Physical and Logical Time

Carlos Baquero
Universidade do Minho

MIEI SDLE 2020

Universidade do Minho

Why should we care?

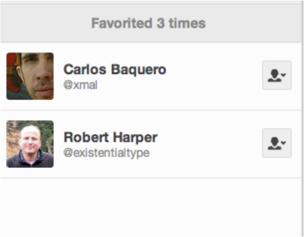
Everything is distributed

"Distributed systems once were the territory of computer science Ph.D.s and software architects tucked off in a corner somewhere. That's no longer the case."

(2014 http://radar.oreilly.com/2014/05/everything-is-distributed)

Anomalies in Distributed Systems

- A buzz, you have a new message, but message is not there yet
- Remove the boss from a group, post to it and she sees posting
- Read a value but cannot overwrite it
- Assorted inconsistencies



Can't we use time(stamps) to fix it?

Time

The problem with time is that

Distributed Computing

March 10, 2015
Volume 13, issue 3

PDF

There is No Now

Problems with simultaneity in distributed systems

Justin Sheehy

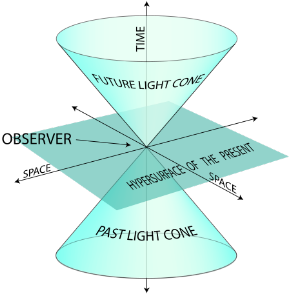
"Now."

The time elapsed between when I wrote that word and when you read it was at least a couple of weeks. That kind of delay is one that we take for granted and don't even think about in written media.

(2015 http://queue.acm.org/detail.cfm?id=2745385)

There is no single universal reference frame

Light speed is causality speed



Some properties of Time

- Time needs memory (*since it is countable change*)
- Time is local (*rate of change slows with acceleration*)
- Time synchronization is harder at distance
- Time is a bad proxy for causality

Wall-clock time
Clock drift

- Clock drift refers to drift between measured time and a reference time for the same measurement unit.
- Quartz clocks: From 10^{-6} to 10^{-8} seconds per second.
- 10^{-6} seconds per second means 1 one second each 11.6 days.
- Atomic clocks: About 10^{-13} seconds per second.
- A second is defined in terms of transitions of Cesium-133. Coordinated Universal Time (UTC), inserts or removes seconds to sync with orbital time.

External vs internal synchronization

External synchronization states a precision with respect to an authoritative reference. Internal synchronization states a precision among two nodes (if one is authoritative it is also external)

External
: For a band $D > 0$ and UTC source S we have $\mid S(t) - C_i(t) \mid < D$

Internal
: For a band $D > 0$ we have $\mid C_j(t) - C_i(t) \mid < D$

An externally synchronized system at band D is necessarily also at $2D$ internal synchronization.

Monotonicity

Some uses (e.g. `make`) require time monotonicity ($t' > t \Rightarrow C(t') > C(t)$) on wall-clock adjusting.
Correcting advanced clocks can be obtained by reducing the time rate until aimed synchrony is reached.

Synchronization: Synchronous system

- Knowing the transit time *trans* and receiving origin time t, one could set to $t + trans$
- However *trans* can vary between *tmin* and *tmax*
- Using $t + tmin$ or $t + tmax$ the uncertainty is $u = tmax - tmin$.
- But, using $t + (tmin + tmax)/2$ the uncertainty becomes $u/2$.

Asynchronous

Now transit time varies between *tmin* and $+\infty$
How to update the clocks?

Synchronization: Synchronous system

- Knowing the transit time *trans* and receiving origin time t, one could set to $t + trans$
- However *trans* can vary between *tmin* and *tmax*
- Using $t + tmin$ or $t + tmax$ the uncertainty is $u = tmax - tmin$.
- But, using $t + (tmin + tmax)/2$ the uncertainty becomes $u/2$.

Asynchronous

Now transit time varies between *tmin* and $+\infty$
How to update the clocks?

Synchronization: Synchronous system

- Knowing the transit time $trans$ and receiving origin time t, one could set to $t + trans$
- However $trans$ can vary between $tmin$ and $tmax$
- Using $t + tmin$ or $t + tmax$ the uncertainty is $u = tmax - tmin$.
- But, using $t + (tmin + tmax)/2$ the uncertainty becomes $u/2$.

Asynchronous

Now transit time varies between $tmin$ and $+\infty$
How to update the clocks?

Synchronization: Asynchronous system
Cristian's algorithm

- Send a request m_r that triggers response m_t carrying time t
- Measure the *roud-trip-time* of request and reply as t_r.
- Set clock to $t + t_r/2$ assuming a balanced round trip
- Precision can be increased by repeating the protocol until a low t_r occurs

Berkeley Algorithm

A coordinator measures RTT to the other nodes and sets target time to the average of times. New times for nodes to set are propagated as deltas to their local times, in order to be resilient to propagation delays.

Synchronization: Asynchronous system
Cristian's algorithm

- Send a request m_r that triggers response m_t carrying time t
- Measure the *roud-trip-time* of request and reply as t_r.
- Set clock to $t + t_r/2$ assuming a balanced round trip
- Precision can be increased by repeating the protocol until a low t_r occurs

Berkeley Algorithm

A coordinator measures RTT to the other nodes and sets target time to the average of times. New times for nodes to set are propagated as deltas to their local times, in order to be resilient to propagation delays.

