The physics of distributed information systems

Erik Aurell
Aalto University School of Science, Helsinki, Finland, ACCESS Linnaeus Centre, KTH, Stockholm Sweden and Dept. Computational Biology, AlbaNova University Centre, 106 91 Stockholm, Sweden
E-mail: eaurell@kth.se

Abstract. This paper aims to introduce Distributed Systems as a field where the ideas and methods of physics can potentially be applied, and to provide entry points to a wide literature. The contributions of Leslie Lamport, inspired by Relativity Theory, and of Edsger Dijkstra, which has the flavor of a growth process, are discussed at some length. The intent of the author is primarily to stimulate interest in the statistical physics community, and the discussions are therefore framed in a non-technical language; the author apologizes in advance to readers from the computer science side for the unavoidable impreciseness and ambiguities.

1. Distributed Systems
This paper aims to present distributed systems as a new (interesting) area of applications of statistical physics, and to make the case that statistical physics can be quite useful in understanding such systems. Most working physicists, as members of the informed general public, are most likely aware of the practical and societal importance of distributed systems. But relatively few physicists are familiar with the problems and main results of the branch of Computer Science called Distributed Systems. To mitigate this state of affairs, to some extent, is the main motivation of this paper.

Our presentation of Distributed Systems is frankly biased and instrumental. For the interested reader, who would like to go further, I recommend the monograph of Tannenbaum and van Steen [1], or the more introductory text of Tel [2]. For personal perspectives I also recommend selected chapters in the collection edited by Edsger W. Dijkstra [3] and the catalogue raisonné of Leslie Lamport [4]. With these caveats, a suitable point of departure is a famous quote, from an e-mail sent by Leslie Lamport in May 1987 [5], which states

A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.

To a physicist, the phrase “a computer you did not even know existed” suggests a division of the world into events inside and outside a light-cone of possible causes. Indeed, the physical light-cone sets the absolute upper limit of information transfer; in a real distributed system however, information transformation is limited also by processing at the sender and at the receiver, as well by the finite speed over wires or wireless connections. It is not hard to imagine a geometry of delays between a set of computers, such that there is a minimum delay $\tau_{AE}$ from computer

In this paper I refer to the academic discipline as Distributed Systems, and to the objects under study as distributed systems.

Published under licence by IOP Publishing Ltd
A to B, and that events in computer A at time \( t \) can hence only influence events at computer B at time \( t + \tau_{AB} \), or later. In the time interval \( t - \tau_{BA} < t' < t + \tau_{AB} \), events at B at \( t' \) cannot be influenced by events at A at \( t \), and neither can events at B at \( t' \) influence events at A at \( t \). Such events are called concurrent in Distributed Systems, and the problems facing a designer or manager of a system with concurrent events is called concurrency \(^2\). By analogy with Relativity, in Physics we would call such events simultaneous, with respect to the particular geometry of delays. The space-time picture has been a major influence in Distributed Systems, as we will see below in Section 3.

The first and last notions in Lamport’s definition, “one in which the failure” and “render [...] unusable”, stress that things may fail, and that failures have unexpected, and negative, consequences. Failures and how to handle them is a classic problem in Distributed Systems. In section 4 I will describe how a very large numbers of simultaneous failures can be handled by algorithms called “Byzantine” after [6], and which certainly appear to the neophyte to be of Byzantine complexity. I will also describe other classic algorithms that handle (non-Byzantine) failures, some of which are at least conceptually simple, and even have a physical flavor. From today’s perspective, in a very large system concurrent changes are unavoidable, and a failure is just a name for a change which we do not like. The process by which a large system is continuously subject to change is called churn. A system under churn differs from a system with changes isolated in space-time in that under churn the system, as a whole, does not have time to recover from a change (somewhere) before another change occurs (somewhere else). How to cope with churn is a main topic of Distributed Systems over the last decade or so, and I will give a few suggestions for further reading in that direction.

The outline of the paper is as follows. In Section 2 I introduce in a non-technical manner the concept of state machines, and in Sections 3 and 4 I describe work by Leslie Lamport (mainly) and others (partly) in this direction, still in a non-technical manner. In Section 5 I introduce the distributed self-stabilizing algorithm of Edsger Dijkstra and the concept of “self-star”, and in Section 6 I give some further pointers to the literature.

2. State machines, and the replicated state machine approach
I base the discussion in this Section on the concept of a state machine. I distinguish between a big state machine (one for each system) and small state machines (one for each element in the system) \(^3\). At this point, the physically inclined reader can think of a big state is a microstate, a big state machine is a complete description of the time evolution of a microstate, and a small state machine is the local dynamics of local state variables.

