Universidade do Minho

Ricardo Jorge Tomé Gonçalves

Logical Clocks for Could Databases

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Doutor Carlos Miguel Ferraz Baquero Moreno

Agosto 2011

Declaração

Nome: Ricardo Jorge Tomé Gonçalves

Endereço Electrónico: tome@di.uminho.pt

Telefone: 910467537

Bilhete de Identidade: 13303430

Título da Tese: Logical Clocks in Cloud Databases

Orientador: Doutor Carlos Miguel Ferraz Baquero Moreno

Ano de conclusão: 2011

Designação do Mestrado: Mestrado em Engenharia Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, 30 de Agosto de 2011

Ricardo Jorge Tomé Gonçalves

Resumo

Ambientes de computação na nuvem, em especial sistemas de base de dados na nuvem, estão rapidamente a aumentar em importância, aceitação e utilização entre as grandes aplicações (*web*), que precisam de alta disponibilidade e tolerância a partições por razões de escalabilidade, para isso sacrificando o lado da coerência (teorema de CAP). Com esta abordagem, o uso de paradigmas como a Coerência Inevitável tornou-se generalizado. Nestes sistemas, um grande número de utilizadores têm acesso aos dados presentes em sistemas de dados de alta disponibilidade. Para fornecer bom desempenho para utilizadores geograficamente dispersos e permitir a realização de operações mesmo em presença de partições ou falhas de nós, estes sistemas usam técnicas de replicação optimista que garantem apenas uma coerência inevitável. Nestes cenários, é importante que a identificação de escritas concorrentes de dados, seja o mais exata e eficiente possível.

Nesta dissertação, revemos os problemas com as abordagens atuais para o registo da causalidade na replicação optimista: estes ou perdem informação sobre a causalidade ou não escalam, já que obrigam as réplicas a manter informação que cresce linearmente com o número de clientes ou escritas. Propomos então, os *Dotted Version Vectors (DVV)*, um novo mecanismo para lidar com o versionamento de dados em ambientes com coerência inevitável, que permite tanto um registo exato e correto da causalidade, bem como escalabilidade em relação ao número de clientes e número de servidores, limitando o seu tamanho ao factor de replicação. Concluímos com os desafios surgidos na implementação dos DVV no Riak (uma base de dados distribuída de chave/valor), a sua avaliação de comportamento e de desempenho, acabando com uma análise das vantagens e desvantagens da mesma.

Abstract

Cloud computing environments, particularly cloud databases, are rapidly increasing in importance, acceptance and usage in major (web) applications, that need the partition-tolerance and availability for scalability purposes, thus sacrificing the consistency side (CAP theorem). With this approach, use of paradigms such as Eventual Consistency became more widespread. In these environments, a large number of users access data stored in highly available storage systems. To provide good performance to geographically disperse users and allow operation even in the presence of failures or network partitions, these systems often rely on optimistic replication solutions that guarantee only eventual consistency. In this scenario, it is important to be able to accurately and efficiently identify updates executed concurrently.

In this dissertation we review, and expose problems with current approaches to causality tracking in optimistic replication: these either lose information about causality or do not scale, as they require replicas to maintain information that grows linearly with the number of clients or updates. Then, we propose Dotted Version Vectors (DVV), a novel mechanism for dealing with data versioning in eventual consistent systems, that allows both accurate causality tracking and scalability both in the number of clients and servers, while limiting vector size to replication degree. We conclude with the challenges faced when implementing DVV in Riak (a distributed key-value store), the evaluation of its behavior and performance, and discuss the advantages and disadvantages of it.

Acknowledgements

I would like to express my special gratitude to Professor Carlos Baquero, for being my advisor and supporting my work with great advise and encouragement, through the last several months. Also a special thanks to Professors Paulo Sérgio Almeida and Vítor Fonte, both heavily involved in this project. Without forgetting Nuno Preguiça from *Universidade Nova de Lisboa*, the consulter of this project. To all of them, thank you for the continuous support and encouragement throughout this work and for the counseling provided. I learned a lot. Without their guidance and dedication this dissertation would not have been possible.

Thanks to my colleagues and friends at the laboratory, for the pleasant environment created. Thanks to Pedro Gomes for the exchange of ideas and brainstorming that we often had. It was of great value.

Additionally, I am grateful to my family and Ana, for all the love and support given, and for always having confidence in me.

Finally, a special thanks to *Fundação para a Ciência e a Tecnologia (FCT)* for supporting this work through project CASTOR (Causality Tracking for Optimistic Replication in Dynamic Distributed Systems), under Research Grant (BI) number: BI1-2010_PTDC/EIA-EIA/104022/2008_UMINHO

Contents

Contents xi
List of Figures xiv
List of Tables xv
List of Acronyms xvii

1 Introduction **1**
1.1 Context 2
1.2 Problem Statement 3
1.3 Contributions 4
1.4 Dissertation Outline 4

2 Literature review **7**
2.1 CAP Theorem 7
2.1.1 The CAP systems 8
2.2 Order, Cause and Effect 10
2.2.1 Order Theoretical Concepts 11
2.2.2 Causality 11
2.2.3 Happens-before 12
2.3 Logical Clocks 13
2.3.1 Scalar Clocks 14
2.3.2 Vector Clocks 16
2.3.3 Version Vectors 17

2.3.4 Interval Tree Clocks 19
2.4 Summary 22

3 Causality Tracking in Cloud Databases 25
3.1 System model 25
3.2 Cloud Databases 26
3.2.1 Dynamo 26
3.2.2 Cassandra 32
3.2.3 Riak 38
3.3 Common approaches to Causality tracking 42
3.3.1 Causally compliant total order 44
3.3.2 Version vectors with per-server entry 46
3.3.3 Version vectors with per-client entry 49
3.4 Related Work and Summary 50

4 Dotted Version Vectors 53
4.1 A Kernel for Eventual Consistency 53
4.1.1 Using the kernel operations 54
4.2 Dotted Version Vectors 58
4.2.1 Definition 58
4.2.2 Partial order 59
4.2.3 Update function 61
4.2.4 Correctness 62
4.3 Summary 64

5 Implementing and Evaluating DVV 67
5.1 Implementation 67
5.1.1 Operation Put in Riak 67
5.1.2 Changing Operation Put to use DVV in Riak 69
5.2 Evaluation 73

5.2.1 Basho Bench 73
5.2.2 Setup 74
5.2.3 Generic Approach 75
5.2.4 TPC-W Approach 79
5.3 Summary 81

6 Conclusion 85

6.1 Future Work 86

References 88

Appendix 94

A Implementation of Dotted Version Vectors in Erlang 95

List of Figures

2.1 Design choices regarding conflict handling [37]. 9
2.2 Two communicating processes 13
2.3 Scalar Clocks 15
2.4 Vector Clocks 16
2.5 Version Vectors 18
2.6 ITC id [5] 19
2.7 ITC event [5] 20
2.8 ITC stamp [5] 21
2.9 ITC run case example [5] 21

3.1 Summary of techniques used in Dynamo and their advantages [13]. 27
3.2 Dynamo ring 28
3.3 Default values to prune vector clocks in Riak. 41
3.4 Three clients concurrently modifying the same key on two replica nodes. Causal histories. 43
3.5 Three clients concurrently modifying the same key on two replica nodes. Perfectly synchronized real time clocks. . . . 45
3.6 Three clients concurrently modifying the same key on two replica nodes. Per-server entries. 47
3.7 Three clients concurrently modifying the same key on two replica nodes. Per-client entries. 50

4.1 A get operation using DVV. 55
4.2 A put operation using DVV. 57
4.3 Three clients concurrently modifying the same key on two replica nodes. Dotted version vectors. 59
4.4 Server clock in replica B 65
4.5 Client Clock for a concurrent update in replica B, using (a) Version Vectors and (b) Dotted Version Vectors 65

5.1 A put operation in Riak, using VV per-client entry. 68
5.2 Finite State Machine Diagram for PUT operations in Riak, using VV. 70
5.3 Finite State Machine Diagram for PUT operations in Riak, using DVV. 71
5.4 CPU, disk I/O and network bandwidth measurements in on machine, S2k, S316 76
5.5 Performance summary for DVV, S361, S2K 77

List of Tables

5.1 Scenario 1 (S1K) with generic approach. 76
5.2 Scenario 2 (S2K) with generic approach. 77
5.3 Scenario 3 (S5K) with generic approach. 78
5.4 Scenario 1 (S1k) with TPC-W approach. 80
5.5 Scenario 2 (S2k) with TPC-W approach. 80
5.6 Scenario 3 (S5k) with TPC-W approach. 80

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

API Application Programming Interface.

BASE Basically Available, Soft state, Eventually consistent.

CAP Consistency, Availability, Partition-tolerance.

DVV Dotted Version Vectors.

HTTP Hypertext Transfer Protocol.

ITC Interval Tree Clocks.

NoSQL Not only SQL[Structured Query Language].

NTP Network Time Protocol.

TPC Transaction Processing Council.

VC Vector Clocks.

VV Version Vectors.

Chapter 1

Introduction

Time is a very important aspect in distributed systems. Due to hardware limitations, software limitations or both, it is often impossible in (asynchronous) distributed systems to have a global clock like perfectly synchronized physical computers clocks. Even more advanced solutions like GPS time or NTP synchronization or are not feasible due to possible unpredictable delays by hardware or software limitations.

This absence of global clock is a problem for ordering events and for collecting information on the state of the system. But even if this global clock is possible, what it gives us is the total time of events. It does not deliver information on relations between events, like which events led to another event, and the events that had nothing to do with it, thus not capturing inherently concurrency between them [15].

Because it is often too difficult or even impossible to have a global clock, it is useful to know relative order of events. In fact, this relative order may be an actual requirement for some systems. Causality is the relationship between two events, where one *could* be the consequence of the other (cause-effect) [28, 14]. Due to restraints in global clocks and shared memory in distributed systems, mechanisms for capturing causality to partial order events are needed.

Relative order of events can be achieved simply by observing the causality relation between events, even in asynchronous systems that have no real time clock. This causality relation can be captured by the *happens-before*

relation, which was the basis for the subsequent logical time mechanisms introduced [30, 14, 28, 12].

Causality tracking mechanisms - or simply logical clocks - provide this *happens-before* relation for events, which is quite important in resolving a vast range of problems such as distributed algorithm design [9, 27], tracking of dependent events [12, 26], knowledge about the progress of the system [43, 2] and as a concurrency measure [11]. Schwarz and Mattern [38] has an extensive discussion of causality and its applications.

1.1 Context

The design of Amazon's Dynamo system [13] was an important influence to a new generation of databases, such as Cassandra [22], Riak[1] and Voldemort[2] focusing on partition tolerance, write availability and eventual consistency. The underlying rationale to these systems comes from the observation that when faced with the three conflicting goals of *consistency*, *availability* and *partition-tolerance*, only two of those can be achievable in the same system [8, 16]. Facing wide area operation environments where partitions cannot be ruled out, these systems relax consistency requirements to provide high availability.

Instead of providing the ACID properties, they focus on implementing what it is called a BASE system [33]. A BASE system has weaker consistency model, focuses on availability, uses optimistic replication and because of all of this, it is faster and easier to manage large amounts of data, while scaling horizontally.

This means that eventually, all system nodes will be consistent, but that might not be the true at any given time. In such systems, optimistic replication is used to allow users to both successfully retrieve and write data from replicas, even if not all replicas are available. By relaxing the consistency level, inconsistencies are bounded to occur, which have to be detected with minimum overhead. This is where Logical Clocks are useful.

[1] http://www.basho.com/Riak.html
[2] http://project-voldemort.com/

1.2 Problem Statement

The mentioned systems follow a design where the data store is always writable. A consequence is that replicas of the same data item are allowed to diverge, and this divergence should later be repaired. Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [23, 30, 38, 36]. In particular, for data storage systems, version vectors [30] enables the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. A replica version that is determined to be obsolete can be replaced by a more recent replica version. Merging concurrently modified replicas usually requires *semantic reconciliation*. By semantic, we mean that it is not an automatic reconciliation, it uses some sort of higher user-based logic to resolve it. This is achieved by sending to users (or to higher level application logic) the set of concurrent replica versions, the metadata context, and have them write a new version that supersedes the provided versions.

When accurate causality tracking and handling of concurrent replica versions is considered too complex for a given application domain, systems such as Cassandra resort to physical timestamps derived from node or client clocks, upon which they establish what replica version is considered the most recent. The drawback of this simplification is that it enforces a last writer wins strategy where some concurrent updates are lost. In addition, if the clocks are poorly synchronized some nodes/clients might always lose their competing concurrent updates.

Even in systems where a full-fledged characterization of causality is sought, there are important limitations to either system scalability or to the correctness of the causality tracking in present implementations. In this thesis, first we analyze these problems, with a special focus on solutions used in replicated key-value stores designed for cloud computing environments, such as Dynamo, Cassandra and Riak.

1.3 Contributions

We propose a novel mechanism, *Dotted Version Vectors (DVV)*, that can provide an accurate and scalable solution to track causality of updates performed by clients. Our approach builds on version vectors; however, unlike previous proposals, it does not require an entry per-client, but only an entry per-server that stores a replica, i.e., according to the degree of replication.

An full prototype of a real data storage system - Riak - using DVV is provided. Along with it, DVV implementation in Erlang [3] can also be used to different systems, serving as an aid to future implementations in different languages. Furthermore, we evaluated both the original Riak implementation and our implementation. Performance, metadata size growth and conflicts rates were all measured on different settings for an all around comparison.

Finally, two papers were written in the course of this thesis: 1) *Dotted Version Vectors: Logical Clocks for Optimistic Replication*, where Dotted Version Vectors are described in detail, and 2) *Evaluating Dotted Version Vectors in Riak*, where it is presented an overview of DVV implementation, and a comparison of DVV and Riak's Version Vectors.
The former is not yet published, but available at `http://arxiv.org/abs/1011.5808`; whereas the latter was accepted at Inforum 2011 [4].

1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 presents some literature review on the CAP Theorem and its implication on some distributed systems, following by an overview of some traditional logical clocks to capture causality relations. Chapter 3 discusses cloud databases, which ones were studied, and presents a more in depth presentation of classical approaches to capture causality, specifically in these systems. Chapter 4 presents Dotted Version Vectors (DVV), a novel mechanism to causality tracking, that scales in cloud databases without compromising its causality accuracy. In Chapter 5, we describe briefly the challenges when implementing DVV in Riak, and present

[3] `http://www.erlang.org/`
[4] `http://inforum.org.pt/INForum2011`

an evaluation of DVV with various benchmark settings, discussing the results in the end. Finally, in Chapter 6 we conclude our work by describing the objectives achieved and presenting some ideas for future research.

Chapter 2

Literature review

System Model For further scenarios, it will be assumed a model where the system is an asynchronous distributed system. This means that there is no global clock and no shared memory between processes. Processes (or Nodes) execute sequentially and local operations are atomic and instantaneous at the level of observation. It also implies that there are no guaranties that messages are delivered in a specific time, nor it is guarantied that the order by which they arrive to destiny are correct (the order they were sent may not be the order that they were received). Messages are sent via unicast, so they only have one destiny. This is also the only way for the processes to communicate.

This model is generic enough and represents well the traditional distributed system, since no assumptions are made on hardware, communication topology, network speed, CPU speed, etc. It can also support multicast messages by grouping a set of atomic sent messages.

2.1 CAP Theorem

Eric Brewer introduced in Brewer [8] the CAP conjecture, an acronym which stands for Consistency, Availability and Partition-tolerance. Later, the conjecture was formally proved by Seth Gilbert and Nancy Lynch [16]. This theorem says that only two of the three stated properties, can be *fully* supported in a distributed system. Lets described each one.

Consistency Having consistency means that all system nodes are up-to-date and are in the same state. There is no inconsistency, be it data inconsistency or anything else. This is the **Strong consistency**. Of course, this property is not "binary", i.e., there is a middle-ground, where we can have different levels of consistency. Strong consistency is hard and costly to guarantee when scaling, since when writing, for the operation to complete, all replicas must ensure a successful write. Therefore, other types of consistency started to gain visibility. One example of this is the **Eventual consistency**, where it is not guaranteed that nodes are always in sync, but *eventually*, they will converge and be consistent.

Availability Availability is property that is depends on how available data is across the system, at a given time. For example, if my system has two nodes and one of them crashes, my system would have 50% of availability. Of course, if data is replicated in both nodes, availability is 100% if at least one node is available. Therefore, to maximize availability, some degree of replication is used across data, but that also worsens the consistency complexity.

Partition-tolerance Lets say we have a distributed system, located in two parts of the globe. If internet fails to link them for some reason, we have a system with two partitions, each with an arbitrary node configuration. For example, if we require quorum to correctly write data, and each partition is equally divided, the system could be unable to write. Partition-tolerance is the degree to which a system can support or tolerate *correctly* these type of failures, where systems are divided in arbitrary partitions.

2.1.1 The CAP systems

The CAP theorem essentially tells us that we can have three system categories: CP, AP and CA. However, this has been criticized [1] because Partition-tolerance is something that systems want to have, because if a partition occurs

[1] http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

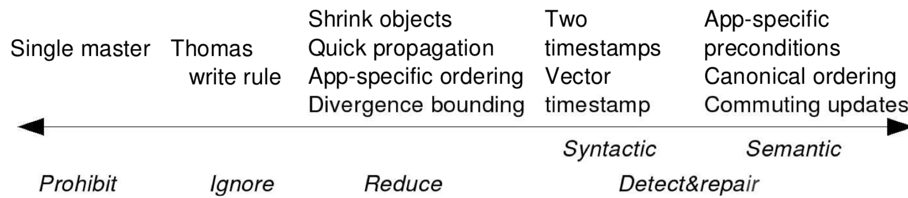


Figure 2.1: Design choices regarding conflict handling [37].

and it is not tolerated, then availability is also affected. Therefore, only CP and AP systems are actually used. The CAP theorem is also a spectrum, i.e., there are several levels of consistency, availability and partition tolerance. And in many cases, systems are actually adjustable or tunable to these properties, so that it can fit better to more use cases.

An example of a CP system is Google's Bigtable [10]. Bigtable does not replicate data, instead it relies that to its Google file system level which stores all the data. Therefore, is a node fails, that data is unavailable until that node is brought back online.

Amazon Dynamo [13] is a systems that in contrast to Bigtable, chooses availability and partition tolerance. Instead of Strong Consistency, it has Eventual Consistency. This means that when a node fails, its data is still available in a replica else where. Even when a partition occurs, data is still to be read if present in that partition, and also data can be written, even if not all replicas are available. This means that data can diverge momentarily, but eventually all data converges.

These AP systems are also called BASE systems. Its a system where the focus is not on ACID properties. Since availability is more important than consistency here, we trade the isolation and consistency of ACID, for performance and availability of BASE (Basically Available, Soft state, Eventually consistent) system [33]. With this type of system, data might not always be fresh, but given sufficient time and updates will propagate to all nodes. This is the design that web-scale systems or cloud systems use today to face increasing demand in scalability, performance and availability.

NoSQL systems are databases that usually apply a BASE approach instead of an ACID approach. They appeared exactly to face the increasing

scalability concerns. To do so, they applied what is called Optimistic Replication approaches [42, 7, 37]. They increase systems availability by allowing conflicts to occurs, instead of try to avoid them. Later they are detected and resolved. One of the focus of optimistic replication is how to handle conflicts. They happen when operations fail to satisfy their preconditions. In figure 2.1 we can see the different approaches to deal with conflicts. We can simply prevent it from happening, by aborting or blocking operations if necessary. But this hurts availability. Systems that use a single master do this, because only the master can write. Of course, conflicts would not occur, but the availability is very limited. Conflict rates can be reduced if updates are propagated faster, or by deconstructing multi-update transaction in individual updates. Ignoring updates is also a possibility. When conflict happens, the older data is overwritten by the new update. This is often called writer wins strategy [40]. If conflict rates are low, this can be a viable solution for some use cases. But what we really want is the ability to detect conflicts. Syntactic detection relies on ordering relations between operations and techniques for expressing them. These techniques are independent of the data or the use case. In contrast, semantic approaches are more complex to implement, because they differ with each case. However, having semantic repair can reduce conflicts and provide a better user experience, by resolving some conflicts automatically.

In the next section, we review ordering relations and the techniques to express them, usually called causality tracking mechanisms or logical clocks.

2.2 Order, Cause and Effect

Logical Clocks are about ordering events, so some Order Theory must be presented. More specifically, lets concentrate on Total Order and Partial Order. Given that, we must define by which property we should order clocks. In logical clocks we use the causality relation, which gives some sort of cause and effect between events.

