TEXT ONLY

Why Logical Clocks are Easy

SOMETIMES ALL YOU NEED IS THE RIGHT LANGUAGE

CARLOS BAQUERO AND NUNO PREGUIÇA

Any computing system can be described as executing sequences of actions, with an *action* being any relevant change in the state of the system. For example, reading a file to memory, modifying the contents of the file in memory, or writing the new contents to the file are relevant actions for a text editor. In a distributed system, actions execute in multiple locations; in this context, actions are often called events. Examples of events in distributed systems include sending or receiving messages, or changing some state in a node. Not all events are related, but some events can cause and influence how other, later events occur. For example, a reply to a received mail message is influenced by that message, and maybe by prior messages received.

Events in a distributed system can occur in a close location, with different processes running in the same machine, for example; or at nodes inside a data center; or geographically spread across the globe; or even at a larger scale in the near future. The relations of potential cause and effect between events are fundamental to the design of distributed algorithms. These days hardly any service can claim not to have some form of distributed algorithm at its core.

To make sense of these cause-and-effect relations, it is necessary to limit their scope to what can be perceived

A distributed system has no way of knowing that the first reservation has actually caused the second one

inside the distributed system itself—*internal causality*. Naturally, a distributed system interacts with the rest of the physical world outside of it, and there are also cause-and-effect relations in that world at large. For example, consider a couple planning a night out using a system that manages reservations for dinner and a movie. One person makes a reservation for dinner and lets the other person know with a phone call. After receiving the phone call, the second person goes to the system and reserves a movie. A distributed system has no way of knowing that the first reservation has actually caused the second one.

This *external causality* cannot be detected by the system and can only be approximated by *physical time*. (Time, however, totally orders all events, even those unrelated—thus, it is no substitute to causality—and wall clocks are never perfectly synchronized.[11,16]) This article focuses instead on internal causality—the type that can be tracked by the system.

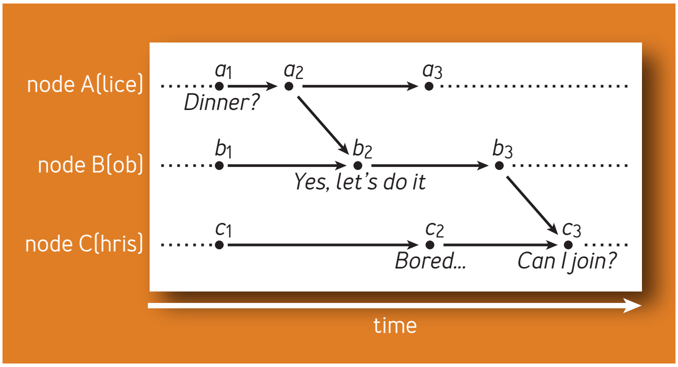
HAPPENED-BEFORE RELATION

In 1978 Leslie Lamport defined a partial order, referred to as *happened before*, that connects events of a distributed system that are potentially causally linked.[8] An event *c* can be the cause of an event *e*, or *c* happened before *e*, iff (if and only if) both occur in the same node and *c* executed first, or, being at different nodes, if *e* could know about the occurrence of *c* thanks to some message received from some node that knows about *c*. If neither event can know about the other, then they are said to be concurrent.

Using the example of dinner and movie reservations, figure 1 shows a distributed system with three nodes. An arrow between nodes represents a message sent and delivered. Both Bob's positive answer to the dinner suggestion by Alice and Chris's later request to join the party are influenced by Alice's initial question about plans for dinner.

In this distributed computation, a simple way to check if an event *c* could have caused another event *e* (*c* happened before *e*) is to find at least one directed path linking *c* to *e*. If such a connection is found, this partial order relation is marked $c \rightarrow e$ to denote the happened-before relation or potential causality. Figure 1 has $a_1 \rightarrow b_2$ and $b_3 \rightarrow c_3$ (and, yes, also $a_1 \rightarrow c_3$, since causality is transitive). Events a_1 and c_2 are concurrent (denoted $a_1 \parallel c_2$), because there are no causal paths in either direction. Note $x \parallel y$ if and only if $x \nrightarrow y$ and $y \nrightarrow x$. The fact that Chris was bored neither influenced Alice's question about dinner, nor the other way around.

