Measuring Temporal Lags in Delay-Tolerant Networks

Arnaud Casteigts*, Paola Flocchini*, Bernard Mans† and Nicola Santoro‡

* University of Ottawa, Canada,
`{casteig,flocchin}@site.uottawa.ca`
† Macquarie University, Sydney, Australia,
`bernard.mans@mq.edu.au`
‡ Carleton University, Ottawa, Canada,
`santoro@scs.carleton.ca`

Abstract—**Delay-tolerant networks (DTNs) are characterized by a possible absence of end-to-end communication routes at any instant. Yet, connectivity can be achieved over time and space, leading to evaluate a given route both in terms of *topological* length or *temporal* length. The problem of measuring temporal distances in a social network was recently addressed through post-processing contact traces like email datasets, in which all contacts are *punctual* in time (i.e., they have no duration). We focus on the *distributed* version of this problem and address the more general case that contacts can have arbitrary durations (i.e., be non-punctual). Precisely, we ask whether each node in a network can track in real-time how "out-of-date" it is with respect to every other. Although relatively straightforward with punctual contacts, this problem is substantially more complex with arbitrariry long contacts: consecutive hops of an optimal route may either be disconnected (intermittent connectedness of DTNs) or connected (i.e., the presence of links overlap in time, implying a *continuum* of path opportunities). The problem is further complicated (and yet, more realistic) by the fact that we address *continuous-time* systems and *non-negligible* message latencies (time to propagate a single message over a single link), however this latency is assumed fixed and known. We demonstrate the problem is solvable in this general context by generalizing a time-measurement vector clock construct to the case of "non-punctual" causality, which results in a tool we call T-CLOCKS, of independent interest. The rest of the paper shows how T-CLOCKS can be leveraged to solve concrete problems such as learning *foremost* broadcast trees, network backbones, or *fastest* broadcast trees in periodic DTNs.**

Index Terms—**C.2.1.d Distributed networks; C.2.1.j Store and forward networks; C.2.8 Mobile Computing.**

I. INTRODUCTION

Highly-dynamic networks, and in particular delay-tolerant networks (DTNs), are characterized by a possible absence of contemporaneous end-to-end communication routes (routes in which every next hop follows on directly, also called *direct journeys*). In most cases, however, communication can still be achieved over time and space through disconnected routes (*indirect journeys*) using store-carry-forward-like mechanisms. This particularity led researchers to develop a number of routing techniques based for example on proactive knowledge on the network schedule [5, 23], probabilistic [29] or encounter-based strategies [11, 17, 24]. A taxonomy is presented in [34].

On the analytical side, the time-dimension has had a strong impact on research that focused mainly on extending usual graph concepts to a temporal version, e.g, paths and reachability [2, 25], distance [5, 21], diameter [10], connectivity [1, 4],

or necessary conditions [6]. Of particular interest in this paper are the concepts of *journey* and *temporal distance* (terminology from [5]), which appeared independently in several fields under various other names, *e.g. schedule-conforming path* [2], *time-respecting path* [18, 25], or *temporal path* [10] for the concept of journey, and *reachability time* [18], *information latency* [27], or *temporal proximity* [28] for that of temporal distance. In this context, the duration of a given route no more depends on the sole number of hops separating nodes. Questions that immediately arise are *how far apart in time the nodes can be?* Such a value is obviously itself time-dependent: it varies depending on the date considered. *Can this temporal distance be measured, for every node, in each point in time?*

This type of question was recently addressed in a number of works from the field of social network analysis [18, 27, 28]. Indeed, social networks are in essence very similar to DTNs, and unlike these latter, often generate analizable datasets. In [27], Kossinets *et al.* ask how out-of-date each node could be with respect to every other node. They provide a *centralized* algorithm to process a known sequence of contact history and measure these lags based on an adaptation of vector clocks.

Besides looking at the question from a centralized point of view, these studies assumed that contacts between nodes are punctual in time (*i.e.,* they have no duration), and generally given as triplets (u, v, t) where u and v are two entities (nodes) and t a date of contact between them. This assumption was due to datasets where interactions are punctual in time, such as email exchanges or message posts on community websites. The situation in DTNs is typically different, because contacts between nodes can have arbitrarily durations and possibly overlap in time with each other. This aspect renders computation of exact temporal distances more complex because it implies the possible co-existence of indirect routes on the one hand, and *continuums* of direct routes on the other hand. Typical DTNs exhibit a mixture of both in various proportions.

In such a context, we look at the *distributed* version of the problem and ask: *is it possible for a node to know precisely, and in real time, how out-of-date it is with respect to every other node?* At first sight this problem bears some resemblance with that of clock synchronization in distributed networks (see e.g. [9, 20, 32]), however it is different in essence since we do not require (nor aim to achieve) a common time referential. In fact clock synchronization in our case would be straightforward to achieve using the assumption of

fixed latency for message propagation over any link (i.e., time required to propagate a single message over a single link). In this paper, we answer positively to the above questions in the case that contacts have arbitrary durations that take place over the continuous time domain. Feasibility is demonstrated through an algorithm that extends the one from [27] to a distributed setting and generalizes it to non-punctual contacts (and non-negligible, but fixed latencies). Doing this we design an abstraction tool called T-CLOCKS (for Temporal-lags Vector Clocks), of independent interest.

The second part of our work is dedicated to illustrating how the knowledge of temporal lags could be used as a building block to solve more concrete problems, such as distributed learning of temporally optimal broadcast trees (BTs) in periodically-varying networks (following a line of work initiated in [7]), of particular relevance in the field of satellites communication (e.g. to build delay-tolerant spanning structures) or public transportation (e.g. to propagate emergency messages or schedule updates in a temporally efficient way). In particular, we provide algorithms to learn *foremost* BTs: a set of broadcast trees that vary with the emitter and the emission date (modulo the period), guaranteeing the *earliest* possible delivery time at all nodes; and *fastest* BTs: a set of broadcast trees that vary with the emitter and guarantee that the time spent between first message emission and last message reception is minimum, even if it means waiting before the first emission. Both algorithms exploit a network abstraction provided by T-CLOCKS, each in a different way. Interestingly, the union of all foremost BTs for a given emission date also corresponds to what the authors of [27] refer to as a *network backbone*. Our foremost broadcast algorithm therefore computes all network backbones (*i.e.,* backbones for all emission dates) as a straight by-product.

Throughout the paper, we use the *time-varying graph* (TVG) formalism proposed in [8] to describe the environment and interaction between entities (nodes), as well as to analyze the protocol correctness. This formalism, which is semantically equivalent to that of *evolving graphs* [12], offers in comparison an *interaction-centric* perspective that proves more convenient to express and manipulate temporal aspects this work requires, *e.g.* focusing on the evolution of an edge independently from that of the entire graph (which proves convenient for various problems ranging from broadcast [7] to failure detectors [16]).

The paper is organized as follows: in Section II, we describe the model and terminology. Section III discusses the problem of measuring temporal lags in the general case of arbitrary long contacts in continuous time, assuming a fixed latency on each edge. We show the problem solvable by providing an algorithm called T-CLOCKS. Section IV shows how T-CLOCKS could be turned into a building block to solve more complex problems, and provides some guidance regarding its (object-oriented) suggested implementation. Finally, Section V illustrates a concrete use of T-CLOCKS to solve complex problems in periodic DTNs, namely the construction of foremost BTs (V-B), network backbones (V-C), and fastest BTs (V-D).

II. MODEL AND TERMINOLOGY

Consider a set V of *nodes*, making contacts with each other over a (possibly infinite) time interval $\mathcal{T} \subseteq \mathbb{T}$, called *lifetime* of the network; the temporal domain $\mathbb{T}$ corresponds here to $\mathbb{R}^+$ (continuous-time). Let the contacts between nodes define a set of intermittently available undirected edges $E \subseteq V^2$ such that $(x,y) \in E \Leftrightarrow x$ and y interact at least once in $\mathcal{T}$.

Following [8], we represent the network as a *time-varying graph* (*TVG*, for short) $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$, where $\rho : E \times \mathcal{T} \to \{0, 1\}$, called *presence* function, indicates whether a given edge is available at a given time, and $\zeta : E \times \mathcal{T} \to \mathbb{T}$, called *latency* function, indicates the time it takes to propagate a message over a given edge at a given date. In this work, we assume the latency function to be fixed for all edges and presence times, and thus denote it as a constant value ζ. If a message is sent less than ζ time before the disappearance of an edge, it is lost. The duration of an edge presence, on the other hand, can be arbitrarily long and possibly vary among several appearances of the same edge (*i.e.,* arbitrary long contacts between nodes). Given an edge e, we allow the notation $\rho_{[t_1, t_2)}(e) = 1$ to signify that $\forall t \in [t_1, t_2), \rho(e, t) = 1$.

Given a TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$, we denote by $G = (V, E)$ its *underlying graph*, that is, in a sense, the *static* counterpart of $\mathcal{G}$ (one might think of it as its "footprint"). A sequence of couples $\mathcal{J} = ((e_1, t_1), (e_2, t_2) \ldots, (e_k, t_k))$, where $e_1, e_2, ..., e_k$ is a walk in G and $t_i + \zeta \leq t_{i+1}$ for $1 \leq i < k$, is a *journey* in $\mathcal{G}$ iff $\rho_{[t_i, t_i + \zeta)}(e_i) = 1$. We denote by $departure(\mathcal{J})$, and $arrival(\mathcal{J})$, the starting date t_1 and last date $t_k + \zeta$ of $\mathcal{J}$, respectively. Journeys can be thought of as *paths over time* from a source to a destination and thus have both a *topological* and a *temporal* lengths. The *topological length* of $\mathcal{J}$ is the number $|\mathcal{J}|_h = k$ of couples in $\mathcal{J}$ (i.e., number of *hops*), and its *temporal length* (or *duration*) is $|\mathcal{J}|_t = arrival(\mathcal{J}) - departure(\mathcal{J}) = t_k - t_1 + \zeta$. For example the journey $(ac, 2), (cd, 5)$ in Figure 1 has a topological length of 2, and a duration of $3 + \zeta$ units of time.

Let us denote by $\mathcal{J}_{\mathcal{G}}^*$ the set of all journeys in TVG $\mathcal{G}$, and by $\mathcal{J}_{(u,v)}^* \subseteq \mathcal{J}_{\mathcal{G}}^*$ those journeys starting at node u and ending at node v. Clearly, the concept of journey is not symmetrical: the existence of a journey from u to v does not imply the existence of a journey from v to u; this holds regardless of whether the edges are directed or not, because the time creates its own level of direction. For example, in the TVG of Figure 1, there are several journeys from a to d whereas $\mathcal{J}_{(d,a)}^* = \emptyset$.

