Brief Announcement: Synchronization in Anonymous Networks Under Arbitrary Dynamics

Rida Bazzi
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Anya Chaturvedi
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Andréa W. Richa
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Peter Vargas
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Abstract

We present the δ-Synchronizer, which works in non-synchronous dynamic networks under minimal assumptions. Our model allows for arbitrary topological changes without any guarantee of eventual global or partial stabilization and assumes that nodes are anonymous. This deterministic synchronizer is the first that enables nodes to simulate a dynamic network synchronous algorithm for executions in a semi-synchronous dynamic environment under a weakly-fair node activation scheduler, despite the absence of a global clock, node ids, persistent connectivity or any assumptions about the edge dynamics (in both the synchronous and semi-synchronous environments). We make the following contributions: (1) we extend the definition of synchronizers to networks with arbitrary edge dynamics; (2) we present the first synchronizer from the semi-synchronous to the synchronous model in such networks; and (3) we present non-trivial applications of the proposed synchronizer to existing algorithms. We assume an extension of the PULL communication model by adding a single 1-bit multi-writer atomic register at each edge-port of a node. We show that this extension is needed and that synchronization in our setting is not possible without it. The δ-Synchronizer operates with memory overhead at the nodes that is asymptotically logarithmic on the runtime of the underlying synchronous algorithm being simulated – in particular, it is logarithmic for polynomial-time synchronous algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Synchronization, Anonymous Dynamic Networks, Arbitrary Dynamics

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.49

Funding *Anya Chaturvedi*: NSF-CCF-2312537 and 2106917; U.S. ARO (MURI W911NF-19-1-0233).
Andréa W. Richa: NSF-CCF-2312537 and 2106917; and U.S. ARO (MURI W911NF-19-1-0233).
Peter Vargas: NSF-CCF-2312537 and 2106917; and U.S. ARO (MURI W911NF-19-1-0233).

1 Introduction

Modern distributed systems, such as wireless sensor networks, mobile peer-to-peer systems, and biologically inspired swarm networks, often exhibit constantly changing and unpredictable communication dynamics that make achieving coordinated behavior between agents challenging, even for simple tasks. Achieving coordinated behavior in such a setting can be further compounded by asynchrony and agent anonymity (no identifiers). To simplify the design of distributed algorithms, researchers developed *synchronizers* that can transform algorithms designed under strong synchrony assumptions into algorithms that work correctly under weaker assumptions [2, 19, 15]. Previous work on synchronizers considers systems in which agents have unique identifiers and the network is either static or is dynamic for some

time but eventually stabilizes. In contrast, this paper considers a system of anonymous agents that communicate through a network with *arbitrary dynamics*, i.e., with *no restrictions on topological changes*, including *no assumptions on eventual (local or global) stabilization.*

In such networks, three different synchrony models are considered [13]. In the *synchronous* model, time is divided into stages and all nodes are active in every stage. In the *semi-synchronous model* (see, e.g., [14]), time is also divided into stages, but some nodes might not be active in a given stage. In both models, an active node executes one action – which involves communication with their neighbors and a bounded amount of computation – per stage, and topology changes occur only at the beginning of a stage. In other words, the *semi-synchronous* mode is one where time is synchronous but nodes are activated asynchronously. In the *asynchronous* model, there is no synchronization of time nor node activations, so actions can take an arbitrary bounded amount of time to execute, and topological changes and node activations can happen at arbitrary times.

In this paper, we introduce our *Arbitrary-Dynamics Synchronizer*, or *δ-Synchronizer* for short. The δ-Synchronizer is the first *deterministic* synchronizer that allows algorithms designed for a synchronous anonymous network with arbitrary adversarial dynamics to execute correctly in a semi-synchronous anonymous, arbitrary dynamics network under a *weakly-fair node activation scheduler*. Specifically, *our δ-Synchronizer transforms any algorithm $\mathcal{A}_{sync}$ designed for a* synchronous *dynamic network under arbitrary edge-dynamics given by a time-varying graph $\mathcal{G}_{sync}$ into an algorithm $\mathcal{A}_{semi}$ that correctly simulates $\mathcal{A}_{sync}$ under a* weakly-fair scheduler *in a* semi-synchronous *dynamic network under arbitrary edge-dynamics given by a time-varying graph $\mathcal{G}_{semi}$.*

Unlike other synchronizers that assume eventual stabilization and for which one can compare executions of the original algorithm and the transformed algorithm on the same stabilized network, in our setting, there is no guarantee that the network ever stabilizes and there is no guarantee that $\mathcal{G}_{sync}$ and $\mathcal{G}_{semi}$ are identical, which should not be surprising. This necessitates a *non-triviality* requirement on synchronizers for networks with arbitrary dynamics. The overall requirements for such synchronizers are captured by the following three general conditions (in bold), which we then indicate how they are satisfied by our δ-Synchronizer. We formalize our guarantees in Section 4.