Causality – Happens-before
Lamport 78

Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

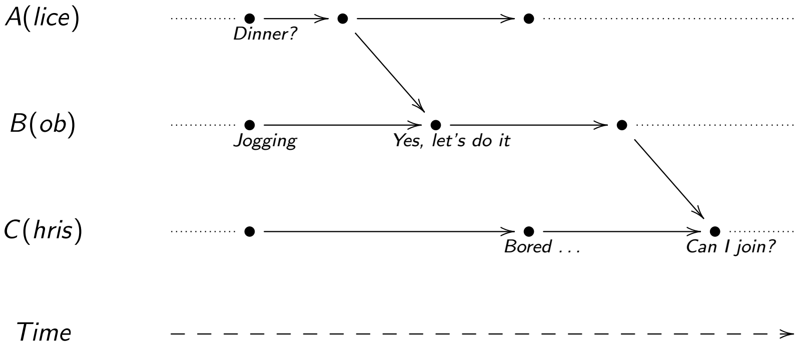
(1978 http://amturing.acm.org/p558-lamport.pdf)

Causality

A social interaction

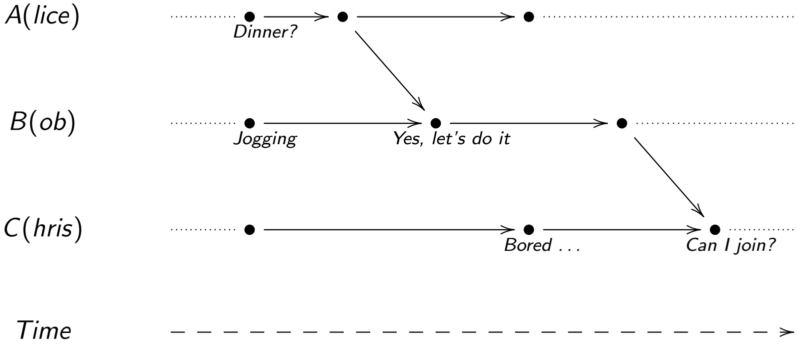
- Alice decides to have dinner
- She tells that to Bob and he agrees
- Meanwhile Chris was bored
- Bob tells Chris and he asks to join for dinner

Causality is a partial order relation



Causally: "Alice wants dinner" $\parallel$ "Chris is bored"
Timeline: "Alice wants dinner" $<$ "Chris is bored"

Causality is only potential influence



Causally: "Bob is jogging" $\rightarrow$ "Bob says: Yes, let's do it"

. . . but they are probably unrelated

Causality is only potential influence

Past statements only potentially influence future ones

```
a = 5;
b = 5;
if (a > 2) c=2;
```

Causally: "`b = 5;`" $\rightarrow$ "`if (a > 2) c=2;`"
...but in fact not

How to track it?

Causality relation

How to track it? Maybe read Vector Clock entry in Wikipedia?

Partial ordering property [edit]

Vector clocks allow for the partial causal ordering of events. Defining the following:

- $VC(x)$ denotes the vector clock of event x, and $VC(x)_z$ denotes the component of that clock for process z.
- $VC(x) < VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$
  - In English: $VC(x)$ is less than $VC(y)$, if and only if $VC(x)_z$ is less than or equal to $VC(y)_z$ for all process indices z, and at least one of those relationships is strictly smaller (that is, $VC(x)_{z'} < VC(y)_{z'}$).
- $x \rightarrow y$ denotes that event x happened before event y. It is defined as: if $x \rightarrow y$, then $VC(x) < VC(y)$

Properties:

- If $VC(a) < VC(b)$, then $a \rightarrow b$
- Antisymmetry: if $VC(a) < VC(b)$, then $\neg VC(b) < VC(a)$
- Transitivity: if $VC(a) < VC(b)$ and $VC(b) < VC(c)$, then $VC(a) < VC(c)$ or if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

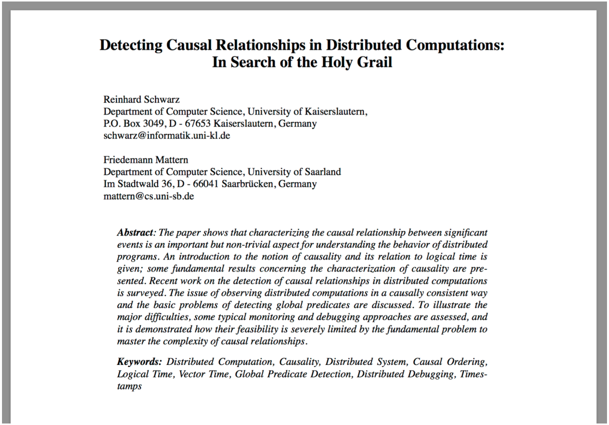
Relation with other orders:

- Let $RT(x)$ be the real time when event x occurs. If $VC(a) < VC(b)$, then $RT(a) < RT(b)$
- Let $C(x)$ be the Lamport timestamp of event x. If $VC(a) < VC(b)$, then $C(a) < C(b)$

(2015 https://en.wikipedia.org/wiki/Vector_clock)

Maybe start with something simpler: **Causal histories**

Causal histories
Schwarz & Mattern 94



Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail

Reinhard Schwarz
Department of Computer Science, University of Kaiserslautern,
P.O. Box 3049, D - 67653 Kaiserslautern, Germany
schwarz@informatik.uni-kl.de

Friedemann Mattern
Department of Computer Science, University of Saarland
Im Stadtwald 36, D - 66041 Saarbrücken, Germany
mattern@cs.uni-sb.de