2.2.1 Order Theoretical Concepts

Both Total Order and Partial Order are binary relation on a set. In this case, let E denote the set of events in a distributed computation, and denoted ≤ the binary relation.

Antisymmetry If a ≤ b and b ≤ a then a = b

Transitivity If a ≤ b and b ≤ c then a ≤ c

Totality a ≤ b or b ≤ a

Reflexivity a ≤ a

Both Total an Partial Orders have the antisymmetry and transitivity properties. What separates them is that a Total Order has a Totality property, i.e, all events in E are mutually comparable (≤). A Partial Order has a weaker form of Totality, which is Reflexivity, i.e., every event in E holds a relation to itself. this implies that when E is a Partial Order, some events in E are not comparable (≤). Given that a totality implies reflexivity, every Total Order is also a Partial Order.

2.2.2 Causality

Causality is a binary relation on partial order sets. Informally, this relation implies a cause-effect, where one event is the cause for another. Since all logical clocks are mechanisms to track causality, lets provide a clear definition of this relation [23]. → is the smallest transitive relation satisfying:

1. $a \rightarrow b$, if a and b are events in the same activity and a occurred before b;

2. $a \rightarrow b$, if a is the event of sending a message and b is the corresponding event of receiving that message.

Additionally, if and only if $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$ then a and b are *concurrent* ($a \parallel b$). Or in other words, if it cannot be said that a causality precedes b

and vice-versa.

Given the causality definition, lets introduce two other concepts: *characterize* and *consistent* [27, 28, 38]. Let E denote the set of events in a distributed computation, and let $(S, <)$ denote an arbitrary partially ordered set. Let $\theta : E \rightarrow S$ denote a mapping.

1. $(\theta, <)$ is said to be consistent with causality, if for all $a, b \in E$. $\theta(a) < \theta(b)$ **if** $a \rightarrow b$.

2. $(\theta, <)$ is said to characterize causality, if for all $a, b \in E$. $\theta(a) < \theta(b)$ **iff** $a \rightarrow b$.

As an example, real time (total order) is consistent with causality (partial order), since if $a \rightarrow b$, then a occurred before b. But real time does not characterize causality, because if a occurred before b, it does not mean that $a \rightarrow b$. Additionally, real time is a total order and total orders do not characterize partial orders.

2.2.3 Happens-before

Happens-before was introduced by Lamport [23] and characterizes the causality relation. It captures the relation between two events, where one event occurs before the other. This means that the first event *can potentially be* the cause for the second event (potential causality), but it also *might not be*. In other words, they are causally related (not concurrent), but not necessarily is one the cause for the other. However, this can only be confirmed with semantic information. Nevertheless, this potential causality is consistent with causality, only extending it with more events. Further, potential causality will be referred simply as causality.

Within a process p_1, a local event e_1 causally influences e_2 if the last happens after the first, since in a process, events occur sequentially. An event in process p_1 can only causally influence another event in process p_2, if e_1 is the event that sends a message to p_2 and e_2 is the event that receives that message

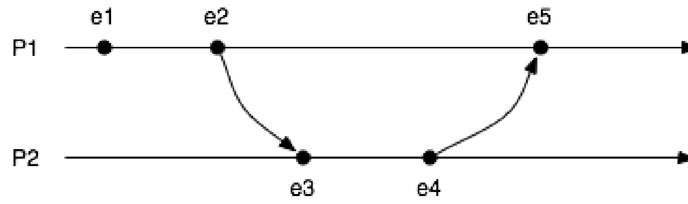


Figure 2.2: Two communicating processes

in p_2. This relation is also transitive, which means that events can causally influence others indirectly by transitivity.

Given the events e_1 and e_2, from now on, the *happens-before* relation can be formally denoted as $e_1 \rightarrow e_2$, when e_1 *happens-before* e_2, which only occurs when:

1. e_1 and e_2 are from the same process and e_1 occurs before e_2;
2. e_1 is the event that sends a message m to another process and e_2 is the event that receives m;
3. If exists an event e where $e_1 \rightarrow e$ **and** $e \rightarrow e_2$.

Figure 2.2 serves as example to these 3 conditions. For example: $e_1 \rightarrow e_2$ by (1); $e_2 \rightarrow e_3$ by (2); $e_1 \rightarrow e_4$ by (3).

As with causality, if neither one event *happens-before* the other, they are concurrent (||).

2.3 Logical Clocks

In this section, it will be presented some logical clock mechanisms, which are mechanisms that track causality (happens-before relation) among processes/nodes in a distributed system.

A way to capture the happens-before relation is to have a tag in each event. Lets have a function LC_i that maps events from process p_i to that tag,

thus representing a clock. This clock does not have anything to do with real time clocks, because it does not relate events in an absolute fashion. Instead it relates them in a relative and logical way, therefore being called *logical*. Also, this logical clock increases monotonically its value. To capture this relation, a comparison (<) of these tags must be done to partial order them. So for every pair of events, if $e_1 \rightarrow e_2$ then $LC(e_1) < LC(e_2)$. Attiya and Welch [6] described this as *clock condition*.

Lets introduce some basic terminology used in these contexts:

- **Dominate**: consider these two clocks LC_1 and LC_2; LC_2 dominates LC_1 if $LC_1 < LC_2$, i.e., if all elements of LC_1 are less or equal to all elements of LC_2, and at least one element of LC_2 is greater than some element of LC_1;

- **Conflict**: two clocks are in conflict when neither one dominates the other, thus there is no relative order, they are concurrent; sometimes these conflicting clocks are also called **siblings**;

- **Syntactical reconciliation**: when we can automatically reconcile two clocks, i.e., when one of the clocks dominates the other;

- **Semantical reconciliation**: when we *cannot* automatically reconcile two clocks, i.e., both clocks are in conflict.

2.3.1 Scalar Clocks

Scalar Clocks mechanism, also know as *Lamport Timestamps*, was introduced by Lamport [23]. This was the first and most simple mechanism that opened the way to future and more developed mechanisms. The concept is rather simple: each process has a local counter, which is an non-negative integer. Initially is set to zero. Each time an event occurs, the counter is updated as follows:

- In a local event or a send message event, the counter is incremented by one;

- In a sent message, it goes attached with it the current local counter (already updated by one);

- If the event is a received message, the new local counter is the maximum between the current local counter and the counter attached to the message, plus one.

In figure 2.3 we can observe examples of the rules above. For example, in process P_1 the first event is a local event and increments the previous value counter, which was zero, by one. Again, it can be seen that the second event in P_1 is a message sent to P_2, attached with the updated local counter, which is 2. Finally, the second event of P_3 was a received message by P_2, and the value was calculated by $max(1, 4) + 1$ which results in 5, the updated value of the counter on this event.
Note that this is an asynchronous distributed system, there are no guaranties in the order of the messages, as it can be seen in figure 2.3.

It might be a necessity to attach along with scalar clock, the ID of the process, for differentiating between events that occurred in different processes but have the same logical time.
Still, this mechanism is only consistent with causality (total orders do not characterize partial orders). As an example, using scalar clocks, the second event of P_1 is greater (>) than the first event of P_3. But they are not causally related, they are concurrent (||), which is not captured by this mechanism.

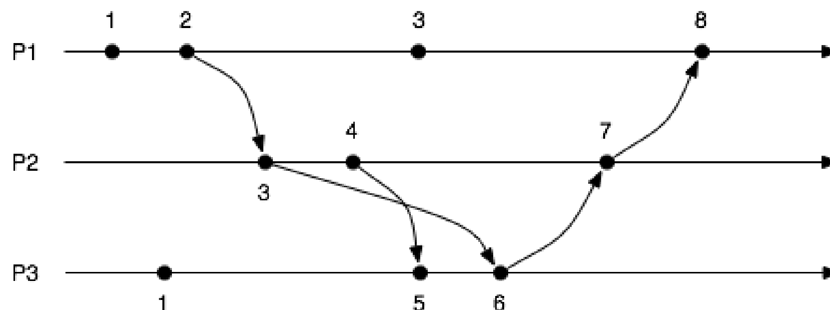


Figure 2.3: Scalar Clocks

2.3.2 Vector Clocks

Vector Clocks (VC) mechanism was introduced simultaneously and independently by Fidge [14] and Mattern [28]. Instead of having a counter per process, we have a list (or vector) of counters (non-negative integer, initially set to zero) per process. Each counter has associated the process ID that it represents. Local events are represented in the local vector and they "synchronize" when exchanging messages. Each time an event occurs, the vector is updated as follows:

- In a local event or a send message event for a process P_i, the value in $LC_{Pi}[i]$ is incremented by one;
- In a sent message, it goes attached with it the current local vector LC_{Pi} (already updated);
- If the event is a received message for a process P_i, the value in process P_i is incremented by one. Considering the message came from P_m, for every position j, where j ≠ i, the updated value is calculated by: $max(LC_{Pm}[j], LC_{Pi}[j])$.

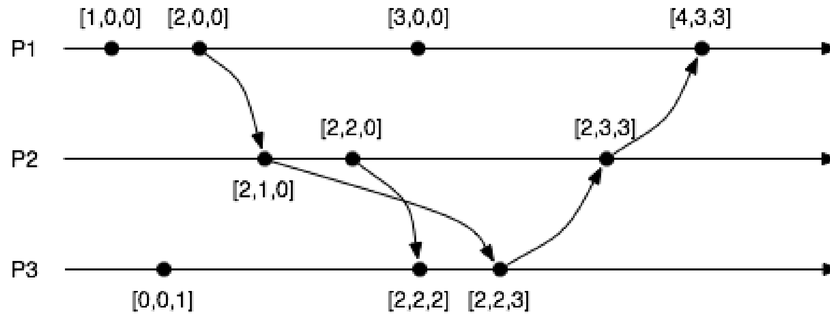


Figure 2.4: Vector Clocks

In figure 2.4 lies examples of the rules above. For example, in process P_1 the first event is a local event and increments by one the previous value of (in this case) the first position of the local vector. Again, it can be seen that the second event in P_1 is a message sent to P_2, attached with the updated local counter. Finally, the second event of P_3 was a received message by P_2, and

the local vector was updated by one in position three. As for positions 1 and 2, the calculation was the same in this case: $max(0, 2)$ which resulted in the value 2 for both positions.

Vector clocks are partial orders, in contrast with scalar clocks that were total orders. Partial order can be defined as follows: $\boldsymbol{LC_{P1}} \leq \boldsymbol{LC_{P2}}$ if $\forall i, LC_{P1}[i] \leq LC_{P2}[i]$ and $\boldsymbol{LC_{P1}} < \boldsymbol{LC_{P2}}$ if $\forall i, LC_{P1}[i] \leq LC_{P2}[i] \wedge \exists j, LC_{P1}[j] < LC_{P2}[j]$. So if $\neg(LC_{P1} \leq LC_{P2}) \wedge \neg(LC_{P2} \leq LC_{P1})$ then these vectors are *incomparable*.
So if LC_{P1} and LC_{P2} are incomparable vectors, then they are concurrent. On the other hand, if $LC_{P1} < LC_{P2}$ then $LC_{P1} \rightarrow LC_{P2}$ (Attiya and Welch [6] call this the strong consistency condition). So, it can be concluded that vector clocks characterize causality.

An important property of this mechanism is that the minimality result by Charron-Bost [12], indicates that vector clocks are the most concise characterization of causality among process events. However, until now we described the process causality, but other forms of causality exist and the minimality result does not applies to it (see chapter 2.3.3).

This mechanism is designed to work in static and well-connect scenarios, thus they do not scale very well for dynamic systems, where processes can be added or removed as desired. This happens because vector clocks has to store information for every node that was added, regardless of its retirement or not, causing potential unbounded growth.

2.3.3 Version Vectors

Vector clocks are the most concise characterization of causality among process events, but process causality is not the only causality. *Data causality*, were local events are updates to the local data, and messages are used to synchronize clocks between nodes, which leads to a common state between them. Examples of these systems are partitioned replicated systems with strong consistency in each partition, replicated file systems, databases and some classes

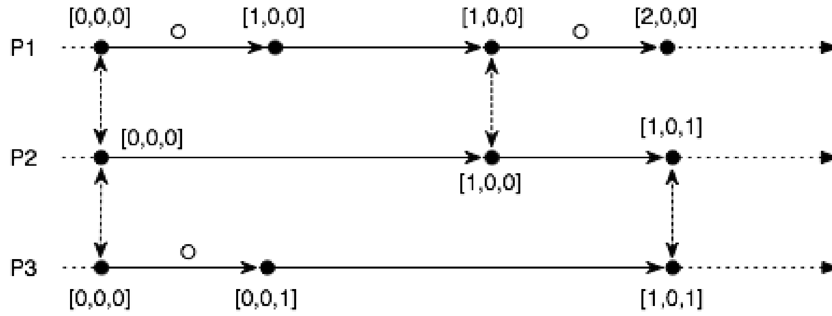


Figure 2.5: Version Vectors

of code version control systems.

Parker et al. [30] introduced Version Vectors (VV), which shared equivalent structure from Vector Clocks. But its purpose was, however, different. It aimed at the detection of mutual inconsistency between replicas in optimistic replication systems [37]. Being structurally equivalent to Vector Clocks, each counter here represents the last known update for its respective owner ID. E.g., a version vector of pairs clientID, counter, represents the number of updates that each client made. Similarly, a version vector of pairs nodeID, counter, represents the number of updates that each node made. There are two main operations: the *update* operation that updates local nodes, and the *synchronization* operation, where two replicas synchronize, thus resulting in common state between them. Both are viewed as atomic operations. The algorithm to update version vectors is a little different from vector clocks:

- In a local update of node P_i, the value in $v_i[i]$ is incremented by one;
- In a synchronization event between nodes P_1 and P_2, both clocks are compared and can only conclude one of three thing: (1) P_1 is more recent ($LC_{P1} \rightarrow LC_{P2}$), thus keep this version in both nodes; (2) P_2 is more recent ($LC_{P2} \rightarrow LC_{P1}$), thus keep this version in both nodes; (3) P_1 and P_2 have conflicting data ($LC_{P1} \parallel LC_{P2}$). The way that conflicts are resolved is an application specific problem. Logical clocks propose it to comparing clocks and detecting conflicts, not necessarily resolve those conflicts, but it may help.

Figure 2.5 represents a possible scenario of execution. First of all, arrows with a circle on top are local updates; vertical arrows are synchronization between nodes. Nodes start at the same synchronized state [0,0,0] (in the example, the relative position of the counter, indicates which node represents, e.g., the first position represents P_1 counter). P_1 for instance, first suffers an update on its data, thus incrementing its position in the vector [1,0,0]. Next, P_1 and P_2 synchronize, which leads to both of them having, in the end, the same vector [1,0,0]. Here is a major difference between Vector Clocks and Version Vectors: in this case, the synchronization did not update the local counter in the P_2 vector, because it was not a local update to its replica, whereas in the vector clock mechanism, this would be seen as an event, thus incrementing local counter by one.

One particular common use of these mechanisms is tracking causality between coexisting replicas, i.e., between replicas forming a consistent cut [28] of the system state [38, 3]. Still, in these cases, version vectors carry information about causality, concurrency and past events. So it would serve for every possible consistent cut, but normally that much information is not required. Almeida et al. [2] described *Bounded Version Vectors*, a different version of this mechanism where past information is discarded.

2.3.4 Interval Tree Clocks

$$(1,(0,1)) \sim$$
$$((0,(1,0)),(1,0)) \sim$$

Figure 2.6: ITC id [5]

Interval Tree Clocks (ITC) was introduced by Almeida et al. [5]. It was the result of the evolution of mechanisms like Version Stamps [3] and Dynamic Map Clocks [4]. It can be seen as a generalization of vector clocks. Therefore it suits well static scenarios. However, its strength is the support for dynamic systems since processes/nodes can be added or retired in a decentralized manner. It consists in each node having a stamp, which is constituted by a pair of an unique identifier and an event counter:

$$ITCstamp = \{Id, Event\}$$

This mechanism differs from classical ones for two reasons:

1. Unique identifier that does not map into integers, nor it have a pre-defined set of values. Instead it is a tree structure, that dynamically adapts to the number of replicas/processes, which can be created or retired locally.

2. Classical mechanisms use functions over a finite domain, in contrast functions in ITC are used over an infinite domain R - mainly [0,1[- that can be divided and joined as needed.

The unique identifier serves the same purpose as the IDs that were tagged along each counter in vector clocks and version vectors. The unique identifier in ITC can be represented by:

$$i ::= 0 \mid 1 \mid (i_1, i_2)$$

Figure 2.6 demonstrates two possible structures of the id component. On the right side lies a visual representation of these structures. The id can be the reunion of various subsets of the the interval [0,1[. A node can only update areas were it has value of 1, or in the visual representation, it can only update areas that have a bar.

The event component is where the causal history is stored. It serves the same purpose as the counters in vector clocks. Its structure is a binary tree, where each tree node has a counter. This component can be formally described as:

$$e ::= n \mid (n, e1, e2)$$

$$(1, 2, (0, (1, 0, 2), 0)) \sim$$

Figure 2.7: ITC event [5]

Graphically, we can view each counter node as a height, where the the root serves as the base height and each branch sums its height on top of it recursively, either in left half or in the right half, depending if it is the left or right branch, respectively. Figure 2.7 demonstrates two possible representations of the event component. At the right lies a graphical representation of this structure. The event can assume infinite forms, because the tree can create new leafs infinitely.

$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \sim$$

Figure 2.8: ITC stamp [5]

Figure 2.8 demonstrates the final structure of an ITC stamp. The left pair is the id component which represent where this node can update the event structure. In the graphical representation, this id component are the bottom bars. The right side of the stamp is the event structure that represents causality tracking that is known by this node. Graphically it is all the bars above the id component.

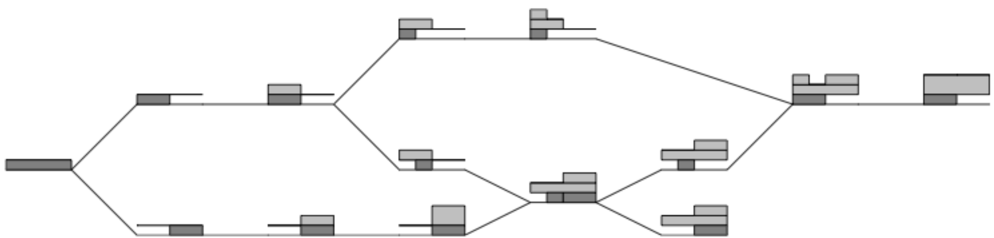


Figure 2.9: ITC run case example [5]

Finally, figure 2.9 demonstrates a hypothetical run of this mechanism. It starts with one node that has all id space available for updates, since its the only entity in the system. Then there was a need to create/add a new node, so the initial node was *forked*, which implies dividing equally the id space between the two nodes and copying the event structure to both. This event structure then diverges from there, since the space where they can update the

event structure is completely different (the id space for each is different for both). Eventually, the retirement of a node can be necessary, in which case the *join* operation will be performed on both the ITC structure of the node that will be removed and a random node (or not, there might be optimizations here, where a carefully selected node can result in a "better" *join*). This operation *join* is achieved by the reunion of the id spaces (visually it is the overlapping of the two id bars) and the reunion of event structures (visually it is the maximum height between them for all the spectrum).

Since the implementation size is one of the most important aspects of a logical clock, a few optimizations are used when updating the event tree, thus making this process rather tricky. First, it is used the operation *Fill* on the space that is going to be updated. *Fill* either succeeds in doing one or more simplifications, or returns the unmodified event tree; it never increments an integer that would not lead to simplifying the tree. If the previous operation did not update the event tree, then the operation *Grow* is used to update it. Grow performs a dynamic programming based optimization to choose the inflation that can be performed, given the available id tree, so as to minimize the cost of the event tree growth. It is defined recursively so that:

- incrementing an integer is preferable over expanding an integer to a tuple;
- to disambiguate, an operation near the root is preferable to one farther away.

2.4 Summary

We introduced the CAP theorem and how systems are using it to achieve better performance and scalability. These systems discard the ACID constraints, in favor of a more relaxed approach - BASE. Optimistic Replication is often used to create more available systems. In specific, they allow data do diverge, which often creates conflicts. Detecting these conflicts is crucial. To aid in this task, we introduced logical clocks.

Logical Clocks are version tracking solutions and their use in cloud storage systems are rooted on Lamport' seminal work on the definition and role of causality in distributed systems [23]. This work was the foundation for subsequent advances in causality's basic mechanisms and theory, including the introduction of version vectors [30] for tracking causality among replicas in a distributed storage system and vector clocks [14, 28] for tracking causality of events in a distributed systems.