Algorithmic complexity theory lies outside the scope of this review, but as a motivational device I invoke the Turing machine, which most readers will know as a (the) universal model of computation. A Turing machine reads and writes on a tape. The state of a Turing machine is specified by the content of the tape, an internal state, and the position of the read/write head on the tape. Most (and perhaps all) computational devices can similarly be described by a set \( S \) of states, a set of initial states \( I \), and a next-state relation \( N \), such that \( I \subseteq S \) and \( N \subseteq S \times S \). Such an abstract device is called a state machine, and the computations it can perform are sequences of states \( s_1 \to s_2 \to s_3 \to \cdots \), or state behaviors \([7]\). In a deterministic state machine, e.g. a Turing machine, the next-state relation is a function.

A distributed system can be seen as one big state machine, where \( S \) is the set of internal states of all its elements (including, if so desired, the state of the communication medium), \( I \) is the set of initial states and \( N \) specifies the combined behavior of all the elements. The reader can imagine \( S \) as the set of spins in a lattice and \( N \) as some lattice dynamics. This is a

---

\(^2\) I here base concurrency on which messages could have been sent. The alternative definition, based on messages actually sent, is taken up in Section 3.

\(^3\) May I be excused of this notational innovation.
deterministic state machine if the spins are always updated in the same order, and if the local update rule is deterministic; otherwise the state machine is non-deterministic. A distributed system can also be seen as many small state machines, one for each element, where \( S_i \) is the set of internal states of element \( i \). The reader can in this case think of a single spin in a lattice. In contrast to the big state machine picture, the many small state machine picture is incomplete, since in all cases of interest the next-state relation of element \( i \) depends on the states of at least some other elements. The reader may think of of the update rule of a single spin which depends on the neighboring spins.

Distributed systems are typically open systems, which interact with an external world (communication media, hardware states, system users, other distributed systems etc.). While it would be possible to include the relevant states of the external world in the state description, as in Physics it is often more convenient to separate the two. To the set of states is then added a set \( \mathcal{A} \) of actions, such that the next state relation is a subset of \( S \times \mathcal{A} \times S \), and a state-action behavior is a sequence \( s_1 \rightarrow \alpha_1 s_2 \rightarrow \alpha_2 s_3 \rightarrow \alpha_3 \ldots \) \([7]\). If the state-action machine is deterministic, then the state machine will change its state in one way only under a given action, and the state machine then obeys the actions as externally issued commands. If the state-action machine is not deterministic, then the action sequence does not determine the state behavior, but only which state behaviors are possible.

I separate big states and small states by the convention that in a big state actions are only generated by events deemed external to the system. The absence of any external events is a null event, which also generates an action, and the big state can therefore change, even if there are no external events (but it does not have to do so). In a small state, actions can be generated both by external events relevant to this element, and by other elements in the system changing their states. The set of all the small state-action machines can obviously be as a complete description (if done correctly) of the system as the big state-action machine. When describing a real distributed system it is largely a matter of art to chose what set of states to include as small states in the big state picture, and which states of the world will be deemed external to the system.

As touched upon above, distributed systems have several natural failure modes which distinguish their study from other computational devices. One is that some elements may fail entirely. This is similar to some spins in a lattice for some reason using the wrong (or random) update rule. Another mode is that the elements and/or the connections between them may work at different rates, and so get unsynchronized. This is analogous to the spins in a lattice updating in random order, and some spins updating twice or many more times before some other spins have even updated once, which, if one is not careful, could lead to incorrect thermodynamics. It is also largely a matter of art whether a particular deviation from a normative behavior should be described by an action, or as a failure mode of the system.

The reader may at this point think of a system which is large, but not in the thermodynamic limit, and on which are placed strict conditions that some bad behavior should never occur. More concretely, the reader may think of the servers running the on-line banking system of a very large bank, and the unwanted behavior that some client manages to withdraw more money than he or she has in his or her account. Such a system can consist of hundreds or thousands of servers in many countries. At least sometimes some of these servers may be down (temporary failures), most likely the servers are not identical (running at different speeds), and almost surely there are unpredictable delays in the inter-server communications (which translates to different effective speeds, and perhaps also to effective failures). Showing that the conditions are fulfilled can then be translated as a Distributed Systems task of the type “Prove that distributed system \( X \), if it has property \( S \) at time \( t_0 \), will have property \( A \) for all times \( t > t_0 \)”.

