Time-Aware Reactive Storage in Wireless Edge Environments

Conference Paper · November 2019

DOI: 10.1145/3360774.3360828

5 authors, including:

João A. Silva
Universidade NOVA de Lisboa
14 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Hervé Paulino
Universidade NOVA de Lisboa
72 PUBLICATIONS **176** CITATIONS
SEE PROFILE

João Lourenço
Universidade NOVA de Lisboa
84 PUBLICATIONS **345** CITATIONS
SEE PROFILE

João Leitão
Universidade NOVA de Lisboa
31 PUBLICATIONS **340** CITATIONS
SEE PROFILE

Time-Aware Reactive Storage in Wireless Edge Environments

João A. Silva, Hervé Paulino, João M. Lourenço, João Leitão, and Nuno Preguiça
NOVA Laboratory for Computer Science and Informatics, Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2829-516 Caparica, Portugal
jaa.silva@campus.fct.unl.pt {herve.paulino, joao.lourenco, jc.leitao, nuno.preguica}@fct.unl.pt

ABSTRACT

Nowadays, smart mobile devices generate huge amounts of data in all sorts of gatherings. Much of that data has localized and ephemeral interest, but can be of great use if shared among co-located devices. However, these devices often experience poor connectivity, leading to availability issues if applications' storage and logic are fully delegated to a remote cloud infrastructure. In turn, the edge computing paradigm pushes computations and storage beyond the data center, closer to end-user devices where data is generated and consumed. Thus, enabling the execution of certain components of edge-enabled systems directly and cooperatively on edge devices. In this paper, we address the challenge of supporting reliable and efficient data storage and dissemination among co-located wireless mobile devices without resorting to centralized services or network infrastructures. We propose Thyme, a novel time-aware reactive data storage system for wireless edge networks, that exploits synergies between the storage substrate and the publish/subscribe paradigm. We present the design of Thyme and evaluate it through simulation, characterizing the scenarios best suited for its use. The evaluation shows that Thyme allows for reliable notification and retrieval of relevant data with low overhead and latency.

CCS CONCEPTS

• **Software and its engineering** → *Publish-subscribe / event-based architectures*; • **Human-centered computing** → *Mobile devices.*

KEYWORDS

distributed storage, publish/subscribe, wireless networks, mobile devices, edge computing

ACM Reference Format:
João A. Silva, Hervé Paulino, João M. Lourenço, João Leitão, and Nuno Preguiça. 2019. Time-Aware Reactive Storage in Wireless Edge Environments. In *16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), November 12–14, 2019, Houston, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3360774.3360828

1 INTRODUCTION

We are witnessing a rapid growth of both the capabilities and amount of mobile devices worldwide [**?**]. As such, there is a wide adoption of smartphones and tablets for performing the most diverse activities, from leisure to work-related tasks. Hence, the volume of data, like user-generated content and sensor data, generated by these devices is growing rapidly [**?**].

Much of the data generated by mobile devices in all sorts of social gatherings (like sports events, protests, music festivals) has *localized* and *ephemeral* interest. People in such events are usually interested in similar types of information (e.g., statistics and videos at sports events), and such interest typically diminishes over time. Thus, swift and spontaneous data storage and dissemination among neighboring mobile devices can be of great use. For instance, smartphones carried by people in such gatherings can collect lots of useful data that, when shared among co-located devices, may help others discover new points of interest, enjoy videos of key moments (from multiple viewpoints), or avoid waiting lines.

In many scenarios, making information available may be of paramount importance (e.g., disaster situations [**?**]), or just helpful (e.g., crowded events [**?**]). Being dependent on infrastructure access to support such use cases is unfeasible due to their potential overload or destruction. Even assuming the availability of infrastructure, transferring large amounts of data to and from the cloud can lead to network congestion, processing delays, and possible monetary costs. Furthermore, in those scenarios, mobile devices often experience poor connectivity, leading to *availability* issues if applications' storage and logic are fully delegated to a remote cloud infrastructure. Still, the non-negligible costs associated with network infrastructure setup (i.e., adding access points) further motivates the need to have devices interact through an infrastructure-less or ad-hoc network. Thus, the main question we address in this paper is: *how to support reliable and efficient data storage and dissemination among co-located wireless mobile devices without resorting to centralized services and subsisting with no network infrastructure?*

Together, the ubiquitous smart mobile devices, the opportunistic gathering of users, and the growing pervasiveness of edge computing environments [**?**], have enabled novel opportunities for data storage and dissemination at the *network edge*. In fact, it is more efficient to communicate and distribute information among *nearby* devices than to use distant centralized intermediaries [**?**]. By storing data near its source, applications can be more responsive, while *relieving some of the load* from cloud and network infrastructures.

Allowing systems' components to actively and directly collaborate in the edge requires some form of distributed data repository as to share and disseminate information. Thus, we propose Thyme, a novel time-aware reactive data storage system for networks of mobile devices, that exploits *synergies* between the storage substrate and the publish/subscribe (P/S) communication paradigm. It fuses the storage interface with a P/S abstraction, enabling co-located mobile devices to store and disseminate data reliably among them.

MobiQuitous, November 12–14, 2019, Houston, TX, USA

Contrary to previous solutions, queries are in the form of *subscriptions* that have a specific *time scope* defining when they are active. Leveraging this novel time-aware abstraction, Thyme is able to achieve robust, efficient and timely data storage, dissemination, and querying. It also allows both the notification and retrieval of relevant data with low overhead and latency, using limited bandwidth, and under message losses and node failures.

In typical storage systems [**?** **?** **?**], users are required to *actively* and *explicitly* search for the desired data. Since the kind of distributed environments we target are highly volatile and dynamic, we adopt a *reactive* and *loosely coupled* data dissemination mechanism [**?**]. By integrating a P/S abstraction, users (or applications) can register their interests, being subsequently notified of any data items matching those interests. This allows users to quickly discover what data exists in the system in a reactive manner, and *only* be notified about data they are interested in.

In the kind of gatherings we are addressing, individual moments are intrinsically tied by time relations (e.g., the band performing at time x in the music festival, or the second speech in a rally). Also, people are often interested in information with these associated time references (e.g., find photos of the opening band). Hence, Thyme considers *time* to be a first order dimension. Subscriptions include a time frame that defines their *active time-span*, either in the future, in the present, or in the past, effectively providing the full *time decoupling* of the P/S paradigm [**?**].