Most of this initial work dealt with a fixed, mostly small, number of participants. Later, several systems introduced mechanisms for the dynamic creation and retirement of vector entries to be used when a server enters and leaves the system. While some techniques required the communication with several other servers [17], others required communication with a single server [31]. Interval Tree Clocks [5] are able to track causality in a dynamic, decentralized scenario where entities can be autonomously created and retired.

Chapter 3

Causality Tracking in Cloud Databases

3.1 System model

Cloud Databases are storage systems for cloud computing environments. They can be seen as a set of interconnected server nodes that provide a data read-/write service to a much larger set of clients. We can consider a standard key-value store interface that exposes two operations: GET(K) and PUT(K,V). (A delete operation can be implemented, for example, by executing a put with a special value.)

A given key is replicated in only a subset of the server nodes, which we call the replica nodes for that key. For our analysis, the approach used to decide which nodes will replicate a given key (e.g., consistent hashing) is not important. Depending on the system, in each replica node, for each key, the system maintains either a single value or multiple concurrent values. We name each of these values, a replica version or simply a version when no confusion may arise.

These systems usually rely on an optimistic replication approach [37], allowing client operations to complete without coordination. In case of concurrent updates to the same key, these systems usually guarantee eventual consistency by either relying on a last writer wins strategy [40] (e.g. Cassan-

dra [22]) or by maintaining multiple versions for the concurrent updates to the key (e.g. Amazon's Dynamo [13], Depot [24], Riak). In the latter case, conflicts can be solved by issuing a new update that supersedes the concurrent versions, which is usually done by the client (but could also be automatically executed by application code running on a server).

For achieving this execution model, these systems must include some form of causality tracking. Lets analyze some cloud databases and their approach on tracking causality.

3.2 Cloud Databases

Logical Clocks are mechanisms almost ubiquitously used for versioning data. Particularly, in storage systems that provide high availability by relaxing its consistency levels. Such systems provide an eventual consistency model for data replication that trade-offs consistency for availability and partition-tolerance. This is done to build distributed systems at a worldwide scale. We call this type of consistency *Eventual Consistency*.
Amazon's Dynamo is one good example of a highly available storage that uses Eventual Consistency [13]. It also inspired a few other distributed databases like Cassandra and Riak. As described in Dynamo's paper, it uses Version Vectors [23]. Same goes for Riak, although with significantly different approaches. Cassandra as of 0.8.4, still uses timestamps.

Lets describe some Cloud Databases, explaining some of their internals, and more importantly what they use for versioning data.

3.2.1 Dynamo

Dynamo is a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use. Dynamo's architecture is mostly described in DeCandia et al. [13].

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Figure 3.1: Summary of techniques used in Dynamo and their advantages [13].

Figure 3.1 shows a summary of techniques employed, depending on the problem.

System Interface Dynamo has two operations - *get(key) -> value|[values], Clock* and *put(key, context, value) -> ok|value*. The **Get** operation is straightforward - providing the key, it returns the corresponding value, or a list of values in case of conflicting values in the system, and along with this, it also returns a context, which is an opaque object to the client that contains metadata such as the logical clock of the value. The **Put** operation requires three parameters - the key, the value and the context that was obtained in a previous get operation. The node that handles these operations is called the **coordinator**.

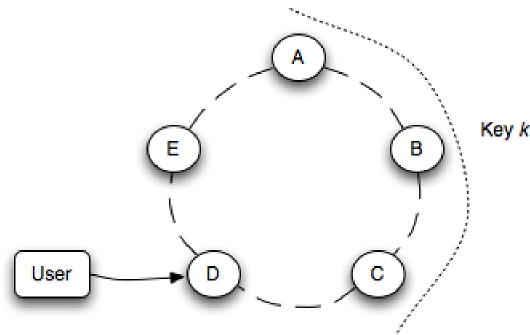


Figure 3.2: Dynamo ring

Partitioning Dynamo uses consistent hashing for the keys, so that the output in that hash function results in a circular ring, where the max value wraps around the min value. Figure 3.2 represents such ring. Each physical node in the system is represented by one or more virtual node in Dynamo's ring. This number of virtual node varies for each physical node, in such a way that the system can load balance itself (more capacity/power equals more virtual nodes, thus accounting for infrastructure heterogeneity), resist to node failures without overloading the neighbor nodes and scale dynamically when adding physical nodes (by accepting almost same load for very other node in the system). From here on, nodes are virtual nodes and not physical node.

Replication For replication, Dynamo uses a configurable replication factor N per-instance, which means that every value is replicated for N hosts. When a key/value is initially added to the system, it is assigned to the coordinator. Then it is replicated to the N-1 clockwise successor nodes in the ring. There also exists a preference list, that are the nodes responsible for storing a particular value. This list is constituted by the top N nodes described earlier and also with some extra nodes, in case some of the top N nodes fails. This mechanism for handling temporary node or network failures is called *Hinted Handoff*. In a put operation, if some top N node fails, the next node in the preference list receives the value and a hinted in the metadata, that indicates what node this value was intended to. When the failed node comes back to the system, the backup node copies the value to the original and can now delete

the local replica of that value.

Of course, applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store.

Note that a physical node can have more than one virtual node, so to ensure that in the preference list there are only distinct physical nodes, it is constructed by skipping positions in the ring.

Versioning For every pair key/value, Dynamo also has a corresponding logical clock, more specifically a Version Vector (VV). These VV are arrays of pairs $\{NodeId, Counter\}$. *NodeId* is the identifier of the node and *Counter* is the respective counter. For every value in Dynamo, VV has the pairs corresponding to the nodes that have a replica for that value. However, Dynamo applies a truncation to the VV, which means that if the array exceeds a specific value, then the oldest pair of the array is removed. This can be done because attached to each pair is a timestamp, that indicates the last time that node updated the value. Dynamo's paper [13] does not actually states what value is used for truncation, but it hints to 10. This truncation approach is used because VV could grow too much, if too many servers coordinate writes for that clock and if node failures result in new IDs for the server.

Consistency To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that $R + W > N$ yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Get/Put Operations Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. Using this framework, a client can contact any Dynamo node by either using a load balancer that will decide based on current load, which node to contact, or by using a partition-aware library that forwards the request directly to nodes that are responsible that for that key.

A node handling a read or write operation is known as the coordinator, which can be any node in the system for read operations, any node in the preference list for write operations. This restriction happens because nodes coordinating writes have to update to version clock so that it supersedes the older local version. This means that if a node receives a write operation and it is not in the top N nodes of the preference list for that key, then it forwards the request to the first node of the top N list.

On new write operations, the node creates a new VV, which then sends it to the top N nodes. Then it awaits that W-1 nodes responses arrive, so that this write can be considered successful. Similarly, on reads the coordinating node contacts the top N and awaits for R results before responding to the client. If more than one version is receive from the nodes, it returns to the client the version that are in conflict. It is the client responsibility to resolve the conflict and submit the new version through a write.

Data Versioning

Data Versioning can be resumed by these points:

- Dynamo uses Version Vectors for tracking causality between replicas, using server ids;
- Reconciliation for divergent replicas is done by the client. In the get operation it receives the conflicting values and theirs clocks, then applies some logic to solve the divergence, and writes back to Dynamo through a put operation;
- The actual size of the VV is truncated when exceeding some value, therefore losing the oldest entries in the clock.

Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. When the system is facing failure scenarios such as node failures, data center failures, or network partitions, conflicting updates may occur. It also may occur when the system is handling a large number of concurrent writers to the same data item, and different nodes end up coordinating the updates concurrently. Clearly, it is best to maintain the number of conflicting versions as low as possible, given the performance concerns and the complexity that it brings. If the versions cannot be (syntactically) reconciled based on VV alone, they have to be passed to the business logic for (semantic) reconciliation. This introduces additional load on services, so it is desirable to minimize the need for it. Additionally, without the non-truncated VV to decide, this semantic reconciliation can lead to errors because it is not always easy or even possible to decide the correct version from a business logic level.

Dynamo's paper states that in a live production environment, 99.94% of requests saw only one version of the data, 0.00057% of requests saw 2 versions, 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. Concurrent writes was the main reason for the conflicts.

Given that writes are coordinated by the healthy top N node in the preference list, and in case of failures, coordinated by the proceeding nodes in the preference list, we can say that it a fair assumption that not many nodes rather then the top N, will coordinate a write for a given key. Thus, the actual size of the VV does not tend to be very large, so the probability of the VV being truncation is slim (given a reasonably truncation size).

While theoretically the truncation approach could be an issue by having false conflicting updates, with such a low divergence probability and given the fact that VV tend to not grow very much, we can pragmatically assume that this would not be an issue in actual production sites. What can be a problem is when clients try to write a value that was already overwritten by others, thus creating conflict. Being Dynamo a closed source software, not much can be said other than what it is written in the paper. Regarding this issue, it is not clear if this issue is even accounted for. Maybe they choose to discard writes that were read from past versions. Nevertheless, it is a problem, given that

VV using server IDs do not have a way to express conflict in the same server, for the same key.

3.2.2 Cassandra

Facebook runs the largest social networking platform that serves hundreds of millions users at peak times using tens of thousands of servers located in many data centers around the world. To meet the reliability and scalability needs of such massive system, Facebook developed Cassandra. Cassandra uses a synthesis of well known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. Inbox Search is a feature that enables users to search through their Facebook Inbox. This meant the system was required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. Since users are served from data centers that are geographically distributed, being able to replicate data across data centers was key to keep search latencies down [21].

Cassandra was open sourced by Facebook in 2008, and is now developed by Apache committers and contributors from many companies. The Apache Cassandra Project develops a highly scalable second-generation distributed database, bringing together Dynamo's fully distributed design and Bigtable's ColumnFamily-based data model. It is used in sites such has Digg, Facebook, Twitter, Reddit, Rackspace, Cloudkick, Cisco, SimpleGeo, Ooyala, OpenX, and more companies that have large, active data sets. Given the open-source nature of Apache's Cassandra, from here on, we will concentrate on it.

Cassandra's architecture is vastly inspired in Dynamo's architecture. It is a system that was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency. Next it is described the fundamental differences between Cassandra and Dynamo. [1]

Data Model Instead of using the simple key/value approach that Dynamo uses, Cassandra inspired itself in Google's BigTable for its data model. This

[1] Partially based on `http://io.typepad.com/glossary.html`

can be described by the following characteristics [2]:

- Every row is identified by an unique key, which is a string with no limit on its size.
- Every instance of Cassandra has one table which is made up of one or more column families.
- Each column family can contain one of two structures: supercolumns or columns. Both of these are dynamically created and there is no limit on the number of these that can be stored in a column family.
- Columns have a name, a value and an user-defined timestamp associated with them. Columns could be of variable number per key. For instance key K_1 could have 1024 columns/supercolumns, while key K_2 could have 64 columns/supercolumns.
- Supercolumn have a name, and an infinite number of columns associated with them. The number of supercolumns associated with any column family could be infinite and of a variable number per key. They exhibit the same characteristics as columns.

System Interface Cassandra's interface is a bit more complicated that Dynamo, which only had two operations and was done through a HTTP framework. Cassandra has a high level interface named Thrift [3]. Thrift is the name of the RPC client used to communicate with the Cassandra server. Using Thrift, we have to create an explicit connect to a node in Cassandra. Using this open connection, we execute operations available in the Application Programming Interface (API) [4]. Lets concentrate only in the traditionally read and write operations, which have the following signature:

$$get(keyspace, key, column_path, consistency_level)$$

[2] http://www.facebook.com/note.php?note_id=24413138919&id=9445547199&index=9

[3] Several high level APIs build on top of Thrift, for several programming languages, are available and can be found in Cassandra's wiki http://wiki.apache.org/cassandra/ClientOptions#High_level_clients

[4] http://wiki.apache.org/cassandra/API

So, to read a value in cassandra we must specify the keyspace (contains multiple Column Families), the key (a unique string that identifies a row in a ColumnFamily), the column path (the path to a single column in Cassandra) and the consistency level for this operation.

The write operation has the following signature:

$$insert(keyspace, key, column_path, value, timestamp, consistency_level)$$

Similar to the read operation, with the exception of timestamp. This should be the actual system time in microseconds since the Unix epoch (midnight, January 1, 1970).

Replication Cassandra offers a configurable replication factor, which allows essentially to decide how much you want to pay in performance to gain more consistency. That is, your consistency Level for reading and writing data is based on the Replication Factor, as it refers to the number of nodes across which you have replicated data. Replication Factor is configured using the $< ReplicationFactor >$ element.

The replication strategy, sometimes referred to as the placement strategy, determines how replicas will be distributed. The first replica is always placed in the node claiming the key range of its Token. All remaining replicas are distributed according to a configurable replication strategy. The Gang of Four Strategy pattern [5] is employed to allow pluggable means of replication, but Cassandra comes with three out of the box. *RackUnawareStrategy* is the simplest strategy that places replicas the nearest on the ring, ignoring physical proximity. *RackAwareStrategy* places the second replica on a different data center than the first replica, and the remaining replicas in different racks in the same data center. The third is *DatacenterShardStrategy* which you can supply a file called “datacenters.properties” in which you indicate the desired replication strategy for each data center.

[5] `http://en.wikipedia.org/wiki/Strategy_pattern`

Consistency Cassandra uses the same semantics of W, R and N for consistency. But has more options for the level of consistency. This configurable setting allows to decide how many replicas in the cluster must acknowledge a write operation or respond to a read operation, in order to be considered successful. The Consistency Level is set according to your stated Replication Factor, and not the raw number of nodes in the cluster. There are multiple levels of consistency that you can tune for performance. The best performing level has the lowest consistency level. They mean different things for writing and reading.

A brief for the different settings for writes are described below:

- ZERO: Write operations will be handled in the background, asynchronously. This is the fastest way to write data, and the one that offers the least confidence that your operations will succeed.
- ANY: This level was introduced in Cassandra 0.6, and means that you can be assured that your write operation was successful on at least one node, even if the acknowledgement is only for a hint (see Hinted Handoff). This is a relatively weak level of consistency.
- ONE: Ensures that the write operation was written to at least one node, including its commit log and memtable. If a single node responds, the operation is considered successful.
- QUORUM: A quorum is a number of nodes that represents consensus on an operation. It is determined by $< ReplicationFactor > / 2 + 1$. So if you have a replication factor of 10, then 6 replicas would have to acknowledge the operation to gain a quorum.
- DCQUORUM: A version of Quorum that prefers replicas in the same Data Center in order to balance the high consistency level of Quorum with the lower latency of preferring to perform operations on replicas in the same Data Center.
- ALL: Every node as specified in your $< ReplicationFactor >$ configuration entry must successfully acknowledge the write operation. If any

nodes do not acknowledge the write operation, the write fails. This has the highest level of consistency, and the lowest level of performance.

And the following for read operations:

- ONE: This returns the value on the first node that responds. Performs a read repair in the background.
- QUORUM: Queries all nodes and waits for a majority of replies to respond to the client.
- ALL: Queries all nodes and returns the value with the most recent timestamp. This level waits for all nodes to respond, and if one does not, it fails the read operation.

Read Repair This is another mechanism to ensure consistency throughout the node ring. In a read operation, if Cassandra detects that some nodes have responded with data that is inconsistent with the response of other newer nodes, it makes a note to perform a read repair on the old nodes. The read repair means that Cassandra will send a write request to the nodes with stale data to get them up to date with the newer data returned from the original read operation. It does this by pulling all of the data from the node and performing a merge, and then writing the merged data back to the nodes that were out of sync. The detection of inconsistent data is made by comparing timestamps and checksums.

Failure Detection Failure detection is the process of determining which nodes in a distributed fault-tolerant system have failed. Cassandra's implementation is based on the idea of Accrual Failure Detection Hayashibara et al. [18]. Accrual failure detection is based on two primary ideas: that failure detection should be flexible by being decoupled from the application being monitored, and that outputting a continuous level of suspicion, regarding how confident the monitor is that a node has failed. This is desirable because it can take into account fluctuations in the network environment. Suspicion offers a more fluid and pro-active indication of the weaker or stronger possibility of

failure based on interpretation (the sampling of heartbeats), as opposed to a simple binary assessment.

Data Versioning

Cassandra did not implemented Version Vectors like the original Dynamo. Instead each column is associated with a timestamp. These timestamps are supplied by the client, so it is important to synchronize client clocks. The timestamp is by convention the number of microseconds since the Unix epoch (midnight, January 1, 1970). Cassandra compares timestamps to determine which version is newer.

Timestamps have some advantages over mechanisms such as VV with server IDs (like Dynamo). Using timestamps, any node in the system can be a coordinator for a write for any column. This is true since the client can pass the actual timestamp, so that the coordinator only has to propagate this to the replicas. This is how it is done in Cassandra. This actually makes for better performance than the Dynamo approach where in each write, if the node that receives the request is not in the top N nodes for that key, then it has to forward this request to the first healthy node in the preference list. Other advantage is the fact that processing comparisons between timestamps (long integers) is much less cpu heavy them comparing logical clocks such as ITC or VV. Timestamps also have a fixed size, while VV especially and ITC to a lesser extent, tend to be bigger in size and even growing as time goes on.

Although timestamps are easy to use and have some advantages, it has disadvantages as well. For example, it requires a global clock coordination to clients (probably through Network Time Protocol (NTP)). If a malfunction or malicious client sends a timestamp X from the future, then the following writes would be discarded, until they have a greater or equal timestamp than timestamp X. The same goes for a timestamp from the past, where this write would be discarded, if any timestamp on the server was greater. Also, if a client reads a value and holds it for to long then writes, it would probably super-seed writes from other clients that should be considered concurrent, thus losing updates. In short, Cassandra implies a model where the value with greater timestamp wins, and if both timestamps are equal, they apply the

"last write wins" approach [40], where the client updated overwrites server value.

3.2.3 Riak

Riak is developed by Basho Technologies and is heavily influenced by Dr. Eric Brewer's CAP Theorem and Amazon's Dynamo [13]. Written mostly in Erlang, with a small amount of Javascript and C, it is a decentralized, fault-tolerant key-value store, with special orientation to document storage. Being heavily influenced by Dynamo, Riak adopts the majority of its key concepts. Being a truly fault-tolerant system, it has no single point of failure, since no machine is special or central. Next, a brief description of some relevant areas of Riak.

Data Model Riak is a key-value store that has the additional concepts of Bucket and Links. A bucket is a container for keys, with a set of common properties for its contents (e.g. replication factor). This gives the option to use several times the same key in a Riak system. Links are metadata attached to objects in Riak. These links make establishing relationships between objects in Riak as simple as adding a Link header to the request when storing the object.

Replication Inspired by Dynamo, Riak uses the same ring concept for replication. Consistent hashing is used to distribute and organize data. This Riak ring has a 160-bit space size, and by default has 64 partitions, each represented by virtual nodes (vnodes). Vnodes are what manages client's requests, like puts e gets. Each physical node can have several vnodes, depending both on the number of partitions the ring has and the number of physical nodes. The average number of vnodes per node can be calculated by (number of partitions)/(number of nodes). Vnodes positions in the ring are attributed at random intervals, to attempt a more evenly distribution of data across the ring.

By default, the replication factor (n_val) is 3 (i.e., 3 replica vnodes per key). Also, the number of successful reads (R) and writes (W) are by default a quorum number (greater than $n_val/2$), but can be configured depending on

the consistency and availability requirement levels.

System Interface In Riak, all requests are perform over HTTP by RESTful Web Services. All requests should include the *X-Riak-ClientId* header, which can be any string that uniquely identifies the client, to track object modifications with Version Vectors (VV). Additionally, every GET request provides a context, that is meant to be given back (unmodified) in a subsequent PUT on that key. The context contains the VV.

Here are some examples of these operations:

- $GET/riak/bucket/key$: reads an object;
- $PUT/riak/bucket/key$: writes an object;
- $POST/riak/bucket/key$: writes a new object with a random Riak-assigned key;
- $DELETE/riak/bucket/key$: deletes the object;
- $GET/stats$: reports about the performance and configuration of the Riak node to which it was requested;

Data Versioning Riak has two ways of resolving update conflicts on Riak objects. Riak can allow the most recent update to automatically "win" (using timestaps) or Riak can return/store (depends if it is a GET or PUT, respectively) all versions of the object. The latter gives the client the opportunity to resolve the conflict on its own. This occurs when the *allow_mult* is set to true in the bucket properties. When this property is set to false, there is a silent loss update possibility, e.g., when two clients write to the same key concurrently (almost at the same time), one of the updates is going to be discard. Hereafter, assume that *allow_mult* is always true.

Riak Object: A riak object represents the *value* in the *key-value* tuple, i.e., it contains things like metadata, the value(s) itself, the key, the logical clock, and so on. So, from now on, object is the riak object and value

or values are the actual data of an object that being stored, e.g., a text, a binary, an image, etc.