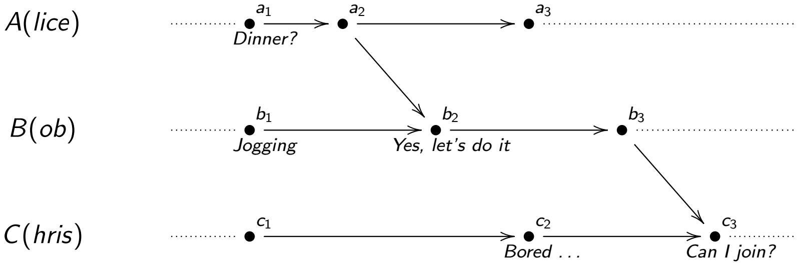
***Abstract**: The paper shows that characterizing the causal relationship between significant events is an important but non-trivial aspect for understanding the behavior of distributed programs. An introduction to the notion of causality and its relation to logical time is given; some fundamental results concerning the characterization of causality are presented. Recent work on the detection of causal relationships in distributed computations is surveyed. The issue of observing distributed computations in a causally consistent way and the basic problems of detecting global predicates are discussed. To illustrate the major difficulties, some typical monitoring and debugging approaches are assessed, and it is demonstrated how their feasibility is severely limited by the fundamental problem to master the complexity of causal relationships.*

***Keywords:** Distributed Computation, Causality, Distributed System, Causal Ordering, Logical Time, Vector Time, Global Predicate Detection, Distributed Debugging, Timestamps*

(1994 https://www.vs.inf.ethz.ch/publ/papers/holygrail.pdf)

Uniquely tag each event

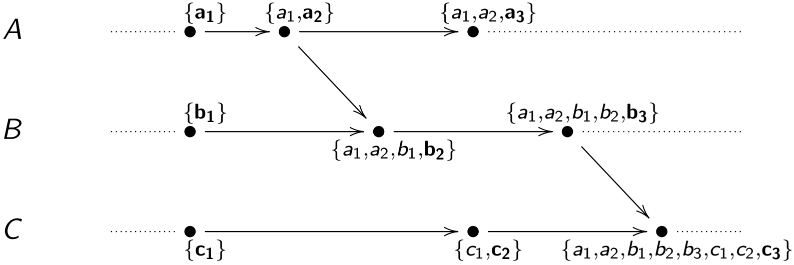
For instance, node name and growing counter



Causal histories

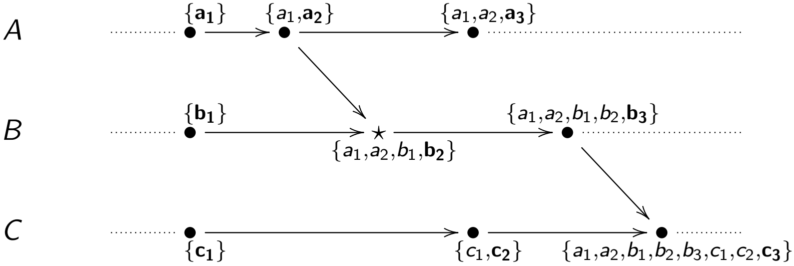
- Collect memories as sets of unique events
- Set inclusion explains causality
  - $\{a_1, b_1\} \subset \{a_1, a_2, b_1\}$
- You are in my past if I know your history
- If we don't know each other's history, we are concurrent
- If our histories are the same, we are the same

Causal histories



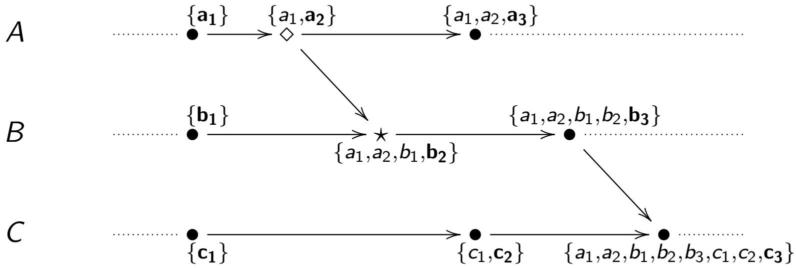
Causal histories

Message reception



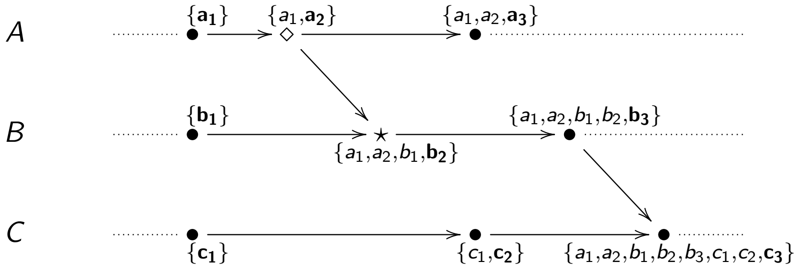
$\star$ receive $\{a_1, a_2\}$ at node b with $\{b_1\}$ yields $\{b_1\} \cup \{a_1, a_2\} \cup \{b_2\}$

Causal histories
Causality check



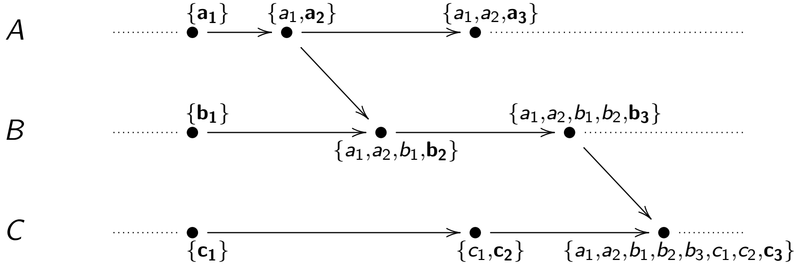
Check $\diamond \rightarrow \star$ iff $\{a_1, \mathbf{a_2}\} \subset \{a_1, a_2, b_1, \mathbf{b_2}\}$

