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

Oliver Vecchini

5th March 2019

Abstract

The following is a summary of the contents, context, and consequences of a seminal paper in theoretical computer science, namely '*Time, Clocks, and the Ordering of Events in a Distributed System*', written by L. B. Lamport and published in 1978 [1]. It is submitted as a part of the assessed coursework component of the COMP0007 'Directed Reading' module at UCL.

1 Background

The 1960s saw growing interest in several concepts that would together later come together to form the field of distributed systems. Computers were being connected together to form networks, even across long distances; the ARPANET connected computers across the U.S. when established in 1969 [2]. Meanwhile, the utility of connecting computers to increase throughput was realised at a similar time, with the McROSS air-traffic simulator program forming a primitive distributed program by moving across the ARPANET and using other computers' idle cycles for calculations to increase throughput [3].

The theoretical study of distributed (or more generally, concurrent) systems began with E. W. Dijkstra's 1965 paper [4], presenting a solution to the *mutual exclusion problem*, the crux of which centers on ensuring that two concurrent processes do not enter their *critical sections* (for instance, access shared resources) at the same time. Dijkstra's paper presents the first ever algorithm to solve the problem for an arbitrary number of processes (an algorithm for the 2-process case was devised by T. J. Dekker several years prior [5]), a simple 'busy-waiting' solution wherein each process essentially 'waits its turn' until a permission token is free. While this does technically solve the problem, it makes the non-trivial assumption that underlying (read-and-write) machine operations are atomic; in addition, the target of the allocation of permission is randomly determined, and so is neither fair nor starvation-free (i.e. some processes may never progress).

While other solutions to Dijkstra's problem had since been published, Lamport himself wrote a solution in 1974, the 'bakery' algorithm, with several useful properties [6]. The premise is that successive processes attempting to enter the critical section are allocated successively increasing numbers, and at any one time only the process with the lowest number may enter its critical section. The evident FIFO fairness of the algorithm ensures starvation-freedom, however even more significant is that the correctness of the algorithm is not reliant on correctness of the value of memory read operations when they overlap with memory writes;

that is, the algorithm is able to implement mutual exclusion without requiring it of the underlying machine operations, the first algorithm of its kind to do so. In addition, it allows for the failure of individual components without impedance of the others, again a new feat.

Mutual exclusion is an example of a problem that can be solved with the principles of Lamport's paper. Another is the problem of synchronising physical clocks; distributed systems often have difficulty even approximately synchronising these due to message latency between nodes. Although he cites some general techniques observed beforehand [7], it appears that prior to his's paper no explicit (major) algorithm existed for synchronising physical clocks between nodes of a distributed system.

2 Contents of the article

Lamport begins with some definitions, namely of a *distributed system*, consisting of processes that are spatially separated and communicate by messages (with the resulting implication that messages take non-zero time to transmit), followed by the more formal definition of the *happened-before* relation, but not before some preliminary definitions. An *event* represents some aspect of execution, such as a machine operation or the sending/receiving of a message, and a *process* is defined as a set of events with a total ordering of time of occurrence (i.e. a sequence of events). The happened-before relation is thus defined as being the smallest relation satisfying the following: $a \to b$ indicates that the event a happened before the event b if both are in the same process and a occurs before b, or if a is the sending of a message and the b is the receipt of that message, with the additional axioms of transitivity and irreflexivity, thus forming a (strict) partial ordering. The novelty of this relation is that this ordering between events is not based on real time using physical clocks, but the ability of one event to influence another; this is likened to the notion of causality from special relativity.

Lamport then defines the notion of a *logical clock*, continuing the trend of disassociation of events from real time. For each process P_i, these are defined as a function of the form $C_i : Event \to \mathbb{N}$, where $C\langle a\rangle$ represents the 'logical time' or 'timestamp' at which a occurs, and more closely resembles a counter than a physical clock (starting at 0 for the first event of the process). The intent of the happens-before relation is expressed by the *Clock Condition*, that is, $if \quad a \to b \quad then \quad C\langle a\rangle < C\langle b\rangle$. To satisfy the Clock Condition, two implementation rules are required (each corresponding to the two conditions of the happens-before relation):

1. Upon executing an event in P_i, C_i is incremented by one.
2. When a message m is sent from P_i to P_j (as event a in P_i),
   (a) Timestamp $T_m = C_i\langle a\rangle$ is included in m.
   (b) C_j is assigned the value of $max(C_j, T_m)$.

These implementations rules (and the clocks they form) are then used to provide a method for constructing a total ordering of all events across all processes in the system, namely by using an augmented version of the happened-before relation such that, for events a, b in processes P_i, P_j respectively, $a \Rightarrow b \quad iff \quad (C_i\langle a\rangle < C_j\langle b\rangle \ or \ (C_i\langle a\rangle = C_j\langle b\rangle \ and \ P_i < P_j))$, for some arbitrary total ordering on processes (at the detriment of fairness), for instance by process ID.