Sibling: A sibling (concurrent object) is created when Riak is unable to resolve the request automatically. There are two scenarios that will create siblings inside of a single object:

- A client writes a new object, that did not come from the current server object (it is not a descendent), conflicting with the server object;
- A client writes a new object, without context, i.e., without VV.

Riak uses VV to track versions of data. This is required since any node is able to receive any request, even if not replica for that key, and not every replica needs to participate (being later synchronized via read-repair or gossiping). When a new object is stored in Riak, a new VV is created and associated with it. Then, for each update, VV is incremented so that Riak can later compare two object versions and conclude:

- One object is a direct descendant of the other.
- The objects are unrelated in recent heritage (the client clock is not a descendent of the server clock), thus considered concurrent. Both values are stored in the resulting object, while both VV are merged.

Using this knowledge, Riak can possibly auto-repair out-of-sync data, or at least provide a client with an opportunity to reconcile divergent objects in an application specific manner.
Riak's implementation of VV tracks updates done by clients instead of tracking updates "written" or handle by nodes. Both are viable options, but what Riak's approach provides are what clients updated the object (and how many times), in contrast to what nodes updated the object (and how many times). Specifically, VV are a list of number of updates made per client (using X-Riak-ClientId), like this: $[\{client1, 3\}, \{client2, 1\}, \{client3, 2\}]$. This VV would indicate that client1 updated the object 3 times, client2 updated the object 1 time, and client3 updated the object 2 times. Timestamp data is also stored in the VV but omitted from the example for simplicity. The reason to use client

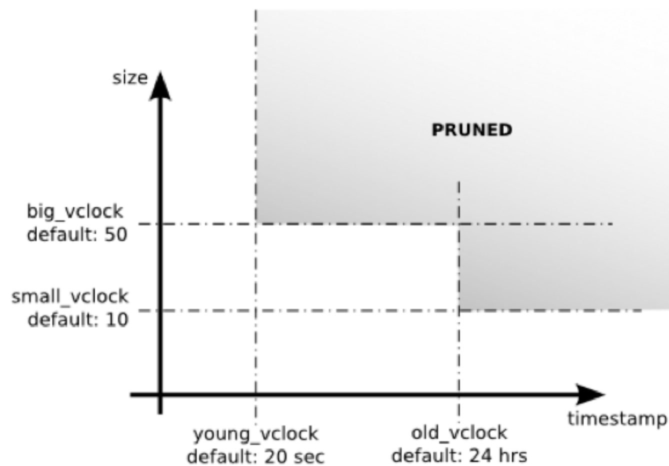


Figure 3.3: Default values to prune vector clocks in Riak.

IDs instead of server-side IDs in VV, is because the latter can cause silent update losses, when two or more clients concurrently update the same object on the same node. [6]

There is a major difference between this and the traditional approach of using the node's IDs, because the number of clients tends to be much greater than the actual number of nodes or replicas. Therefore, the VV would grew in size much faster, probably in an unacceptable way (both size and performance). The solution Riak adopted was to prune VV as they grow too big (or to old), by removing the oldest (timestamp wise) information. The size target for pruning is adjustable, but by default it is between 20 and 50, depending on data freshness. Figure 3.3 [7] shows the default values for the relevant flags that dictate when a VV is pruned. Removing information will not cause data loss, but it can create false conflicts. For example, when a client holds an object with an old unpruned VV and submits it to the server, where the clock was pruned, thus creating conflict, where it should not have happened.

In short, the tradeoff is this: prune to keep the size manageable, thus letting false conflicts happen. The probability of false conflict happening should not be neglected. For example, by having a large number of clients interacting with a specific object, VV can rapidly grow, thus forcing the pruning. This

[6]http://blog.basho.com/2010/04/05/why-vector-clocks-are-hard/
[7]http://wiki.basho.com/Vector-Clocks.html

can lead to cases where clients have to solve false conflicts, which could be later resolved in a not so correct way, i.e., if the value that "wins", is in fact the one that would have been removed if pruning was not applied.

3.3 Common approaches to Causality tracking

In this section, we analyze the main approaches used currently in cloud databases, and discuss their properties and limitations.

An important aspect to consider when reasoning about the scalability of these approaches, is the existence of three different orders of magnitude at play: a small number of replica nodes for each key; a large number of server nodes; a huge number of clients, keys and issued operations. Thus, a scalable solution should avoid mechanisms that are linear with the highest magnitude and, if possible, even try to match the lowest scale.

When considering the system composed by the clients and the storage system, a large number of causal relations are established as the clients issue operations to the servers. Different key-value storage systems trace different sub-sets of these relations.

One simple way to formally illustrate this is to use *causal histories* [38]. Causal histories are simply described by sets of unique update event identifiers. These unique update identifiers can be generated by an unique node identifier and a monotonic integer counter. (We will use replica-based identifiers but client identifiers could be used as well. The crucial point is that identifiers have to be globally unique.) The partial order of causality can be precisely tracked by comparing these sets by set inclusion. Two histories are concurrent if neither include the other: $A \parallel B$ iff $A \nsubseteq B$ and $B \nsubseteq A$.

Consider a simple example, illustrated in Figure 3.4: Clients C_1, C_2, C_3 read the same state from synchronized replica nodes and do independent updates. In this simplified description we omit the keys, implicitly assuming they are the same, and only show the causal information that is committed to each replica node and respective versions, in the same order.

When client C_1 does its first PUT, replica node R_b will record the version associated with the causal history $\{b_1\}$ that includes the update identifier, b_1,

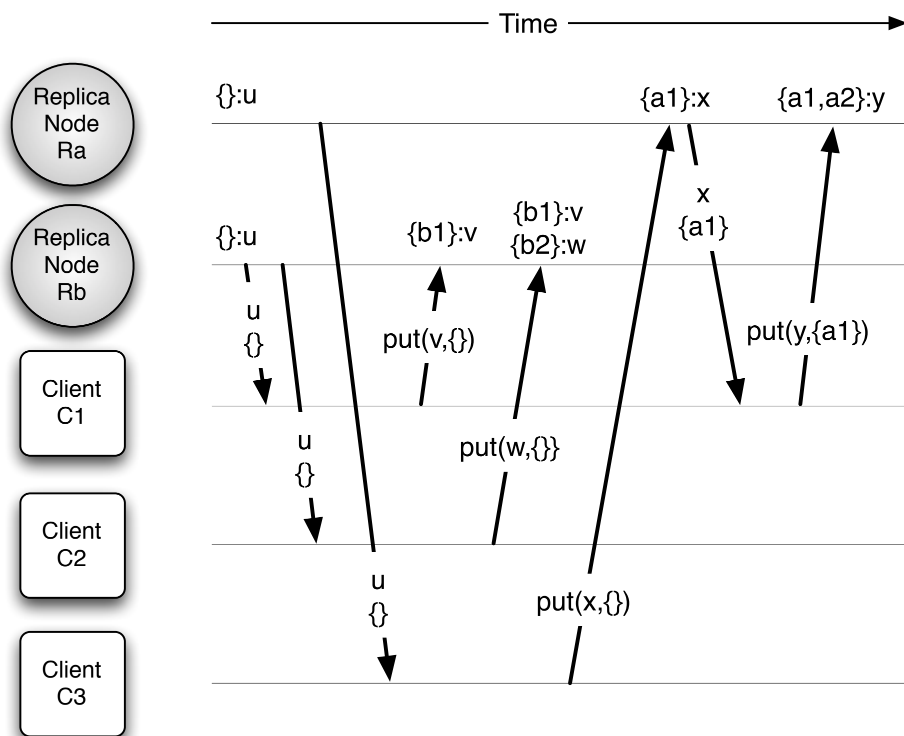


Figure 3.4: Three clients concurrently modifying the same key on two replica nodes. Causal histories.

and the history previously observed by C_1, {} in this case. When client C_2 does its PUT, the causal history associated with the new value will be $\{b_2\}$, which does not include b_1 because C_2 has not observed this version. Thus, R_b ends up with two concurrent versions, as stated by the causal histories. The second PUT from client C_1, handled by R_a, supplies a new version y together with its knowledge of causal history $\{a_1\}$, obtained from its last GET. Replica R_a records this update and adds a_2 to its corresponding causal history. Since $\{a_1\} \subset \{a_1, a_2\}$ the version y will syntactically dominate x and replace it in the committed state in R_a.

At the end of the run we have a value y in R_a than can be detected, by the causal histories, to be concurrent with the two concurrent values, v and w, stored on replica node R_b.

Although conceptually simple, causal histories are not adequate for use in practical systems, since they scale linearly with the number of updates. Next, we survey the mechanisms used in actual systems.

3.3.1 Causally compliant total order

One simple approach is to establish a total order among updates that is compliant with causal dependencies, and use this order to enforce a *last writer wins* policy. The simplest total order is obtained assuming that client clocks are well synchronized and applying real time clock order (simultaneous events are usually further ordered over process ids). In this approach, replica nodes never store multiple versions and writes do not need to provide a get context.

Figure 3.5 depicts the same run used to illustrate the use of causal histories, but now using perfectly synchronized client clocks. One can observe that concurrent events are ordered by the clocks and that the total order is compliant with the causal order: If two values would have causal histories c and c' such that $c \subset c'$ then the real time clocks t and t' are such that $t < t'$. This can be verified, observing values x and y in the run.

The problem is that although causally we have a partial order with $\{a_1, a_2\} : y \parallel \{b_1\} : v \parallel \{b_2\} : w$, this approach ends up ordering all updates. The total order established is compliant with causality, but will order actions that are in

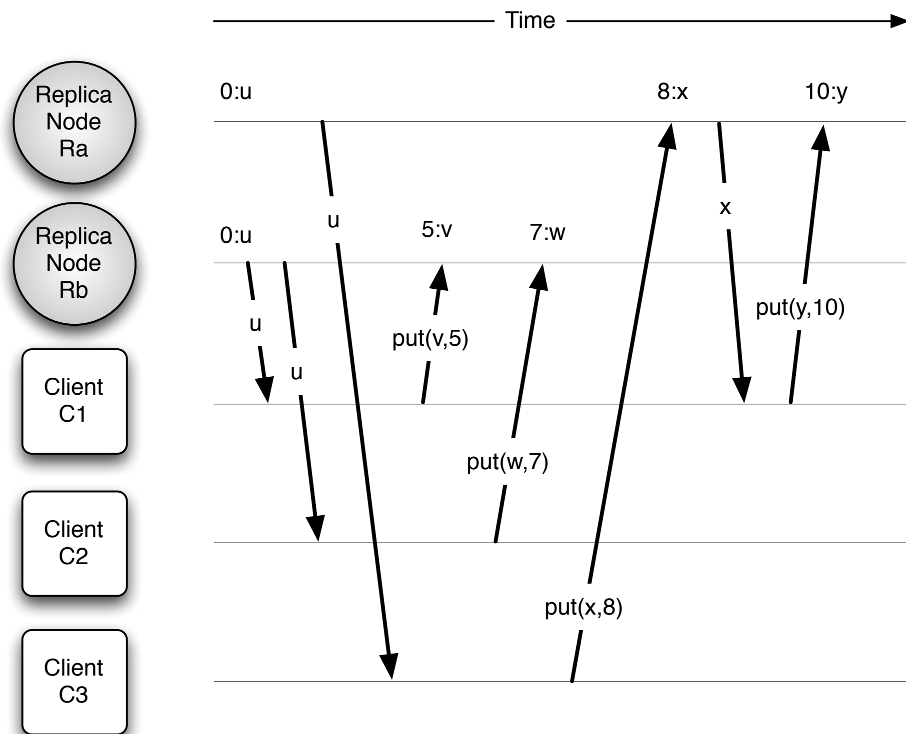


Figure 3.5: Three clients concurrently modifying the same key on two replica nodes. Perfectly synchronized real time clocks.

fact concurrent.

The approach based on client real time clocks is used in Cassandra v0.8.x [22]. Although version vectors are the main mechanism in Dynamo, real time clocks and *last writer wins* are also indicated as an alternative for application settings that tolerate lost updates.

An important drawback with real time is that if client clocks go out of sync the total order might no longer be compliant with causality. It is easy to see that a client with systematically delayed clock values will never see its updates committed and, conversely, that if a clock is always advanced its client updates will always win over concurrent ones.

An alternative approach that avoids real time clock synchronization and the potential anomalies when it fails, would be to use Lamport clocks [23], establishing a total order among updates that is compliant with causal dependencies. Again, this total order would not represent concurrent events and will loose updates.

3.3.2 Version vectors with per-server entry

A second approach is to track causality by using *version vectors* [30] with an entry per server replica node. In this case, each server maintains a version vector where each entry summarizes the sequence of updates it reflects. For example, a causal history of sequential replica events $\{a_1, a_2, b_1, b_2, c_1\}$ is summarized as $\{(a, 2), (b, 2), (c, 1)\}$. In traditional version vectors, for a fixed and ordered set of nodes, this can be further summarized as $[2, 2, 1]$, but this notation is not adequate for dynamic systems where the number of nodes can vary over time.

When the client executes a GET operation, it receives the version vector summarizing the causal history of events reflected in the version(s) received. Later, when the client executes a PUT, it sends the context on which the update is executed, i.e., the version vector previously received. The replica node increments its local counter to reflect the new update, and stores it in the entry of the received vector corresponding to its own identifier. It then checks if this new vector causally dominates any version currently stored, and discards any

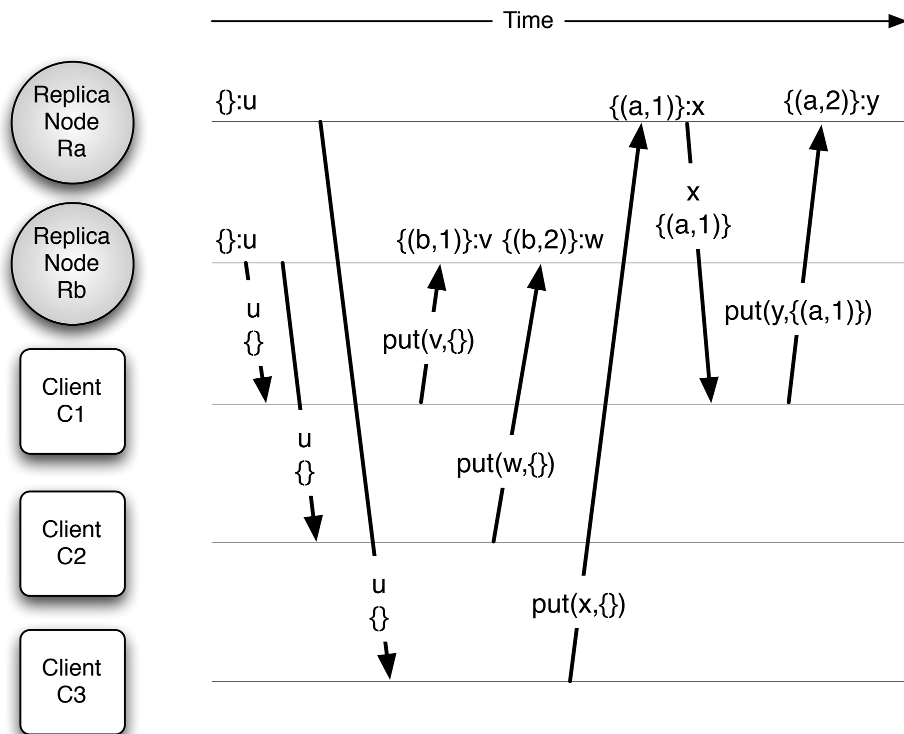


Figure 3.6: Three clients concurrently modifying the same key on two replica nodes. Per-server entries.

version made obsolete.

In this case, it is only possible to track the concurrency among updates that were received in different servers. Figure 3.6 depicts the same example run but now (as opposed to Figure 3.5) updates y and w are correctly detected to be concurrent, since $\{(a,2)\} \parallel \{(b,2)\}$. If a client GET collects these versions from the two replica nodes, this concurrency will be exposed and the client, receiving two versions, can submit back a version that dominates both updates.

However, this approach cannot track causality among updates submitted to the same server. In the example, when the update w from the client C_2 is submitted to replica R_b, it will get registered with the version vector $\{(b,2)\}$ and appear to dominate the previous committed value v with vector $\{(b,1)\}$. By comparing the version vectors of both updates, they will not be considered concurrent. This can be surprising considering the fact that if the second client had submitted the update to a different server it would be considered concurrent.

In practice a last writer wins policy was enforced with respect to concurrent updates handled in the same replica node, and, in this case, one concurrent update was lost. This linearization of concurrent updates, due to the use of less version vector entries than sources of concurrent activity, is formalized in *plausible clocks* [41]. The Dynamo system uses one entry per replica node and thus falls into this category.

The reason for the concurrent updates of the two clients submitted to the same server not being considered concurrent is consequence of the fact that the version vector associated with the second update does not correctly summarizes its causal history. In fact, the vector $\{(b,2)\}$ summarizes updates $\{b_1, b_2\}$, which includes the update v of the first client C_1. One can argue that the replica node R_b could instead verify that the new update is concurrent with its current version by checking that the version vector included in the operation does not dominate the version vector of the current version. In this case, the replica node could reject the update, implementing a conditional write semantics. This approach is used, e.g., in Coda [20] and in the CVS version control system (although not necessarily relying on version vectors).

However this goes against the usual policy of write availability [13], the norm in modern key-value stores.

3.3.3 Version vectors with per-client entry

We have seen that version vectors with one entry per replica node are not enough to track concurrent updates. One natural evolution is to track causality by using version vectors with one entry per client (if servers can also update the data, with server side scripts, an entry for each server should also be included in the version vector). Now the number of entries matches the number of concurrency sources and one no longer faces a plausible clocks setting.

As in the previous approach, updates are associated with a version vector. When a client executes a GET operation, it will receive the version vector associated with the version(s) that it reads. Later, when a client submits a new update, using PUT, the replica node will receive this vector and the client identity.

With per-client entries, the correct way to obtain the integer value used to register the update would be to have state-full clients that maintain a counter, increment it and provide it in each PUT operation, together with the context previously received. A version vector for the new version can be obtained from the context version vector by replacing the entry of the client by the given value. If we want to support the usual model with stateless clients, which only provide the context received by a GET and their unique identifier, we can do so if we have a *read your writes* semantics [39] (obtained, e.g., through read and write quorums), so that the most recent update by a given client is present in the context.

Otherwise, the server can, at most, try to infer the most recent update by that client, by using the maximum of the respective entry in the received context and all vectors at the server for that key. As a more recent update by that client can be stored in other server, this can still lead to lost updates.

In Figure 3.7, in the usual run, we illustrate this problem. Client C_1 when writing v in node R_b, has its updated registered as $(C_1, 1)$. Its later updates

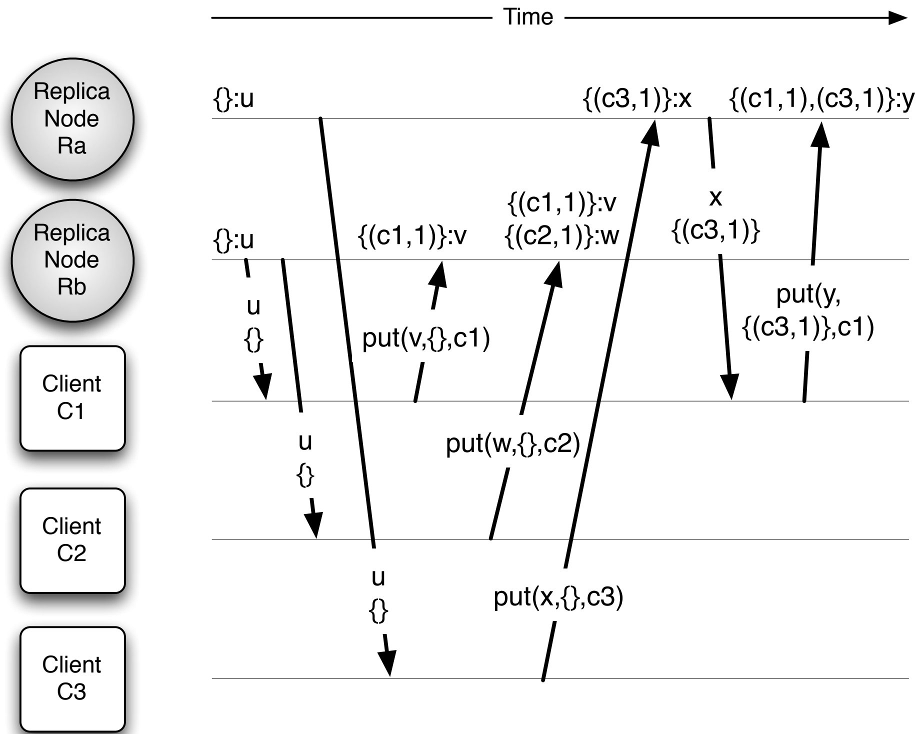


Figure 3.7: Three clients concurrently modifying the same key on two replica nodes. Per-client entries.

will get distinct, and monotonically increasing values as long as the client reads its last written version. However in this run, the client will issue a later update in replica R_a and this update will again be registered with $(C_1, 1)$. The consequence is that now value v seems to be dominated by version y, since $\{(C_1, 1)\} < \{(C_1, 1), (C_3, 1)\}$.