FIGURE 1: **HAPPENED-BEFORE RELATION**



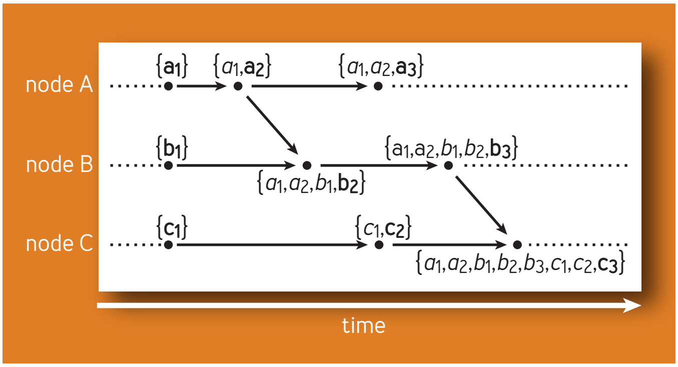
Thus, the three possible relations between two events *x* and *y* are: (a) *x* might have influenced *y*, if $x \rightarrow y$; (b) *y* might have influenced *x*, if $y \rightarrow x$; (c) there is no known influence between *x* and *y*, as they occurred concurrently $x \parallel y$.

CAUSAL HISTORIES

Causality can be tracked in a very simple way by using *causal histories*.[3,14] The system can locally assign unique names to each event (e.g., node name and local increasing counter) and collect and transmit sets of events to capture the known past.

For a new event, the system creates a new unique name, and the causal history consists of the union of this name and the causal history of the previous event in the node. For example, the second event in node *C* is assigned the name c_2, and its causal history is $H_c = \{c_1, \mathbf{c_2}\}$ (shown in figure 2). When a node sends a message, the causal history of the send event is sent with the message. When the message is received, the

FIGURE 2: **CAUSAL HISTORIES**



remote causal history is merged (by set union) with the local history. For example, the delivery of the first message from node A to B merges the remote causal history $\{a_1, a_2\}$ with the local history $\{b_1\}$ and the new unique name b_2, leading to $\{a_1, a_2, b_1, \mathbf{b_2}\}$.

Checking causality between two events x and y can be tested simply by set inclusion: $x \longrightarrow y$ iff $H_x \subsetneq H_y$. This follows from the definition of causal histories, where the causal history of an event will be included in the causal history of the following event. Even better, marking the last local event added to the history (distinguished in bold in the figure) allows the use of a simpler test: $x \longrightarrow y$ iff $x \in H_y$ (e.g., $a_1 \longrightarrow b_2$, since $a_1 \in \{a_1, a_2, b_1, b_2\}$). This follows from the fact that a causal history includes all events that (causally) precede a given event.

VECTOR CLOCKS

It should be obvious by now that causal histories work but are not very compact. This problem can be addressed by relying on the following observation: the mechanism of building the causal history implies that if an event b_3 is present in H_y, then all preceding events from that same node, b_1 and b_2, are also present in H_y. Thus, it suffices to store the most recent event from each node. Causal history $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$ is compacted to $\{a \longmapsto 2, b \longmapsto 3, c \longmapsto 3\}$ or simply a vector [2, 3, 3].

Now the rules used with causal histories can be translated to the new compact vector representation.

Verifying that $x \longrightarrow y$ requires checking if $H_x \subsetneq H_y$. This can be done, verifying for each node, if the unique names contained in H_x are also contained in H_y and there is at least one unique name in H_y that is not contained in H_x. This is immediately translated to checking if each entry in the vector of x is smaller or equal to the corresponding entry in the vector of y and one is strictly smaller (i.e., $\forall i : V_x[i] \leq V_y[i]$ and $\exists j : V_x[j] < V_y[j]$). This can be stated more compactly as $x \rightarrow y$ iff $V_x < V_y$.

