Metadata of the chapter that will be visualized in SpringerLink

Book Title	Algorithms and Architectures for Parallel Processing
Series Title	
Chapter Title	A Dependable, Scalable, Distributed, Virtual Data Structure
Copyright Year	2015
Copyright HolderName	Springer International Publishing Switzerland

Author	Family Name	**Grampone**
	Particle	
	Given Name	**Silvia**
	Prefix	
	Suffix	
	Division	
	Organization	Universidad Católica del Uruguay
	Address	Montevideo, Uruguay
	Email	silviagrampone@gmail.com
Author	Family Name	**Litwin**
	Particle	
	Given Name	**Witold**
	Prefix	
	Suffix	
	Division	
	Organization	Université Paris Dauphine
	Address	Paris, France
	Email	Witold.Litwin@dauphine.fr
Corresponding Author	Family Name	**Schwarz**
	Particle	
	Given Name	**Thomas SJ**
	Prefix	
	Suffix	
	Division	
	Organization	Universidad Centroamericana
	Address	San Salvador, El Salvador
	Email	tschwarz@jesuits.org

Abstract	Cloud computing allows on-demand access to cheap computing resources. This capability can be used for solving problems autonomously by complete enumeration. We present here SDVRP a data structure based on range partitioning that allows to autonomously divide the computing tasks to as many nodes as are needed to meet a user-imposed dead-line (in the order of minutes) despite heterogeneity of nodes. The data structure monitors itself to deal with failures and changes in node capacities. We use simulation for a proof-of-concept of this data structure.

Keywords (separated by '-')	Brute force calculations - Cloud - Data structure - Range partitioning - Scalable distributed data structures - Failure resilience

Author Proof

A Dependable, Scalable, Distributed, Virtual Data Structure

Silvia Grampone[1], Witold Litwin[2], and Thomas SJ Schwarz[3](✉)

[1] Universidad Católica del Uruguay, Montevideo, Uruguay
silviagrampone@gmail.com
[2] Université Paris Dauphine, Paris, France
Witold.Litwin@dauphine.fr
[3] Universidad Centroamericana, San Salvador, El Salvador
tschwarz@jesuits.org

Abstract. Cloud computing allows on-demand access to cheap computing resources. This capability can be used for solving problems autonomously by complete enumeration. We present here SDVRP a data structure based on range partitioning that allows to autonomously divide the computing tasks to as many nodes as are needed to meet a user-imposed dead-line (in the order of minutes) despite heterogeneity of nodes. The data structure monitors itself to deal with failures and changes in node capacities. We use simulation for a proof-of-concept of this data structure.

Keywords: Brute force calculations · Cloud · Data structure · Range partitioning · Scalable distributed data structures · Failure resilience

1 Introduction

Cloud computing has put inexpensive, massively distributed computing at the hands of the masses. If an organization can rent an almost unlimited amount of computing power in various clouds for short blocks of times (ten minutes), then we can use this enormous, temporary computing power to tackle classical optimization problems through complete enumeration, trading cheap on-demand computing power for the more sophisticated, classical algorithms [14]. Take as an example a database that needs to solve a knapsack problem with 50 variables. It took us 15.6 ms to solve a 15 variable knapsack problems on a single core by complete enumeration. The 50 variable problem would then take about 100 days on the single core, but if we can distribute the work over 15000 nodes, it would take less than 10 min. If Google Cloud would rent cores for 10 min only, it would cost $60.00.

We extend here this previous work by presenting a dependable version of a Scalable Distributed Virtual Data Structure (SDVDS), explained in Sect. 2, called Scalable Distributed Virtual Range Partitioning (SDVRP) (Sect. 3). In Sect. 4 we protect SDVRP against non-byzantine node failures that can upset

© Springer International Publishing Switzerland 2015
G. Wang et al. (Eds.): ICA3PP 2015 Workshops, LNCS 9532, pp. 1–12, 2015.
DOI: 10.1007/978-3-319-27161-3_66

Author Proof

the SDVRP's load distribution or overlook an optimal value if not dealt with. Section 5 evaluates failure resilience. Section 6 gives the related work and Sect. 7 concludes.