Although, when used with appropriate quorums that ensure *read your writes*, this approach can fully trace the causality among concurrent updates submitted by different clients, it has the obvious drawback of requiring one entry per client, which makes the size of the vectors now linear with the number of clients that perform PUT operations. Dotted version vectors, described in the next chapter, will provide reliable tracking while avoiding these scalability limitations and quorum formation restrictions.

3.4 Related Work and Summary

Systems such as Dynamo [13] use unsafe techniques to remove entries (pruning), that are expected not to be necessary, based on time.

Tracking causality through version vectors or vector clocks requires a space linear with the number of entities in the system, posing scalability problems for system with a large number of elements [12]. This problem is experienced in practice, for example, in cloud computing storage systems, as discussed in Section 3.3.

Besides the safe techniques previously mentioned to remove entries that are no longer needed, several other directions have been tackled to address this problem.

The Roam system [35] runs a consensus protocol to decrease, in all servers, the value of all entries of the version vector by a constant value. The system only keeps the entries that are larger than zero. The *dependency sequences* [32] mechanism assumes a scenario where dynamic, weakly-connected sets of entities (mobile hosts) communicate through designated proxy entities chosen from a stable, well-connected (mobile service stations). The mechanism maintains information about the causal predecessors of each event. It needs to take periodic global snapshots to prune discardable causality-tracking metadata.

In Depot [24], the version vector associated with each update only includes the entries that have changed since the previous update in the same node. However, each node still needs to maintain version vectors that include entries for all clients and servers.

Other storage systems explore the fact that they manage a large number of objects to maintain less information for each object. In Microsoft's WinFS [25], a base version vector for all objects is maintained for the file system, and each object maintains only the difference for the base in a concise version vector. In Cimbiosys [34], the authors suggest the use of the same technique in a peer-to-peer system. These systems, as they maintains only one entry per server, cannot generate two concurrent version vectors for tagging concurrent updates submitted to the same server from different

clients, as discussed in Section 3.3.

In a separate WinFS work [29], the authors describe a mechanism that allows encoding of non sequential causal histories by registering exceptions to the sequence; e.g. $\{a_1, a_2, b_1, c_1, c_2, c_3, c_7\}$ could be represented by $\{(a, 2), (b, 1), (c, 7)\}$ plus exceptions $\{c_4, c_5, c_6\}$.

Another direction is to use unsafe space-folding approaches that can reduce the storage and communication overhead at the expense of less accuracy of the causality relation captured by these mechanisms. Although devised as an alternative not to version vectors but to vector clocks, *plausible clocks* [41] propose techniques for condensing event counting from multiple replicas over the same vector entry. The resulting order does not contradict the causal precedence relation, but because counters are effectively shared between processes, some concurrent events will be perceived as causally related. In fact, the previously mentioned Lamport clocks [23], are a notable example of plausible clocks.

Additionally, another approach trading off less accuracy of causality-tracking for better scalability is the *hash history* mechanism [19]. It provides a directed graph not of update operations, but of version hashes over the state of each replica. Although independent of the number of replicas in the system, the storage overhead grows linearly with the number of updates. In order to minimize this problem, it truncates the histories, pruning the oldest hashes based on loosely synchronized clocks. Use of hashes, however, can only guarantee statistical correctness, and pruning may cause incorrect perception of concurrency.

Chapter 4

Dotted Version Vectors

4.1 A Kernel for Eventual Consistency

We have seen that, as soon as clients can perform concurrent updates managed by a single replica node, several concurrent versions may have to be kept in that node. These version sets are returned by a get operation, and their clocks are supplied as the context in a put operation.

In this section we argue that the mechanics of a distributed key-value store, in terms of causality tracking, should be based on two core functions on the sets of logical clocks of replicas.

- **sync(S_1, S_2)**: takes two clock sets and returns a clock set. It returns a set of concurrent clocks, each belonging to one of the sets, and that together cover both sets while discarding obsolete knowledge;
- **update(S, S_r, r)**: takes a clock set (S, the context supplied by the client), the set of clocks in the replica node S_r, and the replica node id r, and returns a clock. This clock should: 1) dominate all clocks in S, 2) dominate only those clocks in the system that are already dominated by S and 3) not be dominated by any join of clocks in the system.

The function sync produces a set of concurrent clocks that describe the collective causal past in the parameters. It simply returns elements from the sets in the parameters, and it can have a general implementation, defined only

in terms of the partial order on clocks, regardless of their actual representation:

$$\begin{aligned} \text{sync}(S_1, S_2) &= \{x \in S_1 \mid \nexists y \in S_2 .\, x < y\} \cup \\ &\quad \{x \in S_2 \mid \nexists y \in S_1 .\, x < y\} \end{aligned}$$

The update operation can be more of a challenge because its constraints involve a global condition on the system, but it must be implemented without global knowledge. This is specially the case in dynamic systems, as described in [5], but here we have the challenge of how to avoid the use of client identifiers in clocks.

4.1.1 Using the kernel operations

A key-value store can now implement the operations it intends to make available to clients by using the kernel operations sync and update.

Operation get(k)

When a client asks some node P to perform a get of some key k (step 1 in Figure 4.1):

- P determines the set of replica nodes R for k;
- P asks to a subset of nodes in R for the value for that key. Depending on consistency levels, this subset may contain, for example, a single node or a quorum of nodes (step 2);
- P waits for the replies (step 3);
- P performs a reduce of the replies using the sync operation, and replies to the client (step 4).

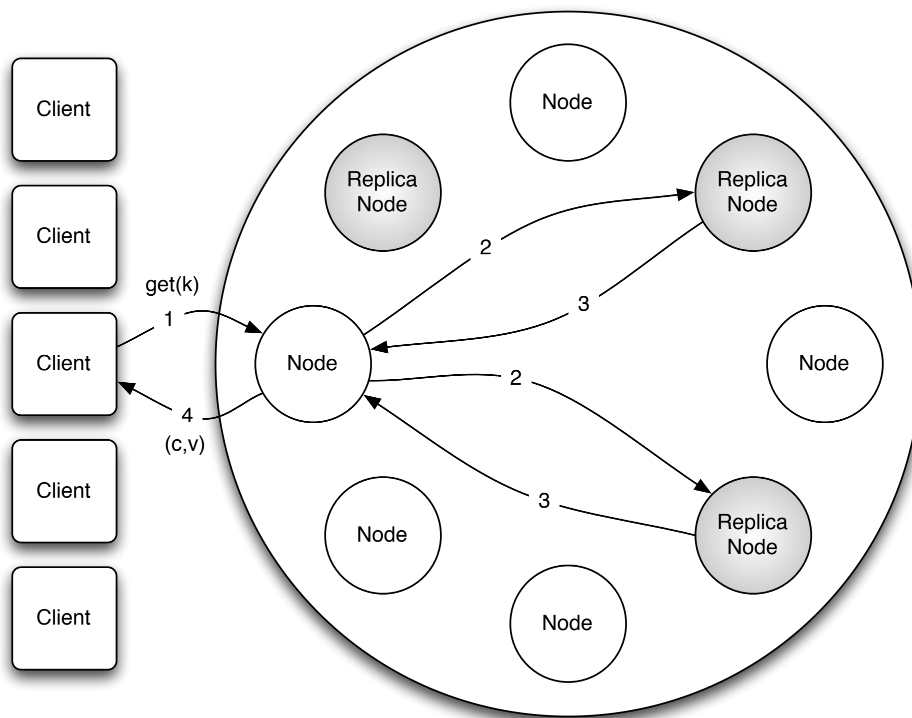


Figure 4.1: A get operation using DVV.

Operation put(k, v, S)

When a client asks some node P to perform a put for some key k, with a clock S (step 1 in Figure 4.2):

- P determines the set of replica nodes R for k;
- if P is a replica node for k, then P will coordinate the request; otherwise P will forward the request to some replica node for k, that will act as coordinator (step 2);
- the coordinator C performs an update operation with its local set of clocks S_C and its Id Id_C, resulting in a new clock value $u = \text{update}(S, S_C, Id_C)$;
- C performs a sync between u and the local set of clocks S_C, and stores the result of the sync $S'_C = \text{sync}(S_C, \{u\})$ (step 3);
- C sends S'_C to a subset of other nodes in R. Depending on consistency levels, this subset may, for example, be empty or contain a quorum of nodes (step 4);
- each of those nodes performs a sync between S'_C and the local set of clocks S_i, stores the result of the sync $S'_i = \text{sync}(S_i, S'_C)$, and acknowledges to C;
- C waits for the replies (if the subset is not empty) (step 5);
- C acknowledges to P (step 6), which in turn acknowledges to the client (or C acknowledges directly if that is possible) (step 7).

Partition-aware client library

We have described the steps used when a generic load balancer or client library is used. While any node can coordinate a get operation, for a put the coordinator must be a replica node. Using a partition-aware client library or load balancer will help in reducing the response time of a put by removing the forwarding hop.

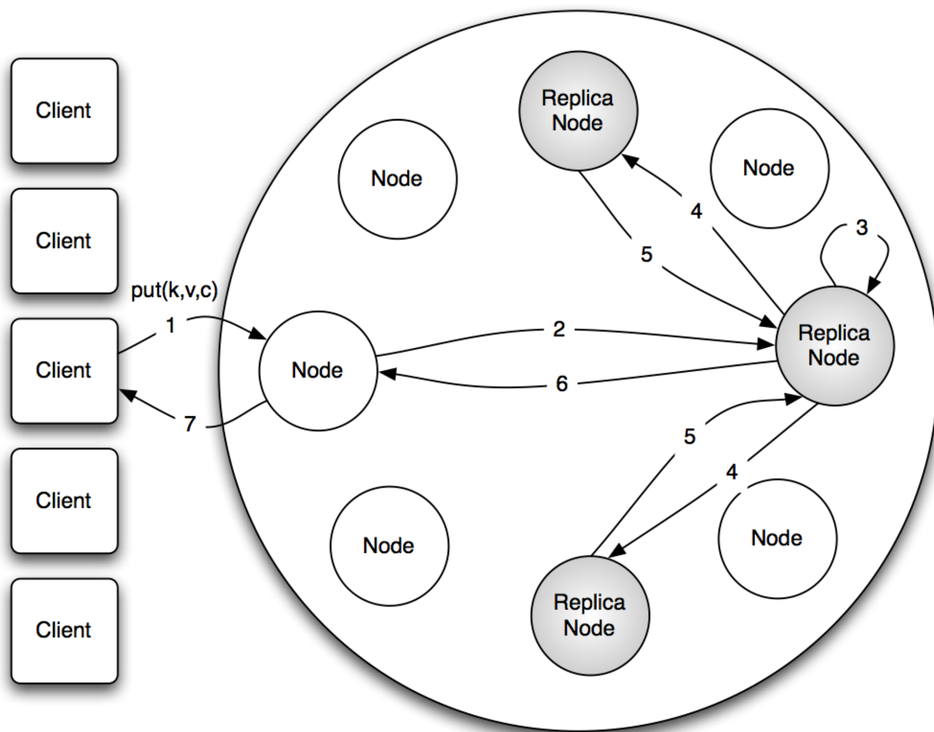


Figure 4.2: A put operation using DVV.

4.2 Dotted Version Vectors

We now present a concise and accurate representation for the clocks to be used as a substitute for the classic version vectors in key-value stores. The mechanism allows a lossless representation of causality (contrary to, e.g., Plausible Clocks) while only using server-based ids, and only a component per replica node, thus avoiding the space consumption explosion that occurs in id-per-client approaches.

While a version vector compresses causal histories by representing, for each component, all events in a range up to a given sequence number, we will be able to represent also individual events that fall outside such ranges.

As an example, a version vector $\{(a,2),(b,1),(c,3)\}$ represents the causal history:

$$\{a_1, a_2, b_1, c_1, c_2, c_3\}.$$

We will be able to represent a causal history like:

$$\{a_1, a_2, b_1, c_1, c_2, c_3, c_7\},$$

where event c_7 falls outside the range from 1 to 3.

DVV are able to represent, for any given component, both a range, and a range plus and individual event (a “dot”). We will see that a range plus a single event (as opposed to arbitrary sets) is enough for the scenario at hand.

4.2.1 Definition

A dotted version vector is a logical clock which consists of a mapping from identifiers to either integers or pairs of integers (m,n). For notational convenience we will use instead a triple (id,m,n) for such elements of the mapping. The events represented by a clock can be characterized by a semantic function from clocks (or sets of clocks) to causal histories:

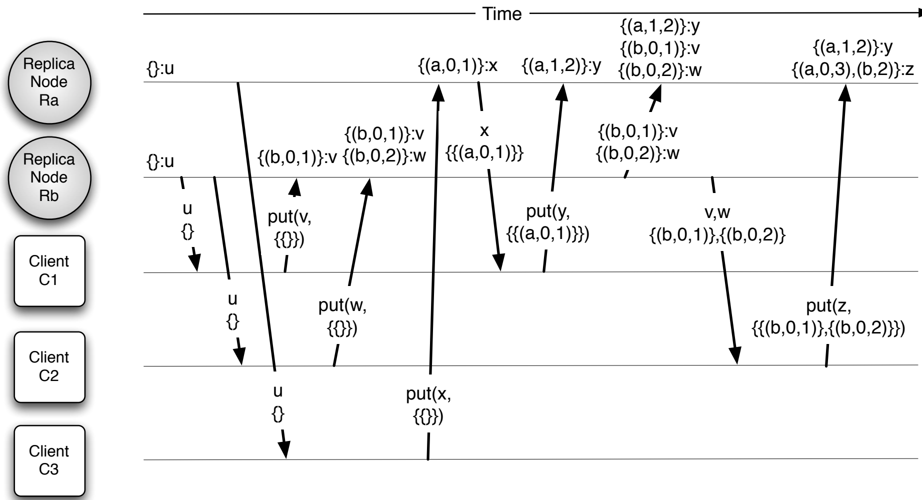


Figure 4.3: Three clients concurrently modifying the same key on two replica nodes. Dotted version vectors.

$$
\begin{aligned}
C[\![(r,m)]\!] &= \{r_i \mid 1 \leq i \leq m\}, \\
C[\![(r,m,n)]\!] &= \{r_i \mid 1 \leq i \leq m\} \cup \{r_n\}, \\
C[\![X]\!] &= \bigcup_{x \in X} C[\![x]\!].
\end{aligned}
$$

In a component (r, m, n) we will always have $n > m$.

With this definition, the causal history:

$$\{a_1, a_2, b_1, c_1, c_2, c_3, c_7\},$$

that cannot be represented in a version vector, will be represented by the dotted version vector $\{(a, 2), (b, 1), (c, 3, 7)\}$.

4.2.2 Partial order

The order on clocks can be defined, as usual, in terms of inclusion of causal histories; i.e.:

$$X \leq Y \iff C[\![X]\!] \subseteq C[\![Y]\!]$$

This can be computed by the function on mappings:

$$X \leq Y \iff \forall x \in X. \exists y \in Y. x \leq y,$$

where the order on individual components of the mapping is defined by the clauses:

$$\begin{array}{rcll} (r,m) & \leq & (r,m') & \text{if } m \leq m', \\ (r,m) & \leq & (r,m',n') & \text{if } m \leq m' \vee m = m'+1 = n', \\ (r,m,n) & \leq & (r,m') & \text{if } n \leq m', \\ (r,m,n) & \leq & (r,m',n') & \text{if } n \leq m' \vee (m \leq m' \wedge n = n'), \\ x & \not\leq & y & \text{otherwise.} \end{array}$$

This order allows concurrent clocks even using only a component from a single replica node. As an example:

$$\{(r,4)\} \parallel \{(r,3,5)\},$$

as they represent the causal histories:

$$\{r_1, r_2, r_3, r_4\} \parallel \{r_1, r_2, r_3, r_5\},$$

This situation will arise when $\{(r,4)\}$ is stored in a replica node and a client, which in the past has read some value and got the context $\{(r,3)\}$, now performs a put using this context. This situation is very common but cannot be handled with current mechanisms using server-based identifiers.

In Figure 4.3, we present our usual run using dotted version vectors. It can be seen that causality is accurately tracked, even tough per-server identifiers are used. We also extend the run so that replica node R_b decides to do some anti-entropy and sends state to node R_a that syncs its. Then, client C_2 does an interaction (with no affinity) where it reads from R_b and does an update z to R_a. We can see that, as expected, z will subsume both v and w, and is registered as concurrent to y.

4.2.3 Update function

An update registered on a replica node r containing the set of versions S_r, can have a reference definition in terms of causal histories using replica node ids plus sequence numbers to distinguish events, as:

$$\begin{aligned} \text{update}(S, S_r, r) &= \bigcup S \cup \{r_{n+1}\} \qquad \text{with} \\ & n = \max(\{0\} \cup \{x \mid r_x \in \bigcup S_r\}). \end{aligned}$$

To define the update function over dotted version vectors, we make use of some auxiliary functions. The ids function gives the set of identifiers in a clock or set of clocks:

$$\begin{aligned} \text{ids}((r, _)) &= r, \\ \text{ids}((r, _, _)) &= r, \\ \text{ids}(X) &= \{\text{ids}(x) \mid x \in X\}. \end{aligned}$$

The $\lceil _ \rceil_{_}$ function takes a clock or set of clocks and a replica node identifier and returns the maximum integer contained in the mapping from that identifier:

$$\begin{aligned} \lceil C \rceil_r &= \max(\{0\} \cup \{m \mid (r, m) \in C \lor (r, _, m) \in C\}), \\ \lceil S \rceil_r &= \max(\{0\} \cup \{\lceil C \rceil_r \mid C \in S\}). \end{aligned}$$

The update function can now be defined:

$$\begin{aligned} \text{update}(S, S_r, r) &= \{(i, \lceil S \rceil_i) \mid i \in \text{ids}(S) \land i \neq r\} \cup \\ & \quad \{(r, \lceil S \rceil_r, \lceil S_r \rceil_r + 1)\}. \end{aligned}$$

It can be seen by this definition that (given that sync does not generate new values) all clocks have exactly one component which is a triple; all the others are the same as in classic version vectors.

In the example of Figure 4.3, each put operation generates a new clock

for the new version. The first PUT from client C_1 generates the clock $(b, 0, 1)$, as no version exists previously in replica node R_b. The same for the first PUT from client C_3 on replica node R_a, which generates the clock $(a, 0, 1)$. A more interesting case is the first PUT from client C_2 on replica node R_b. In this case, as there is a version in replica node R_b with a clock that is not dominated by the context of the PUT, $\{\}$, the clock generated is $(b, 0, 2)$, encoding only the event b_2 of the causal history.

The second PUT from client C_1 exemplifies the situation where a client overwrites the version it has previously read. In this case, the generated clock is $(a, 1, 2)$, as the read context dominates (is equal in this case) to the clock of the version in the replica node.

The most complex example arises in the second PUT from client C_2. This example exemplifies the situation where a client receives two concurrent versions and creates a new version that superseeds the previous concurrent updates. In this case, the context of the PUT is $\{\{(b, 0, 1)\}, \{\{(b, 0, 2)\}\}$, and the clock generated in replica node R_a is $\{(a, 0, 3), (b, 2)\}$. The component $(b, 2)$ encodes the events b_1, b_2 of the causal history, which were represented in the context of the PUT. The component $(a, 0, 3)$ registers the new update event a_3 associated with this PUT operation.

4.2.4 Correctness

First we define the following predicate over clock sets:

$$\text{downset}(S) \iff \forall i \in \text{ids}(S).\, \forall\ 1 \leq n \leq \lceil S \rceil_i.\, i_n \in C[\![S]\!],$$

which is true for sets of clocks for which the union of the corresponding causal histories are downward closed sets, under the order over events $r_i \leq s_j \Leftrightarrow r = s \wedge i \leq j$. In other words, the predicate is true if, for each node r, the set contains all events from r up to some point in time.

The reason that makes it possible to have an accurate representation of causality using dotted clocks is that, as we will show, all sets of clocks, kept at replica nodes or returned to clients, are downsets.