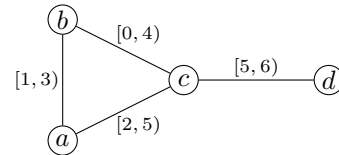

Figure 1. A TVG $\mathcal{G}$; the labels on the edges indicate the time intervals in which those edges are present. The edge latency is $\zeta \leq 1$.

We say that a journey is *direct* if the presence of consecutive edges overlap in time and their use follow on directly (*i.e.,* intermediate nodes do not wait to forward the message); it is said *indirect* otherwise (at least one intermediate node needs

to buffer the message for some time). An example of direct journey in the graph of Figure 1 is $\mathcal{J}_1 = \{(ab, 2), (bc, 2+\zeta)\}$. Examples of indirect ones include $\mathcal{J}_2 = \{(ac, 2), (cd, 5)\}$, and $\mathcal{J}_3 = \{(ab, 2), (bc, 2+\zeta), (cd, 5)\}$.

Nodes can detect the appearance or disappearance of an incident edge *instantly* and react to them by dedicated operations. Since the presence intervals are by convention right-open, we assume that disappearances are always handled before appearances at a given time, which is consistent with looking at journeys whose presence intervals strictly follow as indirect ones (*e.g.* $\mathcal{J}_{(a,d)} = \{(ac, 5-\zeta), (cd, 5)\}$ in the above example). Processing times are neglected. We assume that the fixed latency ζ is known to the nodes, *i.e.,* they know how long it takes to propagate a single message over a single edge. Nodes need not share the same global time, but their clocks advance at a same rate. The network schedule ρ is *not* known and is arbitrary (except for Section V where it is periodic). Finally, nodes have unique identifiers.

III. MEASURING TEMPORAL LAGS

This section is concerned with making nodes aware, at any time, of how *out-of-date* they are with respect to each others. A relevant concept is that of *temporal view*, introduced in [27] for social network analysis. (This concept was simply called "view", but we added the "temporal" adjective to avoid confusion since "view" has a different meaning in distributed computing, e.g. [33].) The temporal view a node v has of a node u at time t, denoted $\phi_{v,t}(u)$, is the latest (i.e., largest) $t' \leq t$ at which a message received by time t at v could have been emitted at u; that is, in our formalism,

$$\phi_{v,t}(u) = \text{Max}\{departure(\mathcal{J}) : \mathcal{J} \in \mathcal{J}^*_{(u,v)} \wedge arrival(\mathcal{J}) \leq t\}.$$

By convention, $\phi_{v,t}(v) = t$ for any node v and time t, and $\phi_{v,t}(u) = -\infty$ if no such journey existed before. Whenever the date or the local node are implicit, we omit the corresponding element in the subscript. The table on Figure 2 summarizes the key concepts and notations seen so far.

Notation	Concept	Meaning
$\rho(e, t)$	presence function	indicates whether edge e exists at time t
$\zeta(e, t)$	latency function	indicates the latency of edge e at time t
$\mathcal{J}_{(u,v)}$	journey	path over time from node u to node v
$|\mathcal{J}|_h$	topolog. length	for a journey $\mathcal{J}$: its number of hops
$|\mathcal{J}|_t$	temporal length	for a journey $\mathcal{J}$: its overall duration
$\phi_{v,t}(u)$	temporal view	that a node v has of a node u at time t: lastest departure at u to reach v by t
$\hat{d}_{u,t}(v)$	temporal distance	min time to reach v from u starting at t

Figure 2. Summary of key concepts and associated notations.

A. Punctual contacts

The question under investigation is: *can the nodes know their temporal views in real time?* That is, can a node v know the exact value of $\phi_{v,t}(u)$ at any time t for any node u? The problem has a known solution if contacts between nodes are punctual (have no duration), which was the case examined

by Kossinets, Kleinberg, and Watts [27]. The solution relies on a "temporal" adaptation of the *vector clock* mechanism. Vector clocks were introduced independently by Fidge [13] and Mattern [31] to track causality relations between events in a distributed system, when no assumption are made with respect to the nodes clocks; they establish a type of logical time that ensures a complete causal ordering of the events in the system. As shown in [27], the same mechanism can be used to measure temporal lags between nodes. We provide in Algorithm 1 a distributed formulation of the solution from [27]. Assume for simplicity that all the local clocks share a same global time (this assumption can be easily removed, as explained at the end of the section). Informally, every node v maintains a vector of all its *current* temporal views $\phi_{v,t} = (\phi_{v,t}(u) : u \in V)$, where t is the local current time (also referred to as $now()$ in the algorithm). The local vector of a node is called its *vector clock*. Initially the vector contains only its own reflexive temporal view. Whenever a contact occurs between two nodes, they exchange their vectors; each node then operates a nodewise maximum between both temporal views (new views are inserted by copy); the resulting vector is considered as the new local vector on both sides.

Algorithm 1 Measuring temporal lags with instantaneous contacts.

1: $VectorClock\ vec \leftarrow \emptyset$

2: ***onContact*** with a neighbor ng:
3: $vec[myself] \leftarrow now()$
4: $send(vec)\ to\ ng$

5: ***onReception*** of a vector clock vec_{ng}:
6: **for all** $u \in vec_{ng}.nodes()$ **do**
7:     **if** $u \notin vec.nodes()$ **or** $vec_{ng}[u] > vec[u]$ **then**
8:        $vec[u] \leftarrow vec_{ng}[u]$

9: ***getView***(Node u):
10: $return\ vec[u]$

In this code, $now()$ returns the current local time; $nodes()$ called on a given vector clock returns the list of nodes within (without the associated views); $myself$ stands for the underlying node.

It is easy to see that the value returned by $getView()$ called with parameter u at node v and time t is indeed equal to $\phi_{v,t}(u)$ in this simple setting.

B. Impact of arbitrary long contacts on the temporal views

The general case of arbitrary long contacts (and non-negligible latencies) is clearly more complex and, until now, no solution exists. Indeed, the existence of arbitrary long contacts between nodes makes it possible for adjacent contacts to overlap in time, thereby producing complex patterns of time lag between nodes. Consider the plots in Figure 3, showing an example of evolution of the temporal view that c has of a in a very simple TVG. Contrary to the case with punctual contacts – where evolution occurs only in discrete steps – there is here a mixture of discrete and continuous evolution. (The reader is

encouraged to spend a few minutes on this example as these concepts are essential in what follows.)
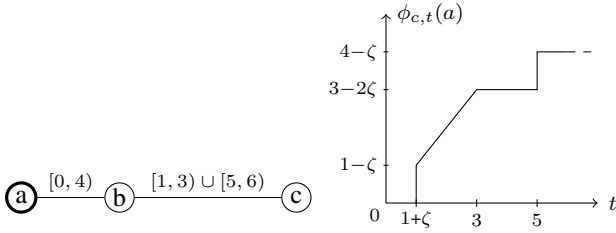

Figure 3. Temporal view that c has of a, as a function of time *(with $\zeta \ll 1$).*

Direct journeys are often faster than indirect ones, but this is not necessarily the case (imagine a very long direct journey, versus a short indirect one whose edges traversals follow closely). As a result, the temporal view ϕ at anytime could be caused by either types of journey. Let us call *direct view* and *indirect view* the views resulting from the best direct and indirect journeys at any given time (keeping in mind that ϕ is always the max between both).

Both types of views are different in nature. Unlike indirect journeys, which produce discrete increases of the view, direct journeys in general belong to a *continuum* of several such journeys, thus producing *continuous* increases of the view. Looking again at Figure 3, the (direct) view c has of a during $[1+\zeta, 3)$ only depends on the *topological* length (*i.e.,* number of hops) of the corresponding journey, here 2 (node c can receive any message emitted by a between times $1 - \zeta$ and $3 - 2\zeta$ after a lag of exactly 2ζ time). Since the duration of a direct journey depends only on the number of hops, it is sufficient for a node v to know the length of the (topologically) *shortest* direct journey currently arriving to it from another node u, called the *level* of v with respect to u, to deduce the corresponding direct view in real time. This particular way of defining a level can be formalized as follows:

$$level_{v,t}(v) =$$
$$Min\{|\mathcal{J}|_h : \mathcal{J} \in \mathcal{J}^*_{(u,v)} \land isDirect(\mathcal{J}) \land arrival(\mathcal{J}) = t\},$$

where $isDirect(\mathcal{J})$ is true iff $\mathcal{J}$ is a direct journey. By convention, $level_{v,t}(u)$ is considered to be $-\infty$ if no direct journey from u is arriving to v at time t. Let us stress that our notion of level is relative to the *reception* side, and is not concerned for example with the fact that some edges of the journey may have disappeared by reception time. What matters is to know whether messages could still be *currently* arriving from a given remote node through a given number of hops.

C. The algorithm

The proposed algorithm tracks evolution of both direct views and indirect views independently, with the resulting temporal view corresponding to their maximum. While indirect views are stored as classical variables, being maintained using a similar technique as in Algorithm 1 (*i.e.,* nodewise maximums upon edge appearances), direct views are deduced when needed from the current time and corresponding levels.

Thus, the algorithm consists in maintaining up-to-date information about two kinds of variables: the *level* of the local node with respect to every other node (for direct views), and the largest $date$ at which a message carried to the local node through a indirect journey could have been emitted at every other node (indirect views). The corresponding vector clock therefore associates to each remote node both a *level* and a $date$ value. Assume again for simplicity that all the local clocks share the same global time; this assumption will be relaxed at the end of the section. The detailed description of the algorithm is in Algorithm 2.

D. Correctness.

In the following, $level_{v,t}(u)$ and $date_{v,t}(u)$ denote the values of variables $vec[u].level$ and $vec[u].date$, respectively, read at node v at time t. The notation $getView_{v,t}(u)$ similarly stands for the result of the function $getView()$ called on v at time t with parameter u. Let us start with a few observations:

Property 1: Whenever an edge appears, the local vector is immediately sent on it (see $onAppearance()$).

Property 2: Whenever the local vector is modified, it is immediately sent to all the current neighbors.

This is true initially; then, the vector can only be modified in $updateVector()$, at the beginning of which the vector is copied and end of which it is sent if different from the copy.