- **(Correctness) The simulated synchronous execution is valid**: We show that for any $\mathcal{G}_{semi}$ and $i \geq 0$, there exists a $\mathcal{G}_{sync}$ such that the state of each node at the end of phase i (where phases are maintained by the synchronizer) of the semi-synchronous execution of $\mathcal{A}_{semi}$ under $\mathcal{G}_{semi}$ and a weakly-fair scheduler is equal to the state of each node at the end of i-th step of a synchronous execution of $\mathcal{A}_{sync}$ under $\mathcal{G}_{sync}$. (Theorem 1)
- **(Non-triviality) Every possible outcome of a synchronous execution can be simulated**: We show that for any $\mathcal{G}_{sync}$ and $i \geq 0$, there exists a $\mathcal{G}_{semi}$ such that the state of each node at the end of step i of the synchronous execution of $\mathcal{A}_{sync}$ under $\mathcal{G}_{sync}$, is equal to the state of each node at the end of phase i in a semi-synchronous execution of $\mathcal{A}_{semi}$ under $\mathcal{G}_{semi}$ and a weakly-fair scheduler. (Theorem 2)
- **(Finite termination) If the synchronous algorithm always terminates in finite time, so do the simulated executions**: We show that the synchronous execution of $\mathcal{A}_{sync}$ terminates in finite time for $\mathcal{G}_{sync}$ if and only if the semi-synchronous execution of $\mathcal{A}_{semi}$ terminates in finite time for $\mathcal{G}_{semi}$, which implies the condition. (Theorem 3)

While the *non-triviality* requirement rules out trivial solutions, e.g., in which $\mathcal{G}_{semi}$ always contains no edges regardless of the actual dynamics, our transformation actually satisfies a **strong non-triviality requirement**: Any edge (u, v) that persists long enough for both nodes u and v to be activated at least once during a phase $i \geq 0$ of the semi-synchronous execution *must* be part of $\mathcal{G}_{sync}$ during the i-th step of the synchronous execution. In particular, if $\mathcal{G}_{semi}$ is static, then the simulation guarantees that $\mathcal{G}_{sync} = \mathcal{G}_{semi}$.

Table 1 Comparison to other synchronizers.

Protocol	Year	Mapping	Network Dynamics	Anonymous
α, β, γ-Synchronizers [2]	1985	Sync to Async	static	No
Afek et al. [1]	1987	Sync to Async	dynamic with eventual-quiescence	No
Awerbuch and Sipser [5]	1988	Sync to Async	dynamic with t_π-stabilization	No
Awerbuch and Peleg [4]	1990	Sync to Async	static	No
Awerbuch et al. [3]	1992	Sync to Async	dynamic with eventual-quiescence	No
ζ-Synchronizer [20]	1994	Sync to Async	static	No
η_1, η_2, θ-Synchronizers [19]	1994	Sync to Async	static	No
σ-Synchronizer [17]	2020	Sync to Async	complete, static	No
Ghaffari and Trygub [15]	2023	Sync to Async	static	No
δ-Synchronizer (this paper)	2025	Sync to Semi-sync	**Arbitrary Dynamics**	**Yes**

We assume the PULL model of communication (see, e.g., [9]), with the addition of a *disconnection detector*, as in [12], and a *1-bit multi-writer atomic register at each edge-port*. The δ-Synchronizer operates with an asymptotic memory overhead that is logarithmic on the number of nodes provided that the underlying synchronous algorithm terminates in polynomial time [7]. We present applications and discuss future work in Section 5, and summarize the most relevant related work in Table 1.