As Physics problems, such tasks have, if anything, the flavor of Mechanics. One way to proceed (in a physical setting) would be to show that the time evolution conserves some invariant.
For example, if a spin update rule preserves an energy function, and if energy is positive at time $t_0$, then energy will be positive for all time. Another way to proceed would be to show that although the system is not forbidden by invariants to reach a certain state, in fact, if one follows the time evolution, one can see that it can never do so. This is obviously the more difficult assignment, and which depends more on system details. Arguably, Mechanics is rather far from the on-line banking example, however, both ways to proceed do have parallels in Distributed Systems, where reasoning about invariants is called an *assertional proof*, and reasoning about the time evolution a *behavioral proof*, and similarly to Mechanics, an assertional proof is much preferred, when it is available. In actual practice, both assertional and behavioral proofs in Distributed Systems can be quite fantastically complicated, even for problems which at first glance look fairly straightforward, and such proofs often cannot be done manually. Physics has few examples to offer of such reasoning, and will hence have little to say to such problems.

On the other hand, if satisfying a condition on no bad behavior is the most important property we want from a system, then Distributed System has the option to change the system under study until the conditions are met. Alternatively, the system can be modified to a kind of normal form, and then the task is to show that some kind of generic condition is satisfied by systems in normal form. The main approach in this direction is called *state machine replication*. This means that one and the same state machine is faithfully replicated either in all the elements of a distributed system, or in a qualified majority of them, or in some other suitable number, depending on the scenarios in which the conditions are to be satisfied. At this point, I leave that number vague; examples will follow. Suppose for illustration that the state machine to be replicated is simple. For instance, to borrow an example from [8], suppose the state is how much money a client has in his or her account and that the client issues withdrawal requests. Leaving out of the description how the client is informed if his or her request is granted, and how he or she actually gets the money, then the state is just given by one number $c \geq 0$, and the request is an action represented by another number $r > 0$. The good behavior is described by a deterministic state machine where $c \rightarrow_r (c - r)$ if $r \leq c$, and $c \rightarrow_r c$ if $r > c$. That the state machine is replicated across a number of computers in a big bank means that they all (or a qualified majority, or some other suitable set) hold the same number $c$, and respond to the same action $r$. If so, the computers will agree between themselves how much money the client has in the account, and the client will not be able to withdraw more than that amount.

The question in normal form is hence whether a state machine can be suitably replicated when the elements of this system run with different speeds and with failures. The answer to this question is a “yes” – provided that the elements properly wait for one another, and/or that there are enough back-up copies to handle failures. I will describe what it means for elements to properly wait for one another, and then how many back-up copies are needed to handle different kinds of failures. But let us also remark that while this approach is general and versatile it also limits what the full system system can do to what one of its components could also do on its own. The main interest is therefore either to build extremely fault-tolerant systems, since the system as whole can be much more robust to failures than one of its elements, and to critical aspects of a system, where the elements must agree – even if they disagree about other things.

3. The space-time view of concurrency

Leslie Lamport, known to most physicists as the designer of \LaTeX, is also the author of a paper entitled “Time, Clocks, and the Ordering of Events in a Distributed System” [9], which, although obviously inspired by Special Relativity, is surprisingly little known among physicists. Lamport solves the problem of replicating any state machine by a distributed system where elements run at different speeds but without failures. The method works by loading timestamps on all commands

---

4 In fact, by an informal survey, the author has not found a single colleague physicist who has read this paper.
and all messages issued by and sent between the elements. Quite strikingly, Lamport begins by pointing out that a set of elements which send messages between themselves can be illustrated by a space-time diagram as in Figure 1. The wide use of space-time diagrams to describe and

![Space-time Diagram](image)

**Figure 1.** This figure reproduces the left panel of Fig 1 of the paper “Time, clocks and the ordering of events in a distributed system” by L. Lamport published in Communications of the ACM 21, 7 (July 1978), 558-565 [9]. Vertical lines are time lines of processes (or processors, or the entities of any distributed system under study) where time is running upwards. Nodes on the time lines are events in the respective processes, event count in each process increasing with time. Diagonal lines are time-and-space traces of messages sent between processes. The line event $p_1$ (in process $P$) to event $q_2$ (in process $Q$) is thus a message sent from process $P$ to process $Q$. Messages may travel at different speeds between processes and may arrive out of order as illustrated by the pair $q_1 \to r_4$ and $q_4 \to r_3$.

reason about communication protocols originates, as far as this author understands, in this paper from 1978 [9, 10]. An event $a$ can causally affect event $b$ if there is a path from $a$ to $b$ in the space-time diagram, always going forwards along world-lines and messages. As in Relativity, potential causality imparts a partial ordering of events. If neither $a$ can causally affect $b$, nor $b$ causally affect $a$, then $a$ and $b$ will be called concurrent events. Note that this notion of concurrency depends on the messages actually sent, i.e. on the space-time diagram.