Causal histories
Faster causality check



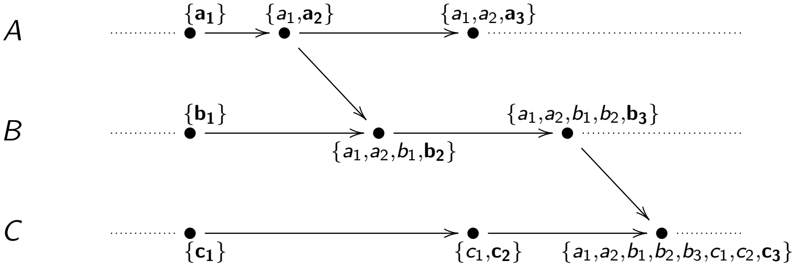
Check $\diamond \rightarrow \star$ iff $\mathbf{a_2} \in \{a_1, a_2, b_1, \mathbf{b_2}\}$

Causal histories



Note: $\{e_n\} \subset C_x \Rightarrow \{e_1 \ldots e_n\} \subset C_x$

Causal histories



Lots of redundancy that can be compressed

Vector clocks

Mattern || Fidge, 88

Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors of clocks. These clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way. The new model of time has a close analogy to Minkowski's relativistic space-time and leads among others to an interesting characterization of the global state problem. Finally, we present a new algorithm to compute a consistent global snapshot of a distributed system where messages may be received out of order.

view of an idealized external observer having immediate access to all processes.

The fact that *a priori* no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like *mutual exclusion, deadlock detection, and concurrency control* are much more difficult to solve in a distributed environment than in a classical centralized environment, and a rather large number of distributed control algorithms for those problems has found to be wrong. *New problems* which do not exist in centralized systems or in parallel systems with common memory also emerge in distributed systems. Among the most prominent of these problems are *distributed agreement, distributed termination detection*, and the *symmetry breaking* or *election problem*. The great diversity of the solutions to these problems some of them being really beautiful and elegant—is truly amazing and exemplifies many principles of distributed computing to cope with the absence of global state and time.

Since the design, verification, and analysis of algorithms for asvnchronous svstems is difficult and error-

Timestamps in Message-Passing Systems That Preserve the Partial Ordering

Colin J. Fidge
Department of Computer Science, Australian National University, Canberra, ACT.

ABSTRACT

Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to the partial ordering inherent in a parallel system. The algorithms do not change the communications graph or require a central timestamp issuing authority.

Keywords and phrases: concurrent programming, message-passing, timestamps, logical clocks
CR categories: D.1.3

INTRODUCTION

A fundamental problem in concurrent programming is determining the order in which events in different processes occurred. An obvious solution is to attach a number representing the current time to a permanent record of the execution of each event. This assumes that each process can access an accurate clock, but practical parallel systems, by their very nature, make it difficult to ensure consistency among the processes.

There are two solutions to this problem. Firstly, have a central process to issue timestamps, i.e. provide the system with a global clock. In practice this has the major disadvantage of needing communication links from all processes to the central clock.

More acceptable are separate clocks in each process that are kept synchronised as much as necessary to ensure that the timestamps represent, at the very least, a *possible* ordering of events (in light of the vagaries of distributed scheduling). Lamport (1978) describes just such a scheme of logical clocks that can be used to totally order events, without the need to introduce extra communication links.

However this only yields one of the many possible, and equally valid, event orderings defined by a particular distributed computation. For problems concerned with the global program state it is far more useful to have access to the entire *partial* ordering, which defines the set of consistent "slices" of the global state at any arbitrary moment in time.

1988 (https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf)
(http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf)

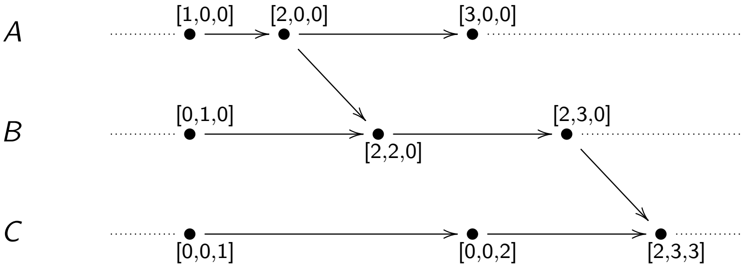
Vector clocks
Compacting causal histories

- $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$
- $\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$

Finally a vector, assuming a fixed number of processes with totally ordered identifiers

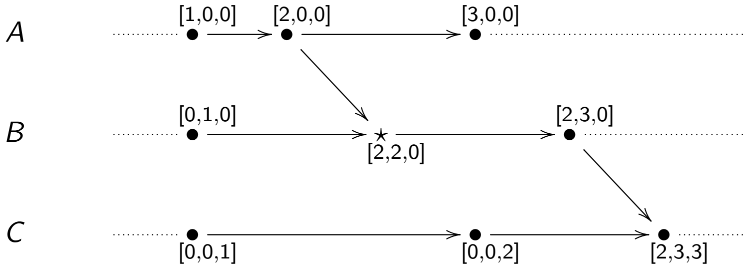
- $[2, 3, 3]$

Vector clocks



Vector clocks
Message reception (step 1)

Set union becomes **join** $\sqcup$ by point-wise maximum in vectors

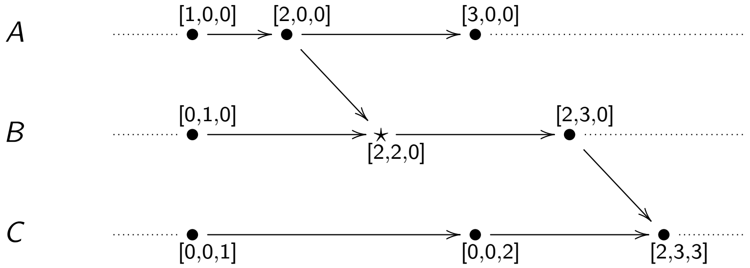


$\star$ receive $[2,0,0]$ at b with $[0,1,0]$ yields $\mathrm{inc}_b(\sqcup([2,0,0],[0,1,0]))$