We present two different approaches to Thyme. The first one, Thyme-LS, follows a simple, yet effective, unstructured approach using local storage and query flooding. The second more intricate one, Thyme-DCS, is inspired by the fact that geographical positions have a close relation to topology in wireless networks, and follows a data-centric storage (DCS) approach [**?**], whereby we build a storage substrate over a geographic hash table (GHT) [**?**].

Although previous systems presented in the literature offer some features similar to Thyme (e.g., tuple spaces [**?** **?**], or peer-to-peer (P2P) systems [**?** **?**]), none provides the same characteristics (as we detail in §2). Thus, to the best of our knowledge, Thyme is the first system to provide reliable reactive storage for wireless edge networks that may be effectively and efficiently used in either small, medium, and large scale scenarios.

In summary, the main contributions of this paper are the following: i) a reactive storage system with intrinsic time-awareness, that fuses a P/S abstraction with the storage substrate, and allows queries within a specific time scope (§3); ii) the design of Thyme, a novel time-aware reactive storage system (§4), and our two approaches to this proposal—the unreliable Thyme-LS (§5), and the reliable Thyme-DCS (§6); and iii) the characterization of the scenarios best suited for the use of the proposed solutions and their reliability trade-offs, through simulation (§7).

2 RELATED WORK

Typical P/S systems are stateless, i.e., producers and consumers only receive data if online at the same time. Hence, the notion of publication persistence has not been addressed in most systems. Some approaches for wired settings exploit the concept of a persistent data repository, by means of distributed buffers [**?**] or traditional databases [**?**]. However, such solutions do not consider time as a first class dimension of the P/S abstraction. Furthermore, solutions for wired scenarios cannot be easily adapted for wireless setting where connectivity is not stable. In the particular context of wireless settings, Chapar [**?**] is, as far as we know, the only persistent P/S system. However, it only assigns time to publications, which are buffered only until their lifetime expires. In Thyme, subscriptions have their time scope assigned, while publications are permanently stored. Thus, new subscribers can always request previously published data. Moreover, Chapar is not functionally symmetric, achieving poor load balancing, an aspect that has been explicitly considered in the design of Thyme.

Krowd [**?**] and Ephesus [**?**] enable content sharing and storage among nearby mobile devices. Both are sustained by classical distributed hash tables and need some kind of network infrastructure for inter-device communication. Of the two, Ephesus is the only to address device mobility or failure, and data availability, via replication. In turn, Thyme supports several wireless technologies, and targets multi-hop environments using a GHT, known for being more suitable in wireless networks. It also employs several replication mechanisms to address mobility and data availability.

PAN [**?**] and Phoenix [**?**] are two systems for reliable storage in mobile ad-hoc networks (MANETs). PAN is an asymmetric system based on probabilistic quorums, and Phoenix uses a round-based simple quorum protocol for one-hop networks only. iTrust [**?**] and PDS [**?**] focus on data discovery and retrieval on co-located devices. iTrust is based on random walk techniques, while PDS is inspired in information-centric networking. PDS's aggressive caching policy can lead to serious storage overheads, and since data is only cached if requested, less popular data may disappear. iTrust does not address data availability issues.

All these storage systems employ the request/reply model, where peers have to proactively search for content. In turn, Thyme explores synergies between the P/S paradigm and the storage substrate to provide both persistent publications and a reactive interaction model, thus allowing applications to react to new data being generated and stored.

Other approaches based on opportunistic and delay-tolerant networking [**?** **?** **?**] provide communication in the presence of intermittent connectivity. Content dissemination is best effort, i.e., it depends on the willingness of interested nodes to carry such content. These systems also provide a reactive interaction model for data retrieval. They are, however, devised for extreme environments that relax temporal restrictions to the order of hours or days.

Systems like TuCSoN [**?**], LIME [**?**], and TOTA [**?**] adapted the tuple spaces model [**?**] for mobile and wireless environments. Besides the model's proactive operations, these systems allow actions to be performed as *reactions* to certain events. Although reactions are similar to Thyme subscriptions, there are significant differences. First, reactions always execute on the client side, i.e., on the host that installed it, and always receive the tuple that triggered the reaction. This does not allow load balancing when executing the reactions and when matching reactions with tuples. It also generates more traffic than actually required, because it is not possible to filter data at the source. In Thyme, subscription matching is executed by random peers that may change in each matching, thus improving load balancing. Another major difference is that tuple spaces do not separate data and metadata management. That is, both have

to be represented as tuples. Since tuples are immutable, the only way of modifying metadata is to remove and insert a new (changed) tuple, which may trigger unwanted reactions. This can be bypassed by making an intricate decomposition of the metadata into several tuples. Although this may work in small scale scenarios, it can quickly become cumbersome, and penalize performance in large scale scenarios, as targeted by Thyme.

TuCSoN was designed for mobility in Internet environments and presents the notion of programmable tuple spaces. It is not easily adaptable to dynamic wireless environments (e.g., it assumes reliable communication), and its reactions do not allow the same kind of behavior as Thyme's subscriptions.

In LIME, when peers are within range, the contents of the tuples spaces of each peer are transiently shared, forming a federated tuple space. The contents of these virtual tuple spaces evolve in time according to the current connectivity pattern. Although reactions enable tuple spaces to react to the insertion of relevant tuples, they are sensitive to hosts' connectivity, which is not sufficient to generally support distributed services or applications. Therefore, LIME was devised for small scale scenarios. In turn, Thyme leverages a lightweight flooding approach or a GHT for ensuring the best possible connectivity in large scale scenarios, and its routing schemes jointly with its replication mechanisms allow the matching of publications against subscriptions of all peers in the network.

In TOTA, tuples can autonomously propagate in the network. Subscriptions only react to changes in a node's local tuple space. To achieve something similar to Thyme, data should be propagated to every network node in order for subscriptions to be matched against the data. Otherwise, some nodes would not be notified about relevant data. TOTA also requires every node to execute the matching of subscriptions against tuples, thus suffering from redundant work and poor load balancing. In contrast, Thyme does not require data to be replicated throughout the network and allows for better load balancing.

Regarding *app stores*, there are several applications for sharing data between mobile devices, e.g., SuperBeam [**?**] and Xender [**?**] allow synchronous one-to-many data exchange. However, data is only available while its owner is online.