2 Model

We consider an edge-dynamic network that we formalize using a time-varying graph [10]. A time-varying graph is represented as $\mathcal{G} = (V, E, T, \rho)$ where V is the set of nodes, E is a (static) set of undirected pairwise edges between nodes, with a lifetime $T \subseteq \mathbb{N}$. A *presence* function $\rho : E \times T \to \{0, 1\}$ indicates whether or not a given edge exists at a given time. We define a *snapshot* of $\mathcal{G}$ at time t as the undirected graph $G_t(V, E_t)$, where $E_t = \{e \in E : \rho(e, t) = 1\}$. At any time, we denote the *neighborhood* of a node $u \in V$ by $N(u)$. For $t \geq 0$, the t-th *stage* lasts from time t to the instant just before time $t + 1$; thus, the communication graph in stage t is G_t.

We assume that each node is equipped with a mechanism to detect disconnections on its ports [12, 16, 18]. When an edge connected to a node u is disconnected, the *disconnection detector* adds the corresponding port label to a temporary set D. The set D is reset to $\emptyset$ at the end of each activation of u. To ensure anonymity among nodes and their neighbors, each node u associates with its neighbors through port labels ℓ, noting that different nodes may connect to u through the same port over time, and that a node may connect at different ports of u over time (respecting that at most one node is connected to any port at any point in time). Each edge (u, v) is assigned specific ports at both nodes u and v. For convenience, we use $u.x$ to denote the value of *variable* x *at node* u.

An algorithm, including the synchronizer, is composed of multiple actions of the form $\langle label \rangle : \langle guard \rangle \to \langle operations \rangle$. An *action is enabled* if its guard (a boolean predicate) evaluates to true, and a *node* is said to be *enabled* if it has at least one enabled action. The scheduler controls the stages when an enabled node is activated and picks exactly one enabled action at the node to execute at the given stage. Our synchronizer algorithm will ensure that we only have one enabled action per node at any stage t.

We assume the PULL model of communication (see, e.g., [9]), where every time a node u is activated at a stage t, u can pull the state information of a neighboring node v. In [7], we show that the classic PULL (or PUSH) model is not powerful enough to support synchronization. Thus, we further equip the PULL model with a *1-bit multi-writer atomic register at each*

edge-port, and allow an activated node u at stage t to perform one atomic write onto each multi-writer register of any neighbor node v such that $(u, v) \in E_t$ during the stage. Each action follows the Read-Compute paradigm, where these two components are executed in this order in lockstep (and writes occur within the compute cycle).

We focus on *semi-synchronous* concurrency, where in each stage, any (possibly empty) subset of enabled nodes is activated concurrently, under arbitrary topological changes. To model this, we assume two adversaries, an *adaptive adversary* that controls the edge presence function at each stage (i.e, controls the edge dynamics), and the scheduler adversary – or simply the *scheduler* – that controls when nodes are activated. We assume a *weakly fair* scheduler that activates nodes such that any continuously enabled node is eventually activated (or equivalently, that every enabled node will be activated infinitely often).

Algorithm 1 δ-Synchronizer.

1: HANDSHAKE: $(\texttt{synch} = 0) \vee (\exists \ell \in P \setminus \tilde{D} \mid \ell.\texttt{block} = 0) \rightarrow$
2: **if** $\texttt{synch} = 0$ **then** ▷ If this is the initialization stage of synchronization
3: **for** $\ell \in N(u)$ **do** ▷ for a phase i, pull information from all neighbors
4: $X(\ell) \leftarrow \text{PULL}(\ell)$ ▷ and initialize $\tilde{D}$, the set of disconnected ports in
5: $\tilde{D} \leftarrow \phi$ ▷ the current phase to the empty set.
6: $P \leftarrow \{\ell \in N(u) \mid$ ▷ Valid neighbors of u to be considered in phase i's simulation are:
7: $[(X(\ell).\texttt{phase} < \texttt{phase})] \vee$ ▷ (1) neighbors that will catch up with phase i, or
8: $[(X(\ell).\texttt{phase} = \texttt{phase}) \wedge$ ▷ (2) neighbors simulating phase i that have either
9: $(X(\ell).\texttt{synch} = 0) \vee$ ▷ (2a) not initialized their valid neighbors, or
10: $(X(\ell).\texttt{port} \in X(\ell).P \setminus (X(\ell).\tilde{D} \cup X(\ell).D))]\}$ ▷ (2b) also consider u as a valid ▷ persistent (i.e., not disconnected) neighbor in phase i.
11: $\texttt{synch} = 1$ ▷ Flag end of the initialization stage for phase i of the simulation.
12: **else** ▷ If u has finished initialization stage of the simulation of phase i,
13: **for** $\ell \in P \setminus (\tilde{D} \cup D)$ **do** ▷ For every persistent valid neighbor in the phase:
14: **if** $X(\ell).\texttt{phase} < \texttt{phase}$ **then** ▷ (a) Pull updated state from neighbors that are
15: $X(\ell) \leftarrow \text{PULL}(\ell)$ ▷ running behind and not yet simulating phase i.
16: **else** ▷ Otherwise, they must be simulating phase i, so
17: $X(\ell).\texttt{ack} \leftarrow \text{PULL}(\ell).\texttt{ack}$ ▷ (b) pull only neighbor's ack value for phase i.
18: $\tilde{D} \leftarrow \tilde{D} \cup D$ ▷ Add disconnections since last activation to the set $\tilde{D}$. ▷ After pulling state and ack information, attempt edge agreement.
19: **for** $\ell \in (P \setminus \tilde{D}) \mid [(X(\ell).\texttt{phase} = \texttt{phase}) \wedge (\ell.\texttt{block} = 0)]$ **do**
20: **if** $X(\ell).\texttt{ack} = 1$ **then** ▷ If a neighbor has already initiated edge agreement, block
21: BLOCK(ℓ) ▷ the edge by simultaneously setting the `block` flag on neighbor's port
22: $\ell.\texttt{block} = 1$ ▷ as well as on own port connected to edge to 1.
23: **else** ▷ Otherwise, indicate that state information for neighbor's
24: $\ell.\texttt{ack} = 1$ ▷ simulation of phase i has been pulled.