Vector clocks

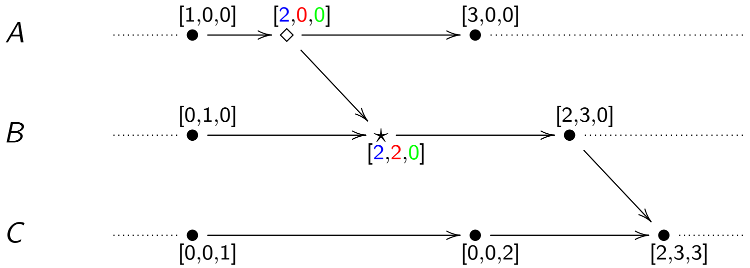
Message reception (step 2)

Set union becomes **join** $\sqcup$ by point-wise maximum in vectors



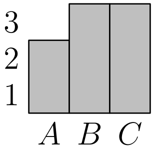
$\mathsf{inc}_b(\sqcup([2,0,0],[0,1,0])) \equiv \mathsf{inc}_b([2,1,0]) \equiv [2,2,0]$

Vector clocks
Causality check



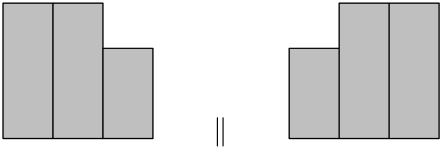
Check $\diamond \rightarrow \star$ iff point-wise check $2 \leq 2, 0 \leq 2, 0 \leq 0$

$$[2, 3, 3] \equiv \{A \mapsto 2, B \mapsto 3, C \mapsto 3\} \equiv$$



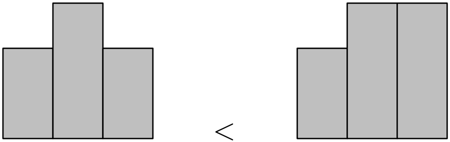
Vector clocks
Graphically comparing

$$[3, 3, 2] \parallel [2, 3, 3]$$



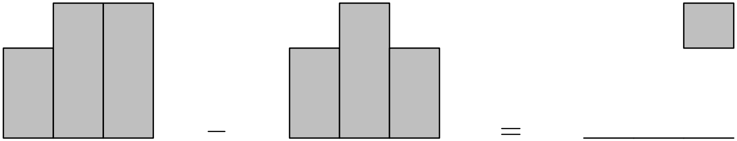
Vector clocks
Graphically comparing

$$[2, 3, 2] < [2, 3, 3]$$



Vector clocks
Graphical difference

$$[2, 3, 3] - [2, 3, 2] = \{c3\}$$

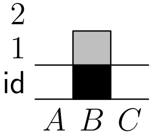


Vector clocks
Graphical message reception

Node b, with $[0, 1, 0]$ is receiving a message with $[2, 0, 0]$

We need to combine the two vectors and update b entry

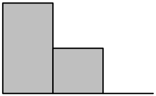
Node b, with $[0, 1, 0]$:



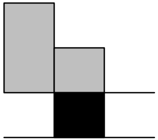
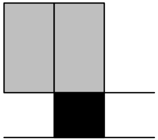
Vector clocks

Graphical message reception

Point-wise maximum of $[2, 0, 0]$ and $[0, 1, 0]$

max(, ) = 

Register a new event on the result

event() = 

Vector clocks
Faster causality check

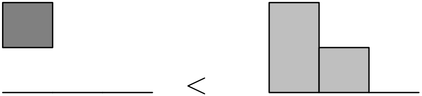
Comparing vectors is linear on the vector size

This can be improved by tracking the last event

Vector clocks with dots

Decouple dot of last event

- $[\mathbf{2}, 0, 0]$ becomes $[1, 0, 0]a_2$
- $[2, \mathbf{2}, 0]$ becomes $[2, 1, 0]b_2$
- The causal past excludes the event itself
- Check $[\mathbf{2}, 0, 0] \to [2, \mathbf{2}, 0]$?
- Check $[1, 0, 0]a_2 \to [2, 1, 0]b_2$ iff dot a_2 index $_2 \leq 2$



Registering relevant events

- Not always important to track all events
- Track only update events in data replicas
- Applications in:
  - File-Systems
  - Databases
  - Version Control

Causally tracking of write/put operations

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

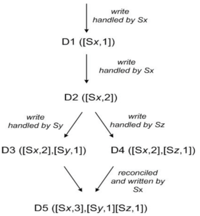


Figure 3: Version evolution of an object over time.

object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

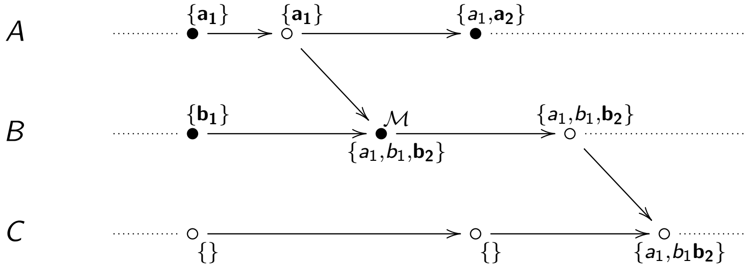
4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read

(http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf)