3 TIME-AWARE REACTIVE STORAGE

Typical storage systems provide a request/reply *proactive* interaction model, while in most publish/subscribe (P/S) systems, publications are *transient*. To overcome such shortcomings, we build strong synergies between the storage substrate and the P/S paradigm. On the one hand, the storage substrate leverages the P/S abstraction to provide a *reactive* interaction model whereby users register their interests through subscriptions and are notified as relevant data is generated. On the other hand, the P/S abstraction takes advantage of the storage substrate to provide *persistent* publications, enabling the time-awareness concept and providing the full time decoupling [**?**].

Our storage interface provides the usual operations: insert, retrieve, and delete. Additionally, due to its integration with the P/S abstraction, it also offers the regular P/S operations: publish, subscribe, and unsubscribe. All operations are asynchronous, receiving their results through callbacks.

3.1 Inserting & Publishing Data

In our model, the insert and publish operations are *merged* together. As a result, the insertion of a data object into storage may trigger the sending of notifications to subscribers.

A data object is the basic unit of work and is seen as an opaque set of bytes. Every object has some associated metadata that consists of the following: the object identifier; a set of tags related with the object, e.g., hashtags used in social networks; a summary of the object, e.g., a thumbnail of an image; the object insertion timestamp; and the owner's node identifier.

Tags are used as topics for subscriptions, thus enabling a *topic-based* P/S system. Although topic-based addressing is not as expressive as content-based systems [**?**], it requires far less filtering and computations, which fits our target environments populated by battery-constrained mobile devices. Still, this tagging feature provides a flexible annotation scheme, e.g., by adding the owner's node identifier to the tags of its own objects, an application can enable the retrieval of all the objects stored by a certain node/user.

3.2 Deleting Data

The delete operation removes an object from storage, making it inaccessible to future subscriptions. Note that subscriptions targeting the past will not see deleted objects, even if these were initially available in the subscription's time frame.

3.3 Subscribing

A subscription consists of the following: its identifier; the query defining which tags are relevant; the timestamps defining when the subscription's time frame starts and expires (ts^{s} and ts^{e}, respectively); and the subscriber's node identifier.

Unlike typical topic-based P/S systems, that only allow one topic per subscription, we support *arbitrary* propositional logic formulas where literals are tags associated with objects.

The ts^{s} and ts^{e} timestamps specify the subscription's time frame, where the special value $\perp$ represents, respectively, the times at which the system started and stopped to exist. With a subscription issued at time t: $ts^{\mathrm{s}} = \perp \wedge ts^{\mathrm{e}} = t$ matches events that happened before the subscription (this allows a typical search or find operation on the storage substrate); $ts^{\mathrm{s}} = t \wedge ts^{\mathrm{e}} = \perp$ matches events after or concurrent with the subscription; and $ts^{\mathrm{s}} = ts^{\mathrm{e}} = \perp$ matches all the past and future events in the system. These parameters can also take any concrete timestamp value.

Due to the unreliable nature of our target (wireless) environment, subscribers are notified of all relevant data in a *best effort* manner. Notifications are triggered upon an insertion, by detecting that the object being stored matches existing subscriptions; and upon a subscription that spans into the past, by detecting that this new subscription matches previously stored objects. Notifications are sent to the respective subscribers carrying *only* the metadata of the matching objects.

The unsubscribe operation revokes a subscription before it naturally expires after its end timestamp, ts^{e}.

When subscribing for a popular tag, that spans into the past, the subscriber might get flooded by a large amount of (past) notifications. To attenuate this problem, when subscribing for a time frame in the past, a subscriber is only notified about the n most recent

objects from a total of x matching objects. Then, if interested, a subscriber can request more of those objects, receiving the notifications in *expressly requested batches*. All the subsequent matching objects will be notified as usual.

3.4 Retrieving Data

Users are notified *only* about data they are interested in, allowing them to discover what data exists in a reactive manner. Even so, a typical search operation can be done by subscribing with timestamps $ts^s = \perp$ and $ts^e = NOW$.

Due to our reactive model, objects can *only* be retrieved as a response to notifications (using the received object metadata), thus revealing a relation between the subscribe and retrieve operations. Received notifications must be acted upon, and may either be discarded, trigger an immediate retrieve operation, or be stored by the application and acted upon later.

4 THE MANY LEAVES OF THYME

The design of a time-aware reactive storage system for wireless edge networks presents a set of interesting challenges. For example, where to place data and how to find it? What are the proper trade-offs between communication and reliability? How and what data to disseminate? And overall, how to integrate the two interfaces—storage and P/S—without losing their principal characteristics, and making the resulting interface easy for developers to grasp and use? Thyme's design, presented next, considers these and other issues.

4.1 System Model

We consider a classical asynchronous model comprised of mobile devices (hereafter named nodes) with no mobility restrictions, other than those imposed by the venue they are in. Our algorithms do not assume any radio technology or routing infrastructure, being practical in several wireless networks. Nodes communicate by exchanging messages through a wireless medium (e.g., Bluetooth, Wi-Fi ad-hoc, Wi-Fi Aware), and should be able to establish communication with (all) their one-hop neighbors. We also consider the classical crash-stop failure model: nodes can fail by crashing but do not behave maliciously.

Data objects are considered immutable. Also, we do not consider security or access control concerns, thus only publicly shareable data is manipulated (e.g., as in social networks). Thyme notifies subscribers of relevant data as completely as possible, i.e., missing some notifications is permitted because applications are not expected to be mission-critical.

Each node has a globally unique identifier and can determine its geographical position, through GPS or other means [**?**]. Thus, nodes can be aware if they are moving or not. We also assume nodes' clocks to be synchronized (with a negligible skew). Both these assumptions are reasonable since we target mobile devices (e.g., smartphones) and nowadays even low-end devices come equipped with GPS and synchronize their clocks with the network providers, while other solutions allow device location even indoors [**?**].

4.2 Architecture

In Thyme, akin to (flat) peer-to-peer (P2P) systems, nodes are functionally symmetric and share the same responsibilities, i.e., there are no specialized components (like P2P super-peers or P/S brokers), and each node can be a publisher, a subscriber, or both.

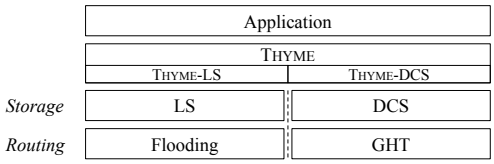


Figure 1: System overview.