A key insight of Lamport was that even without proper relativistic effects, there is no invariant total ordering of events in a distributed system. If two elements actually know absolute Newtonian time they will agree on event order. However, time must be measured by physical clocks, one in each element, and such clocks run at different rates. Unless the clocks be synchronized and all relevant events sufficiently well separated in time, two elements in the system can disagree on the ordering of some pair of events. Furthermore, physical clocks are expensive, and more expensive the better they keep time. Therefore, it is of interest to consider as bad clocks as possible, which Lamport calls “Logical Clocks”. Assume that all events in a
single element are totally ordered, e.g. by an advancing event counter. A set of clocks \( C_i \) with readings \( C_i(a) \), where \( i \) is an element in a distributed system and \( a \) is an event which happens in that element, running at otherwise arbitrary speeds, respect the partial ordering of events when they obey the following two conditions:

**C1** if \( a \) and \( b \) are events in the same element \( i \), and \( a \) comes before \( b \), then \( C_i(a) < C_i(b) \);

**C2** if \( a \) is the sending of a message from element \( i \) and \( b \) is the receipt of the message by element \( j \), then \( C_i(a) < C_j(b) \).

Respecting the partial ordering of events means that if \( a \) can casually affect \( b \), then \( C_i(a) < C_j(b) \). Condition **C1** can be met by advancing the clocks \( C_i \) at least as quickly as the event counters, but it will be appreciated that a set of event counters will not satisfy condition **C2**. However, by introducing a timestamp \( T_m \) of every message \( m \), conditions **C1** and **C2** can be met by the following two further implementation rules:

**IR2(a)** if \( a \) is the sending of a message \( m \) from element \( i \), then \( m \) contains a timestamp \( T_m = C_i(a) \);

**IR2(b)** upon receiving message \( m \), element \( j \) sets \( C_j \) to be greater or equal to its present value, and greater than \( T_m \).

Lamport extends the ordering of events to a total ordering by using logical clocks and any ordering between the elements in the system. Event \( a \) in element \( i \) is then before event \( b \) in element \( j \) if either \( C_i(a) < C_j(b) \) or \( C_i(a) = C_j(b) \) and element \( i \) is before element \( j \). This ordering of events is not invariant, and does not necessarily agree with the order of occurrence of events in Newtonian time. However, it allows any state machine to be simulated by requiring that an element can only execute a command timestamped \( T \) when it has learned of all commands issued by all the other elements with timestamps less than or equal to \( T \). It is therefore a kind of general “waiting for the others” scheme.

In the toy example of above, at time \( T \) any of the bank’s computers \( i \) could receive a request of withdrawal for the amount \( r \) (a deposit could be a withdrawal of \( r < 0 \)). The computer has its own state \( c \) and if \( r > c \) it could just refuse the request and let the client know this. Hence, assume that \( c \geq r \), such that the request looks like a valid one to the computer receiving it. The computer then timestamps this request with \( T \), and puts it on a queue. The action to be taken in response to the client’s request is, when this action is taken, to withdraw money from the account (and send the money to the client) or to refuse the request (and inform the client). The message computer \( i \) needs to send to the others when it grants such a request is therefore “The client has withdrawn \( r \) from the account from me. New balance is \( c \).” That message is timestamped with \( T_r \), the time at which this computer takes this action. In response to such a message, the other computers will also update their value of \( c \) and send the messages “Following the action of \( i \), I have set the balance to \( c \).” timestamped by \( T_a \), the time at which this acknowledgment was sent. The receiving computer puts both kinds of messages together with the client requests on its queue. At any one time, the computer’s queue hence contains a number of pending commands and acknowledgments. That list is ordered such that if two items have different timestamps, the one with earlier timestamp comes first, and if two items have the same timestamp, then a client request received by computer \( i \) takes the order of computer \( i \) itself, while a message from computer \( j \) takes the order of computer \( j \). Computer \( i \) removes any number of leading acknowledgments from the list, and looks at the first command and its timestamp \( T_f \). By above, it cannot execute this command until it has learned of all the other commands in all the other computers with timestamps less than or equal to \( T_f \). This means that there is at least one message on its list timestamped later than \( T_f \) from all computers ranked