Causal histories with only some relevant events

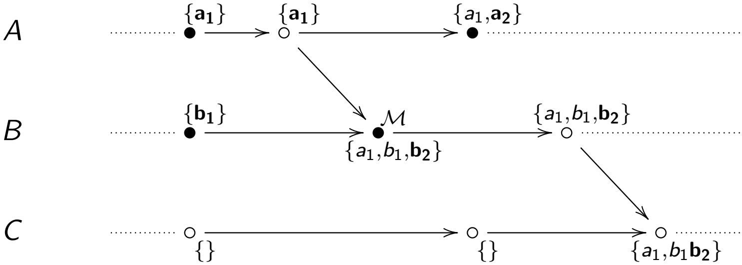
Relevant events are marked with a • and get an unique tag/dot



Other events get a ∘ and don't add to history

Causal histories with only some relevant events

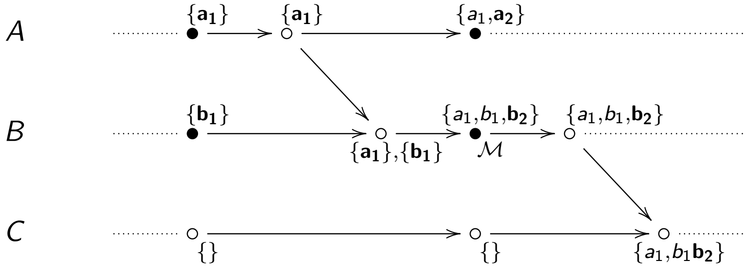
Concurrent states $\{a_1\} \parallel \{b_1\}$ lead to a • marked merge $\mathcal{M}$



Causally dominated state $\{\} \rightarrow \{a_1, b_1, b_2\}$ is simply replaced

Causal histories with versions not immediately merged

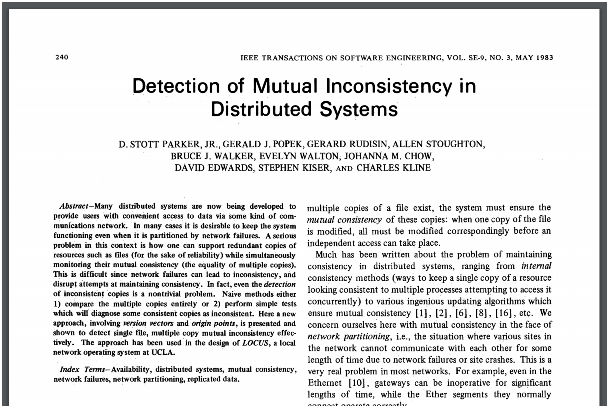
Versions can be collected and merge deferred



Causal histories are only merged upon version merging in a new •

Version vectors

Parker et al 83



240 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-9, NO. 3, MAY 1983

Detection of Mutual Inconsistency in Distributed Systems

D. STOTT PARKER, JR., GERALD J. POPEK, GERARD RUDISIN, ALLEN STOUGHTON,
BRUCE J. WALKER, EVELYN WALTON, JOHANNA M. CHOW,
DAVID EDWARDS, STEPHEN KISER, AND CHARLES KLINE

***Abstract*—Many distributed systems are now being developed to provide users with convenient access to data via some kind of communications network. In many cases it is desirable to keep the system functioning even when it is partitioned by network failures. A serious problem in this context is how one can support redundant copies of resources such as files (for the sake of reliability) while simultaneously monitoring their mutual consistency (the equality of multiple copies). This is difficult since network failures can lead to inconsistency, and disrupt attempts at maintaining consistency. In fact, even the *detection* of inconsistent copies is a nontrivial problem. Naive methods either 1) compare the multiple copies entirely or 2) perform simple tests which will diagnose some consistent copies as inconsistent. Here a new approach, involving *version vectors* and *origin points*, is presented and shown to detect single file, multiple copy mutual inconsistency effectively. The approach has been used in the design of *LOCUS*, a local network operating system at UCLA.**

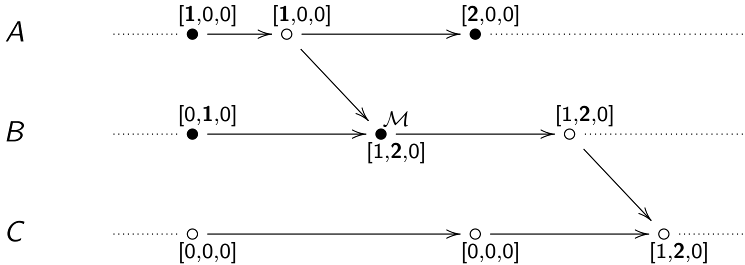
***Index Terms*—Availability, distributed systems, mutual consistency, network failures, network partitioning, replicated data.**

multiple copies of a file exist, the system must ensure the *mutual consistency* of these copies: when one copy of the file is modified, all must be modified correspondingly before an independent access can take place.

Much has been written about the problem of maintaining consistency in distributed systems, ranging from *internal* consistency methods (ways to keep a single copy of a resource looking consistent to multiple processes attempting to access it concurrently) to various ingenious updating algorithms which ensure mutual consistency [1], [2], [6], [8], [16], etc. We concern ourselves here with mutual consistency in the face of *network partitioning*, i.e., the situation where various sites in the network cannot communicate with each other for some length of time due to network failures or site crashes. This is a very real problem in most networks. For example, even in the Ethernet [10], gateways can be inoperative for significant lengths of time, while the Ether segments they normally connect operate correctly.

(1983 http://zoo.cs.yale.edu/classes/cs422/2013/bib/parker83detection.pdf)

Version vectors



Can Causality scale?