The correctness of our algorithm is proven through a sequence of lemmas. Although the intuition behind the algorithm is clear and its formulation relatively compact, proving formally that the temporal view is *exactly* assessed by the nodes *at any point in time* is complex. Here is how the proof is organized. Lemmas 3 and 4 are intermediate properties that are used in the proofs of Lemmas 5 and 6, respectively, which in turn are used in that of Lemma 7 (stating that the computed view is always *larger or equal* to ϕ). Similarly, Lemma 8 is an intermediate property used in the proof of Lemma 9 (the computed view is always *lower or equal* to ϕ). The final theorem concludes equality based on Lemmas 7 and 9.

Lemma 3: For any two nodes v_1, v_2 and time t, if edge (v_1, v_2) is always present during a period $[t, t + \zeta]$, then $level_{v_2, t+\zeta}(u) \leq level_{v_1, t}(u) + 1$ for any u. (Informally, the level of a node cannot be more than that of any of its neighbors plus 1 after any ζ span of existence of the corresponding edge.)

Proof: Two possibilities must be considered, depending on whether the edge (v_1, v_2) appeared before or after $level_{v_1}(u)$ took the value that it has at time t. If it appeared before, Property 1 guarantees that v_2 has received it by time $t + \zeta$; otherwise Property 2 guarantees the same. Every time a vector is received, the function $updateVector()$ is executed. This function goes through all entries of the received vector (among others) and guarantees the property by line 22. ∎

Lemma 4: For any two nodes v_1, v_2 and time t, if the edge (v_1, v_2) is always present during a period $[t, t + \zeta]$, then $date_{v_2, t+\zeta}(u) \geq date_{v_1, t}(u)$ for any u.

Proof: Same ideas as for Lemma 3, but considering line 24 instead of line 22. ∎

In the following, the term "communication message" (or simply "message") does not refer to the "control messages" generated by Algorithm 2; it denotes instead any message that could be exchanged by an application running in the network.

Algorithm 2 Measuring temporal lags in the case of lasting contacts – The T-CLOCKS algorithm.

1: $VectorClock\ vec \leftarrow \emptyset$
2: $Map{<}Node, VectorClock{>}\ neighborsVCs \leftarrow \emptyset$ ▷ *the vector clocks of all neighbors are locally memorized.*

3: *__initialization__*:
4: $vec[myself].level \leftarrow 0$
5: $send(vec)\ to\ N_{now()}$ ▷ *sends the vector to all the current neighbors*

6: *__onAppearance__* of a common edge with neighbor ng:
7: $send(vec)\ to\ ng$

8: *__onDisappearance__* of an edge to neighbor ng:
9: $neighborsVCs[ng] \leftarrow \emptyset$
10: $updateVector()$

11: *__onReception__* of a vector clock vec_{ng} from a neighbor ng:
12: $neighborsVCs[ng] \leftarrow vec_{ng}$
13: $updateVector()$

14: *__updateVector__*():
15: $updateDatesBasedOnLevels()$
16: $VectorClock\ vec' \leftarrow vec$ ▷ *copies the vector for subsequent change detection.*
17: **for all** $v \in vec.nodes()$ **do**
18:   $vec[v].level \leftarrow +\infty$ ▷ *resets all levels.*
19: **for all** $vec_{ng} \in neighborsVCs$ **do**
20:   **for all** $v \in vec_{ng}.nodes()$ **do**   ▷ *go across all entries of all neighbors vectors, and for each source node,*
21:    **if** $vec_{ng}[v].level < vec[v].level - 1$ **then**
22:     $vec[v].level \leftarrow vec_{ng}[v].level + 1$ ▷ *update the underlying level whenever a smaller level is detected.*
23:    **if** $vec_{ng}[v].date > vec[v].date$ **then**
24:     $vec[v].date \leftarrow vec_{ng}[v].date$ ▷ *update the underlying date whenever a larger date is detected.*
25: **if** $vec \neq vec'$ **then**
26:   $send(vec)\ to\ N_{now()}$ ▷ *if the vector has changed, send it to the current neighbors.*

27: *__updateDatesBasedOnLevels__*():
28: **for all** $v \in vec.nodes()$ **do**
29:   **if** $now() - vec[v].level \times \zeta > vec[v].date$ **then**
30:    $vec[v].date \leftarrow now() - vec[v].level \times \zeta$

31: *__getView__*(Node v):
32: **return** $max(vec[v].date, now() - vec[v].level \times \zeta)$

Lemma 5: No communication message could be received by a node v at time t through a direct journey $\mathcal{J} \in \mathcal{J}^*_{(u,v)}$ unless $level_{v,t}(u) \leq |\mathcal{J}|_h$. (Informally, the purpose is to prove that the computed level is never larger that the real level.)

Proof: Let $\mathcal{J} = \{(e_1, t_1), (e_2, t_2) \ldots, (e_k, t_k)\} \in \mathcal{J}^*_{(u,v)}$ where $e_i = (v_i, v_{i+1})$, $1 \leq i \leq k$, with $v_1 = u$ and $v_{k+1} = v$. By Lemma 3, $level_{v_{i+1}, t_i + \zeta}(u) \leq level_{v_i, t_i}(u) + 1$ ($1 \leq i \leq k$); since the journey is direct, $t_i + \zeta = t_{i+1}$. Thus $level_{v, t_{k+1}}(u) = level_{v_{k+1}, t_{k+1}}(u) \leq level_{v_1, t_1}(u) + k$. Since $v_1 = u$ and $level_{u, t_1}(u) = 0$, the Lemma holds. ∎

Lemma 6: No communication message could be received by a node v at time t through an indirect journey $\mathcal{J} \in \mathcal{J}^*_{(u,v)}$ unless $date_{v,t}(u) \geq departure(\mathcal{J})$.

Proof: Let $\mathcal{J} = \{(e_1, t_1), (e_2, t_2) \ldots, (e_k, t_k)\} \in \mathcal{J}^*_{(u,v)}$ where $e_i = (v_i, v_{i+1})$, $1 \leq i \leq k + 1$, with $v_1 = u$ and $v_{k+1} = v$. The proof follows a similar inspiration as that of Lemma 5, but requires an additional intermediate step because in general, $date_{u, departure(\mathcal{J})}(u) \neq departure(\mathcal{J})$. The intermediate step is as follows. Because $\mathcal{J}$ is indirect, then there exists at least one intermediate node v_j that has lost the edge from v_{j-1} before the appearance of the edge to v_{j+1} (that is, v_j is the last node such that $\mathcal{J}_{(u,v_j)} \subseteq \mathcal{J}$ is a

direct journey). This has caused the function $updateVector()$ to execute on v_j, and thus v_j to convert its level w.r.t. u into a date (line 15), which by Lemma 5 is necessarily larger or equal to $departure(\mathcal{J})$. Now, from Lemma 4 we have that $date_{v_{i+1}, t_i + \zeta}(u) \geq date_{v_i, t_i}(u)$. By applying this inequality on the remaining edges (sequentially from v_j to v_{k+1}), we can conclude that $date_{v,t}(u) \geq departure(\mathcal{J})$. ∎

Lemma 7: For any pair of nodes u, v and time t, $getView_{v,t}(u) \geq \phi_{v,t}(u)$. (Informally, the computed view is at least as large as the real view.)

Proof: By contradiction, let there exist a pair u, v and a time t such that $getView_{v,t}(u) < \phi_{v,t}(u)$. This means, by definition, that a message emitted at u at some time $t' \leq t$ could have arrived at v at some time t'' although $getView_{v,t''}(u) < t'$. From the way $getView()$ is computed, this implies both $t'' - level_{v,t''}(u) \times \zeta < t'$ **and** $date_{v,t''}(u) < t'$. Consider now the journey described by such a message. If the journey is direct, then the first inequality is contradicted by Lemma 5; if the journey is indirect, the second inequality is contradicted by Lemma 6. ∎

Lemma 8: For any pair of nodes u, v and time t, $\phi_{v,t}(u) \geq t - level_{v,t}(u) \times \zeta$.

Proof: Let us examine separately the cases that $level_{v,t}(u)$ is $+\infty, 0$, or an integer $n > 0$.

- If $level_{v,t}(u) = +\infty$, then $t - level_{v,t}(u) \times \zeta = -\infty$, which is either equal to $\phi_{v,t}(u)$ when the latter is undefined (by convention), or less otherwise.

- If $level_{v,t}(u) = 0$, then v must be the same node as u (since levels w.r.t. other nodes can only be modified through *incrementing* the value of a neighbor). Thus, $t - level_{v,t}(v) \times \zeta = t$, which is, by definition, the value of $\phi_{v,t}(v)$.

- Let $level_{v,t}(u) = k$ for some integer $k > 0$. Let us first observe that this implies the existence of another node v' such that $level_{v',t-\zeta}(u) = k-1$ and $\rho_{[t-\zeta,t]}(v,v') = 1$ (otherwise the local level would have been decreased by the function $updateVector()$ after the loss of the neighbor causing $level$ to be k). Knowing that $level_{u,t}(u) = 0$ implies, by simple induction, that there is a direct journey $\mathcal{J} \in \mathcal{J}_{(u,v)}^*$ such that $arrival(\mathcal{J}) = t$ and $|\mathcal{J}|_h = level_{v,t}(u)$, which in turn implies that a message emitted at u at time $t - level_{v,t}(u) \times \zeta$ would be received at v at time t. ∎

Lemma 9: For any pair of nodes u, v and time t, $getView_{v,t}(u) \le \phi_{v,t}(u)$. (Informally, the computed view is at most as large as the real view.)

Proof: By contradiction, assume there exist a pair u, v and a time t such that $getView_{v,t}(u) > \phi_{v,t}(u)$. From the way $getView()$ is defined, $getView_{v,t}(u) > \phi_{v,t}(u)$ implies that either $t - level_{v,t}(u) \times \zeta > \phi_{v,t}(u)$ **or** $date_{v,t}(u) > \phi_{v,t}(u)$. The first inequality is contradicted by Lemma 8. As for the second, according to the algorithm, a $date$ variable can only be increased by means of two actions: copying the date variable from a neighbors' vector, or converting the current $level$ into a $date$. The copy cannot generate unappropriate increase of the value because if two nodes are able to exchange their vectors, then they could also exchange the messages they have received so far (such a copy is necessarily consistent). The second action cannot generate a larger date than $\phi_{v,t}(u)$ without contradicting Lemma 8. ∎

Correctness now follows from Lemmas 7 and 9:

Theorem 10: For any pair of nodes u, v and time t, $getView_{v,t}(u) = \phi_{v,t}(u)$.