Lamport illustrates the utility of totally orderering all events in a distributed system by providing an algorithm to solve a synchronisation problem, namely the mutual exclusion problem in the context of a distributed system. The gist of the algorithm is based on the

principles of the FIFO 'bakery' algorithm presented earlier, with availability-checking replaced by processes sending request/release messages to eachother to request entrance of the critical section; the validity of the messaging system is provided by the total ordering provided by the logical clocks. By assuming that all messages sent by each process are received by their recipients in the same order, and also that all messages eventually arrive, Lamport claims the correctness of the protocol he presents (not displayed here for brevity's sake; please refer to the fourth page of [1]). Lamport proceeds to verify correctness of the protocol, and comments on it, noting that the synchronisation of the processes is akin to the parallel execution of a particular State Machine with the same sequence of commands (request/release messages), and thus the power of this mechanism in being able to implement any system as a distributed system. He also notes the lack of robustness, in that the failure of one process halts the entire system. Regardless, the solution presented also provides starvation-freedom.

A flaw of logical clocks is discussed, namely that expected orderings of events (e.g. based on order of occurrence in real time) may be inconsistent with a (logically valid) ordering given by $\Rightarrow$. The two discussed solutions are firstly to allow the explicit indication to the system of potential for causality between two events, and secondly to expand $\rightarrow$ and the Clock Condition to account for non-system events such that $a \mapsto b$ *iff* a causally occurred before b (the 'Strong Clock Condition'), clearly impossibly by logical clocks alone.

To address this, the principles discussed so far are expanded to consider physical clocks, as well as to provide an algorithm for approximate synchronisation of physical clocks in a distributed system, seemingly a first of its kind. Concisely, he states that if the rate of communications and the delay of messages can be controlled, then the 'drift' of two physical clocks can be concretely controlled by an upper bound ϵ.

3 Legacy

The area of distributed systems has greatly expanded since the development of the ARPANET into the modern internet; while generally widespread, powerful examples include SETI@home, with multiple petaFLOPS of computational capacity expendable [8]. Lamport's logical clocks have seen widespread use through distributed systems; again, while generally omnipresent, modern prominent examples would include Google's Paxos consensus protocol [9], and Amazon's Dynamo distributed database (in the form of the vector clocks described below) [10]. Expansions have made to the three major concepts of Lamport's paper, namely the logical clocks system, the distributed mutual exclusion algorithm, and the physical clock synchronisation algorithm; these are listed below.

Lamport's logical were extended to the notion of *vector clocks* in 1987 [11], forming a more powerful mechanism for synchronisation. Here, clocks instead take the form $C_i : Event \rightarrow \mathbb{N}^P$, where the number of processes in the system is P. Each process is assigned a vector clock, where $V_i[i]$ is the number of events passed in P_i, and $P_i[j] \quad (i \neq j)$ is P_i's knowledge of the number of events passed in P_j. As it turns out, there exist implementation rules that allow vector clocks to satisfy the Strong Clock Condition; these rules are:

1. Upon executing an event in P_i, $V_i[i]$ is incremented by one.
2. When a message m is sent from P_i to P_j,
   (a) Timestamp $T_m = V_i$ is included in m.
   (b) For all $1 < k < P$, $V_j[k]$ is assigned the value $max(V_j[k], T_m[k])$.

The distributed mutual exclusion problem has seen many further solutions since Lamport's first provision. Several expand directly upon his work: the Ricart-Agrawala algorithm reduces message complexity by merging the release and request-acknowledgement messages into a 'reply' message (since a request will only ever be granted when another process exits its critical section, that process's release message might as well include its acknowledgement of the other process's request); the Roucairol-Carvalho optmization allows a process to leave and re-enter its critical section multiple times until it sends the appropriate reply message to other requests; and the Maekawa algorithm reorganises processes into request sets (*quorums*) and uses other messages to reduce message complexity to a factor of $\sqrt{N}$ for N processes, although with large constant coefficients. Other approaches also exist that avoid Lamport's timestamp mechanism [12].

Lamport himself made further contributions to the study of physical clock synchronisation, providng more algorithms and also showing that at least 4 clocks are required for indvidual-fault tolerance in synchronisation [13]. Other examples include Cristiano's algorithm, the Clegg-Marzullo algorithm, and many others; however a large number of these rely on central time-servers, as opposed to the 'decentralized' style of Lamport's [14].

References

[1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[2] Michael Hauben. History of arpanet. *Site de l'ISEP*, 2007.

[3] Robert H. Thomas and D. Austin Henderson. Mcross: a multi-computer programming system. In *AFIPS Spring Joint Computing Conference*, 1971.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[5] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.

[6] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[7] C Ellingson and Richard Kulpinski. Dissemination of system time. *IEEE Transactions on Communications*, 21(5):605–624, 1973.

[8] David P Anderson. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 2002.

[9] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[10] DeCandia. Dynamo: Amazon's highly available key-value store. *SIGOPS OS Rev.*, 2007.

[11] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering.* Australian National University. Department of Computer Science, 1987.

[12] Martin G Velazques. *A survey of distributed mutual exclusion algorithms.* CSU, 1993.

[13] Leslie Lamport and P Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM (JACM)*, 32(1):52–78, 1985.

[14] Matt Clegg and Keith Marzullo. Clock synchronization in hard real-time distributed systems,". *UCSD Technical Report CS96-478*, 1996.