2 Scalable Distributed Virtual Data Structures

Scalable Distributed Virtual Data Structures (SDVDS) distribute the work involved of complete enumeration over a large number of nodes in a cloud. We envision them to be used in conjunction with databases and similar applications such as business intelligence decision tools. We now define a complete enumeration problem:

Definition 1. *A complete enumeration problem consists of a (record) range* $I = \{N_0, N_{0+1}, \ldots N_l\} \subset \mathbb{N}$ *and an objective function* $\phi : I \longrightarrow \mathbb{R}$*. A solution to a complete enumeration problem is a value* $\iota \in I$ *that maximizes* ϕ*, i.e.* $\phi(\iota) = \max(\{\phi(x)|x \in I\}$.

An SDVDS solves a complete enumeration problem in phases. The user (application) generates a file scheme S_F that describes (I, ϕ) and hands it to a coordinator node. The coordinator node is only used to begin the process and to communicate the result to the user application and thus does not constitute a bottleneck. In the *set-up* phase, the data structure allocates a sufficient number of nodes to guarantee finding a solution in a maximum time $T_{\max}$ specified by the user (application). Because of multi-tenancy, the speed at which nodes can evaluate ϕ is not constant. Each node therefore evaluates its capacity, the number of records it can evaluate and compares it with its load L, the number of records assigned to it for evaluation. If its capacity is smaller than its load, it reacts, often by recruiting another node to the data structure and then dividing its load with the other node. The set-up phase overlaps with the *scan* phase, where each node evaluates its assigned range of records $\subset I$. Each node also monitors its speed, since the node capacity might not remain constant. At the end of the scan phase, each node sends its value to a predecessor node, which in turn agglomerates the results send to it by selecting the $\iota \in I$ with maximum value $\phi(\iota)$. This *termination phase* finishes when the original node has processed the messages from all of its direct child nodes as well as its own set of records. We previously described the scan phase as having the nodes *materialize* the virtual records and scan them in order to provide the parallelism with Scalable Distributed Data Structures, but this relationship is not necessary in order to understand the basic functioning of the data structures [14]. Different data structures with different properties can be obtained by changing the way in which the record range is divided.

Any SDVDS definition consists of a definition of all the phases, but does not provide a description of the evaluation. An SDVDS is defined by (1) a node allocation process, usually through splitting; (2) an allocation of enumeration ranges to nodes; and (3) the set-up of a hierarchy for reporting the partial results and agglomerating them, and returning them to the user. The second

Author Proof

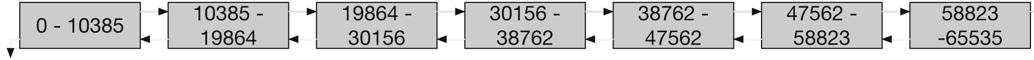

Fig. 1. A small SDVRP structure for the range of $0 \dots 2^{16}$.

and the first step can usually be combined in a single step. An algorithm for an SDVDS consists of (1) a definition of the enumeration range $\subset \mathbb{N}$; (2) instance extraction, which is a method for creating an instance of a possible solution from an index within the range; and (3) an evaluation function.

For example, if we solve a 0–1 integer optimization problem with 40 variables, the range is $R = [0, 2^{40} - 1]$, the instance creation assigns to variable $x_i(n)$ the value of bit i in a number $n \in R$, and the evaluation function ascertains first the truth of a conjunction of inequalities of form $\phi_j(x_1, \dots, x_{40}) < m_j$ and then evaluates the optimization function.

3 SDVRP

Scalable Distributed Virtual Range Partitioning (SDVRP) is a SDVDS based on RP*, the scalable distributed data structure that provides range partitioning. It assigns contiguous sub-ranges of the record range to the nodes. Figure 1 gives a very small example. Each node maintains a left and a right neighbor. A node can move load to one of its neighbors while maintaining the contiguity of the range of records assigned to it. The data structure starts with a single node to which we assign the complete original range. After evaluating its capacity, the node (in all likelihood) decides that its capacity is not sufficient and splits. By local decisions only, the data structure acquires the nodes necessary to perform the enumeration phase within the user-set limit. At all times, the nodes are arranged in a linear list such that consecutive nodes have contiguous ranges of records to evaluate.

3.1 Node Allocation and Subrange Assignment

A node always splits if its load is larger than its capacity. The new node is randomly inserted to the left or to the right of the splitting node. The load is then equally divided between between the splitting and the new node. It takes maybe a second to ascertain the capacity of a node with reasonable accuracy.

