Crowd storage for a distributed database

David Christ
University of Trier
david.christ@uni-trier.de

Abstract

This work lays the foundation to an open social network for cooperative error reporting and analysis. The project's primary goal is to build an error report database, which is enriched with semantic information by its users. In order to provide an independent storage system, in this paper the feasibility of outsourcing a database to the end-users is going to be shown. First, suitable and available computers are evaluated. Afterwards, a setup of Cassandra—a scaleable, fault-tolerant, distributed database—is presented, and benchmarked. It is shown that the database could run as a background process, while not constraining the user in utilizing its computer. Furthermore, the number of required participants turns out to be much lower than what might be expected. Also, it is pointed out that the storage network might be accompanied by company-owned computers and servers in a way, not inconsistent with the prospect. The paper concludes that providing volunteered crowd storage for a distributed database is feasible.

Categories and Subject Descriptors H.3.4 [*Systems and Software*]: Performance evaluation

General Terms Measurement, Performance

Keywords distributed databases, crowdsourcing

1. Introduction

This paper aims to allow for an open social network providing cooperative analyzing of bugs originating from various software. By making error reports available to the public in a social network, company employees can be accompanied by volunteers in debugging their software, while also providing a convenient channel for communication. Primary goal is to build a structured error report database, which is enriched with semantic information by its users. Such a database facilitates finding patterns and reasoning about cross-references. Like in common social networks, users can get together in groups, here respectively each representing one root cause for a bug. Its members are composed of end-users affected by the bug as well as developers trying to solve it. These groups collect all the error reports associated to their bug and provide a forum for discussion about fixing it. As an emergent behavior, the social network allows for building reputation statistics for users or pointing out experts often solving certain kinds of problems.

In order to collect error reports, the project uses OS-inherent remote crash reporting software. This gathers information about an error as it appears on a computer and sends it back to the manufacturer via internet. The manufacturer in turn aggregates and sorts these reports, before making them available (solely) to the corresponding developers. An open approach allows (small) software manufacturers to operate a large-scale remote crash reporting facility, relying upon OS-inherent feedback mechanisms and storage servers provided by the public without conditions.

As the system should be independent, it targets not to rely on centrally owned and controlled elements. Generating crash reports is done consumer-side anyway, but providing a free storage system is a non-trivial task. Therefore, this paper focuses on the feasibility of a database hosted through end-user computers—a crowdsourcing of storage. Remote crash reporting systems activate on demand. In contrast, a database runs as a daemon. Hence, questions of interest will be, what impact the low availability of end-user's computers has to a storage system and whether the system is lightweight enough to be a background task on them.

After giving an overview on related work, emerging problems are discussed, a proof-of-concept setup is presented, and a benchmark on it is performed.

2. Related Work

The work presented in [7] and [5] reveals internals of *Windows Error Reporting* (WER), Microsoft's mechanism to collect end-user error reports. One of the sys-

CloudDP '13 April 14th, 2013, Prague, Czech Republic

tem's core features is the ability to assign the same unique identifier to different reports originating from the same root cause. Hence, when an error occurs on a computer, only its specific UID is transmitted. If the bug is already known, only a counter gets incremented, else additional data is requested. Similar mechanisms are Apport for Ubuntu Linux or CrashReporter for Mac OS X. Breakpad in contrast is a multi-platform crash reporting system to be integrated in a software directly—used amongst others in Mozilla's Firefox.

Emergent behavior of social networking in combination with software development has been researched for the social coding platform GitHub in [4] and [12]. Given that GitHub users infer information about the activity in a project or the relevance of it to the community via this platform, it can be concluded that similar inferences can be drawn in a social-network-like crash reporting system.

There have been benchmarks on distributed databases, e.g. in [11] and [3]. But existing approaches only focus on common setups of clusters and data centers. They however do not consider the scenario of a globally distributed database.

When designing a distributed system, membership maintenance is important. At a large scale, sharing a complete list of all members is not feasible, especially in a dynamic scenario. In [1] a *gossip* based membership protocol is presented. In this approach a participant does not need to know all members, but a comparatively small subset of them. As in real world gossip, small groups of computers meet and "gossip" about other nodes they know and new or updated data. This way, with each iteration, a node gets a fresh set of nodes for the next meeting and data spreads "epidemically" throughout the network. As a result, a data storage system based on this protocol can never offer the strong consistency traditional database systems provide.

3. Preconditions

Following the given approach, the some preconditions have to be regarded.