Namely, as the clock set S sent from a client is a downset, the dotted

version vector u computed in an update $u = \text{update}(S, S_C, C)$ can represent accurately the appropriate causal history: the union of the causal histories corresponding to clocks in S plus a new event.

We now show that, in a given system containing replica nodes R, each $r \in R$ with a replica set S_r:

$$\forall r \in R. \text{downset}(S_r).$$

It is easy to see that if both X and Y are downsets, then $Z = \text{sync}(X, Y)$ will also be a downset. This means that if we assume that the invariant holds, then the values returned to clients are downsets. It also means that, if S'_C computed by the coordinator in a put operation is a downset, then the values stored in other replica nodes after receiving a store request from the coordinator will also be downsets.

Therefore, to prove that the invariant holds, the only interesting case is what happens in the coordinator in a put operation. The new clock set to be stored locally will be the result of an update followed by a sync: $u = \text{update}(S, S_C, C)$ and $S'_C = \text{sync}(S_C, \{u\})$. Assuming that S and S_C are downsets, S'_C will also be a downset because:

- although $\{u\}$ itself may not be a downset, for any identifier i other than C, the computed mapping $(i, \lceil S \rceil_i)$ represents a contiguous range of events starting from 1 for identifier i in the corresponding causal history of S; the sync between $\{u\}$ and S_C will therefore be a downset in what concerns these identifiers;

- for identifier C, as S_C represents all events from C up to $\lceil S_C \rceil_C$, and u contains only one more event with number $\lceil S_C \rceil_C + 1$, then S'_C represents a contiguous range starting from 1 for id C.

To summarize: even though $\{u\}$ is not necessarily a downset, syncing it with the clock set in the coordinator will result in a downset, as only a successor event is added.

4.3 Summary

We have introduced *dotted version vectors*, a novel solution for tracking causal dependencies among update events. The base idea of our solution is to add the capability to represent an extra isolated event over the downward closed causal history described by version vectors.

As an example, lets consider Replica B with a clock [(A,3),(B,2),(C,2)] as shown in figure 4.4. Now a client tries to write in replica B, a value with a clock that is outdated, lets say [(A,3),(B,1),(C,2)]. This results in a conflict in the replica, but by using VV with server IDs, we don't have a way to express concurrency between both values. As we can see in figure 4.5a, every increment possible to replica's ID, either create two identical clocks, or one of them will dominate the other. Using DVV, we can express this type of concurrency by adding a triple to the clock, as shown in figure 4.5b. This new clock says that the client update knows updates 1 and 3 in B. Since it does not know update 2 of replica B, it is in conflict with the other clock that knows updates 1 and 2. But server value also does not know update 3, so it is also in conflict with the client value.

DVV allow an accurate tracking of causality among updates executed by multiple clients, while using server-based identifiers. Their size is only linear with the number of servers that register the updates, being bounded by the degree of replication. When compared with previous accurate proposals that require client-based identifiers, linear with the number of clients, our solution is much more efficient, as the number of clients tends to be several orders of magnitude larger than the number of servers that register updates for a given data element.

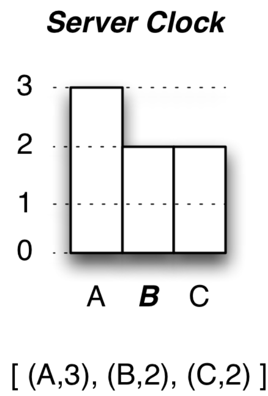
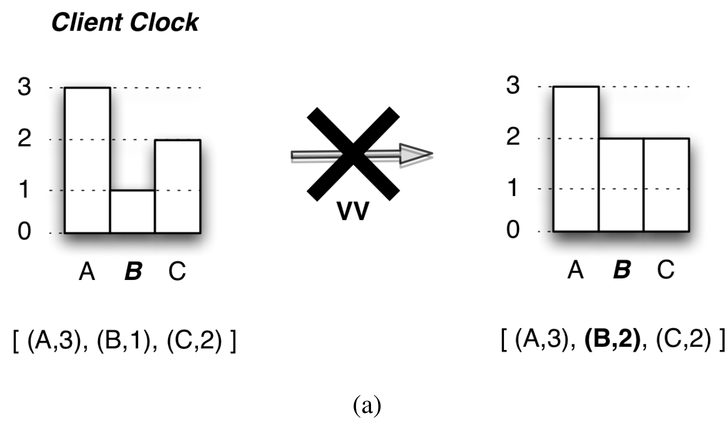
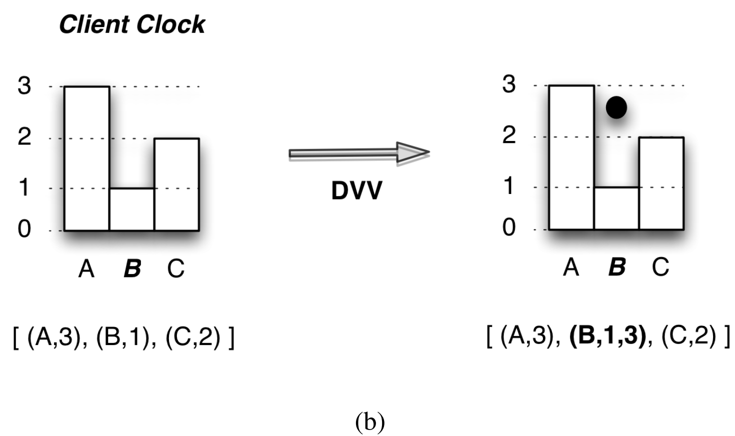


Figure 4.4: Server clock in replica B



(a)



(b)

Figure 4.5: Client Clock for a concurrent update in replica B, using (a) Version Vectors and (b) Dotted Version Vectors

Chapter 5

Implementing and Evaluating DVV

5.1 Implementation

Since Dynamo is not open source, Cassandra and Riak were the ones that were considered to implement DVV. In the end, Riak was chosen. First, it is written in Erlang, a fast prototyping language. Being in Erlang also gave the source code a relatively small size, at least, compared to Cassandra. Most importantly, Riak already uses Version Vectors (VV), which are structurally similar to DVV, thus it would be much easier to implement. Given that Cassandra still does not use or support logical clocks, the code refactoring and adaptation would be tremendously harder and more complex than in Riak's case. The smaller the changes to be made, the better, since it is less prone to code bugs, and the comparison between the original and DVV would be more straightforward. Finally, Riak uses VV with with per-client entry, which was the approach that originally motivated DVV.

5.1.1 Operation Put in Riak

When a client asks some node P to perform a put for some key K, with a clock S (step 1 in Figure 5.1):

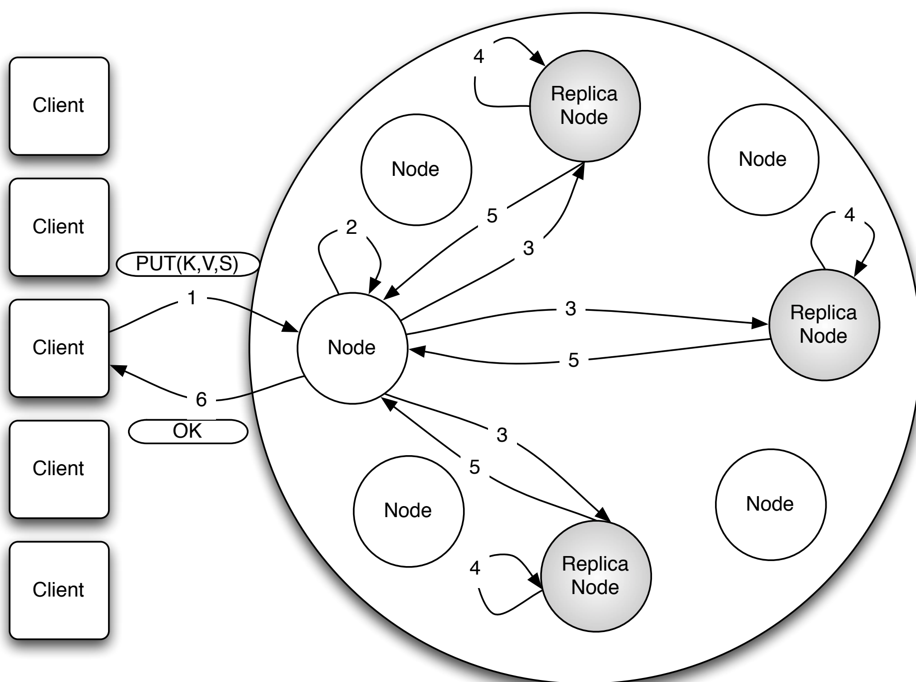


Figure 5.1: A put operation in Riak, using VV per-client entry.

- P updates S with the client Id, resulting in a new clock S' (step 2);
- P determines the set of replica nodes R for K, and forwards the request with S' to all of them (step 3);
- each of those nodes performs a synchronization between the receive S' and local clock, storing the result locally and after acknowledges to P (step 4);
- C waits for a subset of replies from replicas in R. Depending on consistency levels, this subset may, for example, be empty or contain a quorum of nodes (step 5);
- P acknowledges to the client (step 6).

5.1.2 Changing Operation Put to use DVV in Riak

The first thing to be done was an implementation of DVV in Erlang (see Appendix A). It is a single file, that contains all the required API functions and core DVV functions - *update* and *sync*. This file was placed in *Riak Core*, where are all the core mechanisms are implemented (e.g. Merkle Trees or Consistent Hashing). Then *Riak KV* - which is Riak's "business logic" - was modified to use DDV instead of VV. This required some key changes to reflect the core differences between them. One of them was eliminating *X-Riak-ClientId*, since we do not use the clients ID anymore to update our clock. Next are the main changes in specific files, regarding Riak's behavior.

riak_client Here we simply removed the line where the VV was previously incremented in a PUT operation.

riak_kv_put_fsm This file implements a Finite-State Machine (FSM), that encapsulates the PUT operation pipeline. Figure 5.2 represents a simplified representation of this FSM using VV, while figure 5.3 represents the FSM adapted to use DVV. Originally, in the "initialize" state, the node coordinating the request would send it to W number of replicas (remember that W is a parameter provided by the client to tell how many successful responses should

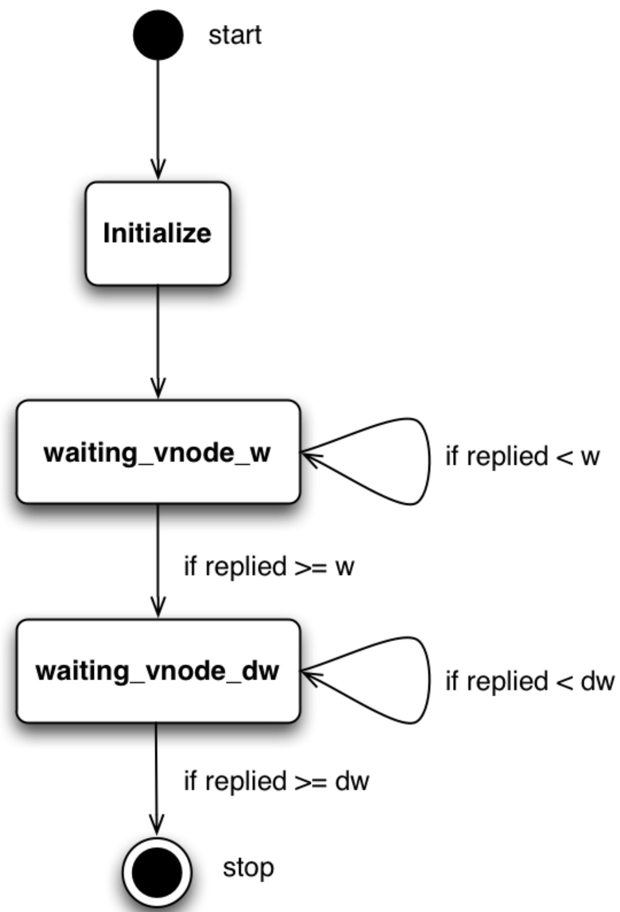


Figure 5.2: Finite State Machine Diagram for PUT operations in Riak, using VV.

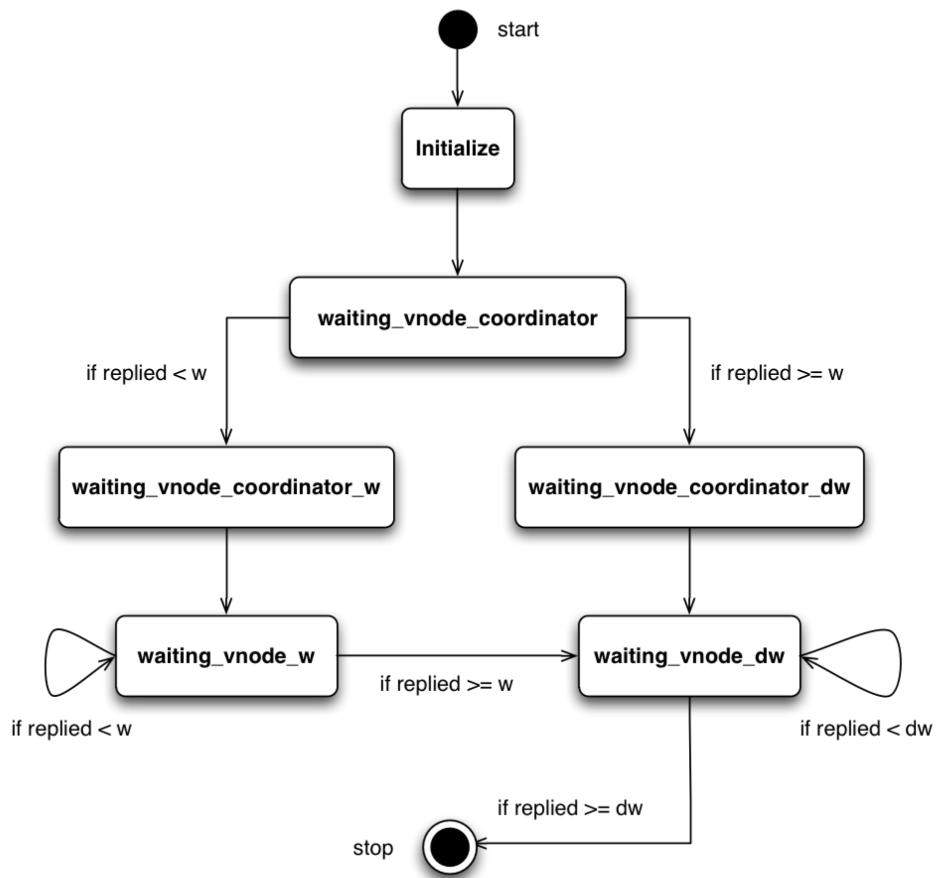


Figure 5.3: Finite State Machine Diagram for PUT operations in Riak, using DVV.

the server receive, before answering back to the client). But now with DVV, we first see if the current node is a replica, and if not, we forward the request to one. When the coordinator is a replica, we execute the PUT locally. We created new states in between “initialize" and “waiting_vnode_coordinator", to handle this local PUT operation. When the this is done, this replica provides the resulting object, with the updated clock and value(s) and sends them to the remaining replicas, where they synchronize their local object with the one that is sent now.

riak_kv_vnode This is where the local put is done. A provided “flag” tells if this node is the coordinator, and thus the one that should do the update/sync to the clock. If this flag is false, the node will only sync the local DVV with the received one. Otherwise, this node is the coordinator, therefore it will run the *update* function with both new and local DVV, and the node ID. Then run the sync function with that resulting DVV and local DVV. Finally, the coordinator sends the results to replicas, but this time not as coordinators, thus they only run the sync function between their local object and the object provided.

riak_object This file encapsulates a Riak object, containing things like metadata, data itself, the key, the clock, and so on. Before, an object only had one clock (one VV), even if there was more than one value (i.e. conflicting values). When conflicts were detected, both VV were merged so that there was only one new VV, which dominated both. This has an obvious disadvantage: the conflicting objects could only be resolved by a newer object. Even if by the gossip between replicas, we found that we could discard some of the conflicting values that were outdated, we could not.

Imagine that for some key, we already have in Riak two conflicting values A and B. If a client tries to put a value C in that key, and C conflicts with A, but is newer than C, then the resulting object would have the three values A, B and C. Even though B dominated C, which makes C dispensable, since we do not have a VV for every conflicting object (only an global one), we cannot know if B dominated C.

With DVV, we change this file so that each value has its own clock. In

the example above, the resulting object would only contain the A and C values. By discarding this redundant values, we are actually saving space and simplifying the complexity of operations, since we manipulate smaller data. It worth noting that this approach to have set of clocks instead of a merged clock, could also be applied to VV. Since DVV was designed to work with set of clocks, it was mandatory to change this aspect, which introduces a little more complexity to the code, but has the advantages stated above.

Others There are several other minor changes throughout the source code, but are not relevant for the discussion, since they are mainly code refactoring and handling set of clocks instead of one clock. Additionally, we made sure that this implementation still passed all unit tests correctly.

5.2 Evaluation

We evaluated this DVV implementation to see especially three things: (1) if the performance was affected; (2) if there was savings in metadata space (smaller clocks); (3) if false conflicts were really gone or diminished. Thus, benchmarks comparing the original version and the DVV version were ran. Additionally, other metrics other than latency were tracked, to provide some insight in what was happening and why. What follows is the benchmarking tool used, the setup for tests and the results performed.

5.2.1 Basho Bench

Basho Bench is a benchmarking tool created to conduct accurate and repeatable performance and stress tests. This tool outputs the throughput (i.e. total number of operations per second, over the duration of the test) and a range of latency metrics (i.e. 95th percentile, 99th percentile, 99.9th percentile, max, median and mean latency) for each operation. Basho Bench only requires one configuration file. The major parameters that were used are:

- Duration: 20 min (was more than enough to see performance stabilization, as seen for example in figure 5.5) ;

- Number of concurrent clients: 250 or 500;
- Requests per client: 1 or 3;
- Types of requests and their relative proportions: various (detailed later);
- Key Space: [0-50000];
- Key Access: Pareto distribution, i.e. 20% of the keys accessed 80% of the time;
- Value Size: fixed 1KB, 2KB or 5KB;
- Initial random seed was the same for all test, to ensure the same conditions;
- Number of replies (R and W for the read and write operations) = 2.

5.2.2 Setup

Seven machines in total were used, all in the same local network. A Riak cluster running with 6 similar machines, while another machine was simulating clients requests. The request rates and number of clients were chosen to try to prevent resource exhausting, since this would create unpredictable results. Resources were monitored to prevent saturation, namely CPU, disk I/O and network bandwidth, as can be seen on figure 5.4. We also used the default replication factor $n_val = 3$, also a $R = W = 2$ (R and W the size of the read and write quorum). Each test took 30 minutes and the same initial random seed was used for all test, to ensure the same conditions.

The following types of requests were issued from clients:

- GET: a simple read operation that returns the object of a given key;
- PUT: a *blind* write, where a value is written to a key, with no causal context supplied, i.e. without a clock. This operation will increase concurrency (create siblings) if the given key already exists, since an empty clock does not dominate any clock, thus always conflicting with the local node value;

- UPD: an update, that is expressed by a GET returning an object and a context (clock), followed by a 50 ms delay to simulate the latency between client and server, and finally a PUT that re-supplies the context and writes a new object, which supersedes the first one acquired in the GET. This operation reduces the possible concurrency (object with multiple values) that the GET brought.

We ran three different combinations of value's size, number of clients and number of request per client per second. And they are:

1. Scenario 1 (S1K): value size 1KB, 500 clients, each with 3 req/s (Table 5.1 and Table 5.4);

2. Scenario 2 (S2K): value size 2KB, 500 clients, each with 1 req/s (Table 5.2 and Table 5.5);

3. Scenario 3 (S5K): value size 5KB, 250 clients, each with 1 req/s (Table 5.3 and Table 5.6);

Each table has the latency of each GET and PUT operations, the mean size of clock metadata and the mean number of values per object (Siblings), i.e., 1 is the minimum, which represents no siblings at all, thus no concurrency. For each metric, we provide de ratio between DVV and VV. From DVV perspective, higher is worse, while lower is better (bolded in the table).

5.2.3 Generic Approach

In this test we adopt a generic and naive approach to the operations distribution. Given the three operations GET/PUT/UPD, four settings were evaluated: 1) 30%/10%/60% (S316), 2) 30%/60%/10% (S361), 3) 60%/10%/30% (S613) and 4) 60%/30%/10% (S631).

In all combinations we find that clock metadata size is always smaller in DVV, even with the (default size) pruning that occurs in VV. We also know that pruning is occurring, because the majority of tests reveal that there were

[1]This particular test was to heavy for the setup used, so the requests per second were decreased from three to one per client, to ensure that resources were not exhausted.