Thyme's design comprises three main layers, depicted in Fig. 1. The bottom layer handles message routing. The middle layer is the storage substrate. The top layer is Thyme itself, providing its interface for applications.

As illustrated in Fig. 1, we propose two different approaches for the two bottom layers (routing and storage). Thyme-LS (§5) uses the nodes' local storage, and query flooding, thus data objects are stored locally by their owners, while subscriptions are fully replicated. Its routing layer provides flooding to the entire network (using UDP broadcast), and (multi-hop) unicast using a typical ad-hoc routing protocol (e.g., DSDV [**?**]).

In turn, Thyme-DCS (§6) follows a data-centric storage (DCS) approach [**?**], using a simple key-value substrate that we built over a cell-based geographic hash table (GHT) for wireless networks [**?**]. Physical space is divided into equally-sized square-shaped cells (see Fig. 2), and all physical nodes within a cell collaboratively act as a virtual node. Messages are addressed to geographic locations, thus routed to the cell that contains the message destination. Messages addressed to a cell are delivered to all physical nodes within the cell. The use of the GHT is two-fold: 1) cells are used to store all the system data; and 2) cells are exploited to match subscriptions and objects, i.e., cells act as virtual P/S brokers.

Wireless communication mediums are known to be subject to many forms of interference, hence messages may be lost and not reach their final destination. However, this layer does not provide any mechanisms to recover from lost messages, delegating this responsibility to the upper layers.

5 THYME-LS

Thyme-LS employs a lightweight unstructured approach. Insert and delete operations are entirely executed locally. Thus, objects are only stored by their owners. On the other hand, subscribe and unsubscribe operations are flooded and executed in every node, hence subscriptions are fully replicated.

Notifications may be triggered in two occasions: upon an insert operation, the node checks if that new object matches any of its locally stored subscriptions; and upon issuing a subscription (when flooding the respective message), each node that receives it checks if that new subscription matches any of its locally stored objects.

Retrieve operations request the desired objects directly from their owners, using the information in the notifications, and the multi-hop unicast provided by the routing layer.

Node mobility is handled transparently by the protocol used in the routing layer. Also, since objects are only stored locally by their owners, Thyme-LS does not guarantee objects' persistence once their owners fail or leave the system.

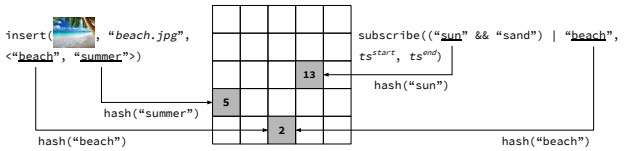


Figure 2: Insert and subscribe operations in Thyme-DCS. The tags' hashing determines the cells responsible for the metadata (cells 2 and 5) and the subscription (cells 2 and 13).

When joining the system, nodes broadcast a join request. To avoid a flooding of replies, only a few (randomly selected) neighbors respond back with their locally stored subscriptions. To avoid collisions, replies are delayed a (configurable) random amount of time. If no replies are received after a maximum number of retries, the joining node assumes it is alone, and starts operating normally.

6 THYME-DCS

By using geographical information, Thyme-DCS provides topology-awareness by design, and allows the inference of the location of relevant data to subscriptions, enabling access to such data using a location-aware strategy.

6.1 Inserting Data

This operation leverages on the cells conveyed by the underlying geographic hash table (GHT). Object data and metadata are managed differently. The later is indexed (and, thus, replicated) in all the cells resultant from hashing the object tags. The actual object data is replicated by all the nodes of the owner's cell (§6.2). This ensures only a small amount of data (i.e., the metadata) is sent through the network, whereas the bulk of the data is kept near its source.

Fig. 2 illustrates an insert operation. The cells resultant from hashing each tag are responsible for managing the object's metadata and checking if subscriptions match the inserted object. If a subscription has matching tags with an object, it will also have overlapping (responsible) cells, guaranteeing the matching and sending of notifications to the subscribers.

6.2 Replication

Since we target dynamic and volatile environments, in order to provide data availability and tolerance to churn, this approach employs two replication mechanisms.

Active replication takes advantage of the virtual nodes provided by the cell-based GHT. Upon an insertion, an object is disseminated inside the owner's cell. Onward, every node inside the cell should be able to reply to retrieve operations for that object. This guarantees that stored content will remain in the system even if their owners' leave. Note that object metadata is also (actively) replicated in the cells resultant from hashing the object's tags (§6.1).

In turn, *passive replication* leverages on the nodes that already retrieved an object to provide more replicas scattered in the network, increasing data availability, and offering a list of multiple locations from where it may be retrieved.

To enable both mechanisms, the system needs to keep track of the whereabouts of each object replica. This is done by listing an object's replica locations in its metadata, in what we call *replication lists* (a list of pairs $\langle id_{node}, cell_{node} \rangle$). These lists are bound to a maximum size, maintaining only the most recent entries. Also, the list only contains one entry for an object's active replica, representing all the nodes inside that cell. Since nodes can move, their location may change over time. After a node stabilizes in a (new) cell, it must update its location for the passive replicas of the objects it holds.

6.3 Deleting Data

In the delete operation, the object metadata indexed by the object tags is removed from the responsible cells. However, while active replicas are also explicitly removed, the same does not happen to passive ones.

6.4 Subscribing

Since the GHT used by Thyme-DCS only routes messages to geographical positions, there is the need to know where to send notifications, i.e., the node's address is not enough. Thus, subscriptions are extended with the location (i.e., cell address) of the subscriber node. This information needs to be updated every time the subscriber node moves to another cell.

Leveraging on the fact that every propositional logic formula has an equivalent in disjunctive normal form (DNF), we employ a divide and conquer strategy of breaking the disjunction into its individual conjunctive clauses, and evaluate each one separately. For a match to occur, it suffices that one evaluates to true. The use of DNF enables load balancing when matching objects against subscriptions, since the work can be split among different cells/nodes. For each conjunction, we randomly select as its key one of its *positive* literals. Hashing that literal determines the cell where to send that part of the query. That cell becomes a (virtual) broker for the subscription, and is responsible for checking if objects match the subscription, and notifying the subscribers. Fig. 2 depicts a subscription of a query with two conjunctions.