25: EXECUTESYNCH: $(\texttt{synch} = 1) \wedge (\ell.\texttt{block} = 1, \forall \ell \in P \setminus \tilde{D}) \rightarrow$
26: $F \leftarrow \{\ell \in P \mid (\ell.\texttt{block} = 1)\}$ ▷ F represents the final set of agreed neighbors of u in phase i
27: Run enabled action of (synchronous) algorithm $\mathcal{A}_{sync}$ with respect to $\{X(\ell) \mid \ell \in F\}$
28: $\texttt{phase} = \texttt{phase} + 1$
29: $\texttt{synch} = 0$ ▷ Clean up variables
30: **for** $\ell \in \{0, \ldots, \Delta\}$ **do** ▷ for new phase.
31: $\ell.\texttt{ack} = \ell.\texttt{block} = 0$

3 Algorithm

The δ-Synchronizer (Algorithm 1) consists of two actions, HANDSHAKE and EXECUTESYNCH, with exactly one of them being enabled for a node at any stage. Recall that the synchronizer works by showing the equivalence between a semi-synchronous execution of $\delta(\mathcal{A}_{sync})$ under an arbitrary dynamic network $\mathcal{G}_{semi} = \{G_0, G_1, \ldots\}$ and a synchronous execution of $\mathcal{A}_{sync}$ under a (potentially different) arbitrary dynamic network $\mathcal{G}_{sync} = \{H_0, H_1, \ldots\}$.

Each node u keeps a local counter, $u.\texttt{phase}$, of the phase number it is simulating and updates it when the simulation of the phase completes and an action of $\mathcal{A}_{sync}$ is executed (Lines 27-28). During the handshake for the simulation of phase i, a node u maintains a set of potential neighbors (identified by the ports to which they are connected) to be included as neighbors in the simulated execution, which is initially equal to the set $u.P$ of all valid neighbors at the start of the handshake, when $u.\texttt{synch} = 0$. The set of potential neighbors can "lose" elements in other stages of the handshake for phase i due to edge disconnections, but does not gain new elements. The crucial property to maintain is *edge consistency*: At the end of the simulation of phase i, the *final set* of neighbors of u, $u.F$, which is a subset of $u.P$, contains node v if and only if the set $v.F$ contains node u. The sets $u.F$, for all nodes $u \in V$, determine the set of (undirected) edges in the simulated graph H_i for phase i.

Initially, the set $u.P$ contains the following nodes:

1. (Running behind) Neighbors that have not yet started simulating phase i. These nodes can be included because node u will wait for them to catch up during the next stages in u's handshake for phase i (unless they disconnect from node u),

2a. (Concurrent nodes in initialization stages) Neighbors that have started simulating phase i but have not yet finished the initialization stage of the handshake. These nodes will also be able to see that node u has not yet finished its initialization stage of the simulation.

2b. (Concurrent nodes past initialization stages) Neighbors that completed the initialization of the simulation of phase i and are still considering node u as a neighbor in their handshake of phase i. These nodes must have started the handshake of phase i before node u but have not detected a disconnection in the edges linking them to node u.