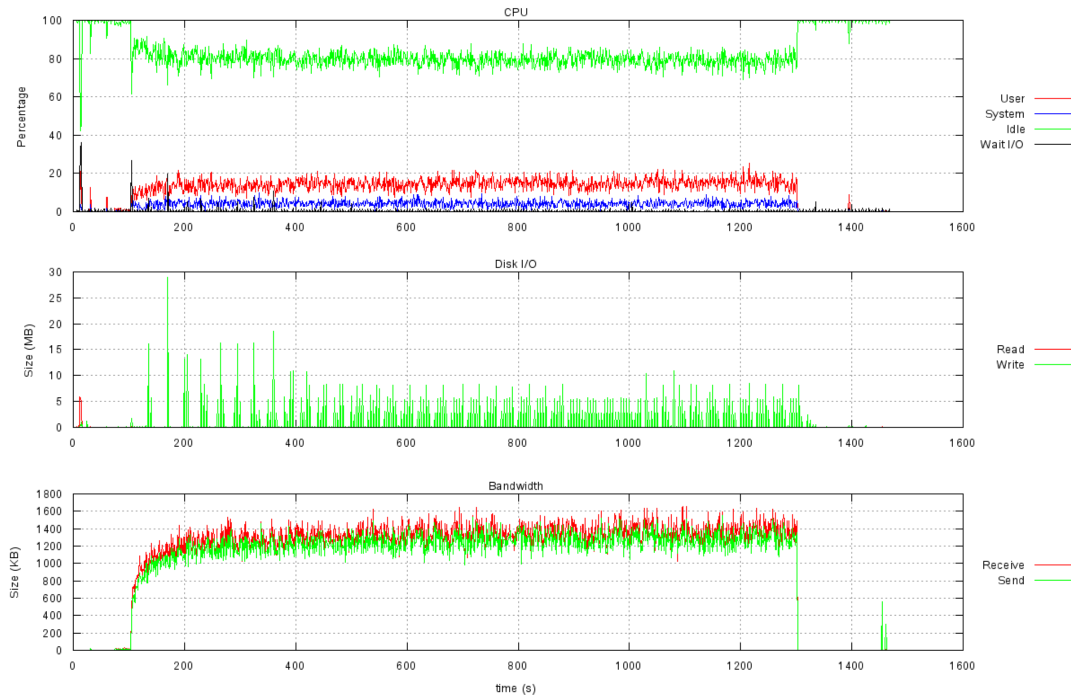


Figure 5.4: CPU, disk I/O and network bandwidth measurements in on machine, S2k, S316

Scenario 1 GET/PUT/UPD	Clock	GET Mean (ms)	GET Median (ms)	GET 95th (ms)	PUT Mean (ms)	PUT Median (ms)	PUT 95th (ms)	Meta (bytes)	Siblings (average)
60/30/10	VV	15.3	13.7	30.5	11.3	9.8	23.4	875	4.35
	DVV	12.4	10.1	28.9	13.8	11.6	28.8	228	3.61
	DVV/VC	**0.81**	**0.74**	**0.95**	1.22	1.18	1.23	**0.26**	**0.83**
30/60/10 [1]	VV	6.73	5.66	15.5	3.62	2.80	8.85	797	5.87
	DVV	12.0	8.14	35.2	14.0	10.3	38.2	312	5.50
	DVV/VC	1.78	1.44	2.27	3.88	3.70	4.32	**0.39**	**0.94**
60/10/30	VV	7.65	6.60	15.9	5.71	5.16	10.1	790	1.34
	DVV	3.16	2.89	5.25	4.31	4.06	6.27	127	1.31
	DVV/VC	**0.41**	**0.44**	**0.33**	**0.76**	**0.79**	**0.62**	**0.16**	**0.98**
30/10/60	VV	10.4	9.07	21.6	7.48	6.74	13.8	859	1.20
	DVV	3.45	3.14	5.83	4.56	4.29	6.59	123	1.16
	DVV/VC	**0.33**	**0.35**	**0.27**	**0.61**	**0.64**	**0.48**	**0.14**	**0.97**

Table 5.1: Scenario 1 (S1K) with generic approach.

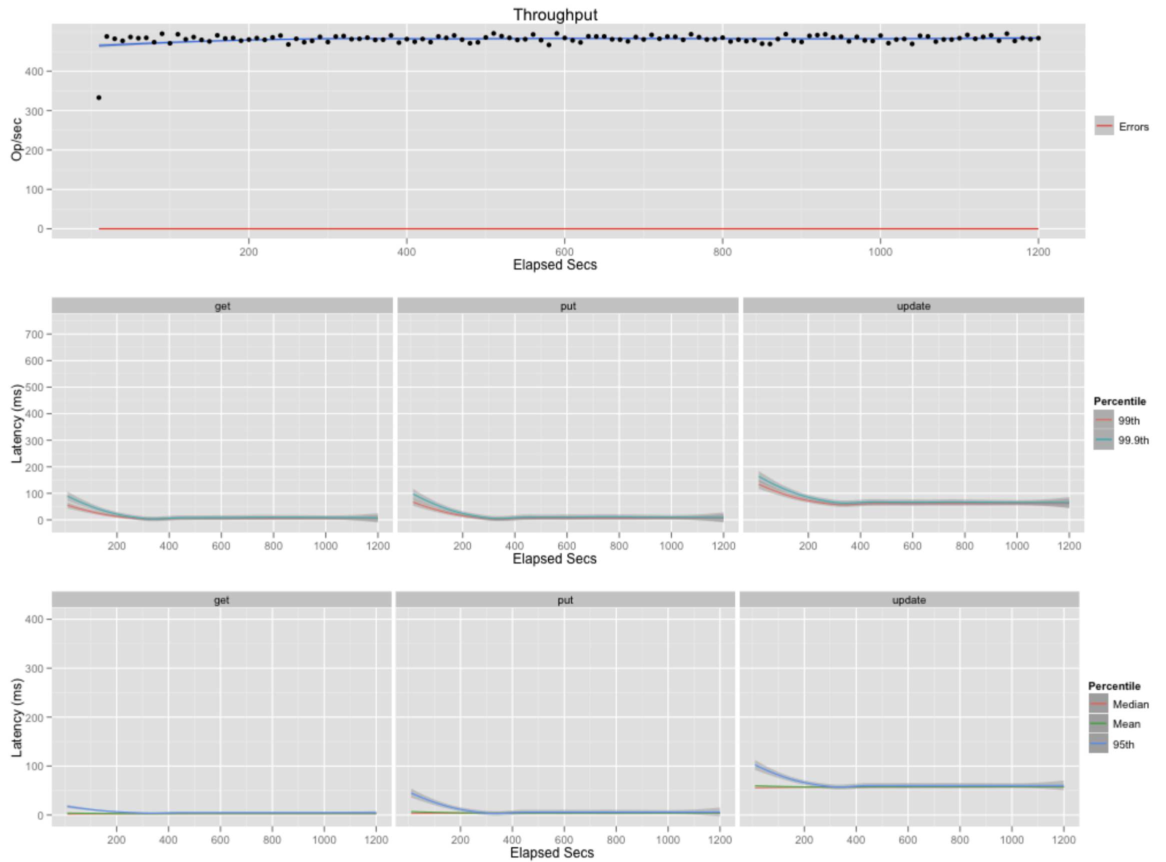


Figure 5.5: Performance summary for DVV, S361, S2K

Scenario 2	Clock	GET			PUT			Meta	Siblings
		Mean	Median	95th	Mean	Median	95th		
GET/PUT/UPD		(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(bytes)	(average)
60/30/10	VV	5.82	4.94	13.4	3.95	3.08	9.61	538	3.19
	DVV	6.15	4.78	16.0	7.66	6.20	18.5	192	3.10
	DVV/VC	1.06	**0.97**	1.19	1.94	2.01	1.93	**0.36**	**0.97**
30/60/10	VV	9.84	7.74	25.0	5.51	3.75	15.8	705	5.32
	DVV	15.6	11.7	42.1	17.8	14.0	44.9	293	5.16
	DVV/VC	1.58	1.52	1.68	3.22	3.73	2.84	**0.42**	**0.97**
60/10/30	VV	3.47	3.21	6.86	2.93	2.54	6.11	424	1.44
	DVV	2.77	2.53	5.15	4.06	3.75	7.54	114	1.28
	DVV/VC	**0.80**	**0.79**	**0.75**	1.38	1.48	1.23	**0.27**	**0.89**
30/10/60	VV	3.87	3.60	7.52	3.22	2.83	6.71	546	1.15
	DVV	2.90	2.62	5.48	4.21	3.87	8.30	113	1.15
	DVV/VC	**0.75**	**0.73**	**0.73**	1.31	1.37	1.24	**0.21**	1.00

Table 5.2: Scenario 2 (S2K) with generic approach.

Scenario 3	Clock	GET			PUT				
		Mean	Median	95th	Mean	Median	95th	Meta	Siblings
GET/PUT/UPD		(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(bytes)	(average)
60/30/10	VV	9.57	7.60	24.3	6.08	4.24	16.1	449	3.06
	DVV	9.97	7.76	26.1	11.4	9.22	27.1	183	2.98
	DVV/VC	1.04	1.02	1.07	1.88	2.17	1.68	**0.41**	**0.98**
30/60/10	VV	18.1	13.8	48.0	9.77	6.01	29.5	625	4.95
	DVV	22.7	17.7	60.5	26.3	21.0	65.5	277	4.89
	DVV/VC	1.26	1.28	1.26	2.69	3.50	2.22	**0.44**	**0.99**
60/10/30	VV	4.30	3.95	8.55	3.73	3.23	7.19	337	1.32
	DVV	3.91	3.48	7.88	5.43	4.94	9.95	110	1.27
	DVV/VC	**0.91**	**0.88**	**0.92**	1.46	1.53	1.38	**0.32**	**0.97**
30/10/60	VV	4.72	4.35	9.15	3.98	3.50	7.64	458	1.20
	DVV	3.92	3.56	7.42	5.50	5.08	9.97	110	1.15
	DVV/VC	**0.83**	**0.82**	**0.81**	1.38	1.45	1.30	**0.24**	**0.95**

Table 5.3: Scenario 3 (S5K) with generic approach.

more siblings in the VV case, when compared with same DVV run. This means that, the major difference between the siblings average results from false conflicts created by pruning. Therefore, we can conclude that, indeed DVV is smaller than VV, while preventing false conflicts from happening. Even if the default size pruning was lowered, metadata would be smaller, but false conflicts rate would rise.

In terms of performance, things are a bit more complex. First, in all S361 scenarios, DVV performance was always worse than VV. Scenarios S361 are write-heavy, and the majority of those writes are new objects, which could generate siblings if that key already had an object. If we look at the siblings average in this case, we see an absurd degree of concurrency happening. Since only a UPD can resolve and simplify concurrency, it is obvious why this is the use case where most concurrency occurs. This in a rather extreme and unrealistic use case, but shows that DVV suffers in fact from the problem of the extra hop to a replica, and also suffers from the fact that it has to send all the possible siblings of the coordinator replica, to the other replicas. Since the number of siblings is extremely high, this fact is augmented in the performance problems shown here. Almost the same can be said to scenarios S631.

Scenarios S316 and S613 are a bit more positive in terms of performance, and only strengthens the previous conclusions. Being read-heavy, or in other

point view write-light, we can see that performance in some cases for DVV, is actually better than VV. Having less siblings, and factoring the smaller clock metadata, on average operations transfer smaller data. In particular GET operations, which do not differ in procedure between DVV and VV. Thus we can see that a read-heavy use case can speed up its read process by having less to read. The write speed can also benefit from DVV when the data is small, like the scenarios where values have 1KB in size. Since the major thing that worsens DVV performance is transferring objects in writes, having a small value relatively to the clock metadata size, can be enough to actually gain performance even in write operations. As we can see, when the data size is bigger like the 2KB or 5KB scenario, the savings on metadata and number of siblings are not enough for writes performance to be better than VV case.

5.2.4 TPC-W Approach

The TPC-W benchmark from the Transaction Processing Council (TPC)[1] is a transactional Web benchmark designed to evaluate e-commerce systems. We did not actually run TPC-W, but we used their workload to give us a more realistic approach. TPC-W uses three different workload mixes, differing in the ratio of read-only to read-write interactions; the browsing mix contains 95% read-only interactions (S95), the shopping mix 80% (S80), and the ordering mix 50% (S50).

We also did not use PUT operations, because we assume that every client follows good behavior, thus reading before every right, so that it can (try to) update the most recent value. These read-write interactions are of course, already defined by the UPD operation.
It is more realistic this way, than doing so many blind puts (PUT), like the previous section. So, we only have GET and UPD operations in these following tests.

Results

The first thing we can see is that concurrency is very low, as it would be expected in a more realistic setting (DeCandia et al. [13] reveals their concur-

Scenario 1	Clock	GET			UPD			Meta	Siblings
		Mean (ms)	Median (ms)	95th (ms)	Mean (ms)	Median (ms)	95th (ms)	(bytes)	(average)
GET/UPD									
95/5	VV	2.15	2.15	3.63	55.0	54.9	57.7	159	1.00
	DVV	2.01	2.00	3.49	55.7	55.7	58.8	89	1.00
	DVV/VC	**0.94**	**0.93**	**0.96**	1.01	1.01	1.02	**0.56**	1.00
80/20	VV	4.23	3.68	8.51	58.6	58.0	64.5	459	1.00
	DVV	2.48	2.32	3.85	56.5	56.4	59.9	106	1.00
	DVV/VC	**0.59**	**0.63**	**0.45**	**0.97**	**0.97**	**0.93**	**0.23**	1.00
50/50	VV	7.70	6.61	16.2	64.4	63.5	74.0	682	1.01
	DVV	2.95	2.68	4.76	57.4	57.1	60.0	113	1.00
	DVV/VC	**0.38**	**0.41**	**0.29**	**0.89**	**0.90**	**0.81**	**0.17**	1.00

Table 5.4: Scenario 1 (S1k) with TPC-W approach.

Scenario 2	Clock	GET			UPD			Meta	Siblings
		Mean (ms)	Median (ms)	95th (ms)	Mean (ms)	Median (ms)	95th (ms)	(bytes)	(average)
GET/UPD									
95/5	VV	1.73	1.84	2.92	54.5	54.6	56.8	74.5	1.00
	DVV	1.72	1.82	2.99	55.3	55.3	58.4	71.8	1.00
	DVV/VC	1.00	**0.99**	1.03	1.01	1.01	1.03	**0.96**	1.00
80/20	VV	2.37	2.39	4.09	55.4	55.3	58.7	202	1.00
	DVV	2.23	2.25	3.77	56.2	56.2	60.1	93.0	1.00
	DVV/VC	**0.94**	**0.94**	**0.92**	1.01	1.02	1.03	**0.46**	1.00
50/50	VV	3.26	3.05	6.42	56.9	56.5	62.5	411	1.00
	DVV	2.59	2.44	4.57	56.8	56.6	61.6	104	1.00
	DVV/VC	**0.80**	**0.80**	**0.71**	1.00	1.00	**0.99**	**0.25**	1.00

Table 5.5: Scenario 2 (S2k) with TPC-W approach.

Scenario 3	Clock	GET			UPD			Meta	Siblings
		Mean (ms)	Median (ms)	95th (ms)	Mean (ms)	Median (ms)	95th (ms)	(bytes)	(average)
GET/UPD									
95/5	VV	1.90	1.81	3.97	55.5	55.3	59.0	52.7	1.00
	DVV	1.89	1.80	3.95	56.2	55.8	60.8	63.2	1.00
	DVV/VC	1.00	1.00	1.00	1.01	1.01	1.03	1.20	1.00
80/20	VV	2.84	2.99	5.00	56.8	56.8	61.0	117	1.00
	DVV	2.77	2.90	4.94	57.7	57.7	62.8	82.0	1.00
	DVV/VC	**0.98**	**0.97**	**0.99**	1.02	1.02	1.03	**0.70**	1.00
50/50	VV	3.48	3.49	5.92	57.8	57.7	62.6	233	1.00
	DVV	3.24	3.27	5.41	58.6	58.6	64.0	95.8	1.00
	DVV/VC	**0.93**	**0.94**	**0.91**	1.01	1.02	1.02	**0.41**	1.00

Table 5.6: Scenario 3 (S5k) with TPC-W approach.

rency rates, and they are very similar to these tests). Therefore, it was only expected that performance would be better compared to the previous tests. Metadata size continue to be much smaller in most cases. In one case, metadata from DVV was actually bigger than VV. Not surprisingly, that case was the one with fewer clients and fewer updates, therefore VV did not actually grow so much (probably with more time, VV would tend to get bigger).

Reading performance was always better, or pretty even between both mechanisms. Updates, although helped by including the reading latencies, were pretty good when the value's size was 1KB. As it been said, given that VV metadata in S50 scenarios rose significantly, compared to DVV metadata, in S1k cases, this was most relevant to write performance. In contrast, when value's size was in average 5KB, updates did not fair nearly as well. Nevertheless, we can see an improvement over the previous tests, which indicate that in fact, having less conflicts helps DVV case, since it does not have to do those extra steps where siblings are sent to replicas.

5.3 Summary

We implemented DVV in Riak with success. It did take code changes, some of which required a more deep refactoring than previously expected. Having done that, we ran benchmarks to see where DVV was doing good and bad. So, lets resume the advantages and disadvantages of using DVV instead of VV.

First, the advantages:

- **Simplify API**: since DVV uses the node's internal 160-bit ID, there is no need for clients to provide IDs, thus simplifying the API and avoiding potential ID collisions;

- **Save space**: DVV are bounded to the number of replicas, instead of the number of clients that have ever done a PUT. Since there is a small and stable number of replicas, the size of DVV would be much smaller than traditional VV;

- **Eliminates false conflicts**: clock pruning does not cause data loss, but it does cause false conflicts, where data that could be discard, is viewed as conflicting. Using DVV, the clock is bound to the number of replicas, therefore pruning is not necessary, thus eliminating false conflicts;
- **Enable partial reconciliation**: DVV were designed to manage not one clock, but a set of clocks. That is to say that when we have conflicts, we save siblings with their own clock. This enables that futures writes can partially resolve a subset of the conflicts in our set of siblings. On the other side, Riak's implementation does not keep individual clocks for each sibling; instead, it merges conflicting clocks into one, without incrementing any value. Then, the only way to resolve these conflicts, is to write a value with a clock that dominates the merged clock;
- **Does not require Quorum**: imagine using consistency of 1 (R=W=1) with VV. The same client can update the same key in different replicas, and they would have the same clock, even tough they were different updates. Since DVV uses server side IDs, this is not a problem since updates in different replicas will always create different clock. Thus, DVV requires less consistency in advance.

And now the disadvantages:

- **More complex write pipeline**: when a *non-replica* node receives a PUT request, it must forward it to a replica node. This overhead can be considerable if the transferred data is big. Which is even worse if the replica is not in the same network as the non-replica. Another thing that may affect negatively the performance is the fact that clock update and synchronization has to first be done in the coordinating replica, and then sent to the remaining replicas, whereas in VV the object goes directly to all replicas simultaneously. This is made worse when in the DVV case, the resulting object of the coordinating replica has siblings, which means that all siblings will be transferred to the remaining replicas. With VV, only the client object is sent to replicas.
- **Serialize operations on same node and same data**: given that DVV uses server side IDs, a little serialization is necessary in the server, i.e.,

when two writes to the same replica and to the same key are concurrent, the second can only start, after the first has written locally. It does not require to wait for the other replicas replies. Since Riak already serializes each type request per vnode, with its own database, this was a non-issue for this implementation.

Chapter 6

Conclusion

This dissertation introduced *Dotted Version Vectors (DVV)*, a novel solution for tracking causal dependencies among update events. DVV allow to accurately track causality among updates executed by multiple clients using information that is only linear with the number of servers that register these updates, i.e., they are bounded by the degree of replication. When compared with previously proposed safe solutions that require information linear with the number of clients, this solution is much more efficient, as the number of clients tends to be several orders of magnitude larger than the number of servers that register updates for a given data element.

Riak implementation of Version Vectors (VV) resorts to pruning, when the number of entries exceeds some threshold, and consequently does not reliably represent concurrency, introducing false conflicts.