Regarding notifications, upon an insertion, cells indexing the metadata check if the new object matches any existing subscriptions; and upon a subscription, cells indexing it check if the locally stored metadata match that new subscription.

When a subscriber moves to a different cell (i.e., each time a node crosses the boundary of a cell), it must update its location for every active subscription it owns. During this situation, notifications sent to moving subscribers may never reach their destination. In such cases, the routing layer returns negative acknowledgments (NACKs) for messages addressed to *individual* nodes that could not be delivered (§6.6). NACKs are used to convey that a node is no longer in its supposed cell, which may be caused by movement or node failure. Node movement will be detected through the subscriber's location update. In such case, we can re-send the notifications that were not previously delivered. Otherwise, we can assume the node has failed and simply stop sending notifications.

When executing an unsubscribe operation, messages are sent to the cells determined by hashing each conjunction key.

6.5 Retrieving Data

From all the locations in the replication list (§6.2) received in the object metadata (with the notification), the requesting node chooses the geographically closest one to itself, and sends a retrieve request

for the desired object. If a negative reply is received, the requester proceeds and tries the next location in the list (until no more options are available, or a maximum of retries is reached). As a last attempt, the cell actively replicating the desired object will be used (if not already tried), because it offers higher chances of success.

The use of geographical routing makes it easier for nodes to make hints on which replicas are better (i.e., closer), using the geographic distance as metric. This approach reduces the distance data has to travel in the network, allowing for a location-aware strategy when retrieving objects.

6.6 Storage Substrate & Routing Layer

The major drawbacks of a routing protocol based on a distributed hash table (DHT) for wireless networks are the mismatch between the logical and physical topologies, and the high maintenance overheads [**?**]. Inspired from both wired [**?**] and wireless [**?**] settings, we adopt a cell-based GHT as our routing protocol. By using geographic information, there is no mismatch between the logical and physical topologies. Also, by leveraging on the control traffic of the geographic routing, the GHT does not add any other costs. Furthermore, the cell-based approach relaxes the requirements for location accuracy, and is robust to topology changes.

We implement a data-centric storage (DCS) substrate on top of this GHT, providing a simple key-value storage abstraction. To make this layer more suitable for the highly dynamic environments we target, we introduce several mechanisms and optimizations.

Our routing scheme is similar to the ones used in [**? ?**]. Routing is done at cell-level, using a variation of the greedy perimeter stateless routing (GPSR) protocol [**?**]. GPSR makes greedy decisions, forwarding messages to the next neighbor geographically closer to the message destination. When such is not possible, the algorithm resorts to forwarding messages around voids in the network. This layer provides a routing mechanism between cells, routing to an individual node (in a specific cell), and broadcast within a single cell. In our implementation, the one-hop broadcast is used as a neighbor discovery service—transmitting *periodic beacons* with the node's current cell—, and as the intra-cell communication primitive. Since broadcast is not acknowledged at MAC-level, this makes it a best effort communication primitive.

Regarding dynamic cell structure, we address empty cells forcing keys to take an entire loop around those cells [**? ?**], stopping in the cell closest to the supposed destination (which becomes a proxy of the destination cell). A cell becoming empty has to deliver all its keys to its proxy cell. In turn, a cell becoming populated receives its keys from its proxy cell, and also all the keys of the empty cells for which it now becomes a proxy.

We argue that moving nodes render routing information volatile, thus only stationary ones actively participate in message routing. Since our target scenarios have mild mobility patterns (i.e., nodes do not move constantly, and some might not even move during the entire event), only stationary nodes form the GHT. When a node starts to move and leaves its current cell, it stops participating in the routing protocol (i.e., it stops forwarding messages). It resumes the protocol when it detects itself as being stationary, by joining the local cell. While moving, nodes still process received periodic beacons, allowing them to keep communicating with the GHT.

Nodes are not individually addressable, but we support the sending of messages to a node in a *specific cell*. To allow the upper layers to react to a node failure or migration from one cell to another, the routing layer replies with a NACK to a *message source*, when a message addressed to an individual node could not be delivered.

For messages that are to be delivered to multiple destinations (e.g., notifications), we optimized our routing scheme by only propagating a single message to those destinations, in what we call *message destination aggregation*. This message is only duplicated when strictly required, which happens when the message's next hop for different destinations is not the same. This contributes to reduce energy consumption and the occupancy of the wireless medium.

When joining the system, a node waits a configurable amount of time. If, during that time, it receives a beacon sent by a neighbor in its own cell, the sender of that beacon is used as an entry point. A join request is then exchanged, and the joining node receives all the cell state. If a maximum number of retries is reached, the node assumes it is alone in the cell, and starts operating normally.

7 EVALUATION

Our evaluation seeks to answer the following questions: 1) which are the trade-offs provided by each approach of Thyme? 2) how does each approach handles with churn? and 3) how does each approach reacts to node mobility?

Each data point reports the average of five randomly generated network topologies, each independently run three times, making a total of 15 runs per data point. As baseline, we devise an approach based on external, centralized storage—Thyme-ES. Storage is external in the sense that it does not belong to the nodes forming the network, i.e., it belongs to a different (server) component. Objects and subscriptions are stored in external storage, and every operation is sent to that server to be executed (and replied back).

7.1 Implementation

In a previous workshop paper [**?**], we presented a proof-of-concept prototype of Thyme. We applied the Thyme-DCS approach to networks of Android devices, and used it to develop a photo sharing application. User can share photos, subscribe to tags of their interest, and subsequently be notified and obtain photos stored with such tags. The experimental results showed adequate response times for interactive usage, and low battery consumption. This prototype substantiates the feasibility of Thyme (in a small network of mobile devices), and helps corroborate the simulation results shown next.

To experiment with large scale scenarios, we resort to simulation and implement the Thyme approaches in the ns-3 network simulator [**?**]. We use ns-3.27 and nodes communicate through 802.11 Wi-Fi ad-hoc (using UDP). Both Thyme-ES and Thyme-LS use DSDV [**?**] as their routing protocol.

In Thyme-DCS, when a cell becomes empty/populated, a state transfer needs to happen between cells (§6.6). Currently, we do not implement such mechanism thus, in our experiments, cell structure is static, i.e., empty/populated cells will remain as such throughout the experiments. This poses some limitations regarding node mobility and churn in Thyme-DCS: nodes may move freely inside a cell, but may only leave a cell if it remains populated afterwards; and nodes may only migrate to previously populated cells.