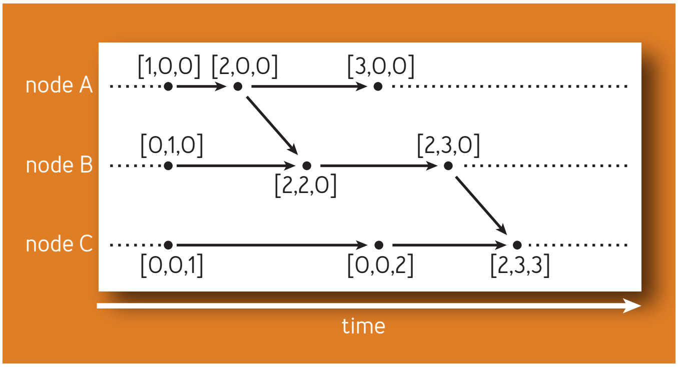
For a new event the creation of a new unique name is equivalent to incrementing the entry in the vector for the node where the event is created. For example, the second event in node *C* has vector [0, 0, 2], which corresponds to the creation of event c_2 of the causal history.

Finally, creating the union of the two causal histories H_x and H_y is equivalent to taking the pointwise maximum of the corresponding two vectors V_x and V_y (i.e., $\forall i : V[i] = \max(V_x[i], V_y[i])$). Logic tells us that, for the unique names generated in each node, only the one with the largest counter needs to be kept.

When a message is received, in addition to merging the causal histories, a new event is created. The vector representation of these steps can be seen, for example, when the first message from *a* is received in *b*, where taking the pointwise maximum leads to [2, 1, 0] and the new unique name finally leads to [2, 2, 0], as shown in figure 3.

This compact representation, known as a *vector clock*, was introduced around 1988.[5,10] Vector comparison is an immediate translation of set inclusion of causal histories. This equivalence is often forgotten in modern descriptions of

FIGURE 3: **VECTOR CLOCKS**



vector clocks and can turn what is a simple encoding problem into an unnecessarily complex and arcane set of rules, going against logic.

DOTTED VECTOR CLOCKS

As shown thus far, when using causal histories, knowing the last event could simplify comparison by simply checking if the last event is included in the causal history. This can still be done with vectors, if you keep track of the node in which the last event has been created. For example, when questioning if $x = [\mathbf{2}, 0, 0] \rightarrow y = [2, \mathbf{3}, 0]$, with boldface indicating the last event in each vector, you can simply test if $x[0] \leq y[0]$ $(\mathbf{2} \leq 2)$ since you have marked that the last event in x was created in node A (i.e., it corresponds to the first entry of the vector). Since marking numbers in bold is not a practical implementation, however, the last event is usually stored outside the vector (and is sometimes called a *dot*): for example, $[2, \mathbf{2}, 0]$ can be represented as $[2, 1, 0]b_2$. Notice that now the vector represents the causal past of b_2, excluding the event itself.

VERSION VECTORS

In an important class of applications there is no need to register causality for all the events in a distributed computation. For example, to modify replicas of data, it often suffices to register only those events that change replicas. In this case, when thinking about causal histories, you need only to assign a new unique name to these relevant events. Still, you need to propagate the causal histories when messages are propagated from one site to another, and the remaining rules for comparing causal histories remain unchanged.

Figure 4 presents the same example as before, but now with events that are not registered for causality tracking denoted with ∘. If the run represents the updates to replicas of a data object, then after nodes *A* and *B* are concurrently modified, the state of replica *a* is sent to replica *b* (in a

FIGURE 4: **CAUSAL HISTORIES WITH ONLY SOME RELEVANT EVENTS**

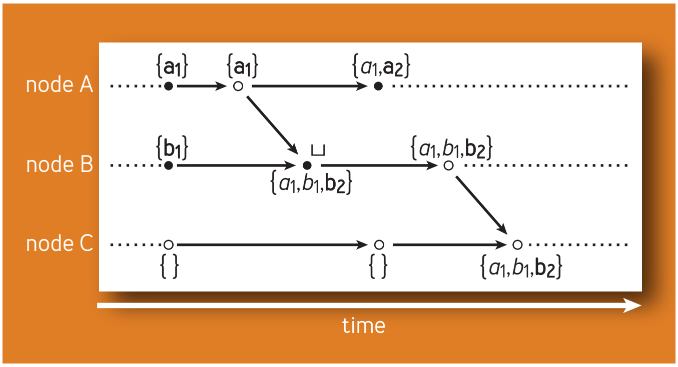
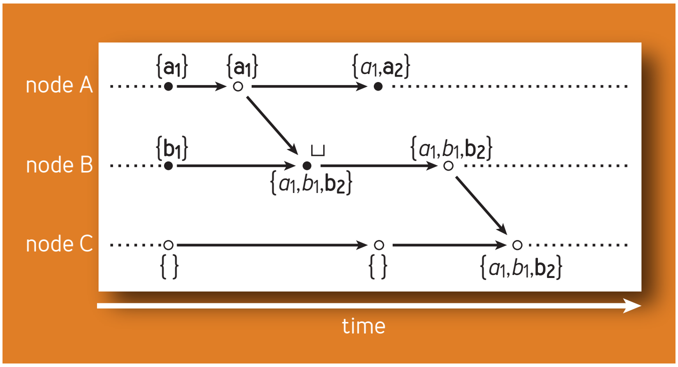