Charron-Bost 91

One entry needed per source of concurrency

Information Processing Letters 39 (1991) 11–16
North-Holland

12 July 1991

Concerning the size of logical clocks in distributed systems

Bernadette Charron-Bost

LITP, IBP, Université Paris 7, 2 Place Jussieu, 75251 Paris Cedex 05, France

Communicated by L. Boasson
Received 18 October 1990
Revised 17 April 1991

Keywords: Distributed computing, clock, logical time, causality, partially ordered sets

1. Introduction

Distributed systems with no known bounds on relative processor speed and transmission delay are called *asynchronous*. In such systems, coordination and synchronization between processes are difficult to achieve. Therefore, the design and the proof of distributed algorithms for asynchronous systems are much more subtle than for a classical centralized environment.

seem very heavy as soon as one is concerned with a distributed system on a large number of processes.

In this paper we prove that vectors of this length are necessary to characterize causality, by constructing an appropriate distributed computation. Then we use classical theorems from the theory of partially ordered sets to give a mathematical interpretation of this result.

(1991 https://doi.org/10.1016/0020-0190(91)90055-M)

Scaling Causality

Scaling at the edge (DVVs)

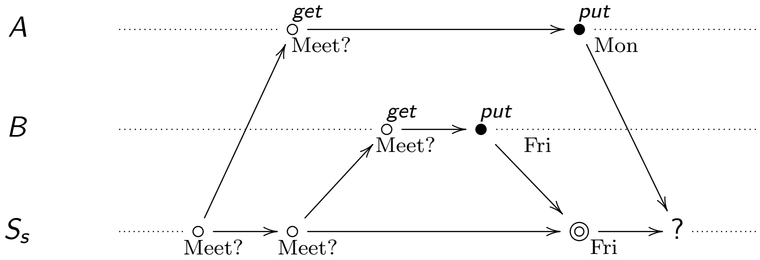
- Mediate interaction by DC proxies
- Support failover and DC switching
- One entry per proxy (not per client)

Dynamic concurrency degree (ITCs)

- Creation and retirement of active entities
- Transparent tracking with minimal coordination
- Causality tracing under concurrency across services

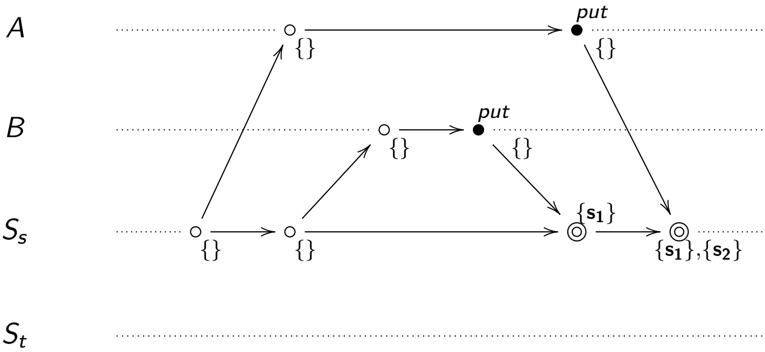
Scaling Causality

Dynamo like, get/put interface



- Conditional writes
- Overwrite first value
- Multi-Value ($[1, 0] \parallel [0, 1]$, one entry per client!)

Dynamo like, get/put interface



Scaling Causality

DVVs. Almeida et al 14

Dotted Version Vectors

Scalable and Accurate Causality Tracking for Eventually Consistent Stores

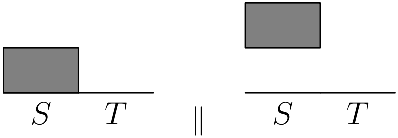
Paulo Sérgio Almeida[1], Carlos Baquero[1],
Ricardo Gonçalves[1], Nuno Preguiça[2], and Victor Fonte[1]

[1] HASLab, INESC Tec & Universidade do Minho
{psa,cbm,tome,vff}@di.uminho.pt
[2] CITI/DI, FCT, Universidade Nova de Lisboa
nuno.preguica@fct.unl.pt

Abstract. In cloud computing environments, data storage systems often rely on optimistic replication to provide good performance and availability even in the presence of failures or network partitions. In this scenario, it is important to be able to accurately and efficiently identify updates executed concurrently. Current approaches to causality tracking in optimistic replication have problems with concurrent updates: they either (1) do not scale, as they require replicas to maintain information that grows linearly with the number of writes or unique clients; (2)

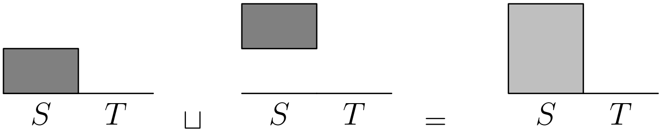
(2014 https://link.springer.com/chapter/10.1007/978-3-662-43352-2_6)

$[0,0]s_1 \parallel [0,0]s_2$



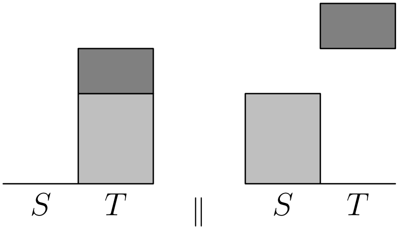
Gets get all values, with compact *causal context*

$$[0,0]s_1 \sqcup [0,0]s_2 = [2,0]$$



Puts can go to alternative servers
Server T can get a put with context $[2, 0]$

$$[0, 2]t_3 \parallel [2, 0]t_4$$



DVVs are used in the Riak Key-value store, deployed in the British NHS, BET365 and other global companies.