Our simulation results in Sect. 5 show, this simple splitting mechanism can be easily improved by trying to use free capacity at a neighbor. In the improved splitting algorithm, a node that needs to split contacts one of its neighbors at random. If this neighbor has free capacity, it takes over part of the range of the splitting node to have its load equal its capacity. The splitting node then tries the same load shifting with the other neighbor. It is possible but unlikely that the splitting neighbor has reduced its load to zero, in which case it deallocates itself.

Author Proof

As always in distributed systems, the algorithm designer needs to prevent race condition. In our algorithm, a node only interacts with its direct neighbors. A node acquires a lock on its neighbor which cannot interact with its other neighbor until the first interaction has finished.

3.2 Result Agglomeration Hierarchy

All nodes but the original node (which becomes the coordinator node) are allocated by another node which becomes its parents. This generate a tree structure that becomes the agglomeration hierarchy. At the end of its scan phase, each node sends its result to its parent, which combines the results of its children with its own and sends the combined result to its parent. The coordinator node sends the result to the user application.

3.3 Fast Allocation

The allocation process of the node is not instantaneous. Each node uses a second or so to assess its own capacity and bases a decision on whether to split on the relation between assigned load and capacity. However, at the beginning of the allocation phase, it is clear that nodes will split. As a variant, we therefore propose *pre-splitting*. The coordinator node determines its load to capacity ratio and decides on a minimum number of nodes necessary. The minimum number is calculated using a safe, upper bound based on its capacity. Thus, if we use assume that node capacity is less than twice the capacity C_0 of the coordinator node, then the number N of nodes needed depends on the total load L by $N > L/(2C_0)$. Since setting up that many nodes in one step also takes time, the coordinator requests M nodes and assigns to the them the enumeration range partitioned into M pieces. Each of the M nodes then allocates N/M new nodes itself that become its children.

4 Providing Failure Tolerance

Cloud nodes are not meant to be very reliable. Instead, the application that uses cloud resources has the task to provide the failure tolerance it needs. Since cloud resources are commodities, the alternative would be a one-size-fits-all strategy that would not deliver for some applications or over-provision for other applications. According to Birman, it is even considered acceptable for a data center administrator to randomly shut down nodes in order to escape from an unstable situation [4].

Failure tolerance for distributed complete enumeration task is different than for many other applications. While an adversary might destroy a complete calculation by giving a false optimum and causing the other nodes to throw away their work, a byzantine malfunction for such a simple programming pattern is unlikely. If we can assume (as we argue that we can) that nodes do not create false results by either overlooking a local optimum or by reporting one where

Author Proof

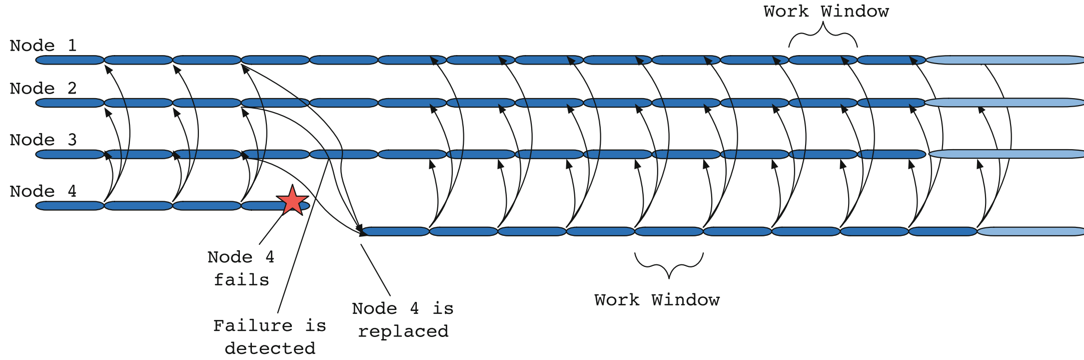

Fig. 2. A time line of the handling of a node failure. All nodes divide their work into work windows and send a resumé of their results to all other nodes in the group. The figure only shows the updates sent by Node 4. When Node 4 later fails, the failure of receiving the latest resumé or the lack of acknowledgment to resumés sent to Node 4 leads to a replacement of the node. The replacement node is brought up to the point of the last resumé of Node 4 so that the work of the first three work windows is not lost.

none exist, then the only thing that can go wrong is in the division of labor by the nodes. Complete enumeration is an "at least once" task. If the data structure assigns part of the range to two nodes (for example because it wrongly assumed that one has failed), then the only damage consists performing the same work twice and having to pay the rental costs of the superfluous node. The validity of the overall result is not affected.