Finally, observe that the simplifying assumption that clocks share the same global value is not necessary. Indeed, if the nodes keep track of *when* each $date$ was locally received, they can easily add information about how long the date was locally stored when they transmit it further. Combining this information with the edge latency ζ (which is known) allows to convert any received date into the local referential.

IV. USING T-CLOCKS AS A NETWORK ABSTRACTION

Knowledge of temporal lags, by means of both direct and indirect views, can be instrumental to solve concrete problems in DTNs (as shown in the examples of Section V). We suggest in this section a possible architecture to build on top of T-CLOCKS, and provide guidance relative to its implementation.

Building on top of T-CLOCKS

One way of using T-CLOCKS is to consider them as an *abstraction* providing high-level information on the temporal views – in our case, information to track both direct and indirect views, which proves sufficient to solve problems like learning foremost broadcast trees, network backbones, and fastest broadcast trees in periodically-varying DTNs. Technically, the abstraction consists of an intermediate layer between the network and some higher application (see Figure 4), which it informs by means of generating the two following events: `levelChanged()`, reflecting the evolution of a direct view, and `dateImproved()`, reflecting that of an indirect view.
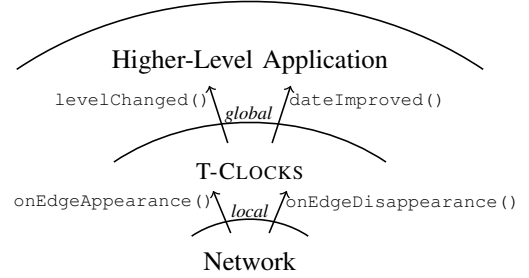

Figure 4. T-CLOCKS as an abstraction to track temporal views.

Whenever one of these events occurs, the higher algorithm should be able to identify which local neighbor is responsible for making the view evolve (from a practical point of view, this information is crucial to learn routing paths in the network). Both this feature and that of notifying higher algorithms require adaptations of the T-CLOCKS algorithm from Section III (Algorithm 2), which we now describe.

a) Tracking the best proxies: The current version of Algorithm 2 does not *identify* which neighbors are responsible for current views. Instead, only the value of the best $levels$ (direct views) or $dates$ (indirect views) are determined in $updateVector()$ (see line 19 and subsequent). Identifying these particular neighbors – or *proxies* – can be done by adding and maintaining two additional variables in the vector clocks, which account respectively for the best "level proxy" (direct view), and the best "date proxy" (indirect view). These variables should be updated whenever the corresponding values change, *i.e.,* when lines 22 or 24 execute.

b) Notifying higher algorithms: The notifications are to be raised whenever either type of view has changed. The corresponding test is already performed at the end of $updateVector()$. Note that the two vectors compared on line 23 should be considered as different not only when a level or a date has been updated, but also when a proxy has changed (which may occur without any change of the level, *e.g.* when two local neighbors are providing the same direct view relative to a given remote node, and one of them disappears). We envision notifications in the spirit of the *observer-observable* design pattern between two objects, namely the T-CLOCKS algorithm (observable) and the higher algorithm (observer). Concretely, this means that the higher algorithm subscribes (typically upon initialization) to the events of the T-CLOCKS algorithm by calling a $register()$ function (so that the T-CLOCKS algorithm becomes aware this particular entity must be notified). Then, whenever events occur, the T-CLOCKS algorithm calls the corresponding dedicated functions

on the higher algorithm side (in our case, $levelChanged()$ and $dateImproved()$), in which the desired response to the event is encapsulated. (If unclear, this mechanism shall become clearer by looking at the examples in Section V.) The notifications are to be raised as follows:

- $levelChanged$(Node src, Integer $level$, Node $proxy$): called whenever the level value or the level proxy with respect to a remote node (also called the *emitter* or the *source*) has changed during $updateVector()$. The corresponding source, level, and proxy are transmitted as parameters.
- $dateImproved$(Node src, Integer $date$, Node $proxy$), called whenever the date value relative to an emitter (src) has increased during the loop, <u>iff</u> this date is larger than the direct view (that is, larger than $now() - level(src) \times \zeta$).

Ideally, the T-CLOCKS algorithm should be able to run independently from higher algorithms, and tolerate several observers simultaneously. To enable registration on top of an already running instance of T-CLOCKS, we consider the following extra argument of the $register()$ function.

Property 11: The call to $register(boolean\ param)$ causes T-CLOCKS to call immediately $levelChanged()$ on the higher algorithm for every source with respect to which the *current* level is not $+\infty$, if $param = true$. This extra notification does not occur if $param = false$.

Complexity

Clearly, using T-CLOCKS does *hide* a substantial amount of complexity to higher applications. Our purpose in this paper was to demonstrate theoretical feasibility of the problem and its relevance to other concrete problems. As such, we did not focus on improving its complexity. Trivial improvements include avoiding to send the complete vector every time a change occur or an edge appears (*cf.* Algorithm 2), by means of sending only the differences with previously sent vectors.

V. APPLICATIONS – LEARNING FOREMOST BROADCAST TREES, NETWORK BACKBONES, AND FASTEST BROADCAST TREES IN PERIODIC DTNs

This section illustrates how temporal lags can be leveraged to solve concrete problems such as the construction of foremost broadcast trees, network backbones, and fastest broadcast trees in TVGs whose schedule (presence function ρ) is periodic. We first define the problems in general terms, then motivate the assumption of periodicity based on known negative results. Then we provide algorithms that solve them by exploiting the T-CLOCKS abstraction discussed in Section IV.

A. Definitions and background

As mentioned in Section II, the length of a journey can be measured both in terms of *hops* or *time*. This gives rise to two distinct definitions of distance in a graph $\mathcal{G}$:

- The *topological distance* from u to v at time t, noted $d_{u,t}(v)$, is defined as $Min\{|\mathcal{J}|_h : \mathcal{J} \in \mathcal{J}^*_{(u,v)} \wedge departure(\mathcal{J}) \geq t\}$. Given a date t, a journey whose departure is $t' \geq t$ and topological length is $d_{u,t}(v)$ is called *shortest* ;

- The *temporal distance* from u to v at time t, noted $\hat{d}_{u,t}(v)$ is defined as $Min\{arrival(\mathcal{J}) : \mathcal{J} \in \mathcal{J}^*_{(u,v)} \wedge departure(\mathcal{J}) \geq t\} - t$. Given a date t, a journey whose departure is $t' \geq t$ and arrival is $t + \hat{d}_{u,t}(v)$ is called *foremost*; one whose temporal length is $Min\{\hat{d}_{u,t'}(v) : t' \in \mathcal{T}_{[t,+\infty)}\}$ is called *fastest*.

Informally, a *shortest* journey is one that minimizes the number of hops; a *foremost* journey minimizes the arrival date; and a *fastest* journey minimizes the time spent between departure and arrival (however late the departure is). The problem of computing shortest, fastest, and foremost journeys in DTNs was solved in [5] as a centralized (*i.e.,* combinatorics) problem, given complete knowledge of $\mathcal{G}$. Precisely, the corresponding algorithms build the optimal set of journeys from the emitter to every other nodes (one algorithm for each metric).

The *distributed* problems of performing shortest, fastest, or foremost *broadcasts* without knowing the network schedule was recently investigated in [7], with following definitions:

- *Foremost broadcast*: every node must receive the message at the earliest possible date following broadcast initiation.
- *Shortest broadcast*: every node must receive the message by means of a minimum number of hops.
- *Fastest broadcast*: the overall time between first emission (at the initial emitter) and last message reception (anywhere in the network) must be minimized.

We require the emitter to detect termination of the broadcast (*i.e.,* all the nodes received the message), although this termination needs not be itself foremost, shortest, or fastest. In [7], the authors study the feasibility of these three metrics depending on various assumptions and knowledge on $\mathcal{G}$. Three cases are considered: (i) no assumption is made on the schedule; (ii) edges are recurrent, that is, if they appear at some time, then they will re-appear at some unknown future date; and (iii) the re-appearance is bounded by some known duration. The feasibility and complexity of each metric with respect to these assumptions varies. In particular, foremost broadcast becomes feasible in (ii), but the corresponding trees change constantly and therefore cannot be *learnt* for later use even in (iii). Shortest broadcast turns out to become both feasible and learnable at once in (iii). As for fastest broadcast, it remains unfeasible even in (iii) (and *a fortiori* not learnable). This motivates to consider a stronger assumption on the schedule of $\mathcal{G}$, such as that of *periodicity*, which makes foremost BTs learnable and fastest BTs both feasible and learnable, thereby completing the results of [7].

The following sections address the problems of learning foremost BTs and fastest BTs in TVGs whose schedule is periodic with known period p, that is, graphs such that $\forall e \in E, \forall t \in \mathcal{T}, \rho(e,t) = \rho(e, t + kp)$ for all integer k. The periodic assumption holds in networks whose entities have periodic movements (*e.g.,* satellites, subways, guards tour) or sleeping schedule (sensors). Other works in periodic DTNs include exploration by mobile agents [3, 14, 15, 22] or scalable routing [26, 30]. This assumption is also made in part of [27].

B. Application 1 – Foremost broadcast trees

Among all metrics, "foremost" is probably the one whose interest is most obvious. It is natural to ask what set of journeys a message emitted at a given source (or *emitter*) should follow to reach all nodes the earliest. Clearly, this choice depends on which date the broadcast is initiated at (*initiation* date), and even then, several options may exist. A nice property of the foremost metric is that among all the possible foremost journeys, there is (at least) one whose *prefixes* are themselves foremost journeys, *i.e.,* every intermediary node is reached in a foremost fashion. (This property may seem obvious, but it does not hold for fastest journeys, as we will see in Section V-D.) This allows to consider, for a given initiation date, a *tree* of foremost journeys that we refer to as a *foremost broadcast tree* (foremost BT, for short) for that particular date.

