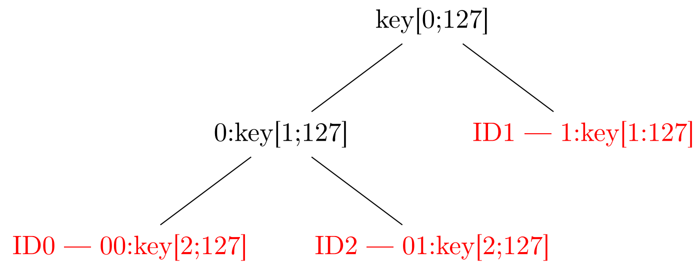

Figure 2: ECHT virtual partitioning tree for three nodes

system minus one. Then it contacts the node which ID matches its own ID the best, notifies it of its arrival, and asks for the data that now falls into its key space. This is illustrated in the following example.

Assume we currently have 6 nodes in the system. The node IDs can be coded on 3 bits, and can therefore match on 0, 1, or 2 bits. Prefix matching is done with the IDs coded in little-endian (least significant bit first). Now a node is joining and gets the ID $6 = 011$. The node computes the unique smaller ID which prefix matches on exactly 2 bits, and finds that it is $2 = 010$. Then it contacts this node of ID 2 (which was up to now managing all data with keys starting with 01), and asks for the data of keys starting with 011 (statistically one half of all the data the node 2 is managing). Now the new node updates the global routing table, and every request for those keys will be directed to node 6.

Node departure is handled symmetrically. When a node is leaving, the node with the highest ID merges back the keys it was managing with the node which best matches its ID. Then, either the node leaving was the node of highest ID and the operation is over, either it was some other node which will now trade its ID and all the keys it was managing with the node of highest ID. This effectively reduces the highest ID by one, and keeps all IDs contiguous. The following example illustrates the node departure protocol.

Assume we have 7 nodes in the system, and the node 5 is leaving. It notifies the node of highest ID (6) that it needs to merge its data with another node and then trade IDs. The node 6 searches for the ID best matching its own one and finds that it is 2 (cf. the above example). Then it transfers all of its data to node 2, gets the data managed by node 5, assumes node 5's identity by changing its own ID, and updates the routing table accordingly. It then finally notices the former node 5 that it can terminate.

4.5 Fault tolerance

Fault tolerance needs mostly to address two potential problems. The first is the loss of the global state of the system: the number of participating nodes and the routing table. This is clearly not an issue as this information is fully replicated on each node, using transactions for consistency management.

The second is the loss of data due to node failure. This can be managed in a rather efficient way by using the already-existing fault-management of ECRAM, by keeping all the data structures used in ECHT as ECRAM global objects.

This allows nodes to give the object ID pointing to the root of their data to a few backup nodes, all the actual data replication being done by ECRAM. When a failure is detected (either by ECRAM or by an ad-hoc implementation-specific technique), the backup nodes for the failed node determine which node will temporarily take over the failed node's data (again, this is implementation-specific). This node will then engage in the node departure protocol while assuming the identity of the failed node.

4.6 Consistency management

In order to keep the global information consistent, we manage the node counter and routing table by using the transactions of ECRAM. Whether end-to-end locking of the information is necessary when a node is joining or leaving or if parallel requests for key-space partitioning can be satisfied is for the moment left as an implementation choice, but we think the parallel option may be challenging when strict consistency of the global information is needed, considering the contiguity constraint of the node IDs (cf. section 4.2).

However, when key duplicates are acceptable for a short period of time, quicker parallel updates of the global information could be possible during the joining process, by using less-than-strict consistency for the transactions. In this case, contending nodes would eventually need to merge their data, and all but one of them would have to start a new joining process.

Weak consistency could also be used when a node is leaving. In this case, nodes may need to pull up-to-date global information when they are unable to contact some node, in order to ensure that it is still part of the system.

All cases need however to be further investigated before bringing a definitive answer.

5 Experimental results for ECHT

We implemented an early prototype of ECHT using ECRAM in order to assess its scalability and to compare actual behaviour with what the analysis of the design would predict.

All tests were performed on up to 32 nodes of a small cluster. Larger-scale testing would be of course interesting, but was not possible due to a limited amount of time and a lack of immediately available suitable testing infrastructure.

The cluster nodes were fit with AMD Opteron 244/246 CPUs with 2 to 8 GB of RAM. The first 20 nodes shared the same 1 Gbit switch, the 12 others being plugged on another similar switch.