3.1 Weak Consistency

The CAP-Theorem states that a distributed system can only satisfy two of the following three features [6]:

Consistency All nodes hold the same data at a time, thus making parallel communication to different nodes seem like serial communication with one.

Availability Every request to an available node results in a response (or a failure notice by that node).

Partition Tolerance Even if internal messages get lost or nodes fail, the system still responds correctly (or gives a failure notice).

Interviewees	Online
30%	0 h/d
21%	< 1 h/d
25%	1 - 2 h/d
18%	2 - 5 h/d
6%	5 - 10 h/d
1%	> 10 h/d

Table 1. Online time in hours per day

Providing the latter two, in the recent years the family of *NoSQL* databases has evolved, allowing for a high flexibility in the context of cloud computing. As shown in [10], NoSQL databases are designed for *horizontal scaling* and find application when it is sufficient for clients to get a possibly outdated (but nonetheless valid) response. In contrast to the traditional ACID properties (Atomicity, Consistency, Isolation, Durability), NoSQL calls its model BASE (Basically Available, Soft state, Eventually consistent). Additionally, NoSQL databases usually use a non-relational storage scheme, are optimized for heavy read/write access, and provide spreading and replication of data across multiple machines in order to share load and provide redundancy.

3.2 Human factors

Before considering whether running a storage is bearable for end-users, it has to be examined whether their computers are useful for this in the first place.

One part of this question regards availability. Professional database clusters provide an availability beyond 99%, whereas home computers run a few hours a day. In addition, they are not able to serve during their whole runtime, due to the join and leave overhead. If e.g. a node of the database goes online after many hours of absence, it first has to update its local, outdated data before being able to serve requests. As a first guess, only computers with online times beyond six hours a day are considered suitable. According to [2], German end-users are online as shown in table 1. Assuming similar numbers for other countries, a maximum of approximately 5% of all home computers fulfill the requirement.

The other part of the question regards volunteering. Only a few of the users, whose computers do satisfy the first prerequisite, will make them available to the public without any demands. A rough number on the quantity of volunteers can not be foreseen at the moment.

3.3 Technical factors

The model of a distributed database is not applicable to all kinds of end-user machines, because acting as a server does not allow for deeper sleep states. As a result, notebooks, tablets and smartphones do not fit the needs of the system—while constantly rising in market share.

Next, the low bandwidth and high latency of the average household's internet connections might be an issue. Condensing the sent data to a UID makes transmitting even thousands of reports possible. Hosting data however demands for a reasonably higher network throughput, especially as upload rates are usually lower than download rates on these connections.

3.4 Comparisons

To proof the feasibility of a crowdsourced database, it has to be shown that a small fraction of the world's computers can host a sufficiently reliable database for potentially all of the world's error reports.

According to [7], 60 servers processed "well over 100 million error reports per day" arising from all Windows PCs worldwide in 2009. If the same load is distributed to 10,000 participants, it might drop to a negligible level.

Also, while the system is designed to be usable by everyone—like non-professional developers or small companies lacking an excessive IT infrastructure—, access is nevertheless available to big companies likewise, who might want to contribute to the project. In addition, universities are often willing to share spare resources. As a result, end-user computers might be backed up by office and pool computers, or even servers. As these machines only accompany the end-users' ones, this possibility does not contradict with the precondition of an decentralized, independent infrastructure.

Finally, projects like SETI@home show that enough volunteers get together to justify the effort of implementing a volunteer-computing-platform.

Motivated by these facts, finding enough participants for a distributed storage system is not assumed to be problematic.

4. Realisation

WER provides a technically mature remote crash reporting system. A simple program was implemented to clone these reports to get a realistic set of testing data.

The NoSQL database Apache Cassandra—which uses a gossip based membership protocol [8])—fitted the needs for a distributed storage system best, as it:

- provides linear scalability,
- has a fine-grained replication mechanism, and
- is very fault-tolerant.

And since it is possible for a client, to connect to any node of a Cassandra grid, this database already provides the complete network infrastructure.

In Cassandra, each node in the network is given a unique ID at its first join. The node to store a specific record on is determined by this ID and the key of the record. To ensure balance, the ID of a node is chosen such that it takes over half of the records of the node that currently is responsible for the most records. Providing fault-tolerance, replicas of records can be placed on other nodes, also determinable by record key and node ID. Additionally, to distribute network and system load, read and write access to records can be distributed across all these replica nodes as well.