As an example, the foremost BTs corresponding to the graph of Figure 5 for emitter a are shown in Figure 6 as a function of the initiation date. These trees do not indicate, strictly
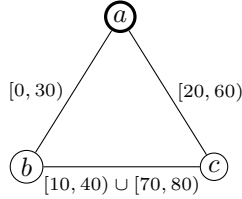

Figure 5.   Example of periodic TVG (*with period* 100 *and latency* $\zeta = 1$).

speaking, what journeys to follow (*i.e.,* paths together with crossing dates), but only what are the underlying paths, which is enough (every edge being used as early as possible). For example, the foremost tree corresponding to initiation date 50 goes through b, so a forwards the message to b at date 50, then b knows it must forward to c, which occurs at 70 when the corresponding edge appears.
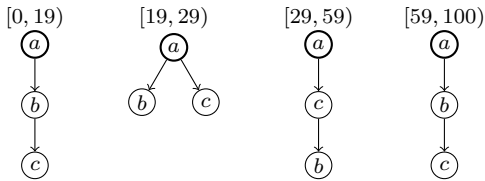

Figure 6.   Foremost BTs corresponding to the graph of Figure 5, for emitter a and initiation dates modulo 100.

The assumption of periodicity is not strictly needed to perform a foremost broadcast (as shown in [7]), but it allows for the trees to be learnt (and reused at lower cost) because the optimality of journeys holds modulo p. Here, for example, the first tree is optimal for any initiation date in $[0, 19)$, or $[100, 119)$, or $[900, 919)$, *etc.* It is thus sufficient to build all foremost BTs relative to one period, then use them forever.

It is important to keep in mind that forwarding choices are relative to the *initiation date*. Take for instance initiation date 58.5, where a forwards the message to c who receives it at 59.5 ($\zeta = 1$ in this example). The date c must consider to make forwarding choices is not 59.5, but well and truly 58.5 (inducing decision to forward to b). Another fact is that a node

needs not knowing the whole tree to make local choices. It only needs to know what neighbors it must forward a message to, based on the source and initiation date. The corresponding information relative to source a is shown in the table of 7.

on a	Initiation date	$[0, 19)$	$[19, 29)$	$[29, 59)$	$[59, 100)$
	Children	$\{b\}$	$\{b, c\}$	$\{c\}$	$\{b\}$
on b	Initiation date	$[0, 19)$	$[19, 59)$	$[59, 100)$	
	Children	$\{c\}$	$\emptyset$	$\{c\}$	
on c	Initiation date	$[0, 29)$	$[29, 59)$	$[59, 100)$	
	Children	$\emptyset$	$\{b\}$	$\emptyset$	

Figure 7.   Set of children relative to emitter a.

Learning children tables for all potential emitters in the network is the purpose of our algorithm, whose informal strategy is as follows.

1) High-level informal strategy: The algorithm actually starts the other way around, with nodes determining their set of optimal *parents* in the trees, relative to one complete period of initiation dates and all sources. This is done based on information provided by the T-CLOCKS abstraction described in Section IV. The resulting information is stored in a structure equivalent to the table on Figure 8. Once a node

on b	Initiation date	$[0, 29)$	$[29, 59)$	$[59, 100)$
	Parent	a	c	a
on c	Initiation date	$[0, 19)$	$[19, 59)$	$[59, 100)$
	Parent	b	a	b

Figure 8.   Set of parents relative to emitter a.

has determined its set of parents with respect to a complete period of initiation dates, it notifies each parent by sending the corresponding intervals. Since the network is periodic, these notifications cannot, locally to a notifying node, last more than one period. On the parent side, the intervals are processed upon reception to fill in their children table. The way the intervals can be sent and processed is straightforward. Thus, we focus on explaining how the tables of parents are built.

2) Detailed strategy: There is a clear connexion between the problem of determining which parent is best to obtain a message in a foremost fashion, and the concept of temporal view discussed in Section III. In fact, the relation is clear: for a given source s and initiation date t, the parent that a node should select is precisely the first of its neighbors to provide a temporal view $\phi(s) \geq t$. The T-CLOCKS abstraction described in Section IV allows precisely to track this information. The detailed process is described in Algorithm 3. Its basic principle consists in monitoring the evolution of both direct and indirect views, and record the corresponding neighbors as parents as follows. Whenever the temporal view is improved by means of an indirect journey ($dateImproved()$), the corresponding neighbor is associated with the provided initiation date. More precisely, it is stored in a table that associates it with this date, which corresponds in fact to the *end* of the interval this parent covers (the beginning of the interval being the end of the previous one, circularly). This strategy relies on the fact that if a node can provide a message initiated at t, then it can

also provide any message initiated before t. As for the improvement of the temporal view by means of direct journeys, the algorithm maintains a dedicated variable to remember the current levels (table $level$) and the corresponding neighbors (table $directProxy$). Whenever a notification occurs, whether related to a change in date or level, this variable can be used in $updateRecord()$ to determine what largest initiation date t could have already been covered by these direct journeys. If the date constitute an improvement, it is stored in the parent table together with the corresponding neighbor. A few

---

Algorithm 3 Learning Foremost BTs (as table of parents)

1: $Map <Node, <Date, Node>> parents \leftarrow \emptyset$
2: $Map <Node, Node> directProxy \leftarrow nil$
3: $Map <Node, Integer> level \leftarrow nil$
4: $Date\ startD \leftarrow nil$

5: ___init()___:
6: $startD \leftarrow now()$
7: $TClocks.register(true)$ $\quad \triangleright$ register to T-CLOCKS *with immediate notification of the current level (if any)*

8: ___dateImproved(Node src, Integer date, Node proxy)___:
9: $updateRecord(src)$
10: $parents[src].add(date, proxy)$

11: ___levelChanged(Node src, Integer level, Node proxy)___:
12: $updateRecord(src)$
13: **if** $level \neq +\infty$ **then**
14: $\quad directProxy[src] \leftarrow proxy$
15: $level[src] \leftarrow level$ $\quad \triangleright$ *keeps a local copy of the levels.*

16: ___currentDirectView(Node src)___:
17: $return\ now() - level[src] \times \zeta$ $\quad \triangleright$ *based on the local copy.*

18: ___updateRecord(Node src)___:
19: **if** $directProxy[src] \neq nil$ **then**
20: $\quad$ **if** $currentDirectView() > parents[src].lastDate()$ **then**
21: $\qquad parents[src].add(currentDirectView(),$
$\qquad\qquad\qquad\qquad directProxy[src])$

22: ___when___ $now() == startD + p$:
23: $terminate.$

---

additional remarks:

- In some cases, the algorithm may lead to record consecutively a same parent; if so, intervals can simply be merged.
- Last but not least, the process can be started independently on each node, as long as Algorithm 2 is assumed to have already run on *all* the nodes for at least a duration of $max(|\mathcal{J}|_t : \mathcal{J} \in \mathcal{J}_{\mathcal{G}}^*)$, that is, the *temporal diameter* of the network. This is to ensure that all the nodes know their respective temporal views.

3) Correctness: The correctness of Algorithm 3 essentially follows from periodicity and the correctness of Algorithm 2.

Theorem 12: The execution of Algorithm 3 in a periodically-varying graph $\mathcal{G}$ with known period p results in all nodes selecting the correct set of parents with respect to all Foremost BTs in $O(p)$ time.

Proof: The idea is to prove that the recorded parent for any initiation date t and source s is indeed (any of) the first neighbor to provide a temporal view of s that is greater or equal to t. For any t, this view is either direct or indirect. The information relative to the direct view – level and corresponding neighbor – is locally stored through

lines 14 and 15 whenever it changes (as per Algorithm 2). This allows $currentDirectView()$ to indicate, at any time instant (thanks to Property 11), the corresponding direct view. As for the indirect view, the desired property follows from the way parents are recorded in the parents table: when an higher indirect view is provided ($dateImproved()$), first the neighbor responsible for the current direct view is stored for all initiation dates that it has already been able to cover ($updateRecord()$), then the one responsible for the new indirect view is stored in turn. The same update operation is executed when the level changes, but instead of being stored in turn, the new level proxy replaces the previous one. Due to the periodicity, the algorithm necessarily terminates in $O(p)$ time (in fact, in exactly one period p) because the first parent reappears and can deliver the same initiation dates as before, plus p. ∎

4) Example traces: The tables below show some execution traces based on the example graph of Figure 5 (with respect to emitter a). These traces consider that c starts at time 50 (modulo 100), and b at time 65 (modulo 100). These dates are chosen to reflect a variety of initial conditions and behaviors. The traces include the list of dated notifications and modifications of the parents table for both nodes (Figures 9 and 10), and the resulting tables of parents, using both their original and interval-based representations (Figures 11 and 12).

Date	Event	Parents table for source a
71	$dateImproved(a, 59, c)$	$parents[a].add(59, c)$
1	$levelChanged(a, 1, a)$	
30	$levelChanged(a, 2, c)$	$parents[a].add(29, a)$
40	$levelChanged(a, +\infty, nil)$	$parents[a].add(38, c)$
165	*end of the period*	

Figure 9. Relevant traces with respect to node b and emitter a.

Date	Event	Parents table for source a
50	$levelChanged(a, 1, a)$	
60	$levelChanged(a, +\infty, nil)$	$parents[a].add(59, a)$
11	$levelChanged(a, 2, b)$	
21	$levelChanged(a, 1, a)$	$parents[a].add(19, b)$
50	*end of the period*	

Figure 10. Relevant traces with respect to node c and emitter a.

Initiation date	$\to 59$	$\to 29$	$\to 38$	
Parent	c	a	c	
Initiation date	$[0, 29)$	$[29, 38)$	$[38, 59)$	$[59, 100)$
Parent	a	c	c	a

Figure 11. Resulting table of parents for node b.

Initiation date	$\to 59$	$\to 19$	
Parent	a	b	
Initiation date	$[0, 19)$	$[19, 59)$	$[59, 100)$
Parent	b	a	b

Figure 12. Resulting table of parents for node c.

C. Application 2 – Network backbones

A translation of the concept of *network backbone* in dynamic networks was proposed in [27] as the "subgraph consisting of edges on which information has the potential to flow

the quickest". This concept is *time-dependent* – the backbone varies depending on the emission date that is considered. Precisely, an edge belongs to the backbone relative to time t (denoted $\mathcal{H}_t$), if and only if it is *essential* ("lies on the minimum-delay path between some pair of nodes x and y") relative to time t. Rephrased in our context, an edge is essential with respect to time t iff it is used by a foremost journey starting at time t. It follows that

Theorem 13: The union of all foremost BTs relative to emission date t is a network backbone with respect to t.

As a result, the algorithm presented in Section V-B computes at once, and as a by-product, the backbones relative to all possible dates. Furthermore, these backbones are generalizations of those of [27] in the sense that our context addresses arbitrary long contacts and non-negligible latencies.