---

5 This is formalized as implementation rule IR1 in [9].
before, and at least one message timestamped $T_f$ or later from all computers ranked after. If now computer $i$ can execute this first command and it is a message from another computer, then it updates the account to the new value $c$, removes the first command from the list, and sends the acknowledgment message to all the other computers. If, on the other hand, the first command is a client request put on the list by computer $i$ itself, then computer $i$ takes the appropriate action (based on $r$ and its current value $c$), sends a message to the other computers if the request is granted, and removes that command from the list. One can reason about the correctness of this protocol by following the fate of a request. First, a computer never grants a request unless it thinks itself that this is OK. Second, it never grants a request until it has learned of all the withdrawals granted by the other computers and which are timestamped earlier, and of all of the withdrawals with the same timestamp granted by computers which come before. Third, from the opposite direction, until the computer has dealt with the request, no other computer will grant a request which is timestamped later, or which has the same timestamp but arrives in a computer which comes later. This is so because when the computer deals with the request it sends a acknowledgment message with this timestamp, and all the other computers have to wait for a message with such a timestamp. It has been assumed that no computers fail, no messages are lost, and no messages are received out of order.

The paragraph is an example of a behavioral proof written informally. The subtlety of Lamport’s procedure is perhaps better appreciated by the concrete example he presents, which is that of a fixed collection of elements sharing a single resource (a problem which will be taken up in another version and from another angle in Section 5). Suppose that the resource ought to be granted to the elements in a manner which satisfies the following three conditions:

I An element which has been granted the resource must release it before it can be granted to another element;

II Different requests for the resource must be granted in the order in which they were made;

III If every element which is granted the resources eventually releases it, then every request is eventually granted.

The solution is built on timestamped request and release commands and messages. Note that as in the toy example, the order of requests in condition II above is then that of the total ordering. But why such complexity? For one thing, a central scheduling process, which grants requests in the order in which they are received, will not satisfy condition II. The interested reader is encouraged to look at [9] for the (short) proof that this is impossible, as well as the (slightly longer) description of the distributed algorithm which does work.

What about physical clocks? This question is also treated in [9] by asking how small $\epsilon$ in a synchronicity condition $|C_i(t) - C_j(t)| < \epsilon$ can be, if it is to hold for all clock pairs $C_i(t)$ and $C_j(t)$ and for all time $t$, and if single clocks, running without resetting, are accurate to a fraction $\kappa$. Note that the clock speeds are assumed to vary unpredictably within the bounds set by $\kappa$ as otherwise the clocks readings could be re-calibrated. Note also that a naive algorithm, with a master clock which the other clocks adjust to periodically after receiving messages transmitted with maximum delay $\xi$ every $\tau$ seconds will achieve $\epsilon \approx 2\kappa \tau + \xi$ (the master sends its time to element $i$ which receives it $\xi$ seconds later, and while waiting for the next time, element $i$ and the master can drift an additional $2\kappa \tau$ seconds apart). Such an algorithm will however not satisfy clock condition C1 (if clock $C_i(t)$ has outrun the master clock and needs to be set back), and requires that all clocks communicate directly with the master. I refer to to [9] for a precise statement of the theorem Lamport proves, and just mention that if messages can be sent on every link every $\tau$ seconds, and under other suitable assumptions, then $\epsilon \approx d(2\kappa \tau + \xi)$, where $d$ is the diameter of the graph. In a complete graph ($d$ equal to one), a fully distributed algorithm which obeys both clock conditions C1 and C2, and where all elements send the same number of messages as the master in the naive algorithm, hence achieves the same synchronicity.
4. Coping with failures
One way to cope with failure is to have a back-up, and, in some sense, this must be the only way. However, things can fail in different ways, and they can fail singly or fail together. For instance, trivially, if some elements fail together, then there must be at least one back-up left. This Section will describe methods invented to deal with single, multiple, and more or less malicious failures in distributed systems which replicate state machines.

The main distinction is between an arbitrary failure and limited failure mode. An arbitrary failure is called Byzantine after the parable of the Byzantine Generals told in [11], and other failures are accordingly called non-Byzantine. A collection of nodes which fails in a Byzantine manner can be imagined to have been taken over by an adversary who is malicious, i.e., has the objective to disrupt the function of the state machine. The adversary must not all-powerful, e.g., as in the method described below, be computationally limited in the style of cryptographic analysis, but is at least able to send any message he wants at any time to any of the other nodes. As told by Lamport in [12] the term “Byzantine” was inspired by the “Chinese Generals Problem” and the problem was originally called “The Albanian Generals Problem” but modified to “Byzantine” lest no Albanian would be offended. As an aside, the dictionary term “Byzantine” can refer to an inhabitant or to the art of Byzantium, the Eastern half of the Roman empire, which survived until 1453 A.D., but also has, in most European languages, the meaning of a procedure of great and even dizzying but largely or completely unnecessary complexity. The use of Byzantine in [11] was certainly apt also from this perspective. Starting with the Oral Message (OM) algorithm of [11] (which will not be described here), these methods are complex in themselves, and later improvements (of which I will describe one) were intended for use in real implementations. The original publications can therefore describe what is in effect system designs (even benchmarked prototypes), and typically then contain several protocol variants and incremental additions. A presentation in full detail is out of scope of this paper; the reader should however be aware of that there is more (and sometimes much more) in the sources we follow, mainly [13].