NHS launches upgraded IT backbone Spine, powered by Riak

Posted September 9, 2014 | Category: Press Releases

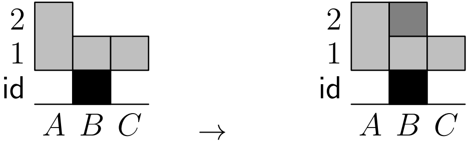
London, Sept. 9 2014 – the Spine – the electronic backbone of the UK's National Health Service – has been successfully rebuilt to harness new technology, including the use of Basho Technologies' distributed database, Riak Enterprise.

The Spine – a collection of national applications, services and directories – connects clinicians, patients and local service providers throughout England to essential national services, such as electronic prescriptions and patient health records.

Spine is used by more than 20,000 organizations that provide health care across England, including primary and secondary care sites, pharmacies, opticians and dentists. Riak, the open source distributed database, is key to providing the reliability and scalability for the platform to drive efficiency and improve patient care.

Dynamic Causality

Tracking causality requires exclusive access to identities



To avoid preconfiguring identities, id space can be split and joined

Dynamic Causality
ITCs. Almeida et al 08

Interval Tree Clocks

A Logical Clock for Dynamic Systems

Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte

DI/CCTC, Universidade do Minho
Largo do Paço, 4709 Braga Codex, Portugal
{psa,cbm,vff}@di.uminho.pt

Abstract. Causality tracking mechanisms, such as vector clocks and version vectors, rely on mappings from globally unique identifiers to integer counters. In a system with a well known set of entities these ids can be preconfigured and given distinct positions in a vector or distinct names in a mapping. Id management is more problematic in dynamic systems, with large and highly variable number of entities, being worsened when network partitions occur. Present solutions for

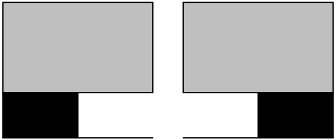
(2008 https://link.springer.com/chapter/10.1007/978-3-540-92221-6_18)

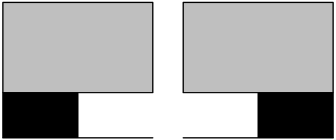
A seed node controls the initial id space

id 

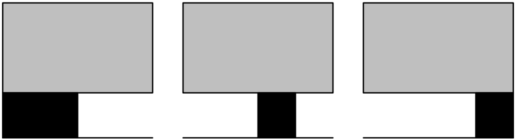
Registering events can use any portion above the controlled id

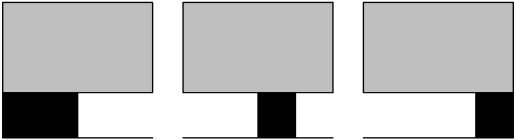
Ids can be split from any available entity



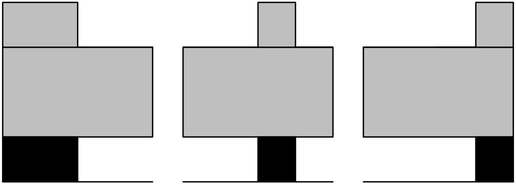


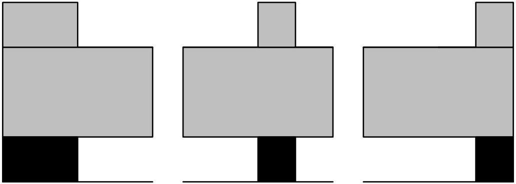
...and be split again



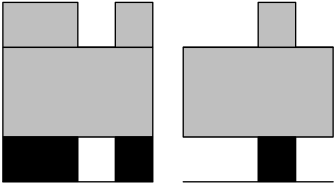


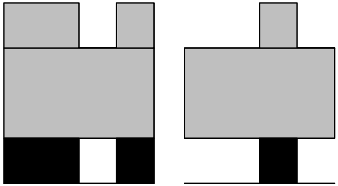
Entities can register new events and become concurrent



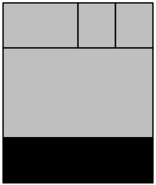


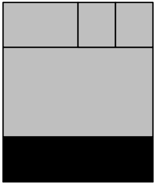
Any two entries can merge together



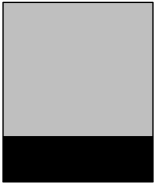


. . . eventually collecting the whole id space





. . . and simplifying the encoding of events



Global Invariants on IDs

Causality characterization condition

Each entity has a portion of its identity that is exclusive to it. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities.

$$\forall i.\ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

Entity events must use at least a part of the exclusive portion.

Disjoint condition

A less general but more practical condition is that all identities are kept disjoint.

$$\forall i_1 \neq i_2.\ i_1 \cdot i_2 = \mathbf{0}.$$

Any portion of the id can be used to register events.

Dynamic Causality
Interval Tree Clocks

ITCs are used in concurrency tracing and debugging, and in distributed settings within the Pivot tracing system.

Pivot tracing: dynamic causal monitoring for distributed systems

Full Text: PDF

see source materials below for more options

Authors: Jonathan Mace Brown University
Ryan Roelke Brown University
Rodrigo Fonseca Brown University

2015 Article

Best Paper

Bibliometrics

Published in:

- Proceeding

SOSP '15 Proceedings of the 25th Symposium on Operating Systems Principles
Pages 378-393

Monterey, California — October 04 - 07, 2015
ACM New York, NY, USA ©2015
table of contents ISBN: 978-1-4503-3834-9
doi>10.1145/2815400.2815415

Closing notes

- Causality is important because time is limited
- Causality is about memory of relevant events
- Causal histories are very simple encodings of causality
- VC, DVV, ITC do efficient encoding of causal histories
- All mechanisms are only encoding of causal histories

Graphical representations allow visualization of causality, this is useful in comparing existing mechanisms and to develop novel ones.