Second, the result of a a partial enumeration can be resumed in very little space. If a node has scanned a subrange, the results of this scan can be subsumed in a description of the range scanned (usually the upper and the lower bound), and the pair $(\iota, \phi(\iota))$ consisting of the argument and best value seen so far. We call this the *resumé* of the partial evaluation at a node. Each resumé contains additionally information about the node such as the addresses of its neighbors and in case of the coordinator node, the address of the application. It functions essentially as execution checkpoints in languages such as Erlang or Scala.

In order to provide failure tolerance, we use the old idea of process groups [3] that has several processes in a distribute system monitor each other and provide solutions if one process has failed. In our setting, we divide the nodes into groups, which we call *buddy groups*. Buddy groups run a distributed membership and consensus protocol such as a version of Paxos or Raft [5,16,21]. We use the sending of resumés as a heart-beat.

In more detail, each node divides its work into work windows of about equal time (such as one minute). At the end of a work window, the node sends a resumé of its scan results up to now to all other nodes in the buddy group. Resumés should be small; in general, they will consist of the range scanned and the argument of the best result(s) seen in this range. If a resumé from another buddy group member does not arrive in time, then the membership protocol is triggered to ascertain whether the node has in fact failed. The buddy groups replaces a failed node with a new node requested from the cloud. The replacement

Author Proof

node resumes the work of the failed work from the moment it sent its last resumé and does not start over, Fig. 2. Note that the resumé contains all the information needed by the replacement node.

Buddy group size is limited by the difficulties of running a membership protocol, since these protocols do not scale well, but they should not be so small that failure of all nodes in a buddy group in relatively short time is an event that is worth while to worry about. The splitting algorithm actually makes it unlikely that neighboring nodes are allocated immediately after the other, so that it is unlikely that neighboring nodes are physically located in the same server. We assume that buddy group sizes between four and seven provides the needed reliability without creating too much overhead.

Since our data structure is one-dimensional, forming groups is quite simple. A naïve solution would have the leftmost node starts a counting process that assigns the first k nodes to the first group, the second k neighbors to the second group, etc. and deals with a last group that is smaller than k by merging it with the previous group. This nave algorithm is perfectly suited to small instances, but would take too long for a large data structure.

A more efficient algorithm generates buddy groups by local coagulation. After a node has finished the allocation phase, it becomes leader of a budding buddy-group. Each buddy group aims at having a membership between four and seven nodes. If it does not have the required number of nodes, the leader associates the group to one of its neighboring groups. If the resulting group has more than seven elements, it splits into two contiguous parts, each having at least four elements. This simple procedure works rapidly, guaranteeing that each group reaches the required amount of nodes in at most three join attempts. We avoid race conditions by locking leaders that are negotiating a merger.

5 Evaluation

5.1 Variability and Multi-Tenancy

We use simulation to evaluate the construction principles of SDVRP. First, we evaluated the effects of node capacity variation on node utilization. Node utilization (the ratio of total load divided over total capacity) measures the inefficiency of our work assignment algorithm. Given the nature of cloud computing, where resources are cheap, but are not free (as in P2P), we want utilizations over 50 %, but do not worry too much if they are not close to 100 %. We modeled variation in the capacity of a node by using the beta distribution with parameters $\alpha = 2$ and $\beta = 2$ to generate capacities distributed between $[0.5, 1.5]$, $[0.75, 1.25]$ and $[0.9, 1.1]$.

As Fig. 3 shows, the utilization oscillates between 0.5 and 1 with peaks where the initial load is close to a power of two. If there is no variation and all nodes have capacity exactly one, then the resulting utilization graph is a sawtooth graph. The utilization is one if the initial load is an integer power of two, and 1/2 if the load is increased by an infinitesimal amount, since in this case all nodes have to split.