Let us start by considering a distributed system of \( N \) small state-action machines, and ask how many machines \( f \) with Byzantine failures can the system tolerate. The external actions address one, some or all small state machines simultaneously and identically, and are called requests. The system functions correctly if sufficiently many of the small state-action machines change state correctly in response to an action in a finite amount of time. These properties are called safety and liveness. State change is witnessed by a response. Requests and responses can be thought of as messages exchanged between the system and an external client, and can be specified as in [13]. The property to have failed is externally given. Mechanisms for a failed node to recover are not included in the description. The question could therefore perhaps be more properly phrased as what is the smallest number \( N_r(f) \) of correctly functioning small state-action machines necessary that the system can cope with \( f \) faulty machines. That is, all the \( N_r(f) \) correctly functioning machines stay correct in the time interval between a request and a response.

Contrary to what one would perhaps naively expect, \( N_r(f) = 2f + 1 \) (not \( f + 1 \)). This result, first shown in [6], follows from the following argument in [13]. If the client gets \( N_r(f) \) messages as responses, then this must be enough for the client to be sure that the system has executed the state transition correctly. This is a liveness requirement, because the failed machines may fail in the way of not sending any responses at all. However, any of the failed machines may also fail in a way where it does send a response (Byzantine failure), and the client has no way of knowing which of the \( N_r(f) \) messages it has received which are from the failed ones. Hence, the responses from the correctly functioning machines must always outnumber the failed ones, within a total number of \( N_r(f) \) responses. This means \( N_r(f) > 2f \), which is a safety requirement. The result is usually quoted as \( N = 3f + 1 \).
To argue the correctness of more complex algorithms of this nature is not easy, even if many more details are given, and I have to refer the interested reader to [13] for an example where there are three rounds of messages exchanged among the machines themselves, a number of messages exchanged growing quadratically with the number of machines, and with three checkpoints, two among the machines, and one at the client when it receives the responses. Nevertheless, this algorithm has been reported to work well as a basis for a distributed file server (BFS, a Byzantine-fault-tolerant NFS service) with only a quite small overhead compared to the standard NFS system [13].

5. The self-star approach
Edsger Dijkstra more than a generation ago invented an algorithm to ensure cooperation between processors without central control [14]. Dijkstra takes cooperation in general to mean that the system is some set of legitimate states, defined such that one out of processors cooperation.

NFS system [13]. By Byzantine-fault-tolerant NFS service) with only a quite small overhead compared to the standard NFS system [13].

The self-star approach
Edsger Dijkstra more than a generation ago invented an algorithm to ensure cooperation between processors without central control [14]. Dijkstra takes cooperation in general to mean that the system is some set of legitimate states, defined such that one out of processors cooperation.

NFS system [13]. By Byzantine-fault-tolerant NFS service) with only a quite small overhead compared to the standard NFS system [13].
$c'$ is positive it sets $c_i$ equal to $c'$, erases the requests from the ledger, and sends something like “Requests $r_1, r_2, \ldots, r_n$ accepted, new balance is $c'$”. Otherwise the processor looks for the largest $k$ such that the partial sum $c' = c_1 - r_1 - r_2 - \cdots - r_k$ is positive, sets $c_i$ to this $c'$, erases requests $r_1, r_2, \ldots, r_k$, and sends something like “Requests $r_1, r_2, \ldots, r_k$ accepted, new balance is $c'$. Requests $r_{k+1}, r_{k+2}, \ldots, r_n$ still pending, please put more money in the account”. Only after doing all this the does processor $i$ set $x_i$ equal to $x_{i-1}$ (processor $0$ sets $x_0$ equal to $(x_0 + 1) \mod K$). If the processor $i$ receives another request while it is has the privilege it is put on a buffer, and dealt with after the processor has given up the privilege. If the state is legitimate it cannot happen that the left-hand neighbor of $i$ changes its current account value after processor $i$ has evaluated the condition $x_i = x_{i-1}$ to false, but before processor $i$ has given up the privilege. When the bank closes for the day, the processor with the privilege holds the current account. It can be objected that the requests are not necessarily processed in the same order as the client feels it has issued them. But this was not so in the approach using state machine replication either. It can also be objected that this bank based on Dijkstra’s algorithm does not behave in exactly the same way as the other bank based on Lamport’s algorithm. But both implementations respect in some sense the condition that the bank does not pay out money the client does not have in his or her account, and unless we have given more precise specifications, it is not clear which implementation is better. The real advantage of the Lamport bank is that it will function correctly if some processors fail, while the Dijkstra bank – at least this toy one — will not.