FIGURE 4: **CAUSAL HISTORIES WITH ONLY SOME RELEVANT EVENTS**



message). When the message is received in node *B*, it is detected that two concurrent updates have occurred, with histories $\{a_1\}$ and $\{b_1\}$, as neither $a_1 \rightarrow b_1$ nor $b_1 \rightarrow a_1$. In this case, a new version that merges the two updates is created (merge is denoted by the join symbol $\sqcup$), which requires creating a new unique name, leading to $\{a_1, b_1, b_2\}$. When the state of replica *b* is later propagated to replica *c*, as no concurrent update exists in replica *c*, no new version is created.

Again, vectors can compact the representation. The result, known as a *version vector*, was created in 1983,[12] five years before vector clocks. Figure 5 presents the same example as before, represented with version vectors.

In some cases when the state of one replica is propagated to another replica, the two versions are kept by the system as conflicting versions. For example, in figure 6, when the message from node *A* is received in node *B*, the system keeps each causal history $\{a_1\}$ and $\{b_1\}$ associated with the

FIGURE 5: **VERSION VECTORS WITH ONLY SOME RELEVANT EVENTS**

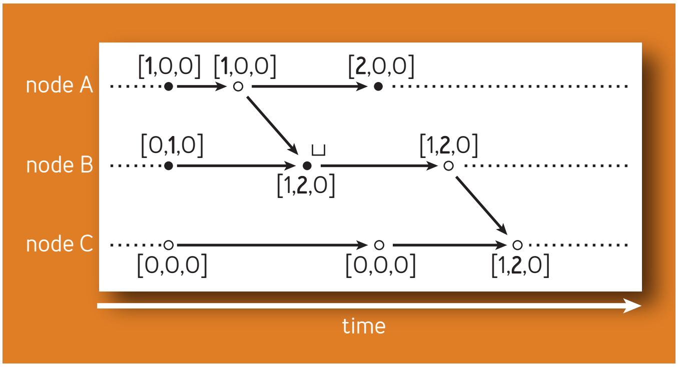
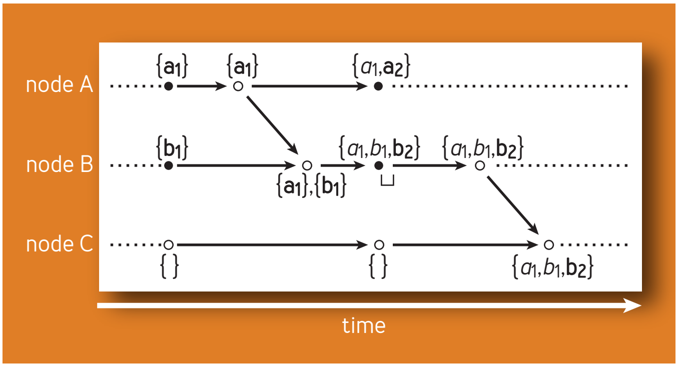


FIGURE 6: **CAUSAL HISTORIES WITH VERSIONS NOT IMMEDIATELY MERGED**



respective version. The causal history associated with the node containing both versions is $\{a_1, b_1\}$, the union of the causal history of all versions. This approach allows later checking for causality relations between each version and other versions when merging the states of additional nodes. The conflicting versions could also be merged, creating a new unique name, as in the example.