D. Application 3 – Fastest broadcast trees

The difference between foremost and fastest broadcast seems not obvious at first sight, due to the fact that both relate to time. While *foremost* refers to minimizing the arrival date, *fastest* refers to minimizing the overall time spent in the system. As such, one might be willing to delay the effective starting date of a broadcast in the purpose of making it faster (which makes sense e.g. in communication networks whose medium is shared exclusively or to minimize a trip duration in the context of transportation networks).

A fundamental difference between fastest and foremost journeys is that finding fastest journeys whose prefixes are themselves fastest may not always be possible. Consider the example in Figure 13, assuming node a is willing to broadcast at time 0. Reaching d in a fastest way requires to send the first message at $49 - \epsilon$ and then propagate the message from b to c anywhere between 60 and 69, thereby implying a duration of at least 12 from a to c. However, faster journeys of duration 2 could exist earlier from a to c. This observation is crucial to formulate the problem of fastest broadcast. Attempting to reach each node using fastest journeys may be relevant in the case of point-to-point communication (and this objective was the one considered in [5]); but it is clearly less relevant in a context of *broadcast* for the aforementioned reason, since this would imply a same message is sent several times over a same edge (e.g. over (bc), once as part of the fastest journey to c, another time as part of the fastest journey to d).
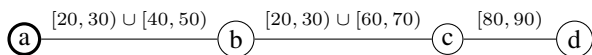

Figure 13. A simple TVG where fastest broadcast does not match fastest journeys (*edge latency* $\zeta = 1$).

Hence, we address the more natural problem of minimizing the *overall duration* of the broadcast, that is, the time elapsed between the first message emission and the last message reception in the whole network. As such, a *fastest broadcast tree* (or fastest BT) corresponds to a union of journeys which may or may not be individually fastest. The main subproblem here is to determine *when* the emitter has the potential to reach all nodes the fastest, *i.e.,* when the "longest" foremost

journey from the emitter to any other node (also called its *temporal eccentricity*), is minimum. Note that learning minimum temporal eccentricities in distributed networks is an interesting problem in its own right, and may be used in other purposes than broadcasting (e.g., electing leader nodes based on their ability to reach all other nodes quickly).

Once the time of minimum eccentricity is known by the emitter, fastest broadcast reduces to perform a *foremost* broadcast at this particular date (by definition, this is optimal). This can be done using the outcomes of the foremost BT algorithm in previous section, or by building a single foremost BT for the selected date (for a single date, foremost BTs can be built by means of a flooding whereby all the nodes record which of their neighbor gave them the message first, followed by local acknowledgments of these relations). In both cases, broadcast itself does not bring original difficulty, we thus focus on the mechanisms by which a node can learn its temporal eccentricity in a periodic TVG. For clarity, we solve the problem respective to a single emitter (contrary to the foremost BT algorithm that dealt with all emitters at once).

1) Learning temporal eccentricities over one period: The temporal eccentricity (or simply eccentricity below) of a node u at date t is formally defined as

$$ecc_u(t) = max\{\hat{d}_{u,t}(v) : v \in V\}, \tag{1}$$

that is, the maximum among all temporal distances (or simply distances below) from u to all other nodes at time t, see Section V-A for definition of the temporal distance and its connection with temporal view.

There is a strong connection between temporal distances and temporal views. Both actually refer to the same quantity seen from different perspectives: the temporal distance is a duration defined locally to an emitter at an emission date, while the temporal view is a date defined locally to a receiver at a reception date. In fact, we have

$$\hat{d}_{u,t_e}(v) = t_r - \phi_{v,t_r}(u) \tag{2}$$

where t_e is an emission date, and t_r is the corresponding earliest reception date.

2) High-level informal strategy: The algorithm consists of inferring (and recording) temporal distances at every node relative to a given emitter, based on the evolution of temporal views monitored through T-CLOCKS and the equivalence relation of Equation 2. Precisely, for a given emitter u, every node $v \neq u$ infers $\hat{d}_u(v)$ from $\phi_v(u)$ over one period and records it in a *distance table*. Since we deal with continuous-time and possibly overlapping contacts, this information is recorded as a set of *intervals* that correspond to the different phases of evolution of the distance (discrete or continuous). Once the distance tables are computed all over the network with respect to emitter u, they are opportunistically aggregated along a tree rooted in u (the aggregation tree may be arbitrary). Aggregation of a children table consists of an segment-wise *maximum* against the local distance table (segments are artificially split, if needed, to be aligned with each other). Once the emitter has aggregated the table of its last child, the final result corresponds to its eccentricity over time (Equation 1).

Any of the minimum values is finally selected and used as initiation date for the broadcast. We now describe how tables of distances are built on the receiver side using T-CLOCKS.

3) Detailed strategy for computing the distance: Let us examine closer the relation between temporal view and temporal distance, through three key properties established in Lemmas 14, 15 and 16.

Lemma 14: Every discrete increase of $\phi_v(u)$ by k corresponds to a continuous decrease of $\hat{d}_u(v)$ during k time units.

Proof: A discrete increase of the view by k at time t (that is, $\phi_{v,t}(u) = \phi_{v,t-\epsilon}(u) + k$), implies the existence of a journey $\mathcal{J}$ from u whose departure is $\phi_{v,t}(u)$ and arrival is t. Switching to the emitter viewpoint, the considered increase implies that *no* journey starting during $[t_1 = \phi_{v,t-\epsilon}(u), t_2 = \phi_{v,t}(u))$ could have arrived before t (otherwise such a journey would imply an intermediate increase of the view by less than k). Therefore, the temporal distance at any point in $[t_1, t_2)$ is fully determined by the arrival of $\mathcal{J}$, and thus for all t' in $[t_1, t_2)$, we have $\hat{d}_{u,t'}(v) = \hat{d}_{u,t_2}(v) + (t_2 - t')$, which corresponds to a continuous decrease of the distance during $t_2 - t_1 = k$ time units. ∎

Lemma 15: Each continuous increase of $\phi_v(u)$ during k time units corresponds to a stagnation of $\hat{d}_u(v)$ during k time units.

Proof: Continuous increases of the view are due to continuums of direct journeys of same level. Such increase during some interval $[t, t+k]$ thus implies a continuum of direct journeys departing over $[\phi_{v,t}(u), \phi_{v,t}(u)+k)$ (since ζ is constant). We thus have $\forall t_e \in [\phi_{v,t}(u), \phi_{v,t}(u)+k), \hat{d}_{u,t_e}(v) = level \times \zeta$ (where $level$ is the level of the considered journeys), which corresponds to a constant during k time units. ∎

Lemma 16: All initiation dates are covered either by a discrete or a continuous increase of the view.

Proof: By nature of the view, which is *past-inclusive, i.e.,* the fact that a message emitted at time t_e could have arrived by some time t_r implies that *any* message emitted before t_e could have also arrived by t_r. (In essence, there is no "gap".) Besides, an increase is necessarily discrete or continuous. ∎

Combination of Lemmas 14, 15 and 16 allows us to state the following general theorem on temporal distances (with constant edge latency ζ).

Theorem 17: The evolution of $\hat{d}_u(v)$ can be fully captured by a sequence of segments of two possible types: *flat* segments (stagnation of the value) and *slope* segments (continuous decrease of the value). This sequence can be inferred at v by associating every continuous (*resp.* discrete) increase of $\phi_v(u)$ to a flat (*resp.* slope) segment of $\hat{d}_u(v)$.

This strategy is the one considered by Algorithm 4, whose underlying principle is to detect and record every transition between two segments of distance by means of transitions in the evolution of the temporal view, using T-CLOCKS. Each transition is recorded as a triplet containing an emission date, the corresponding distance, and the type of segment starting at this emission date (*flat* or *slope*). An example of such sequence is given on Figure 14, representing the distance from node a to node c in the graph of Figure 5 (triangle TVG).

The subtlety here is that the value of the distance at the beginning of a slope segment, as well as the duration of both

t	$\hat{d}_{a,t}(c)$	trend
9	2	flat
19	2	slope
20	1	flat
59	52	slope
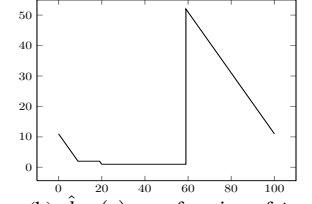
(a) Sequence of triplets for $\hat{d}_{a,t}(c)$

(b) $\hat{d}_{a,t}(c)$ as a function of t

Figure 14. Temporal distance from a to c, as a function of emission date.

Algorithm 4 Computing temporal distances at the receiver, relative to a given emitter.

1: $Map <Date, Distance>\ table \leftarrow \emptyset$
2: $Integer\ currentLevel \leftarrow +\infty$
3: $Date\ startD \leftarrow nil$
4: $Date\ pendingED \leftarrow nil$

5: ***init()***:
6: $\overline{TClocks.register(false)}$ ▷ *register to* T-CLOCKS *without immediate notification of the current level*

7: ***dateImproved (Node** src**, Date** date**, Node** proxy**)**:
8: **if** $src = emitter$ **then**
9: $\quad update(date)$

10: ***levelChanged (Node** src**, Level** level**, Node** proxy**)**:
11: **if** $src = emitter$ **then**
12: $\quad update(now() - level \times \zeta)$
13: $\quad currentLevel \leftarrow level$

14: ***update (Date** newED**)**:
15: **if** $startD = nil$ **then**
16: $\quad startD \leftarrow now()$
17: $\quad pendingED \leftarrow newED$
18: **else**
19: $\quad updateFlat()$
20: $\quad updateSlope(newED)$
21: $\quad$ **if** $now() = startD + p$ **then**
22: $\qquad terminate$

23: ***updateFlat ()***:
24: **if** $currentLevel < +\infty$ **then**
25: $\quad Date\ bestED \leftarrow now() - currentLevel \times \zeta$
26: $\quad$ **if** $bestED > pendingED$ **then**
27: $\qquad$ ▷ *A flat segment is detected*
28: $\qquad table.add(pendingED, currentLevel \times \zeta, "flat")$
29: $\qquad pendingED \leftarrow bestED$

30: ***updateSlope (Date** newED**)**:
31: **if** $newED > pendingED$ **then**
32: $\quad$ ▷ *A slope segment is detected*
33: $\quad table.add(pendingED,\ now() - pendingED,\ "slope")$
34: $\quad pendingED \leftarrow newED$

---

types of segments, becomes known only *at the end* of the corresponding temporal view segment; therefore, the algorithm always records values relative to a previously pending emission date ($pendingED$). Precisely, whenever an event related to the temporal view occurs, whether it be caused by the arrival of a better indirect journey ($dateImproved()$) or a change in the level ($levelChanged()$), the same update function is called involving the following sub-routines:

- $updateFlat()$: If the current level ($currentLevel$) was not infinite, then the corresponding continuum may have delivered new emission dates (checked in lines 24 to 26). If this is the case, then a flat segment is inserted for the pending date using distance value $currentLevel \times \zeta$ (see Proof 15).