To recover from lost messages, all approaches employ a retransmission mechanism. After a configurable amount of time without receiving the expected replies, the operation is retried. If a (configurable) maximum number of retries is reached, the operation fails with a timeout error code.

7.2 Simulator Setup and Methodology

Unless stated otherwise, all parameters were left with the simulator's default values. We used Wi-Fi 802.11g configured with a constant rate manager and a data rate of 6 Mbps.

We emulate an application similar to an online social network on top of Thyme (akin to Twitter). We generated trace files with the operations to be executed, using crawled tweets that were issued during the 2016 UEFA European Championship final. Tweets were used as data objects, where: the tweet id was used as the object identifier; the text was used as the object data; the timestamp was used as the object insertion time; and the hashtags were used as the object tags. The top-k most active users were chosen, and every other operation was generated from that, using exponential distributions configured with different λ values (i.e., rates). Subscriptions were generated taking into account the tags of the inserted objects, and the top 60% most popular tags were used for the subscriptions' queries (for simplicity, each subscription subscribed to one tag chosen at random). Subscriptions were generated in two forms: time independent ($ts^s = ts^e = \perp$); and in the future ($ts^s = NOW$ and $ts^e = \perp$). Time independent subscriptions where generated with a probability of 60%. During the first half of the game, subscriptions were generated with a rate of three per user per hour, and reduced to one for the remainder of the event. Delete and unsubscribe operations, which are expected to be rare, were generated with a rate of 0.5 and 0.2 per user per hour, respectively, during the second half of the game. We crawled a total of three hours, starting at 20:00 2016-07-10. To make the simulation execution more lively, we compressed the three hours into ten minutes of simulated time.

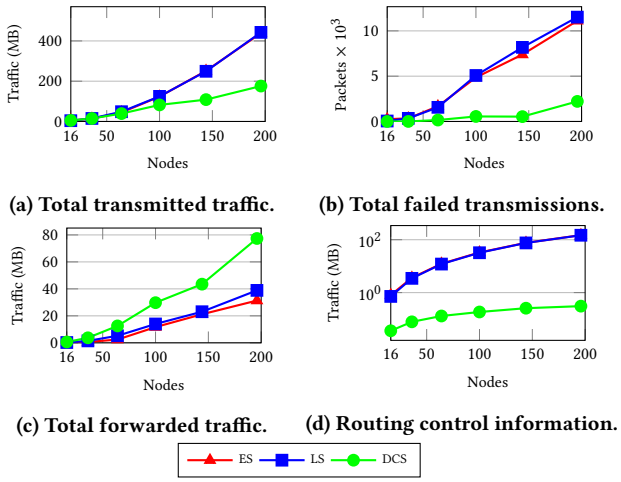
The simulation area has a rectangular shape. For Thyme-DCS, cell size is 40x40m, which entails a radio range of ±113m (roughly the range in our Wi-Fi setting), and we set an average density of 2 nodes per cell. All nodes were placed uniformly at random.

In our simulations, the application running on the nodes started only after 30s. Nodes joined randomly in the next 30s, and operations started being issued only after that. At the end of the simulation, nodes shutdown 60s after operations stopped being issued, for a total simulation time of 720s. All Thyme approaches executed the same traces and used the same methodology.

Since notifications are not user-triggered operations, we use *recall* (i.e., how many relevant items are selected) as a measure of success. However, we use the number of matching objects for a *perfect* execution of the trace, where operations *always* succeed and are executed *instantly*. Take into account that, for instance, if an insertion fails, all the subscriptions matching that object will not be matched against it, thus achieving 100% recall is practically impossible for our comparison baseline.

7.3 Static and Stable Nodes

In Fig. 3, we can observe the impact of each Thyme approach on the lower layers of the network stack. Figure 3a reports the total



(a) Total transmitted traffic. **(b) Total failed transmissions.**

(c) Total forwarded traffic. **(d) Routing control information.**

Figure 3: Lower layers metrics (static scenario).

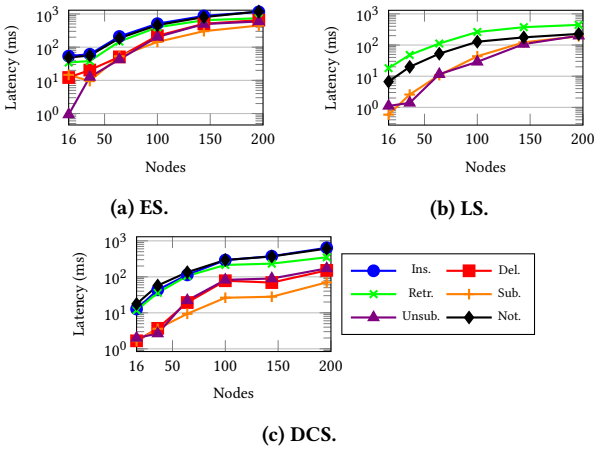
traffic transmitted by all nodes (at the physical layer—PHY), during the simulation. ES and LS overlap, and both exhibit a reasonable overhead. With 196 nodes, they report more than 2× the traffic of DCS. Looking at these values in an energy perspective, ES and LS will spend twice the energy to do roughly the same work as DCS.

Fig. 3b depicts values reported by the link layer, and it shows the total number of packets that exceeded the maximum retransmission attempts. The standard IEEE 802.11 Wi-Fi MAC layer implements CSMA/CA and a per hop retransmission mechanism. Thus, this figure depicts the interference observed in each approach. Both ES and LS require many more retransmissions than DCS to overcome the loss of messages that is inevitable in a wireless communication medium. This is due to the proactive nature of DSDV, frequently sending update messages, and thus causing more interference.

Next, Fig. 3c depicts the total traffic forwarded by every node in the system. In some sense, this value reports the amount of work nodes have to do on behalf of the system. In this case, DCS forwards more traffic because its messages are forwarded through longer routes than ES and LS (that use DSDV). This is even more exacerbated by the fact that some DCS messages may need to loop around voids in the network (§6.6).

Fig. 3d shows the total amount of control information the routing protocols transmit. While DSDV needs to exchange bulky routing tables to compute the shortest paths to every other node in the network, the geographic routing used by DCS routes messages using only local information (§6.6). With 196 nodes, a quarter of all the transmitted traffic of ES and LS was control traffic.