Dijkstra’s paper eventually gave rise to the whole field of self-stabilization, and this idea of “self-” has more recently resulted in “self-management”, “self-configuration”, “self-healing” and even many more “self-”, such as “self-organization”. Hence the present term “self-star” or “self-$\star$” for any system property which is deemed “self-”. I are not aware of any definition of “self-” which is both well established in the Distributed Systems literature, and simple enough for the purposes of this paper. So I offer one:

A distributed system is self-$P$ for some property $P$ if it achieves $P$ without replicating a state machine which has $P$.

Dijkstra’s algorithm is of course “self-stabilizing”, as well as (at least) “self-configuring”, since privilege is not even defined as a property of a small state only.

Physics has quite a few examples which can be thought of as “self-$\star$”. In Statics, a system of weights, springs and spokes is at rest if the forces on each weight balance. This is self-$\star$ because the forces depend on the states of the other weights (e.g. their heights), and typically are not the same on each. A hydrodynamic velocity is self-$\star$, because the single particles move with very different velocities, and hydrodynamic stability and the whirls and jiggles of fluid flow are therefore both self-$\star$. In fact, practically any property in Condensed Matter Physics is self-$\star$, because it is defined by the interactions of the constituents. For instance, to move beyond the classical domain, the lattice mass of a conducting electron in a metal is self-$\star$ (as well as the whole band structure) because the single electrons are particles with another mass.

All the physical properties evoked above are however self-$\star$ because they are macroscopic. A microscopic system of weight, strings and spokes, such as a molecular motor, will not be at rest, because it is driven by thermal processes. Likewise, in a small enough sample of a liquid or a gas, average velocity fluctuates too much to be a useful concept, and a small enough collection of metal atoms will not form a crystal and will not have a clear band structure. Therefore, self-$\star$ properties of physical systems are more or less the same as emergence, and the difference between the microscopic and the mesoscopic or the macroscopic. This then suggests the following:

Physics is useful to analyze self-$\star$ properties of some distributed system which hold only almost surely if the system is large enough.
The work outlined below in Section 6 was done in this spirit. Let us note that Dijkstra’s algorithm as a physical device would also only be “thermodynamically self-*”. Since it does not obey detailed balance it must be a driven system, therefore cannot be at thermodynamic equilibrium at zero temperature, and therefore must have fluctuations. In a microscopic device these fluctuations would translate into occasional mistakes, and such a device would therefore at times leave the set of legitimate states. This suggests then that the effects of failures in a (large) distributed system can be analyzed by the methods used in Statistical Physics to go between the microscopic and the macroscopic, and in particular the mesoscopic. I have found this a fruitful approach.

6. A guide to further reading
The discussion above has attempted to convey the impression that it is dealing with concurrency and failures together with makes Distributed Systems a complex science. I have also sketched that while methods to cope with any number of failures exist, those methods are expensive, and their cost grows fairly quickly with system size. One reason that Distributed Systems is a complicated science is that it has aimed to solve very difficult problems, that is to provide performance guarantees under very general perturbations to the system. In a very large system concurrent changes are unavoidable, and a failure is just a name for a change which we do not like. The process by which a large system is continuously subject to change is called churn. In Physics we are accustomed to much weaker adversaries than in distributed systems (certainly not Byzantine Generals), but on the other hand have techniques to handle very large systems. One does not expect to be able to follow every event on the atomic scale in a liquid or a gas, and even if one could, one would not know how to use that information. But we can hope to predict average properties, which will then almost surely hold in a single, but large enough, system. To do so we must describe the statistically stationary state where changes, of different kinds, and on the average, balance. For a distributed system to function under churn the stationary state must furthermore be one of acceptable expected functionality. Contrary to the possibility evoked by Lamport in the opening quote defining a distributed system, your own computer should not be rendered unusable by a single failure in another computer, or even by many such failures, except at most very rarely. Such a property could loosely be described as robustness.