Author Proof

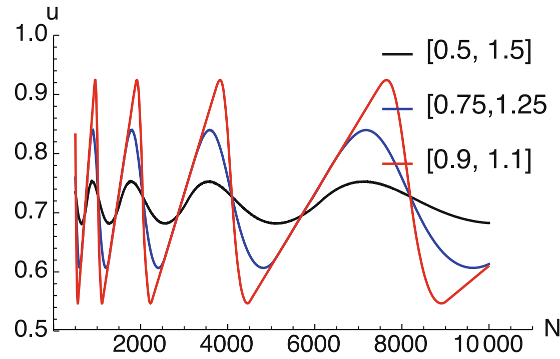

Fig. 3. Utilization u using the basic node splitting scheme depending on the load N

As the capacities vary more, the sawtooth curve flattens out, but oscillation in the utility still occurs. We observed the same behavior when we experimented with different beta distributions, even if the distributions were no longer symmetric, i.e. when the parameters α and β of the curve were not equal.

If we used the more sophisticated splitting algorithm, we still observe oscillations with periods given by integer powers of two, but the behavior now is more involved, especially if we used smaller variations. In general, the more sophisticated splitting algorithm results in appreciable higher utilization.

We also compared the utilization with the one obtained after 90 % of the nodes change capacity and use the advanced algorithm to rebalance the load. The rebalancing is not entirely successful, but the difference is minute for the smaller variations in node capacity. Figure 4 gives our results. We used a light-gray fill to indicate the difference of the before and after values of the utilization. We can see that the difference becomes small as we move to systems with less variability, but the wave of changes that we introduced definitely lower the utilization, though not by much.

5.2 Reliability

An accurate, general failure model for nodes in a cloud data center does not exist. Two incidents show that cloud failures can be drastic and lead to complete outages. In 2011 an automatic misdiagnosis to a bad configuration resulted in a massive recovery effort for servers that had not failed [2]. Gmail suffered a massive service interruption in 2009 when routine maintenance resulted in a larger than expected change in the traffic load at some routers, who became overloaded and unresponsive [7].

Cloud outage based on hardware failure follow a Weibull distribution, but at the scale of minutes, failure rates are constant. In the absence of better information for modeling, we test our reliability scheme against two different scenarios. The first is a scenario where a large proportion of nodes suddenly becomes unavailable. The second scenario assumes independent failures at a given failure rate and models failures as a Poisson process. The truth is some combination of both scenarios and we need to test our scheme against both extremes. Finally,

Author Proof

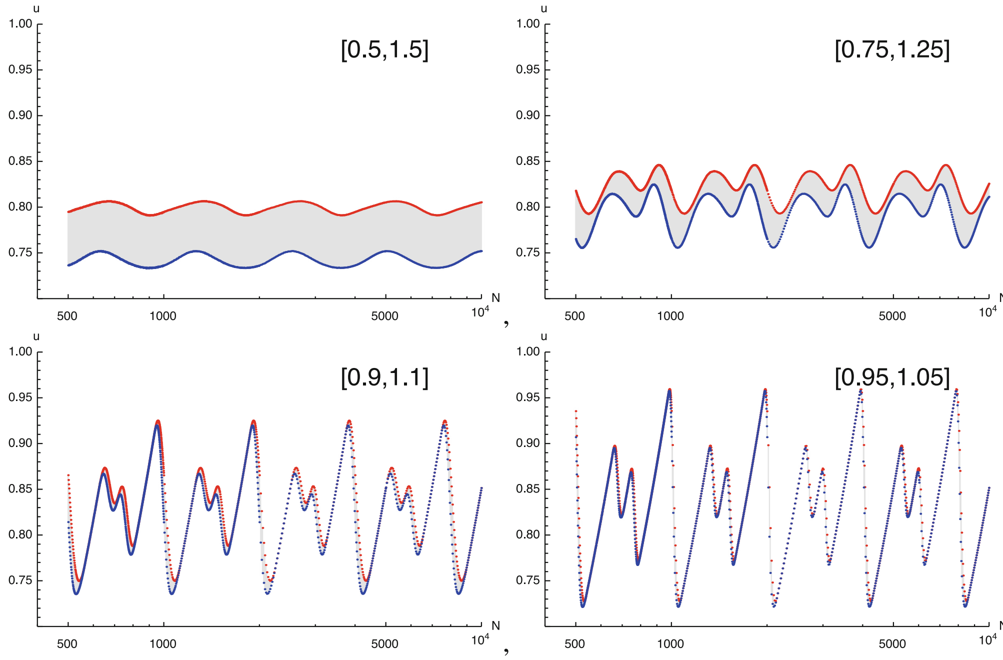

Fig. 4. Utilization u using the advanced node splitting scheme depending on initial load N. The upper graph shows the original utilization, the lower graph the utilization after 90 % of the nodes changed capacity. The node capacities vary between 0.5 and 1.5, 0.75 and 1.25, 0.9 and 1.1, and 0.95 and 1.05 respectively. The x-axis is logarithmic to bring out the oscillatory behavior.