5.1 Evaluation of the keyspace partitioning scheme

In this section we try to validate the claim we made about the well-balanced property of the keyspace partitioning. We set up a system with a few ECHT nodes and instructed each of them to insert 1000 random keys. Then, we compared how many keys were managed by each node, thereby assessing the behaviour of the partitioning scheme.

The results have been plotted in figure 3, where we can observe the minimum and maximum number of keys obtained for a single node in function of the total number of nodes, along with the 50th and 90th percentile. The average number of keys stored in a node is unsurprisingly steady at 1000, as few conflicts from the random-number generator are to be expected at this rate, and none from the hash function itself.

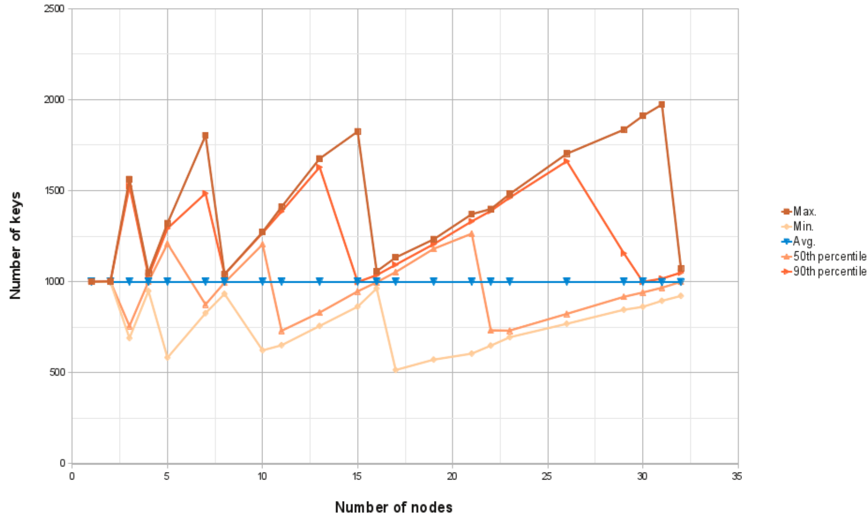

Figure 3: ECHT keyspace partitioning

These results mirror well what we expected: in the best case, when the number of nodes is a power of two, every node shares the same amount of key; otherwise, some node will assume responsibility for more keys than others, but never more than twice the least managed amount.

5.2 Performance of basic operations

In this section we focus on the average time needed to put and retrieve some data from ECHT, depending on the number of nodes. We are also interested in how this time can vary from a node to another.

From the design, we would expect the access times of a single node to be split in two groups: local and remote accesses. A remote access is of course more expensive than a local one, but keeps the overhead at the minimum as only one pair of remote messages needs to be exchanged. We should however note than due to the current state of ECRAM, an additional pair of messages always needs to be exchanged with another node, regardless of the locality of the access.

Therefore, considering that the more nodes, the more remote accesses, we expect the average access time to increase slightly with the number of nodes, while being bounded by the access time for a node performing all accesses remotely.

We performed our experiments by setting up an ECHT system and repeatedly asking every node to perform a few hundreds of the operation we wanted to measure.

We plot the average access time for a *get* access in ECHT in function of the number of nodes in figure 4, along with the 10th and 90th percentile of the node population in order to check for variations from one node to another.

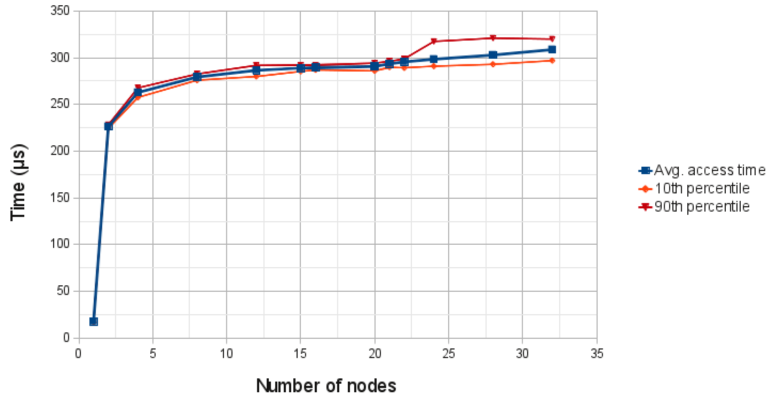

Figure 4: ECHT average *get* access time in microseconds

We believe that the obtained results justify our expectation of a very moderate increase of the access time when the number of nodes grows. It also shows great homogeneity between the nodes, although the data dispersion increases slightly with more than 20 nodes. However, we think this to be directly caused by the heterogeneity of the testing infrastructure itself (cf. section 5), and not by the software.