RowKey	SuperCol.	Column
<ErrorUID>	Details	{Key1: *string*}
		{Key2: *string*}
		...
	Occurrences	{ComputerID1: *int*}
		{ComputerID2: *int*}
		...
	...	
...		

Table 2. Logical table layout
Note: The data model is comparable to a JSON document.

4.1 Eventual consistency

When allowing not only read access, but also write access to one record on different nodes, due to eventual consistency, conflicts can occur if different values have been written for one record on different replica nodes. In Cassandra, to resolve such conflicts during gossip meetings, the most recently written value always wins.

A simple example: The database stores how often each error occurs. If a simple counter is used and it is incremented on different machines, occurrences get lost in the syncing process inevitably. To avoid this, it would be best if a record is written once, and only read henceforth. Instead of a counter, for each occurrence of an error an entry in a list might be added, containing the reporting computers' ID plus a timestamp. This approach would lead to a big amount of data to store.

Table 2 presents a part of the layout implemented in the prototype. Here, an occurrence-record can be written arbitrary times per computer. It can be assumed that one computer only connects to one specific node of the database during one of its online phase. It can also be assumed that between two online phases, there is enough time to spread the written information throughout the storage network. Thus, consistency can be assumed on self-written records for each computer.

4.2 Replicas

To determine how many replicas of a record should be created, a balance between availability and space requirement has to be found. It also has to be taken into account that, with an increasing number of replicas, the synchronization gains complexity. Table 3 gives an overview of the relation between the number of replicas per record and its availability. One replica corresponds

Replicas	Availability	Downtime
1	25%	18 h/d
2	44%	13.5 h/d
4	68%	7.7 h/d
8	90%	2.5 h/d
16	99%	14 min/d
25	99,9%	44 min/month
31	99,99%	4 min/month
39	99,999%	5 min/year

Table 3. Replica availability relation

to one computer, which is calculated with an uptime of six hours per day.

In the benchmarks a replication factor of eight was chosen as middle ground between the former mentioned factors. But while two and a half hours per day seem much of a downtime, the required availability of a record should be assessed first. The root cause of a bug is expected to produce crashes more than once, thus, a report dropped due to unavailability is likely to be submitted again. Ultimately, only the total amount of occurrences will be distorted. Relative counts compared to other errors stay intact, as reports get lost for these too. Thus, statistics keep being representative.

5. Benchmark

In order to proof the feasibility of outsourcing a distributed data storage system to the end-users, it remains to be shown that the load on their computers stays within bearable limits.

The arising system load is expected to correlate with how many storage nodes serve a set of report producing computers (report nodes). It also seems likely that developers (analyzer nodes) download comparatively large sets of data prior to operating on them.

5.1 Testing setup

The previously mentioned software that cloned WER reports was altered to automatically produce a large amount of plausible reports. As the provided computers ran Linux, virtual machines were used to run this software. They were Windows 7 Professional x64 SP1 vanilla installations in VirtualBox 4.1, given 1 CPU core, 1 GB of RAM, and a 20 GB disk image. Only the .NET runtime and the report software were installed.

The computers themselves ran Debian Wheezy (testing) x86_64 on Kernel 3.0 with Cassandra 0.8 on OpenJDK 6b24. dstat 0.7.2 was used to collect statistics. The statistics shown bellow origin from machines with a 3 GHz Core2Duo, and 4 GB of RAM.

A joining database node or a client can connect to any running node of a Cassandra cluster. But to simulate the case that the they do not connect with a uniform distribution, one of them was the dedicated connection point to all other nodes (seed node). Thus, the benchmark also shows load in case of poor balance.

The 58 computers where used as follows:

poolpc-remote Remote controller via SSH and simulation of the analyzer node's read access

poolpc10 Seed node

poolpc11-25 Dedicated storage nodes

poolpc26-50 Storage and report nodes

poolpc51-66 Dedicated report nodes

The database on poolpc10 was initialized with one million error reports to provide initial information to share with joining storage nodes. Afterwards, infcip-remote executed the following steps:

1. Start Cassandra on poolpc10-20 to show data distribution load on an idle system.
2. Start VirtualBox on poolpc26-66, ten each twenty minutes, to show increasing load with a decreasing ratio of storage nodes per report node.
3. Start Cassandra on poolpc21-50, ten each twenty minutes, to show decreasing load with an re-rising amount of storage nodes.
4. Stop all VirtualBoxes
5. Repeatedly start a script reading all records and shut down ten Cassandra instances until all are stopped, to simulate different ratios of storage nodes per analyzer node.