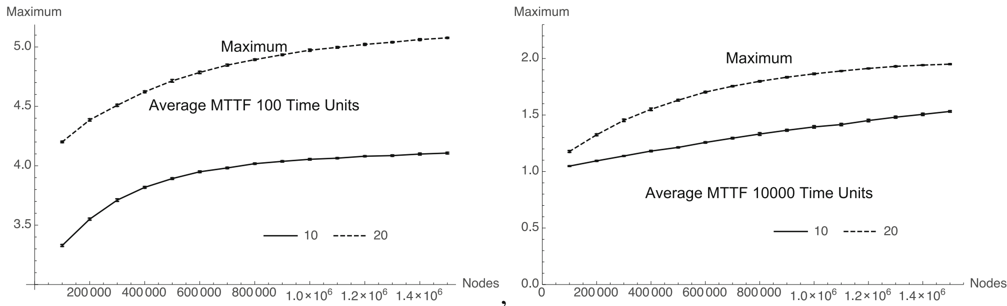

Fig. 5. Maximum number of restarts of a node's evaluation depending on the total number of nodes employed (x-axis). The failure rate during a time unit is a high 0.01 (left) and a more realistic 0.0001. If a time unit is a minute, this corresponds to a mean time to failure of less than two hours and 167 hrs. The calculation is slated to take 10 and 20, respectively, time units. We give error bars at the 99 % confidence level based on dividing the simulation runs into 20 batches.

a massive service disruption at the scale that were suffered by EC2 and Gmail is not controllable.

We first look at the effects of independent node failures. Typical maximum computation time is in the order of minutes or at most an hour, and at this scale, time between node failures can be assumed to be exponentially distributed.

Author Proof

We first determined the maximum number of restart that a single node would suffer in a system with N nodes, Fig. 5. Our experiment assumed a high failure rate of 1 % per time unit. The total calculation is broken into ten and twenty work windows respectively. Each failure has to be discovered (usually at the end of the work window, unless we change the protocol to monitor nodes more aggressively) and the new node has to be allocated (and its load possibly partially distributed if its capacity is lower than its failed predecessor). Still, even at unrealistic node numbers of a million or more and a very high failure rate per time unit (corresponding in order of magnitude to a minute), the most the calculation is slowed down is by less than a fourth of the total time. Since a node suffers a double failure during its ten or twenty work windows only very rarely (or rather, the replacement node for a failed node suffers itself another failure during the rest of the work), we can allocate two instead of one replacement node to bring the maximum time to be spend for resuming work on a failed node to two work windows plus some allocation time.

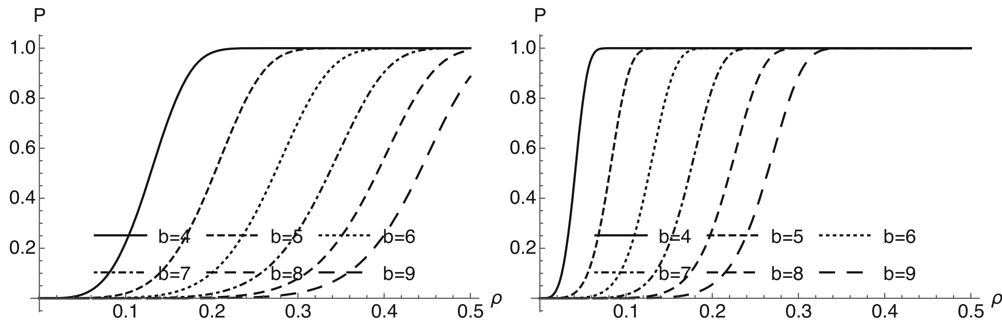

Fig. 6. Probability (P) that at least one buddy group has completely failed if a portion ρ of the nodes in an ensemble with 10,000 (left) and 1,000,000 nodes (right) has failed.