- $updateSlope()$: If the new event implies a discrete improvement of the view (line 31), then a slope segment must be created, and only then the distance at the pending emission date becomes known. The segment being a continuous decrease, the value corresponds to the distance at the newly received emission date plus the time elapsed between that date and the pending date (see Proof 14), *i.e.,* $(now() - newED) + (newED - pendingED) = now() - pendingED$ (*c.f.* line 33).

Observe that nothing prevents a same event from inducing both a flat *and* a slope segments (e.g. when c's level relative to a changes at time 21 in the graph of Figure 5). Figure 15 shows a final example of execution trace corresponding to the most relevant steps at node c corresponding to the values in Figure 14. The beginning of execution at node c was arbitrarily set anytime between date 21 and date 60 (modulo $p = 100$).

Date	Event	Action
60	$levelChanged(+\infty)$	$startD \leftarrow 60$
		$pendingED \leftarrow 59$
		$currentLevel \leftarrow +\infty$
111	$levelChanged(2)$	$updateFlat()$
		$updateSlope(109)$
		$\mathbf{table.add(59, 52, "slope")}$
		$pendingED \leftarrow 109$
		$currentLevel \leftarrow 2$
121	$levelChanged(1)$	$updateFlat()$
		$\mathbf{table.add(109, 2, "flat")}$
		$pendingED \leftarrow 119$
		$currentLevel \leftarrow 1$
		$updateSlope(120)$
		$\mathbf{table.add(119, 2, "slope")}$
		$pendingED \leftarrow 120$
		$currentLevel \leftarrow 1$
160	$levelChanged(+\infty)$	$updateFlat()$
		$\mathbf{table.add(120, 1, "flat")}$
		$terminate$

Figure 15. Relevant traces at node c relative to emitter a (assuming an arbitrary start of execution between dates 21 and 60).

Theorem 18: The execution of Algorithm 4 in a periodically-varying graph $\mathcal{G}$ with known period p, relative to emitter u, results in every node v capturing correctly the evolution of $\hat{d}_u(v)$ over one complete period, in $O(p)$.

Proof: The argument is based on the correctness of T-CLOCKS combined with the strategy of Theorem 17 that Algorithm 4 implements. Theorem 17 states that each segment of evolution of the temporal distance corresponds to a segment of evolution of the temporal view (the precise characterization of this correspondence being given by the proofs of Lemmas 14 and 15). Correctness follows from detecting all transitions in the evolution of the temporal view (events $levelChanged()$ and $dateImproved()$), guaranteed by T-CLOCKS, and transposing the corresponding segments into segments of the evolution of the temporal distance by means of procedures $updateFlat()$ and $updateSlope()$ in Algorithm 4 (both of which are called for every such transition, and each of which checks for the need to create the corresponding type of segment according to Lemmas 14 and 15). One period exactly after the first record is inserted, a last record completes the distance table, then the algorithm terminates (lines 21 and 22). Because the dates are taken modulo p, this last record completes the distance table relative to a complete period. ∎

4) Aggregating distance tables back to the emitter: Distance tables relative to a given emitter are aggregated along a tree rooted at that node. Such a tree may be arbitrary and built, for instance, using the same strategy as mentioned above for single foremost BTs (*i.e.,* flooding a dedicated message from the emitter and detecting from which neighbor this message is first received at every node, followed by local acknowledgments of the corresponding relations). Once a node has computed its distance table and aggregated the tables of all its children (if any), it sends the resulting table to its parent. The aggregation of a children table (described below) consists of a segment-wise *maximum* among both tables for all emission dates (see Figure 16 for a visual illustration). Once
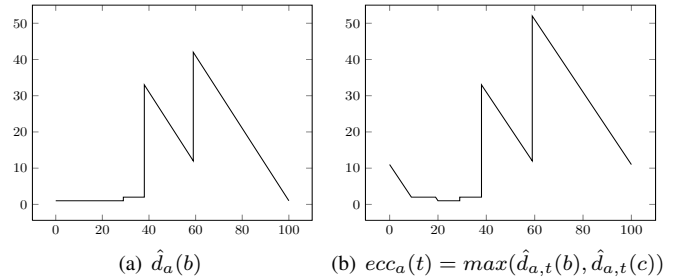

(a) $\hat{d}_a(b)$ (b) $ecc_a(t) = max(\hat{d}_{a,t}(b), \hat{d}_{a,t}(c))$

Figure 16. Aggregation of distances. The left curve (distance from a to b) is combined to the curve of Figure 14(b)) (distance from a to c) in order to yield the eccentricity of node a over one period.

the emitter has aggregated all its children tables within its own table (initially consisting of a single flat segment of value 0), the final result corresponds to the maximum distance in every point in time among all nodes, and therefore (Equation 1) represents the evolution of its eccentricity over one period. Once this table known, it finally selects any of the dates at which the eccentricity is minimum (relative to which the foremost BT is to be built), then terminates. We now describe the aggregation process in more detail.

The purpose of aggregation is to select the maximum value among all distances in every point in time. Aggregating two tables based on their segments would be relatively straightforward if the segments were horizontally aligned (*i.e.,* if the period was split into the same sequence of intervals in both tables) and vertically non-crossing (*i.e.,* given any pair of aligned segments, one of them remains higher than or equal to the other during the corresponding interval). In such an ideal case, the maximum operation between two aligned segments, say $\mathcal{S}_i = (t_i, \hat{d}_{t_i}, trend_i)$ and $\mathcal{S}'_i = (t_i, \hat{d}'_{t_i}, trend'_i)$ is straightforward; it may consist in selecting $\mathcal{S}_i$ iff

$$(\hat{d}_{t_i} > \hat{d}_{t'_i}) \vee (\hat{d}_{t_i} = \hat{d}_{t'_i} \wedge trend_i = flat); \text{ and } \mathcal{S}'_i \text{ otherwise.}$$

Theorem 19: Given two segments whose intervals are aligned and values are non-crossing, the above logic selects the correct maximum segment.

Proof: Let us examine separately the cases that $\hat{d}_{t_i} > \hat{d}_{t'_i}$, $\hat{d}_{t_i} < \hat{d}_{t'_i}$, or $\hat{d}_{t_i} = \hat{d}_{t'_i}$. If $\hat{d}_{t_i} > \hat{d}_{t'_i}$ $\mathcal{S}_i$ is selected irrespective of the trend (left clause). This condition is sufficient because segments cannot cross. If $\hat{d}_{t_i} < \hat{d}_{t'_i}$ the formula evaluates to false, leading to select $\mathcal{S}'_i$, which is correct for the same reason. If $\hat{d}_{t_i} = \hat{d}_{t'_i}$ then three cases are again possible: either

both segments are flat (and therefore equals, $\mathcal{S}_i$ is arbitrarily selected), or both segments are a slope (and therefore equals, $\mathcal{S}_i'$ is arbitrarily selected), or $\mathcal{S}_i$ is flat and $\mathcal{S}_i'$ is a slope (a flat segment starting at the same high as a decreasing slope will necessarily remain higher; $\mathcal{S}_i$ is therefore selected), or $\mathcal{S}_i$ is a slope and $\mathcal{S}_i'$ is flat (symmetrical case, $\mathcal{S}_i'$ will be selected). ∎

In reality, segments do cover intervals of various sizes, and the distance values may also "cross" even if the intervals are aligned (*e.g.* a slope segment starting at a slightly higher value than a flat segment). Rather than complicating the above logic of aggregation, we pre-process the two tables before aggregation as follows:

1) *Split the segments so as to align both tables:* Splitting a segment $(t_i, \hat{d}_{t_i}, trend)$ at date t_i' comes to insert a subsequent entry $(t_i', \hat{d}_{t_i'}, trend)$ such that $\hat{d}_{t_i'} = \hat{d}_{t_i}$ if $trend = flat$ and $\hat{d}_{t_i'} = \hat{d}_{t_i} - (t_i' - t_i)$ if $trend = slope$. Each table undergoes such a split relative to every index date ($t_i's$) that exists only in the other table. An additional split relative to time 0 may also be added for convenience (if not already present).

2) *Split further to eliminate crossing values:* Given two aligned segments of size l, if one is flat and the other a slope, say $(t, \hat{d}_{f_t}, flat)$ and $(t, \hat{d}_{s_t}, slope)$, and $0 < \hat{d}_{s_t} - \hat{d}_{f_t} < l$, then both segments are split at $t + (\hat{d}_{s_t} - \hat{d}_{f_t})$ precisely.

Theorem 20: These pre-processing steps produce tables whose segments are aligned and non-crossing.

Proof: First of all, the fact that a split preserves the consistency of a table is clear from the formulas in step 1 (*i.e.,* using the same distance value if the segment is flat; decreasing it otherwise by an amount equal to the time elapsed since the beginning of the segment). Since each table undergoes a split with respect to every index date that exists only in the other table, the resulting tables must contain the same index dates (aligned segments). As for the crossing segments, let us first observe that two flat segments cannot cross since their value is a constant, neither can two slope segments because their value decrease at a same rate (the rate of time). Therefore, crossings may only occur between segments of different trends. Let $\delta = \hat{d}_{s_t} - \hat{d}_{f_t}$ be the difference of initial value between the slope segment and the flat segment. The segments cannot cross if the slope segment is already below the flat segment at the beginning of the interval (a slope is always decreasing), neither can they if δ is larger than l because the final value of the slope segment will still be above that of the flat segment. Therefore, crossings only occur if $0 < \delta < l$. Besides, both types of segment trend being linear, two given segments cannot cross more than once. It is therefore sufficient to split them with respect to their crossing point $t + \delta$. ∎

Figure 17 illustrates the aggregation of two distance tables that relate again to the same example, *i.e.*, $\hat{d}_{a,t}(b)$ and $\hat{d}_{a,t}(c)$ in the triangle TVG of Figure 5. Note that no situation of crossing segments had to be handled. Based on the final aggregation, the emitter can decide when to initiate the intended fastest broadcast (here, anytime between dates 20 and 29 modulo p).