5.2 Results

CPU load on the storage nodes showed an average below 5%, and few peaks up to 10%, seldom 20%, on heavy read access (nodes joining, reader active). RAM usage stabilized at 1.3 GB. Hard disks show frequent write and rare read peaks. Most reads are served of the cache (RAM), while writes represent cache flushes. Likewise, network send and receive is characterized by peaks; although they reach the 100 Mbit/s of the local network's bandwidth, the load should be bearable with a certain delay by a connection featuring 3 Mbit/s up- and downstream. The overall system load stayed below 0.05 in the five minute average and showed few peaks to 0.2 in the one minute average.

The seed node on the other hand showed much more frequent CPU peaks up to 20%, 1.5 GB RAM usage, and would need a 20 Mbit/s connection to the internet. Five minute average system load peaked often above 0.05. The disk access though did not differ much, also likely due to a good caching mechanism.

During the nearly 14 hours of the test, per node 2 GB of data were written on average and 1 GB of disk space

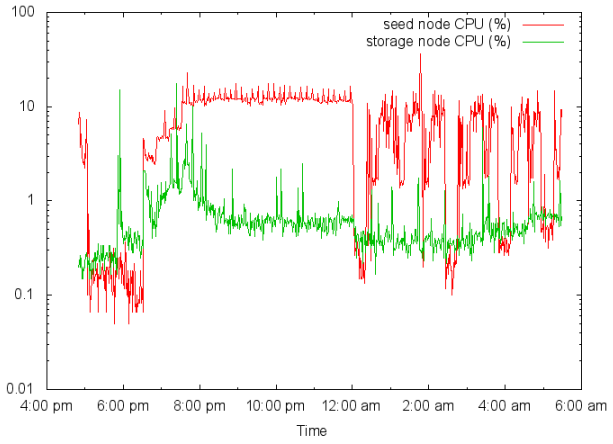

Figure 1. CPU load

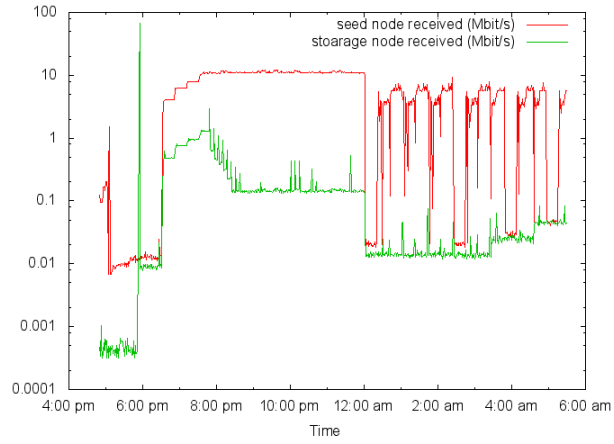

Figure 2. Network load

was allocated. Storage nodes received 2 GB of data via network, 1 GB was sent, whereas the seed node received 33 GB and sent 40 GB.

Figure 1 shows CPU load for the seed node and a storage node. Accordingly, figure 2 compares their network load—only the values for received network traffic are shown, as the values for sent network traffic showed nearly the same numbers.

In the beginning, the seed node shows a notable load due to sharing its records with upcoming nodes. This load decreases, as more nodes come up to share records with each other. Initiating itself, a CPU and network peak is notable at the beginning of the storage nodes' lifetime at 6:00 pm. Between 6:30 and 7:45 pm, both nodes show gradual increases in load, with each step representing the upcoming of a new set of ten report nodes. After 7:45 pm, the load on the storage node decreases as a result of being accompanied by further storage nodes. The load on the seed node remains, because it is still the only connection point for the report nodes. Starting at 12:00 am, the reader connects to the seed node, reading all existing data. Again in steps, the load on the storage node increases with the simulated failing of storage nodes at 3:30 and 4:30 am.

5.3 Analysis

According to [7], Microsoft processed 100 million error reports per day in 2009. If, corresponding to [9], by the end of 2011 2.2 billion internet users existed, an estimate of 1.5 billion Windows users in 2009 seem reasonable. 100.000 million reports per day by 1.5 billion users results in $0.0\overline{6}$ reports per day and computer.

In the test, almost 700,000 records were written in 14 hours which leads to nearly 3,500 records per hour. Thus, the testing machines represented about 1,300,000 computers. Therefore, to simulate 2.2 billion machines, 1,750 report nodes would have been sufficient—43 times more than the 40 computers used in the test.