We now consider the effects of a wave of node failures. We model this phenomenon by assuming the existence of a condition that will cause a portion ρ of all nodes to fail during a work window. If there is a total of N nodes and the buddy group size is b, then the probability that all b members of a single group fail is ρ^b. The total number of groups is N/b, so that the probability that at least one group has failed is

$$1 - (1 - \rho^b)^{N/b}$$

This results in buddy group loss probabilities that quickly shift from almost zero to almost one, Fig. 6. The total number of nodes is – as has to be expected – very influential. For very large number of nodes, the structure does have a significant probability of loosing a complete buddy group with reasonable outage rates ρ. This will be noticed at the end of the calculation which will then be delayed by having to redo the work of the lost buddy group(s). The structure needs to be made more failure tolerant then. There are two possibilities. First, we can restart the work originally assigned to the lost group, but divide it over many more nodes in order to loose not that much time. The other possibility is

Author Proof

to form buddy groups among buddy group leaders. Since buddy group leaders are replaced if they fail, a member of a higher level buddy group only vanishes if all its members in the lower level buddy group have disappeared during a working window, which happens at a rate of ρ^b.

6 Related Work

Complete enumeration has been a tool of desperation since the beginning of solving optimization problems on computers. Scalable Distributed Data Structures (SDDS) were developed in the nineties to marshal the resources of distributed systems (multicomputers) for databases supporting different modes of data access such as linear or extensible hashing [13,20] search trees [15], range queries [18], or R-trees [11]. Much research in SDDS was devoted to provide failure tolerance [17]. Scalable Distributed Virtual Data Structures apply these scans not to records generated by a user but to virtual records, generated directly from the record identifier [14].

P2P systems try to harvest the idle resources of computers connected to the Internet. Early work defined distributed hash tables (Chord, Pastry, Tapestry) to overcome the same scalability problems as SDDS in a more anarchic environment and resulted in mature, efficient structures such as Skip graphs [1] and the Willow DHT [24]. Key is a sophisticated metadata overlay such as SOMO [25]. The difference between our work and failure resilience for P2P systems is that P2P systems already use extensive replication to allow access to data and only need to react if failure rates are too high, whereas we need the results of the work assigned to all nodes.

Cloud computation targets a setting closer to the multicomputer environment envisioned for SDDS and many SDDS structures have come into their own, though not under their original name. Google's BigTable [6] is an SDDS based on range-partitioning, but with more functionality than RP$*$ [19]. The same is true for MS Azure and MongoDB. Amazon's EC2 uses a distributed hash table called Dynamo [10]. VMWare's Gemfire provides its own hash scheme, etc. The novelty of our work lies in applying the scan functionality of SDDS to a completely different type of problem.

The idea of grouping processes in a distributed system into groups that monitor each other is fundamental and has become an accepted tool for reliability in distributed systems. While the use of replica and the topic of replica placement is important for P2P system, there seems to be very little literature on how to form small groups of peers. Slicing in P2P systems comes the nearest, but in general creates much larger groups [12].

Since our proposal is not using the cloud in a traditional way, previous work on cloud failure tolerance does not apply directly. For instance, Dai and colleagues note that grid users care about services that they are using instead of the resources and this is even more the case for cloud services. They therefore develop a holistic model for calculating the probability that a cloud service can successfully complete [8,9]. Similar models are given by Silva et al. and by Thanakornworakij [22,23].

Author Proof

7 Conclusion

We presented here a proof-of-concept for a Scalable Distributed Virtual Data Structure based on Range Partitioning that is failure tolerant. It assumes that cloud providers will eventually rent nodes for short times, controlling the demand by setting rental rates depending on the current demand. SDVDS are built on the principle of maximum autonomy. We applied this philosophy to the implementation of failure tolerance.

SDVDS extend the maximum size of optimization problems without the need to buy and administer special hardware. They will provide a very simple programming interface, as one only has to provide the evaluation function and define the enumeration space. The next step is the implementation.