Regarding the operations success ratio, we verify that DCS is above 99%, except for notifications that fluctuate a little bit and have a success ratio as low as 95%. LS also reports high success ratio: (un)subscribe operations have above 99% success. Only retrieve operations and notifications have a very small reduction as the system grows, having 96% and 95% success, respectively, with 196 nodes. For ES, we see a slight decrease in the success ratio as the system grows, having as low as 78% success with 196 nodes. In every approach, notifications are a type of message that does not



(a) ES. (b) LS.

(c) DCS.

Figure 4: Operations Latency (static scenario).

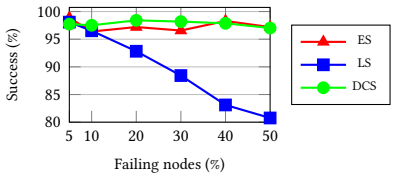
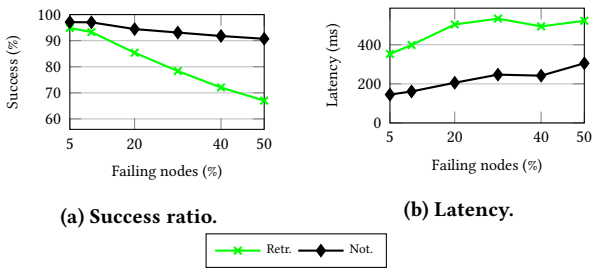


Figure 5: Notification success ratio (crashing, 100 nodes).



(a) Success ratio. (b) Latency.

Figure 6: Application metrics for LS (transient, 100 nodes).

employ an application-level retransmission mechanism, thus they are more susceptible to interferences.

Regarding operations latency (Fig. 4), we can see that for a small number of nodes all approaches behave similarly, with ES having slightly higher latencies. As the number of nodes increases, accompanied by increased interferences (Fig. 3b), we verify that latencies also increase. This is caused by the need for more retransmissions. However, notifications have lower latency in LS, because the geographic routing of DCS cannot compete with the shortest paths of DSDV. On the other hand, retrieve operations in DCS have a slightly lower latency, because DCS causes overall less interferences and it employs a location-aware strategy when retrieving data (§6.5). In ES, the decrease in success ratio is accompanied by an increase in operation latency. This comes from the fact that the majority of operation failures happen due to timeout. Overall, timeouts may indicate a congested network, where operations consistently have to be retried several times. Note that ns-3 metrics do not account for processing time (i.e., the time spent executing the protocols logic). If that was not the case, it would exacerbate ES latencies even more, since it has a central coordination point that with enough incoming requests becomes the system's principal bottleneck.

In summary, Fig. 3 shows that in DCS nodes transmit much less traffic that in both LS and ES (which are similar). This comes at the cost of latency, when compared to LS, as shown in Fig. 4. The centralized approach has the worst behavior as the size of the system increases, with a decreasing success rate and latency much higher that both DCS and LS.

7.4 Static but Failing Nodes

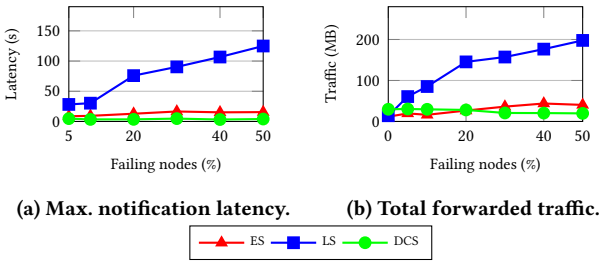
Regarding churn, i.e., the ingress and egress of nodes in the system, we experiment with two different scenarios. We show the impact of nodes leaving the system definitely, e.g., nodes crashing. Secondly, we show the impact of nodes with intermittent failures, thus entering and leaving the system multiple times throughout the simulation. These scenarios allow to evaluate aspects regarding data availability and persistence in the presence of failures.

7.4.1 Permanent Failures. In this scenario, nodes are either publishers or subscribers, and the former choose a random instant (between 200 and 300s of the simulation) to leave the system abruptly.

In LS, insertions are executed locally, thus not requiring communication. However, because only the object owner stores that data, if that node fails, all the data it stores will disappear. Fig. 5 shows exactly that. As more nodes fail, in LS, nodes with relevant data leave the system, thus the matching between subscriptions and objects is not detected. Since DCS employs replication (§6.2), even when object owners leave the system, matching still occurs. ES is not affected because all the system data is stored in external storage. As long as that server component does not fail, data will always be available.

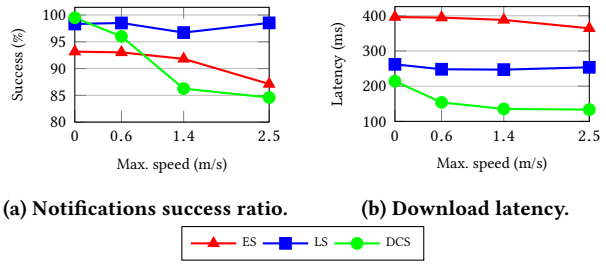
7.4.2 Transient Failures. In this scenario, randomly selected nodes alternate between the on and off states, during 120s and 60s respectively. Nodes have a 75% probability of changing to the opposite state, otherwise they stay in the same state for an equal period of time.

With nodes entering and leaving the system frequently, retrieve and notification operations are the ones that can be more affected, specially in the LS approach. Fig. 6 presents some application metrics of the LS approach for these operations. Since DCS employs replication mechanisms, it is little affected by the intermittent churn, with operation success ratio well above 90%, and latencies consistently between 150-300ms. Due to its central server component, ES is also little affected by the intermittent churn, with operation success ratio above 80%, and slightly higher latencies than DCS, between 400-600ms. LS, however, suffers from low success rate in the retrieve operation (Fig. 6a). Although notifications are detected, when a node tries to retrieve some object, as the amount of failing nodes increases, the probability of the data owner being off also increases. This is also accompanied by an increase in the latency of notifications (Fig. 6b). In LS, the matching between a node's stored objects and subscriptions that were issued when the node was off



(a) Max. notification latency. **(b) Total forwarded traffic.**

Figure 7: Transient scenario, 100 nodes.