To calculate the set $u.P$, a node pulls information from its neighbors and determines which nodes fall into one of the categories above (Lines 7–10). At the end of the first stage of the handshake, node u sets its synch flag to 1 to signal the end of its initialization for phase i.

To ensure *edge consistency*, each node u keeps track of all ports that have experienced a disconnection since the start of the simulation of phase i in its set $u.\tilde{D}$ (initially empty). All neighbors connected through ports in $u.\tilde{D}$ are removed from consideration to be included in the simulated graph H_i. Indeed, if there is a disconnection of an edge (u, v) in a given stage, then nodes u and v will each add the respective port connected to edge (u, v) to its own set $\tilde{D}$ in the first stage in which the node is active after the disconnection, and the edge (u, v) will not be included in the simulated graph H_i by either node. Since nodes are anonymous, any edge that later connects to the port of either u or v earlier connected to (u, v) – including, potentially, a reconnection of u and v – will be ignored in phase i.

A node u checks if the neighbors running behind in the simulation have caught up to phase i (Line 14) or, if the neighbor v has caught up, u only needs to update the respective ack value (since it already pulled the state information from v for phase i the last time it pulled from v in Line 15). Note that when the check is done in Line 14, if the neighbor is not behind, it must be simulating the same phase: The reason is that if the node was previously behind it could not have advanced beyond phase i without first executing the ack/block exchange and the first (and only) time that is done is when the two nodes are in phase i.

A node u completes the handshake of phase i when it determines that all edges linking it to a tentative neighbor in $u.P \setminus u.\tilde{D}$ are blocked (Line 25), implying that such an edge (u, v) will be considered to be in H_i by both u and v. The blocking of such an edge (u, v) can be initiated by u itself or by node v, but in either case it will result in the block flags on the respective ports at u and v being set to 1 during the current stage (while the edge is up). Whichever node blocks the edge-ports, say u, must have detected that the other node (v) set its ack to 1 (Lines 20-22), acknowledging that u and v are both simulating phase i, and that v has pulled u's phase i state information prior to the blocking (u has also pulled v's phase i state during the current stage). Note that there are blocked edges in phase i that disconnect later in the phase: Those edges will be in the respective sets F and thus also in H_i.

4 Our Results

Theorems 1–3, whose proofs appear in [7], ensure that the δ-Synchronizer satisfies the three conditions outlined in Section 1. The *$\mathcal{A}_{sync}$-state* of node u consists of the variables of u that directly pertain to the state variables of $\mathcal{A}_{sync}$.

▶ **Theorem 1** (Correctness). *For any semi-synchronous execution of $\delta(\mathcal{A}_{sync})$ under an arbitrary dynamic graph $\mathcal{G}_{semi}$ and a weakly-fair scheduler, there exists a dynamic graph $\mathcal{G}_{sync}$ such that the $\mathcal{A}_{sync}$-state of each node u at the end of each phase i of the execution of $\delta(\mathcal{A}_{sync})$ under $\mathcal{G}_{semi}$ is equal to the state of u at the end of step i in a synchronous execution of $\mathcal{A}_{sync}$ under $\mathcal{G}_{sync}$, for all $i \geq 0$.*

▶ **Theorem 2** (Non-triviality). *For any synchronous execution of $\mathcal{A}_{sync}$ under an arbitrary dynamic graph $\mathcal{G}_{sync}$, there exists a dynamic graph $\mathcal{G}_{semi}$ such that the state of each node u at the end of each step i of the synchronous execution of $\mathcal{A}_{sync}$ under $\mathcal{G}_{sync}$ is equal to the $\mathcal{A}_{sync}$-state of u at the end of phase i in the execution of $\delta(\mathcal{A}_{sync})$ under $\mathcal{G}_{semi}$, for all $i \geq 0$.*

▶ **Theorem 3** (Finite termination). *Our synchronizer ensures liveness: Any node progresses in finite time to phase i, for any finite $i \geq 0$. Thus, all synchronous executions of $\mathcal{A}_{sync}$ terminate in finite time if and only if all semi-synchronous executions of $\delta(\mathcal{A}_{sync})$ also do.*

5 Applications and Extensions

Designing algorithms for highly dynamic networks without any assumptions on edge dynamics or eventual stabilization is challenging, even in synchronous settings. On the other hand, scenarios with high and unpredictable network dynamics are getting increasingly more common in practice, and practitioners are looking at the benefits of time synchronization in order to better manage the dynamics (see, e.g., [11]). This work aims to bridge this divide.