References

1. Aspnes, J., Shah, G.: Skip graphs. ACM Trans. Algorithms (TALG) **3**(4), 37 (2007)
2. Babcock, C.: When Amazon's cloud turned on itself. Inf. Week 31 (2011) AQ1
3. Birman, K.P.: The process group approach to reliable distributed computing. Commun. ACM **36**(12), 37–53 (1993)
4. Birman, K.P.: Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services. Springer, Heidelberg (2012)
5. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an Engineering perspective. In: Proceedings of ACM symposium on Principles of distributed computing, pp. 398–407 (2007)
6. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
7. Claburn, T.: Gmail outage a big deal, says Google. Inf. Week (2009)
8. Dai, Y.S., Xie, M., Poh, K.L.: Reliability analysis of grid computing systems. In: Pacific Rim International Symposium on Dependable Computing, pp. 97–104 (2002)
9. Dai, Y.S., Yang, B., Dongarra, J., Zhang, G.: Cloud service reliability: modeling and analysis. In: Pacific Rim International Symposium on Dependable Computing (2009)
10. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Syst. Rev. **41**(6), 205–220 (2007). ACM
11. Du Mouza, C., Litwin, W., Rigaux, P.: SD-Rtree: a scalable distributed Rtree. In: IEEE International Conference on Data Engineering, pp. 296–305 (2007)
12. Fernández, A., Gramoli, V., Jiménez, E., Kermarrec, A.M., Rayna, M.: Distributed slicing in dynamic systems. In: International Conference Distributed Computing Systems, p. 66 (2007)
13. Hilford, V., Bastani, F.B., Cukic, B.: EH*-extendible hashing in a distributed environment. In: IEEE Computer Software and Applications Conference, pp. 217–222 (1997)
14. Jajodia, S., Litwin, W., Schwarz, T.: Scalable distributed virtual data structures. In: ASE International Conference on Big Data Science and Computing (2014)
15. Kröll, B., Widmayer, P.: Distributing a search tree among a growing number of processors. ACM SIGMOD Record **23**(2), 265–276 (1994)
16. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)

Author Proof

17. Litwin, W., Moussa, R., Schwarz, T.: LH*RS – a highly-available scalable distributed data structure. ACM Trans. Database Syst. (TODS) **30**(3), 769–811 (2005)
18. Litwin, W., Neimat, M.A.: k-RP*s: a scalable distributed data structure for high-performance multi-attribute access. In: International Conference on Parallel and Distributed Information Systems (1996)
19. Litwin, W., Neimat, M.A., Schneider, D.: RP*: a family of order preserving scalable distributed data structures. Very Large Databases **94**, 12–15 (1994)
20. Litwin, W., Yakouben, H., Schwarz, T.: LH* RS P2P: a scalable distributed data structure for P2P environment. In: International Conference on New Technologies in Distributed Systems. ACM (2008)
21. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings Usenix Annual Technical Conference (ATC) (2014)
22. Silva, B., Maciel, P., Tavares, E., Zimmermann, A.: Dependability models for designing disaster tolerant cloud computing systems. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–6 (2013)
23. Thanakornworakij, T., Nassar, R.F., Leangsuksun, C., Păun, M.: A reliability model for cloud computing for high performance computing applications. In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 474–483. Springer, Heidelberg (2013)
24. van Renesse, R., Bozdog, A.: Willow: DHT, aggregation, and publish/subscribe in one protocol. In: Voelker, G.M., Shenker, S. (eds.) IPTPS 2004. LNCS, vol. 3279, pp. 173–183. Springer, Heidelberg (2005)
25. Zhang, Z., Shi, S.-M., Zhu, J.: SOMO: Self-Organized Metadata Overlay for resource management in P2P DHT. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735. Springer, Heidelberg (2003)

Author Proof

Author Queries

Chapter 66

Query Refs.	Details Required	Author's response
AQ1	Kindly provide full information for Reference [2, 7].	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	• • • under matter to remain	Ⓥ
Insert in text the matter indicated in the margin	⋏	New matter followed by ⋏ or ⋏ⓧ2
Delete	/ through single character, rule or underline or ⊢—⊣ through all characters to be deleted	ꝺ/ or ꝺ/ⓧ2
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢—⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌞⁄
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	~ under matter to be changed	~
Change to bold italic	≂ under matter to be changed	≂⁄
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⟊
Change bold to non-bold type	(As above)	⟊
Insert 'superior' character	/ through character or ⋏ where required	Y or ⋏ under character e.g. Y² or ⋏²
Insert 'inferior' character	(As above)	⋏ over character e.g. ⋏₂
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	Y' or ⋏' and/or Y' or ⋏'
Insert double quotation marks	(As above)	Y" or ⋏" and/or Y" or ⋏"
Insert hyphen	(As above)	⊢⊣
Start new paragraph	⌐	⌐
No new paragraph	⤺	⤺
Transpose	⊔⊓	⊔⊓
Close up	linking ⁀‿ characters	⁀‿
Insert or substitute space between characters or words	/ through character or ⋏ where required	Y
Reduce space between characters or words	| between characters or words affected	⊤