Regarding the results, 43 seed nodes and 473 to 1,763 storage nodes could bear the world's errors, likely showing similar load. Combining reporting with analyzing nodes, these numbers should according to the tests at most triple. Furthermore, at least the load on the non-seed storage nodes seems bearable for end-users.

6. Further Work

Cassandra revealed two issues during the test.

First, as mentioned above, an ID is assigned to a first time joining node. If a further node joins the network while the load is still being balanced to the former, the latter gets the same ID assigned. If several nodes with the same ID exist in the grid, only one of them is used. A balance time of seven minutes before starting the next node worked out well in the test. In the intended scenario however, nodes join randomly. Using a mechanism of mutual exclusion would lock joining of nodes for an intolerable long period of time. Further insight in the joining and ID assigning mechanism will be needed to solve this issue.

Second, also the leaving of nodes is problematic. Cassandra stores which nodes with which IDs have been online. Thus, a specific node keeps being responsible for a specific set of records. As nodes will leave the network for good eventually, at some point records will be lost. Although Cassandra is able to sign nodes off, this has to be done explicitly. A solution might be a periodically executed script, kicking out nodes last seen longer than a certain period of time ago. In a distributed, server-less grid, one of the running nodes might be chosen for that task through a distributed election algorithm.

Other possible issues, like the connection over slow networks or scaling to 10,000 nodes, have not been researched yet, and might reveal further problems.

An authorization mechanism was omitted in this work. On the one hand, following the conception of crowd storage, everyone should get read and write access to the records; on the other hand, intended or unintended misuse might become a problem. A first solution might be a lightweight, storage-side front end to Cassandra, which offers only a subset of Cassandras possible operations and checks queries for plausibility.

Next, it was not considered, how running storage nodes are announced to the public. A name-service-like mechanism might be used. This creates the need for a server, but as it just serves this one purpose, it might not demand for a professional infrastructure. In a server-less scenario, nodes might store the addresses of all nodes seen in gossip meetings, hoping at least one of them might be available on their next start.

If the above concerns can be resolved, further work to put the concept into practice can be started.

Having servers available, as stated in 3.4, makes a hybrid setup possible. These servers could be intentional seed nodes. Their IPs could be provided to the public, superseding the solutions suggested above and they might also run the script that cleans up orphaned nodes.

Making Cassandra GeoIP-aware will increase the reliability of the network. Replicating a record all over the globe makes its availability more independent of day- or nighttime and also makes it possible to work around local connection problems.

As Cassandra integrates well in Solr, a search platform powered by the Lucene search engine, and Hadoop, a MapReduce framework for distributed computing, searching the reports or applying further heuristics on the data can be realized using these software tools.

Last, a social-network-like front end to the database has to be implemented. It should provide features like the ones outlined in the introduction.

7. Conclusion

While the results are promising, vanilla software was used without modifications. Customization to the specific demands of the intended use will work around (some of) the problems and improve performance.

Furthermore, today's computers are often overpowered for a casual "browser and word processor"-user. And among the 2.2 billion internet users worldwide, finding a few thousand volunteers with above-average broadband internet access should not be much of a problem. Thus, operating a purely crowdsourced database has to be called plausible.

References

[1] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 292–301, New York, NY, USA, 2005. ACM.

[2] BITKOM. Press information. Survey on behalf of BITKOM, Sept. 2010. URL `http://www.bitkom.org/60376.aspx?url=BITKOM_Presseinfo_Dauer_der_Internet-Nutzung_05_09_2010.pdf`.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143—154, New York, NY, USA, 2010. ACM.

[4] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[5] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[6] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[7] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, page 103–116, New York, NY, USA, 2009. ACM.

[8] A. Lakshman and P. Malik. Cassandra. *SIGOPS Operating Systems Review*, 44(2):35—40, Apr. 2010.

[9] Miniwatts Marketing Group. World Internet Usage Statistics News and World Population Stats, Aug. 2012. URL `http://www.internetworldstats.com/stats.htm`.

[10] J. Pokorny. Nosql databases: a step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[11] Y. Shi, X. Meng, J. Zhao, X. Hu, B. Liu, and H. Wang. Benchmarking cloud-based data management systems. In *Proceedings of the second international workshop on Cloud data management*, CloudDB '10, pages 47—54, New York, NY, USA, 2010. ACM.

[12] J. T. Tsay, L. Dabbish, and J. Herbsleb. Social media and success in open source projects. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 223–226, New York, NY, USA, 2012. ACM.