In a classic application of our δ-synchronizer, we extend the applicability of the synchronous algorithm for maintaining a spanning forest that approximates the minimum possible number of spanning trees in arbitrary dynamic networks of [6] to semi-synchronous environments (after adapting the algorithm from the PUSH to PULL model). Another application is in the context of *minority dynamics* [8], a stateless protocol in which each node samples a random subset of neighbors and adopts the minority opinion observed. The impact of our synchronizer is not about enabling synchronous algorithms in semi-synchronous environments, but about providing an exponential speed-up in runtime when doing so, as we describe in [7].

In future work, we plan to investigate whether we can extend the δ-Synchronizer also to work in asynchronous environments, determine its overhead in terms of runtime and bits exchanged, and investigate whether it can work under certain classes of (non-stabilizing) network dynamics (e.g., edge recurrent, snapshot connected, etc.).

References

1 Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 358–370, 1987.

2 B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

3 B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. E. Saks. Adapting to asynchronous dynamic networks. In *Proc. of the ACM Symp. on Theory of Computing (STOC)*, pages 557–570, 1992.

4 B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *31st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 514–522, 1990.

5 B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *29th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 206–220, 1988.

6 M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. Neggaz. Maintaining a spanning forest in highly dynamic networks: The synchronous case. In *OPODIS*, pages 277–292, 2014.

7 R. Bazzi, A. Chaturvedi, A. W. Richa, and P. Vargas. Synchronization in anonymous networks under continuous dynamics. *CoRR*, 2025. doi:10.48550/arXiv.2506.08661.

8 L. Becchetti, A. Clementi, F. Pasquale, L. Trevisan, R. Vacus, and I. Ziccardi. The minority dynamics and the power of synchronicity. In *Proc. of the 2024 ACM-SIAM Symp. on Discrete Algorithms, (SODA)*, pages 4155–4176, 2024.

9 M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proc. of ACM Intl. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, pages 93–104, 2017.

10 A. Casteigts. *A Journey through Dynamic Networks (with Excursions)*. Habilitation à diriger des recherches, U. of Bordeaux, 2018. URL: https://tel.archives-ouvertes.fr/tel-01883384.

11 J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013. doi:10.1145/2491245.

12 J. J. Daymude, A. W. Richa, and C. Scheideler. Local mutual exclusion for dynamic, anonymous, bounded memory message passing systems. In *Proc. of 1st Symp. on Algorithmic Foundations of Dynamic Networks (SAND)*, volume 221 of *LIPIcs*, pages 12:1–12:19, 2022. doi:10.4230/LIPICS.SAND.2022.12.

13 P. Flocchini, G. Prencipe, and N. Santoro, editors. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, volume 11340 of *LNCS*. Springer, Cham, 2019. doi:10.1007/978-3-030-11072-7.

14 P. Flocchini, N. Santoro, M. Yamashita, and Y. Yamauchi. A characterization of semi-synchrony for asynchronous robots with limited visibility, and its application to luminous synchronizer design. *CoRR*, abs/2006.03249, 2020. arXiv:2006.03249.

15 M. Ghaffari and A. Trygub. A near-optimal deterministic distributed synchronizer. In *Proc. of the 2023 ACM Symp. on Principles of Distributed Computing (PODC)*, pages 180–189, 2023.

16 V. Iyer, M. J. Pennell, J. T. West, and M. Beck. Dual termination serial data bus with pull-up current source. U.S. Patent No. 6593768 B1, 2003.

17 G. Pandurangan, D. Peleg, and M. Scquizzato. Message lower bounds via efficient network synchronization. *Theoretical Computer Science*, 810:82–95, 2020. doi:10.1016/J.TCS.2018.11.017.

18 R. Russell. Linux ethtool interface for controlling ethernet devices, 2001. URL: `https://man7.org/linux/man-pages/man8/ethtool.8.html`.

19 L. Shabtay and A. Segall. Low complexity network synchronization. In *8th Intl. Workshop on Distributed Algorithms (WDAG)*, volume 857 of *Springer LNCS*, pages 223–237, 1994. `doi:10.1007/BFB0020436`.

20 L. Shabtay and A. Segall. A synchronizer with low memory overhead (extended abstract). In *Proc. of the IEEE Intl. Conf. on Distributed Computing Systems*, pages 250–257, 1994.