DVV solution is simple and practical: we have modified Riak to use it. Evaluation showed a significant reduction in metadata size, and also the elimination of false conflicts.
Performance wise, results showed that for very small data being saved, DVV excels. However, when data's average size becomes bigger, the operations latency suffers due to the more complex write pipeline. Even so, this is somewhat compensated by the scalability capacity of DVV, concerning the number of clients interacting. The bigger the number of clients, the better DVV perform in comparison to VV. Since DVV are independent of the number of clients, it stays relatively stable, no matter how many clients there are. In

contrast, VV scales linearly with clients, so it resorts to pruning the clock, which ultimately causes false conflicts observed in our tests. In general, the bigger the number of clients interacting, and the more read-heavy workload, the better DVV would compare to VV. In contrast, the bigger the object size, the worse DVV would compare to VV.
Another relevant benefit is a simplification of Riak's API, avoiding the need to generate and transmit globally unique client identifiers.

6.1 Future Work

There are two main reasons that worsens DVV performance, that should be address. One is the extra hop for non-replica nodes, that could be avoid if we use a partition-aware client library, or load balancer that knows which replica to communicate, thus reducing the response time. Another solution is to allow non-replicas to update the clock, instead of forwarding the request to a replica. The downside is the size of DVV would tend to scale linearly with the number of servers, instead of replicas. Even though, being upper bound to number of servers is much more scalable than scaling with number of clients.

The other problem is when the coordinator's local write results in conflicts, thus creating an object with siblings. Having to transfer all the siblings to the others replicas can create an negative overhead in network bandwidth.

To mitigate this problem, we could adopt an optimistic approach, where after a local write by the coordinator, we would only transfer to other replicas the original client value, and the new updated and synchronized local clock. If other replicas upon receiving this update realized that they missed data that should have been transferred (this would be checked using DVV), they would later synchronize through gossip or via a read repair. However, this approach could create a bit of an overhead on the server side, but that remains to be tested.

Another option would be to explore the lesser constraints that DVV puts on consistency levels. More specifically, the tests realized using Riak, did so with consistency level of quorum, since VV using entries per-client requires it. By using entries per-replica, DVV do not requires so many replicas an-

swers (quorum), before returning to the client a positive response.
Taking advantage of this, if we strengthen our consistency level requirement to quorum, maybe the DVV write pipeline *could* be simplified. More specifically, maybe we could send the client update to all replicas, without waiting for the coordinator local write to finish.

Finally, we intend to study how the underlying idea of DVV can be applied to other mechanisms to track causality, such as Interval Tree Clocks (ITC), in order to better handle membership changes in the set of nodes. Recovering lost Ids in clocks, and creating unique Ids without global coordination, are two main properties of ITC that could be used to further strengthen DVV approach.

Bibliography

[1] Yussuf Abu and Shaaban Jane Hillston. A tpc-w-based tool for benchmarking e-commerce programming technologies. In *In Proc. 18th UK Performance Engineering Workshop*, pages 10–11.

[2] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Rachid Guerraoui, editor, *DISC*, volume 3274 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2004. ISBN 3-540-23306-7. URL `http://dblp.uni-trier.de/db/conf/wdag/disc2004.html#AlmeidaAB04`.

[3] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps " decentralized version vectors. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 544–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1585-1. URL `http://dl.acm.org/citation.cfm?id=850928.851897`.

[4] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Improving on version stamps. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'07, pages 1025–1031, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76889-0, 978-3-540-76889-0. URL `http://dl.acm.org/citation.cfm?id=1780453.1780489`.

[5] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 259–274, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-92220-9.

[6] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[7] João Barreto. Information sharing in mobile networks: a survey on replication strategies. Technical report, 2003. URL http://www.gsd.inesc-id.pt/~{}jpbarreto/bib/MobReplicationSurvey_Barreto03.pdf.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: http://doi.acm.org/10.1145/343477.343502. URL http://doi.acm.org/10.1145/343477.343502.

[9] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3 (1):63–75, 1985. URL http://dblp.uni-trier.de/db/journals/tocs/tocs3.html#ChandyL85.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

[11] Bernadette Charron-Bost. Combinatorics and geometry of consistent cuts: Application to concurrency theory. In Jean-Claude Bermond and Michel Raynal, editors, *WDAG*, volume 392 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 1989. ISBN 3-540-51687-5. URL http://dblp.uni-trier.de/db/conf/wdag/wdag89.html#Charron-Bost89.

[12] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39:11–16, July 1991. ISSN 0020-0190. doi: 10.1016/0020-0190(91)90055-M. URL http://dl.acm.org/citation.cfm?id=117603.117606.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1323293.1294281. URL `http://doi.acm.org/10.1145/1323293.1294281`.

[14] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, 1989.

[15] Victor Francisco Fonte. *Causality tracking in dynamic distributed systems*. PhD thesis, University of Minho, Braga, Portugal, January 2009.

[16] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/564585.564601. URL `http://doi.acm.org/10.1145/564585.564601`.

[17] Richard A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5:5–4, 1992.

[18] Naohiro Hayashibara, Xavier Dfago, Rami Yared, and Takuya Katayama. The phi accrual failure detector. *Reliable Distributed Systems, IEEE Symposium on*, 0:66–78, 2004. ISSN 1060-9857. doi: http://doi.ieeecomputersociety.org/10.1109/RELDIS.2004.1353004.

[19] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23nd International Conference on Distributed Computing Systems (ICDCS)*, pages 670–677. IEEE Computer Society, 2003.

[20] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, US, 1991.

[21] Avinash Lakshman and Prashant Malik. Cassandra: a structured storage system on a P2P network. In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA*, page 47. ACM, 2009. ISBN 978-1-60558-606-9. URL http://doi.acm.org/10.1145/1583991.1584009.

[22] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010. ISSN 0163-5980.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[24] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI 2010*, October 2010.

[25] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005. ISBN 3-540-29163-6.

[26] Keith Marzullo and Gil Neiger. Detection of global state predicates. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *WDAG*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer, 1991. ISBN 3-540-55236-7. URL http://dblp.uni-trier.de/db/conf/wdag/wdag91.html#MarzulloN91.

[27] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987. URL http://dx.doi.org/10.1007/BF01782776.

[28] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[29] Lev Novik, Irena Hudis, Douglas B. Terry, Sanjay Anand, Vivek Jhaveri, Ashish Shah, and Yunxin Wu. Peer-to-peer replication in winFS. Technical Report MSR-TR-2006-78, Microsoft Research (MSR), June 2006.

[30] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9:240–247, May 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.236733. URL `http://dx.doi.org/10.1109/TSE.1983.236733`.

[31] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Sixteen ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[32] Ravi Prakash and Mukesh Singhal. Dependency sequences and hierarchical clocks: Efficient alternatives to vector clocks for mobile computing systems. *Wireless Networks*, pages 349–360, 1997. also presented in Mobicom96.

[33] Dan Pritchett. BASE: An acid alternative. *ACM Queue*, 6(3):48–55, 2008. URL `http://doi.acm.org/10.1145/1394127.1394128`.

[34] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.

[35] David Howard Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, 1998. UCLA-CSD-970044.

[36] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, 30:49–56, February 1996.

[37] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37:42–81, March 2005. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1057977.1057980. URL http://doi.acm.org/10.1145/1057977.1057980.

[38] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7:149–174, March 1994. ISSN 0178-2770. doi: 10.1007/BF02277859. URL http://dl.acm.org/citation.cfm?id=1081582.1081586.

[39] Douglas Terry, Alan Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Inormation Systems*, Austin, TX, US, September 1994.

[40] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4:180–209, June 1979. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/320071.320076. URL http://doi.acm.org/10.1145/320071.320076.

[41] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.

[42] M. Wiesmann, F. Pedone, A. Schiper, Kem B., and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS '00, pages 464–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0601-1. URL http://dl.acm.org/citation.cfm?id=850927.851782.

[43] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. *Operating Systems Review*, 20(1):57–66, 1986. URL http://dblp.uni-trier.de/db/journals/sigops/sigops20.html#WuuB86.

Appendix A

Implementation of Dotted Version Vectors in Erlang

```erlang
%% @doc A simple Erlang implementation of dotted version vectors.
%%
%% @reference N. M. Pregui\c{c}a, C. Baquero, P. S. Almeida, V. Fonte,
%% and R. Gon\c{c}alves(2010). "Dotted version vectors: Logical clocks for
%% optimistic replication". CoRR, abs/1011.5808


-module(dottedvv).

-author('Ricardo Goncalves <tome@di.uminho.pt>').

-export([fresh/0,descends/2,sync/2,get_counter/2,get_timestamp/2,
  update/3,all_nodes/1,equal/2,increment/2,merge/1,get_max_counter/2]).


-type dottedvv() :: [dvv_entry()].
-type dvv_entry() :: {dvv_id(), {counterM(), timestamp()}} |
           {dvv_id(), {counterM(), counterN(), timestamp()}} .

```

```
20 % ids can have any term() as a name, but they must differ
      from each other.
21 -type    dvv_id() :: term().
22 -type    counterM() :: integer().
23 -type    counterN() :: integer().
24 -type    timestamp() :: integer().
25
26 % @doc Create a brand new dottedvv.
27 -spec fresh() -> dottedvv().
28 fresh() -> [].
29
30
31
32 % @doc Return true if Va is a direct descendant of Vb, else
      false -- remember, a dottedvv is its own descendant!
33 -spec descends(Va :: [dottedvv()], Vb :: [dottedvv()]) ->
      boolean()
34         ; (Va :: dottedvv(), Vb :: dottedvv()) -> boolean().
35 descends(A,B) ->
36   A2 = lists:flatten(A),
37   B2 = lists:flatten(B),
38   case (A2 =:= B2) and (A2 =:= []) of
39     true -> false;
40     false -> descends1(A,B)
41   end.
42 descends1(_, []) -> true;
43 descends1(S1=[{_,_}|_],S2) -> descends1([S1],S2);
44 descends1(S1,S2=[{_,_}|_]) -> descends1(S1,[S2]);
45 descends1(S1, S2) ->
46   descends2(S1,S2).
47
48 descends2(_,[]) ->
49   true;
50 descends2([],_) ->
51   false;
52 descends2([H|T],S) ->
53   case belongsDelete(H,S) of
54     false -> false;
55     {true,S2} -> descends2(T,S2)
56   end.
57
```

```
58 belongsDelete(_,[]) ->
59   false;
60 belongsDelete(E,[H|T]) ->
61   case descends_aux(lists:flatten(E),lists:flatten(H))
       andalso (equal(E,H)==false) of
62     true -> {true,T};
63     false -> belongsDelete(E,T)
64   end.
65
66
67 descends_aux(_, []) ->
68     % all dottedvvs descend from the empty dottedvv
69     true;
70 descends_aux(Va, [{IdB, {CtrB, _}}|Vbtail]) ->
71     CtrA =
72   case get_counter(IdB, Va) of
73       undefined -> false;
74       {CAm, CAn} -> if CAn == CAm+1 ->
75                CAn;
76             true ->
77                {CAm, CAn}
78            end;
79       CA -> CA
80   end,
81     case CtrA of
82   false ->
83     false;
84   {CtrAm, _CtrAn} ->
85       if CtrB > CtrAm->
86         false;
87     true ->
88         descends_aux(Va, Vbtail)
89       end;
90   _ ->
91       if
92     CtrA < CtrB ->
93         false;
94     true ->
95         descends_aux(Va, Vbtail)
96       end
97     end;
```

```
98  descends_aux(Va, [{IdB, {CtrBm, CtrBn, T}}|Vbtail]) when
        CtrBn == CtrBm+1 ->
99     descends_aux(Va, ([{IdB, {CtrBn, T}}]++Vbtail));
100 descends_aux(Va, [{IdB, {CtrBm, CtrBn, _T}}|Vbtail]) ->
101      CtrA =
102    case get_counter(IdB, Va) of
103        undefined -> false;
104        {CAm, CAn} -> if CAn == CAm+1 ->
105                CAn;
106             true ->
107                {CAm, CAn}
108           end;
109        CA -> CA
110    end,
111
112      case CtrA of
113    false ->
114      false;
115    {CtrAm, CtrAn} ->
116      if  (CtrBn == CtrAn) and (CtrAm < CtrBm) ->
117          false;
118        (CtrBn == CtrAn) and (CtrAm >= CtrBm) ->
119            descends_aux(Va, Vbtail);
120        CtrBm == CtrAm -> %% CtrBn =/= CtrAn
121          false;
122          CtrBn > CtrAm ->
123            false;
124        true ->
125            descends_aux(Va, Vbtail)
126        end;
127    _ ->
128        if
129      CtrA < CtrBm ->
130          false;
131      CtrA < CtrBn ->
132          false;
133      true ->
134          descends_aux(Va, Vbtail)
135        end
136      end.
137
```

```
138
139 merge(S) -> merge2(lists:flatten(S)).
140 merge2([]) -> [];
141 merge2(S) ->
142   S2 = sets:from_list(S),
143   S3 = sets:to_list(S2),
144   Old = [[SB || SB <- S3, descends([SA],[SB])] || SA <- S3],
145     Old2 = flatten(Old),
146     VOld = sets:from_list(Old2),
147     VRes = sets:subtract(S2, VOld),
148   sets:to_list(VRes).
149
150
151
152 %%%%%%%%%%%%%%%%%%%%%%%% sync(S1,S2) -> S
153 % @doc  Takes two clock sets and returns a clock set.
154 %   It returns a set of concurrent clocks,
155 %   each belonging to one of the sets, and that
156 %   together cover both sets while discarding obsolete
      knowledge.
157 -spec sync(Set1 :: [dottedvv()], Set2 :: [dottedvv()]) -> [
      dottedvv()].
158 sync(S1=[{_,_}|_],S2) -> sync([S1],S2);
159 sync(S1,S2=[{_,_}|_]) -> sync(S1,[S2]);
160 sync(S1,S2) ->
161
162 sync2(S1,S2).
163 sync2([], []) -> [];
164 sync2([], S2) -> S2;
165 sync2(S1, []) -> S1;
166 sync2(Set1, Set2) ->
167   S = Set1 ++ Set2,
168   SU = [ sets:to_list(sets:from_list(B)) || B <- S],
169   Old = [[S2 || S2 <- SU,
170     descends(S1,S2)]
171                 || S1 <- SU],
172     Old2 = flatten(Old),
173     VOld = sets:from_list(Old2),
174     VS = sets:from_list(SU),
175     VRes = sets:subtract(VS, VOld),
176   sets:to_list(VRes).
```

```
177
178
179 % @private
180 flatten([]) -> [];
181 flatten([H|T]) -> H ++ flatten(T).
182
183 % @doc Increment DottedVV at Node.
184 -spec increment(Id :: dvv_id(), Dottedvv :: dottedvv()) ->
        dottedvv().
185 increment(Id, Dottedvv) ->
186   update(Dottedvv, Dottedvv, Id).
187
188 %%%%%%%%%%%%%%%%%%%%%%%% update(Sc,Sr,r) -> S
189 % @doc Update dottedvv at Node.
190 -spec update(Sc :: [dottedvv()], Sr :: [dottedvv()], IDr ::
        dvv_id()) -> dottedvv().
191 update(A,B,Id) -> update2(lists:flatten(A),lists:flatten(B),
        Id).
192 update2(Sc, Sr, IDr) ->
193   MaxC = get_max_counter(IDr, Sc),
194   MaxR = get_max_counter(IDr, Sr),
195   case (MaxC == MaxR) of
196     true ->
197       [ {Id, get_max_counter_time(Id, Sc)} || Id <- all_nodes
            (Sc) , Id =/= IDr] ++
198       [ {IDr, {MaxR + 1 , timestamp()}} ];
199     false ->
200       [ {Id, get_max_counter_time(Id, Sc)} || Id <- all_nodes
            (Sc) , Id =/= IDr] ++
201       [ {IDr, {MaxC, MaxR + 1 , timestamp()}} ]
202     end.
203
204
205 %%%%%%%%%%%%%%%%%%%%%%%% ids(X) -> [id]
206 % @doc Return the list of all nodes that have ever
        incremented dottedvv.
207 -spec all_nodes(Dottedvv :: dottedvv()) -> [dvv_id()]
208       ; ([Dottedvv :: dottedvv()]) -> [dvv_id()].
209
210 all_nodes([]) -> [];
211 all_nodes({X,_}) -> [X];
```

```
212 all_nodes(Dottedvv=[{_,_}|_]) ->
213     sets:to_list(sets:from_list([X || {X,{_,_}} <- Dottedvv]
          ++ [X || {X,{_,_,_}} <- Dottedvv])).
214
215
216
217 %%%%%%%%%%%%%%%%%%%%%%%% [S]r -> max(Sr)
218 % @private
219 -spec get_max_counter(Id :: dvv_id(), [Dottedvv :: dottedvv()
     ]) -> counterM().
220 get_max_counter(A,B) -> get_max_counter_aux(A,B,0).
221 get_max_counter_aux(_, [], Acc) ->
222   Acc;
223 get_max_counter_aux(Id, [{Id2,{_M,N,_T}}|Tail], Acc) when Id
     =:= Id2 ->
224   case N < Acc of
225     true -> get_max_counter_aux(Id,Tail,Acc);
226     false -> get_max_counter_aux(Id,Tail,N)
227   end;
228 get_max_counter_aux(Id, [{Id2,{M,_T}}|Tail], Acc) when Id =:=
      Id2 ->
229   case M < Acc of
230     true -> get_max_counter_aux(Id,Tail,Acc);
231     false -> get_max_counter_aux(Id,Tail,M)
232   end;
233 get_max_counter_aux(Id, [_|Tail], Acc) ->
234   get_max_counter_aux(Id,Tail,Acc).
235
236
237
238 get_max_counter_time(A,B) -> get_max_counter_time_aux(A,B,{0,
     timestamp()}).
239 get_max_counter_time_aux(_, [], Acc) ->
240   Acc;
241 get_max_counter_time_aux(Id, [{Id2,{_M,N,T}}|Tail], Acc={N2,
     _T2}) when Id =:= Id2 ->
242   case N < N2 of
243     true -> get_max_counter_time_aux(Id,Tail,Acc);
244     false -> get_max_counter_time_aux(Id,Tail,{N,T})
245   end;
```

```
246 get_max_counter_time_aux(Id, [{Id2,{M,T}}|Tail], Acc={N2,_T2}
      ) when Id =:= Id2 ->
247   case M < N2 of
248     true -> get_max_counter_time_aux(Id,Tail,Acc);
249     false -> get_max_counter_time_aux(Id,Tail,{M,T})
250   end;
251 get_max_counter_time_aux(Id, [_|Tail], Acc) ->
252   get_max_counter_time_aux(Id,Tail,Acc).
253
254
255
256 % @doc Get the counter value in dottedvv set from Node.
257 -spec get_counter(Id :: dvv_id(), Dottedvv :: dottedvv()) ->
      counterM() | {counterM(),counterN()} | undefined.
258 get_counter(Id, Dottedvv) ->
259     case proplists:get_value(Id, Dottedvv) of
260   {M, _} -> M;
261   {M, N, _} -> {M,N};
262   undefined -> undefined
263     end.
264
265
266 % @doc Get the timestamp value in a dottedvv set from Node.
267 -spec get_timestamp(Node :: dvv_id(), Dottedvv :: dottedvv())
      -> timestamp() | undefined.
268 get_timestamp(Node, Dottedvv) ->
269     case proplists:get_value(Node, Dottedvv) of
270   {_, _, TS} -> TS;
271   {_, TS} -> TS;
272   undefined -> undefined
273     end.
274
275 % @private
276 timestamp() ->
277     calendar:datetime_to_gregorian_seconds(erlang:
          universaltime()).
278
279
280
281
282 % @doc Compares two dottedvvs for equality.
```

```
283 -spec equal(Dottedvv :: [dottedvv()], Dottedvv :: [dottedvv()
       ]) -> boolean().
284 equal([],[]) -> true;
285 equal(S1=[{_,_}|_],S2) -> equal([S1],S2);
286 equal(S1,S2=[{_,_}|_]) -> equal(S1,[S2]);
287 equal(S1,S2) ->
288 equal2(S1,S2).
289
290
291 equal2([],[]) ->
292   true;
293 equal2([],_) ->
294   false;
295 equal2(_,[]) ->
296   false;
297 equal2([H|T],S) ->
298   case belongsDelete2(H,S) of
299     false -> false;
300     {true,S2} -> equal2(T,S2)
301   end.
302
303
304 belongsDelete2(_,[]) ->
305   false;
306 belongsDelete2(E,[H|T]) ->
307   case equal3(E,H) of
308     true -> {true,T};
309     false -> belongsDelete2(E,T)
310   end.
311
312 equal3(VA,VB) ->
313     VSet1 = sets:from_list(VA),
314     VSet2 = sets:from_list(VB),
315     case sets:size(sets:subtract(VSet1,VSet2)) > 0 of
316         true -> false;
317         false ->
318             case sets:size(sets:subtract(VSet2,VSet1)) > 0 of
319                 true -> false;
320                 false -> true
321             end
322     end.
```