Theorem 21: The strategy presented in this section correctly leads to the emitter learning the evolution of its temporal eccentricity over one period.

Proof: Follows from the combination of Theorems 18, 19,

ED	Dist	Type
0	1	flat
29	2	flat
38	33	slope
59	42	slope

$\oplus$

ED	Dist	Type
(0)	(11)	(slope)
9	2	flat
19	2	slope
20	1	flat
59	52	slope

$=$

ED	Ecc	Type
0	11	slope
9	2	flat
19	2	slope
20	1	flat
29	2	flat
38	33	slope
59	52	slope

Figure 17. Aggregation of two distance tables, corresponding to $\hat{d}_{a,t}(b)$ and $\hat{d}_{a,t}(c)$ in the graph of Figure 5.

and 20. Each node in the network computes correctly the evolution of the temporal distance from the emitter (Theorem 18); then, whenever two tables of distance are aggregated (which is done opportunistically along an arbitrary delay-tolerant tree rooted at the emitter), they are pre-processed to make the segments of evolutions aligned and non-crossing (Theorem 20), which property is exploited to compute their (segment-wise) maximums (Theorem 19). Aggregation ultimately results in the temporal eccentricity of the emitter (by Equation 1). ∎

VI. CONCLUDING REMARKS AND OPEN PROBLEMS

This paper addressed the question of measuring temporal lags in highly-dynamic networks. We formulated this problem in a distributed setting, and solved it for the general case of arbitrary long (and possibly overlapping) contacts, with non-negligible (though fixed and known) latencies on the edges. The paper also illustrated how temporal lags can be leveraged to solve concrete network problems, such as the construction of foremost and fastest broadcast trees in periodically-varying graphs. Interestingly, the algorithm solving foremost broadcast also solved, as a by-product, the construction of time-dependent backbones, a concept recently formulated in [27]. Feasibility of the main problem was demonstrated constructively, with an algorithm that adapts vector clocks to the task of measuring temporal lags in this general context where both direct and indirect journeys can co-exist in the network. We called this algorithm (or abstraction) T-CLOCKS.

The complexity of T-CLOCKS is presumed high, and implementing it in practical scenario shall require further optimization (same for [27] with punctual contacts). Characterizing and improving the communication costs is all the more relevant if T-CLOCKS is used as a building block for other algorithms. Indeed, any improvement would have instant repercussions – new upper bounds, typically – on *all* derivative problems and algorithms. This raises interesting questions on the analysis of algorithmic complexity in DTNs, which strongly depends on the number of *topological events* during execution (in our case, a majority of actions are triggered by such events). Complexity of T-CLOCKS could also be assessed through simulations, based on publicly available networks traces (see e.g. the Crawdad project [19]). Another interesting question is whether some assumptions could be relaxed such as having fixed latency, which becomes less realistic as the network size increases (in the current version, nodes exchange messages of $O(n)$ size).

Finally, we believe T-CLOCKS could be used in various delay-tolerant reformulations of classical distributed problems

such as election (e.g., electing a leader whose temporal eccentricity is minimum) or compact routing (e.g., through assigning names to the nodes based on their temporal properties).

ACKNOWLEDGMENTS

REFERENCES

[1] B. Awerbuch and S. Even. "Efficient and reliable broadcast is achievable in an eventually connected network," in *Proc.* 3rd *ACM symp. on Principles of Distributed Computing (PODC'84)*, pp. 278–281, 1984.

[2] K. Berman, "Vulnerability of scheduled networks and a generalization of Menger's Theorem," *Networks*, vol. 28, no. 3, pp. 125–134, 1996.

[3] B. Brejová, S. Dobrev, R. Královič, and T. Vinař, "Routing in carrier-based mobile networks," in *Proc. 18th Intl. Conf. on Structural Information and Communication Complexity (SIROCCO)*, pp. 222–233, 2011.

[4] S. Bhadra and A. Ferreira, "Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks," in *Proc.* 2*nd Intl. Conference on Ad Hoc, Mobile and Wireless Networks (ADHOCNOW)*, 2003, pp. 259–270.

[5] B. Bui-Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Intl. J. of Foundations of Comp. Science*, vol. 14, no. 2, pp. 267–285, April 2003.

[6] A. Casteigts, S. Chaumette, and A. Ferreira. Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Proc. 16th Intl. Conf. on Structural Information and Communication Complexity (SIROCCO)*, pp. 126–140, 2009. (Full version in arXiv:1102.5529.)

[7] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro, "Deterministic computations in time-varying graphs: Broadcasting under unstructured mobility," in *Proc. 5th IFIP Conf. on Theoretical Computer Science (TCS)*, pp. 111–124, 2010.

[8] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *Proc.* 10*th Intl. Conf. on Ad Hoc, Mobile and Wireless Networks (ADHOCNOW)*, pp. 346–359, 2011.

[9] B. Choi, H. Liang, X. Shen, and W. Zhuang, "DCS: distributed asynchronous clock synchronization in delay tolerant networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 3, pp. 491–504, 2011.

[10] A. Chaintreau, A. Mtibaa, L. Massoulie, and C. Diot, "The diameter of opportunistic mobile networks," *Communications Surveys & Tutorials*, vol. 10, no. 3, pp. 74–88, 2008.

[11] H. Dubois-Ferriere, M. Grossglauser, and M. Vetterli, "Age matters: efficient route discovery in mobile ad hoc networks using encounter ages," in *Proceedings ACM International Symposium on Mobile Ad Hoc Networking & Computing (MOBIHOC)*, pp. 257–266, 2003.

[12] A. Ferreira, "Building a reference combinatorial model for MANETs," *IEEE Network*, vol. 18, no. 5, pp. 24–29, 2004.

[13] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66.

[14] P. Flocchini, M. Kellett, P. Mason, and N. Santoro, "Searching for black holes in subways," in *Theory of Computing Systems*, pp. 1–27, in press.

[15] P. Flocchini, B. Mans, and N. Santoro, "Exploration of periodically varying graphs," in *Proc. 20th Intl. Symp. on Algorithms and Computation (ISAAC)*, 2009, pp. 534–543.

[16] F. Greve, L. Arantes, and P. Sens. What model and what conditions to implement unreliable failure detectors in dynamic networks? In *3rd W. on Theoretical Aspects of Dynamic Distributed Systems (TADDS)*, 2011.

[17] M. Grossglauser and M. Vetterli, "Locating nodes with EASE: Last encounter routing in ad hoc networks through mobility diffusion," in *Proc. 22nd Conference on Computer Communications (INFOCOM)*, vol. 3, 2003, pp. 1954–1964.

[18] P. Holme, "Network reachability of real-world contact sequences," *Physical Review E*, vol. 71, no. 4, p. 46119, 2005.

[19] D. Kotz and T. Henderson, "Crawdad: A community resource for archiving wireless data at dartmouth," *IEEE Pervasive Computing*, vol. 4, no. 4, pp. 12–14, 2005.

[20] F. Kuhn, T. Locher, and R. Oshman, "Gradient clock synchronization in dynamic networks," in *Proc. of the 21st symp. on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2009, pp. 270–279.

[21] P. Jacquet, B. Mans, and G. Rodolakis, "Information propagation speed in mobile and delay tolerant networks," in *IEEE Transactions on Information Theory*, 56(1), pp. 5001–5015, Oct 2010.

[22] D. Ilcinkas and A.M. Wade, "On the power of waiting when exploring public transportation systems." in *Proc. of 15th Intl. Conf. On Principles Of Distributed Systems (OPODIS)*, Dec 2011.

[23] S. Jain, K. Fall, and R. Patra, "Routing in a delay tolerant network," in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Comp. Comm. (SIGCOMM)*, 2004, pp. 145–158.

[24] E. Jones, L. Li, J. Schmidtke, and P. Ward, "Practical routing in delay-tolerant networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 8, pp. 943–959, 2007.

[25] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," in *Proceedings 32nd ACM Symposium on Theory of Computing*, 2000, p. 513.

[26] A. Keränen and J. Ott, "DTN over aerial carriers," in *Proceedings 4th ACM Workshop on Challenged Networks*, 2009, pp. 67–76.

[27] G. Kossinets, J. Kleinberg, and D. Watts, "The structure of information pathways in a social communication network," in *Proc. 14th Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2008, pp. 435–443.

[28] V. Kostakos, "Temporal graphs," *Physica A*, vol. 388, no. 6, pp. 1007–1023, 2009.

[29] A. Lindgren, A. Doria, and O. Schelén, "Probabilistic routing in intermittently connected networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 7, no. 3, pp. 19–20, 2003.

[30] C. Liu and J. Wu, "Scalable routing in cyclic mobile networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 9, pp. 1325–1338, 2009.

[31] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[32] B. Sundararaman, U. Buy, and A. Kshemkalyani, "Clock synchronization for wireless sensor networks: a survey," *Ad Hoc Networks*, vol. 3, no. 3, pp. 281–323, 2005.

[33] M. Yamashita and T. Kameda, "Computing on anonymous networks: Part I and II," *IEEE Trans. on Par. and Distributed Systems*, vol. 7, no. 1, pp. 69 – 96, 1996.

[34] Z. Zhang, "Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges," *IEEE Communications Surveys & Tutorials*, vol. 8, no. 1, pp. 24–37, 2006.

Arnaud Casteigts is Assistant Professor at the University of Bordeaux, France, where he joined after a four-year postdoctoral stay at the University of Ottawa, Canada. His research interests include distributed computing in static and dynamic networks, private data analysis and visualization of algorithms.

Paola Flocchini is University Research Chair in Distributed Computing at the School of Electrical Engineering and Computer Science (University of Ottawa). Her main research interests are in distributed algorithms, distributed computing, algorithms for mobile agents and autonomous robots, and cellular automata.

Bernard Mans is Professor for the Department of Computing at Macquarie University, Sydney, Australia. His research interests centre on algorithms and graphs for distributed and mobile computing. In 2003, he was the HITACHI Chair 2003 at INRIA, France.

Nicola Santoro is Distinguished Research Professor of Computer Science at Carleton University. He has been involved in distributed computing from the beginning of the field, authoring many seminal papers and founding the main theoretical conferences in the area. He is the author of the book "Design and Analysis of Distributed Algorithms" (Wiley 2007). His main current research is on distributed algorithms for mobile entities and dynamic networks.