In the late 1990’ies the field of Distributed Computing became intertwined with Peer-to-Peer computing which was then widely believed to be the future of computing technology and services [15]. Two key theoretical developments were the development of consistent hashing by Karger and collaborators in [16], allowing for distributed storage that functions under churn, and the Plaxton, Rajaraman and Richa scheme [17] to find and access copies in a distributed storage. A storage architecture developed slightly later and incorporating these ideas was “Chord” where processors and data items have logical addresses in a (large) ring, and ring nodes know of each other in a manner which today would be called “scale-free” (though ordered, not random) [18]. Such systems are robust in that they can provide stable services, between most pairs of nodes in the system, even if other nodes in the meantime arrive or leave. On the other hand, some effort has to be expended by the nodes in the system to keep it functioning, such as correcting incorrect links or look-up tables from information received or by actively polling neighbors a node can talk to directly.

Comparing the performance of different Peer-to-Peer architectures (node conductivities) and network maintenance schemes to the efforts needed to keep the network functioning was of interest, and was quite early evaluated by simulations or by probabilistic techniques [19, 20]. However, they can also, and usually more accurately, be evaluated by statistical physics techniques. Chord under churn can for instance be modeled as a generalized dynamic spin model where the spins indicate the states of different nodes in the system, and the change in system state can be modeled by a master equations [21]. This allows for rather precise quantitative
understanding of the performance-cost trade-off, and can be used to compare different network maintenance strategies [22]. Similar techniques can also be used to evaluate the performance of certain network aggregation systems, with excellent results [23].

Acknowledgments
This paper was started as a section in a longer work with Mikko Alava and Supriya Krishnamurthy which never materialized; I thank them for interesting discussions. I also thank Seif Haridi and Sameh El-ansary for discussions on Distributed Systems and Seif Haridi for valuable suggestions. This work was supported by the Academy of Finland through its Finland Distinguished Program, project 129024/Aurell, and through its Centre of Excellence COIN.

References
[1] Tanenbaum A S and van Steen M 2002 Distributed Systems: Principles and Paradigms (Prentice Hall Inc)
[2] Tel G 2000 Introduction to Distributed Algorithms (Cambridge University Press)
[3] Dijkstra E W 1982 Selected Writings on Computing: A Personal Perspective (Springer Verlag)
[4] Lamport L My writings last modified 20 March 2013
[5] Lamport L 1987 Distribution In ‘My Writings’, item 75, Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87
[6] Pease M, Shostak R and Lamport L 1980 Journal of the Association for Computing Machinery 27 2
[7] Lamport L 2008 Computation and state machines In ‘My Writings’, item 165
[8] Lamport L 2005 Generalized consensus and Paxos. Tech. rep. Microsoft Research
[9] Lamport L 1978 Communications of the ACM 21 558
[10] Lamport L 1978 Time, clocks and the ordering of events in a distributed system In ‘My Writings’, item 27
[11] Lamport L, Pease M and Shostak R 1982 ACM Transactions on Programming Languages and Systems 4 382–401
[12] Lamport L 1982 The byzantine generals problem (with marshall pease and robert shostak) In ‘My Writings’, item 46
[13] Castro M and Liskov B 1999 Proc. Third Symposium on Operating Systems Design and Implementations
[14] Dijkstra E W 1974 Comm. ACM 17 643
[15] Oram A 2001 Peer-to-Peer: Harnessing the Power of Disruptive Technologies (O’Reilly)
[16] Karger D, Lehman E, Leighton T, Levine M and Panigrahy R 1997 ACM Symposium on Theory of Computing 654
[17] Plaxton C G, Rajaraman R and Richa A W 1997 ACM Symposium on Theory of Computing 311
[18] Stoica I, Morris R, Karger D, Kaashoek M and Balakrishnan H 2001 ACM SIGCOMM 149
[19] Li J, Stribling J, Morris R, Kaashoek M F and Gil T M 2005 Proc. 24th Infocom pp 225–235
[20] Lichten-Nowell D, Balakrishnan H and Karger D 2002 ACM Principles of Distributed Computing (PODC) 233
[21] Krishnamurthy S, El-Ansary S, Aurell E and Haridi S 2005 Proc IPTPS, Lecture Notes in Computer Science 2650 93
[22] Krishnamurthy S, El-Ansary S, Aurell E and Haridi S 2008 IEEE/ACM Transactions on Networking 16 814
[23] Krishnamurthy S, Ardelius J, Aurell E, Dam M, Stadler R and Wuhib F 2010 PODC ’10 Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (ACM New York, NY, USA) pp 291–292 ISBN 978-1-60558-888-9