We can also notice that the current cost of an access excluding all network communications is moderate, as can be seen from the access time on one node being slightly less than 20 milliseconds.

We obtained similar results when measuring the time for synchronous *put* accesses, plotted in figure 5, although the nodes look a bit more heterogeneous when performing this operation.

In conclusion, we think that the design for ECHT is sound and allows for efficient implementations, as proven by experimental results. It should be noted, however, that no testing has been done yet to assess the cost of reconfiguring an existing ECHT system by adding or removing some nodes; and this are expected to be the costlier operations of all. Therefore, we state again that we believe our design to be viable when the number of participating nodes is seldom modified, but it is probably not so useful in other cases.

6 Related work

A lot of work has been done on efficient data storage for distributed applications. Lustre [1] is in widespread use among supercomputers, and shares a lot of our objectives. Its approach is however quite different, as it uses RAID for storage and fault-tolerance. It also centralizes metadata management within a single server.

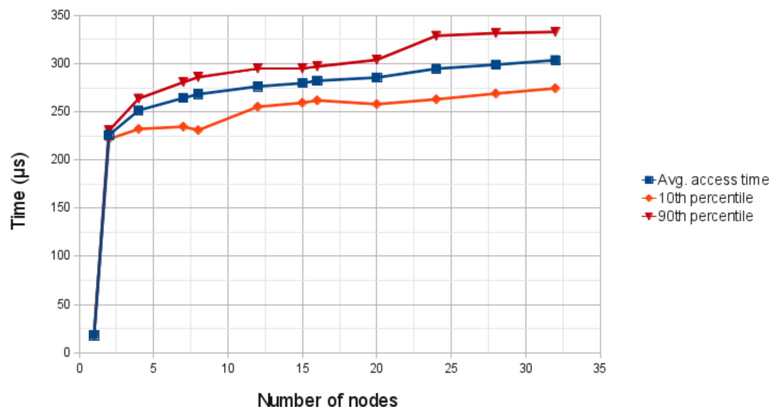

Figure 5: ECHT average synchronous *put* access time in microseconds

Several sophisticated file systems exist to serve as the backend storage for MapReduce applications. They share some of our objectives of high data availability and efficient parallel accesses, but are tuned for much bigger files and different access patterns. Some of them like the Google File System [3] use centralized metadata management. BlobSeer [5] and the Hadoop Distributed File System [10] both distribute their metadata, and BlobSeer also uses a DHT for that purpose. However, both centralize their namespace.

The closest existing file system to ours may be Ceph [12], which can efficiently serve as the storage service for HPC applications. It also distributes its metadata and is tuned for similar access patterns, notably efficient concurrent file creation within a single directory. The main differences of our design is the use of a natively load-balancing DHT for metadata management, and using a transactional memory to maintain the consistency of cached metadata. It is also not an in-memory file system.

7 Conclusion

This report presented an early design for EIS, an in-memory distributed file system to be later implemented with the ECRAM framework. We outlined the technique we use to build a scalable distributed namespace, and its relevance to the targeted use of EIS. We also presented a design and some experimental results for ECHT, a distributed hash-table to be used in the future EIS implementation.

Future work on the subject should focus on implementing a prototype for EIS and conducting significant large-scale benchmarking. In particular, the exact impact of the different possible choices for consistency management should be assessed.

References

[1] Lustre, 2010. http://wiki.lustre.org/.

[2] Ecram, 2011. http://www.cs.uni-duesseldorf.de/AG/BS/english/Research/ECRAM.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[4] Marc-Florian Muller, Kim-Thomas Moller, and Michael Schottner. Commit protocols for a distributed transactional memory. In *Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications. In *IPDPS*, pages 1–11, 2010.

[6] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43:92–105, January 2010.

[7] Kim-Thomas Rehmann, Marc-Florian Müller, and Michael Schöttner. Adaptive conflict unit size for distributed optimistic synchronization. In *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 547–559. Springer Berlin / Heidelberg, 2010.

[8] Ronald Rivest. The md4 message-digest algorithm, 1992. http://tools.ietf.org/html/rfc1320.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001.

[10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–10, 2010.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[12] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design*

and implementation, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.