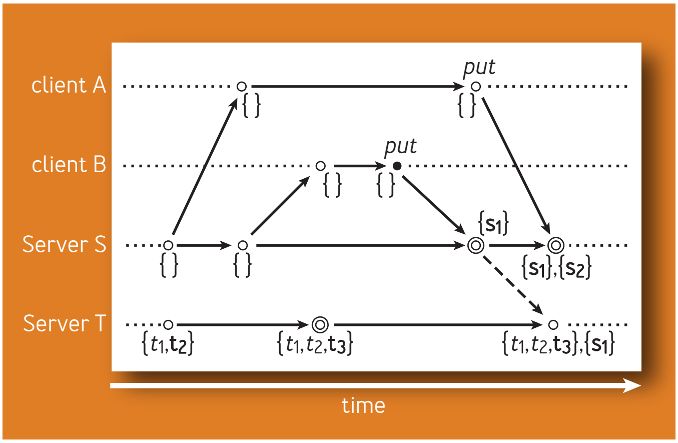
DOTTED VERSION VECTORS

One limitation of causality tracking by vectors is that one entry is needed for each source of concurrency.[4] You can expect a difference of several orders of magnitude between the number of nodes in a data center and the number of clients they handle. Vectors with one entry per client don't scale well when millions of clients are accessing the service.[7] Again, a look at the foundation of causal histories shows how to overcome this limitation.

The basic requirement in causal histories is that each event be assigned a unique identifier. There is no requirement that this unique identifier be created locally or immediately. Thus, in systems where nodes can be divided into clients and servers and where clients communicate only with servers, it is possible both to delay the creation of a new unique name until the client communicates with the server and to use a unique name generated in the server. The causal history associated with the new version is the union of the causal history of the client and the newly assigned unique name.

Figure 7 shows an example where clients A and B concurrently update server S. When client B first writes its version, a new unique name, s_1, is created (in the figure this action is denoted by the symbol ⊚) and merged with the causal history read by the client {}, leading to the causal history $\{s_1\}$. When client A later writes its version, the causal history assigned to this version is the causal history at the client, {}, merged with the new unique name s_2, leading to $\{s_2\}$. Using the normal rules for checking for concurrent updates, these two versions are concurrent. In the example, the

FIGURE 7: **CAUSAL HISTORIES IN A DISTRIBUTED STORAGE SYSTEM**



system keeps both concurrent updates. For simplicity, the interactions of server T with its own clients were omitted, but as shown in the figure, before receiving data from server S, server T had a single version that depicted three updates it managed—causal history $\{t_1, t_2, t_3\}$—and after that it holds two concurrent versions.

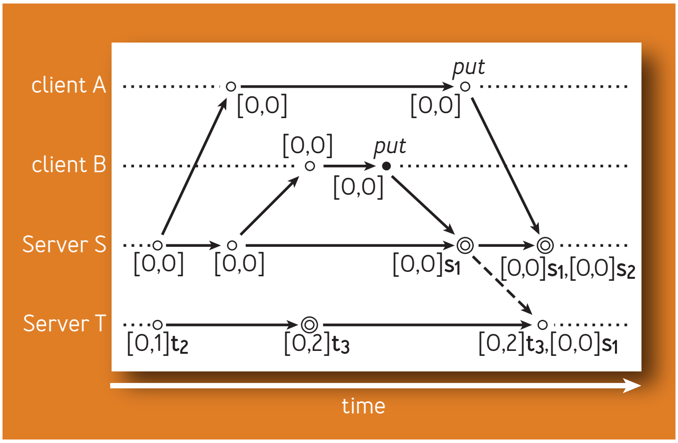
One important observation is that in each node, the union of the causal histories of all versions includes all generated unique names until the last known one: for example, in server S, after both clients send their new versions, all unique names generated in S are known. Thus, the causal past of any update can always be represented using a compact vector representation, as it is the union of all versions known at some server when the client read the object. The combination of the causal past represented as a vector and the last event, kept outside the vector, is known as a *dotted version vector*.[2,13] Figure 8 shows the previous example using this representation, which, as the system keeps running, eventually becomes much more compact than causal histories.

In the condition expressed before (clients communicate only with servers and a new update overwrites all versions previously read), which is common in key-value stores where multiple clients interact with storage nodes via a *get/put* interface, the dotted version vectors allow causality to be tracked between the written versions with vectors of the size of the number of servers.