(a) Notifications success ratio. **(b) Download latency.**

Figure 8: Mobile scenario, 100 nodes, pause 120 seconds.

have to wait for the node to switch state and join the system (§5). When joining the system, a node receives the subscriptions issued by all the other nodes previous to its entrance. Then, the joining node finds the new subscriptions it received and checks if it has matching objects. Fig. 7a corroborates this. The maximum latency for DCS and ES notifications stays stable as the amount of failing nodes increases. But, in LS, the maximum latency for a notification increases to values around 100s with 40% of failing nodes.

Fig. 7b shows a byproduct of the retrieve operation low success ratio. With no churn, DCS forwards more traffic because its insert and delete operations require communication. However, with this kind of intermittent churn, LS forwards much more traffic than DCS. This is due to the fact that retrieve operations are retried (and fail) several times. Also, this entering and leaving of nodes in the network, causes routing tables to become outdated, requiring changes to be made more frequently.

7.5 Mobile but Stable Nodes

We argue that the plain random waypoint mobility model does not mimic the movement pattern people have in the kind of events we target. For instance, in a music concert, people do not move constantly. To make it better resemble our target scenarios, every time a node is about to move, it tosses a coin do decide whether to move or not. If not, the node continues in a pause moment. In this scenario, only 60% of nodes are mobile, and have a moving probability of 80%.

Fig. 8a shows a small caveat of DCS: increasing node speed lowers the notifications success ratio. We reckon this happens because every node inside a cell is supposed to have the same state and work collaboratively as one. But, the intra-cell communication primitive is the unreliable one-hop broadcast. Thus, nodes inside a cell may not receive the same messages. Mobility may create even more entropy in the cell state.

Fig. 8b presents a byproduct of the location-aware retrieval strategy used by DCS. While, ES and LS are required to retrieve data from a specific place, DCS might have different replicas for retrieval at its disposal, and it can choose the one closer to the requester.

7.6 Discussion

Thyme-ES presents a baseline. It has an external, centralized component where all the system's data is stored. Being a centralized server, it presents itself as a bottleneck and a single point of failure. Thus, if it fails, the service will be totally unavailable. If that assumption is not an issue, then Thyme-ES can be an option. However, only for small scenarios, since the evaluation shows that, as the system grows, the server component rapidly becomes the main bottleneck (§7.3). In this approach, the centralized component resides close to the client nodes. If that was not the case, and it would reside in the cloud, we would see even higher latencies.

Due to its flooding approach, Thyme-LS causes far more interferences than Thyme-DCS. This is exacerbated the larger the network is (§7.3). Churn is also a concern for Thyme-LS, because insertions are only executed locally (§7.4). In summary, Thyme-LS can be suitable for smaller scenarios (i.e., with a small number of nodes) with no data availability requirements.

Thyme-DCS leverages geographic routing to employ replication, and location-aware data retrieval. However, one-hop broadcast is unreliable by nature, thus the assumption that every node inside a cell has the same state needs to be relaxed (§7.5). The loose coupling provided by the publish/subscribe (P/S), jointly with the employed replication mechanisms, showed to contribute to the resilience of Thyme-DCS to node failure and churn. Thus, Thyme-DCS is more suitable for larger scenarios with moderate mobility (and can cope with reasonable levels of churn).

In sum, these three approaches have very different characteristics. Which one is appropriate for a specific setting will depend on the conditions of the environment and the nature of the workload. Thus, we stress that Thyme-DCS is not always the approach of choice, but rather that under some conditions it is preferable. In fact, the perfect case is a system that embodies all of these approaches, and users can choose which to use according to the task at hand.

Overall, this reactive storage concept makes a *fundamental overhead shift*. Instead of requiring users to explicitly search for available data, it allows them to register queries (with well defined time scopes) and be notified as relevant data is stored in the system. As a consequence, the overhead of the stakeholders that actually benefit more from this approach—users requesting data—is *moved* to the stakeholders that do not benefit directly from it—users that have data and can provide it. However, we reckon that users usually do not mind sharing their resources just for a greater good (e.g., peer-to-peer (P2P) systems), or if they can also benefit from what the systems have to offer.

Other compelling reasons in favor of this concept are the *volunteer computing* [**?**] and the *crowdsourcing* [**?**] hypes. Volunteer computing uses computing resources (e.g., processing power, storage) donated by the general public to do scientific computing [**?**]. Crowdsourcing is a type of participative online activity in which an entity proposes to a group of individuals the voluntary undertaking of a task entailing mutual benefit [**?**]. This idea has been extensively

used as a cost-effective way of harnessing the collective power of multiple individuals.

With all these aspects in mind, it makes sense to crowdsource the computing and storage resources of a collection of nearby mobile devices to support a new generation of applications. Furthermore, people have shown to be receptive to the idea of harnessing the individual resources in order to make sense of the old motto "unity is strength".

8 CONCLUSION

In this paper, we present the concept of a time-aware reactive storage, that fuses the P/S paradigm with the storage substrate, providing persistent publications and allowing queries (i.e., subscriptions) within a specific time scope. The insert operation of the storage substrate is merged with the publish operation of the P/S system, enabling applications to be notified as relevant data is generated and stored. We also describe Thyme, a novel time-aware reactive data storage system for wireless edge networks. We detail two different approaches. Thyme-LS follows a lightweight unstructured approach using local storage and query flooding, while Thyme-DCS employs a DCS approach using a storage substrate built over a cell-based GHT for wireless networks.

Our evaluation shows that Thyme allows the reliable and efficient notification and retrieval of relevant data with low overhead and latency, even under node failures. However, each approach displays a different behavior and may be best suited for scenarios with certain characteristics (§7.6).

This work can be seen has a first step towards a data storage and dissemination system for a wide-area setting, like a campus or a music festival. In this scenario, data will still be stored in the devices, and communication will mostly be device-to-device to offload it from the network infrastructure.

As future work, we highlight the evaluation of our Android prototype [**?**] in realistic scenarios; the integration of our approach with opportunistic infrastructure support [**?**]; tackling the load imbalance issues resulting from large amounts of data referencing only a few very popular tags; and privacy and security concerns (mainly access control and trust).

ACKNOWLEDGMENTS

This work was partially supported by FCT-MCTES via project PTDC/CCI-COM/32166/2017 (DeDuCe), UID/CEC/04516/2019, and grant SFRH/BD/99486/2014; and by the European Union via project LightKone (grant agreement n°732505).