FINAL REMARKS

Tracking causality should not be ignored. It is important in the design of many distributed algorithms. And not

FIGURE 8: DOTTED VERSION VECTORS IN DISTRIBUTED STORAGE SYSTEM



respecting causality can lead to strange behaviors for users, as reported by multiple authors.[1,9]

The mechanisms for tracking causality and the rules used in these mechanisms are often seen as complex,[6,15] and their presentation is not always intuitive. The most commonly used mechanisms for tracking causality—vector clocks and version vectors—are simply optimized representations of causal histories, which are easy to understand.

By building on the notion of causal histories, you can begin to see the logic behind these mechanisms, to identify how they differ, and even consider possible optimizations. When confronted with an unfamiliar causality-tracking mechanism, or when trying to design a new system that requires it, readers should ask two simple questions: (a) Which events need tracking? (b) How does the mechanism translate back to a simple causal history?

Without a simple mental image for guidance, errors and misconceptions become more common. Sometimes, all you need is the right language.

Acknowledgments
We would like to thank Rodrigo Rodrigues, Marc Shapiro, Russell Brown, Sean Cribbs, and Justin Sheehy for their feedback. This work was partially supported by EU FP7 SyncFree project (609551) and FCT/MCT projects UID/CEC/04516/2013 and UID/EEA/50014/2013.

References
1. Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., Veeraraghavan,

K. 2015. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland. Usenix Association.

2. Almeida, P. S., Baquero, C., Gonçalves, R., Preguiça, N. M., Fonte, V. 2014. Scalable and accurate causality tracking for eventually consistent stores. In *Proceedings of the Distributed Applications and Interoperable Systems,* held as part of the Ninth International Federated Conference on Distributed Computing Techniques, Berlin, Germany: 67–81.
3. Birman, K. P., Joseph, T. A. 1987. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* 5(1): 47–76.
4. Charron-Bost, B. 1991. Concerning the size of logical clocks in distributed systems. *Information Processing Letters* 39(1): 11–16.
5. Fidge, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10(1): 56–66.
6. Fink, B. 2010. Why vector clocks are easy. Basho Blog; http://basho.com/posts/ technical/why-vector-clocks-are-easy/.
7. Hoff, T. 2014. How League of Legends scaled chat to 70 million players—it takes lots of minions. High Scalability; http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.
8. Lamport, L. 1978. Time, clocks, and the ordering of events

in a distributed system. *Communications of the ACM* 21(7): 558–565.

9. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D. G. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, New York, NY: 401–416.
10. Mattern, F. 1988. Virtual time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Gers, France: 215– 226.
11. Neville-Neil, G. 2015. Time is an illusion. acmqueue 13(9). 57 - 72
12. Parker, D.S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., Kline, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 9(3): 240–247.
13. Preguiça, N. M., Baquero, C., Almeida, P. S., Fonte, V., Gonçalves, R. 2012. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors. In *ACM Symposium on Principles of Distributed Computing,* eds. D. Kowalski and A. Panconesi: 335–336.
14. Schwarz, R., Mattern, F. 1994. Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing* 7(3): 149–174.
15. Sheehy, J. 2010. Why vector clocks are hard. Basho Blog;

http://basho.com/posts/ technical/why-vector-clocks-are-hard/.
16. Sheehy, J. 2015. There is no now. *acmqueue* 13(3): 20-27.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Carlos Baquero *is assistant professor of computer science and senior researcher at the High-Assurance Software Laboratory, Universidade do Minho and INESC Tec. He obtained his MSc and PhD degrees from Minho Universidade do in 1994 and 2000. His research interests are focused on distributed systems, in particular causality tracking, data types for eventual consistency, and distributed data aggregation. He can be reached at cbm@di.uminho.pt and as @xmal on Twitter.*

Nuno Preguiça *is associate professor in the Department of Computer Science, Faculty of Science and Technology, Universidade NOVA de Lisboa, and leads the computer systems group at NOVA Laboratory for Computer Science and Informatics. He received a PhD in computer science from Universidade NOVA de Lisboa in 2003. His research interests are focused on the problems of replicated data management and processing of large amounts of information in distributed systems and mobile computing settings. He can be reached at nuno.preguica@fct.unl.pt and as @nunopreguica on Twitter.*