CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2016; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

# Total Order in Opportunistic Networks

Mihail COSTEA[1], Radu-Ioan CIOBANU[1], Radu-Corneliu MARIN[1], Ciprian DOBRE[1]*, Constandinos X. MAVROMOUSTAKIS[2], George MASTORAKIS[3], Fatos XHAFA[4]

[1]*Department of Computer Science, University Politehnica of Bucharest, Romania* [2]*Department of Computer Science, University of Nicosia, Cyprus* [3]*Department of Informatics Engineering, Technological Educational Institute of Crete, Greece* [4]*Department of Computer Science, Multimedia, and Telecommunication, Open University of Catalonia, Barcelona, Spain*

SUMMARY

Opportunistic network applications are usually assumed to work only with unordered immutable messages, like photos, videos or music files, while applications that depend on ordered or mutable messages, like chat or shared contents editing applications, are ignored. In this paper, we examine how total ordering can be achieved in an opportunistic network. By leveraging on existing dissemination and causal order algorithms, we propose a Commutative Replicated Data Type algorithm based on Logoot for achieving total order without using tombstones in opportunistic networks where message delivery is not guaranteed by the routing layer. Our algorithm is designed to use the nature of the opportunistic network to reduce the metadata size compared to the original Logoot, and even to achieve in some cases higher hit rates compared to the dissemination algorithms when no order is enforced. Finally, we present the results of the experiments for the new algorithm by using an opportunistic network emulator, mobility traces and Wikipedia pages. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: opportunistic networks; delay tolerant; consistency; total order

## 1. INTRODUCTION

Opportunistic networks (ONs) [28] offer a decentralized communication infrastructure in environments where, due to high mobility of participating nodes, a path cannot be apriori established between a source and a destination. In this case, nodes exchange data only when connections occur opportunistically (i.e., when nodes come in contact) over wireless protocols such as IEEE 802.11, Bluetooth, ZigBee, WiFi Direct, etc. They use what is today known as the store-carry-and-forward (SCF) paradigm [29]: a node can produce data, and transmit it to another node when an opportunity occurs. Each node then forwards the data to another one having possibly a higher chance of succesfully delivering the message[†] to its destination, and/or cache it for a limited time period.

Up to now, most work in ONs has been directed towards message forwarding or dissemination, where messages are usually considered immutable.

A system that supports messaging in ONs can be quite practical as a means to disseminate information quickly. However, for practical applications, dissemination should be accompanied with the means to manage *the meaning of the information*. For example, for a chat application, it makes

*Correspondence to: University Politehnica of Bucharest, Spl. Independentei 313, Romania, E-mail: ciprian.dobre@cs.pub.ro
[†]In opportunistic networks, data items are generally called messages.

sense for a user participating in a conversation to see his friend's answer *only after* he sees the actual question being posed. For a social application, a user expects to see a comment to a photo or status update his friend shares *only after* he sees the actual share. While these examples might look simple (they can be solved by simply numbering the order of events and re-constructing the correct order at the destination), they can be extended to messages showing collaborative operations such as editing or deletion. For example, imagine in the social application the user has an option to edit his own post. In this case, it becomes important whether one sees the comment to the original post, or to the edited one. In general, in collaborative applications, besides fast dissemination, *the order of actions* associated with various messages becomes important.

Only a few researchers have tackled the problem of mutable content that requires a given order [11, 32, 4], even though important user applications are based on it [26]. Examples of such applications, as shown above, include distributed discussion forums or social networks (i.e., ONs can solve problems related with censorship and would allow fast convergence of messages horizontally in co-located islands of people).

In distributed discussion forums, thus, users post news or questions in discussion threads, and others respond to them. While different discussion threads are independent from each other and do not require any order, messages in the same thread *should respect a total order*, such that every user sees the same (logic of) discussion. Similarly to discussion forums we have social networks, where users post something about themselves, or an interesting article, melody or citation they have found on the Internet or somewhere else, which also requires a total order for future responses. Other types of applications that require a total order are collaborative editing applications, like Google Docs or wiki pages. Every participant in a collaborative editor, or even a simple viewer of a wiki page, should have a consistent view of the contents. To refer to another example of why the order is important, consider a message being sent to two different users, and one user receives "I hate water", while the other one receives "I hate water less than I hate flying" (i.e., completely different meaning). This can happen if the text message is being collaborative and simultaneously edited by different users, and does not correctly propagate to other nodes.

The problem we pose in this article is as follows: *In an opportunistic network supporting message exchanges between sets of mobile devices, is it possible to come up with a solution as to enforce total order between messages received?* In total order, if multiple users are updating[‡] the same messages simultaneously, all users that receive them will consistently see similar resulted messages.

Total ordering is important for applications with mutable (i.e., that can change, even accidentally and with potential negative effect, as in the above example) contents, recently considered in ONs as well. Mobility support and the (potential) lack of centralized control (as opposed to traditional networks) make ONs especially attractive as an infrastructure for applications. In recent years, users have shifted from desktops to laptops and smartphones, and more recently to wearables such as smartwatches. Consequently, many traditional applications are already being migrated towards ONs - e.g., during the Hong Kong protests, FireChat[§] successfully replaced more traditional messaging apps such as Facebook Messenger or WhatsApp [13, 10]. Except for the ever-more-present access to mobile device, FireChat also solved the problem of circumventing government-enforced censorship.

Another reason for the ON adoption is sometimes the lack of a proper network/communication infrastructure. Emerging countries are not properly connected to the global network (this is also the case with disaster scenarios, when telecommunications infrastructure can be partially or totally destroyed). But, as smartphones have become affordable through programs such as Android One [16], ONs represent a viable solution for connecting together ad-hoc such devices - which leads to a request for applications similar to the ones mentioned above.

Finally, ONs can provide a better environment when it comes to privacy. ONs are able to work without connecting to the global network, providing an isolated environment. If participant nodes in

[‡]In contrast to causal order, that assumes users can produce new messages, in case of total order they can also update existing messages using insert or delete operations.
[§]We acknowledge FireChat [24] as one pioneer application in the domain, even though it does not use an ON infrastructure per se.



DOI: 10.1002/cpe

ONs are not allowed to read the contents of the messages they carry unless they are the destination, government agencies or hackers will not be able to easily eavesdrop, as messages are exchanged only between the nodes themselves, without the help of an infrastructure that can log messages whenever a third party wants [3].

In this article, we tackle the problem of ordering ONs messages so they follow total order before being delivered to the application layer. Our contributions consist of two parts: firstly, we propose a new Commutative Replicated Data Types / Conflict-Free Replicated Data Type (CRDT) [34] algorithm for achieving total order based on Logoot [39]. Using the high inter-contact times which are normal in ONs, our proposed algorithm reduces the metadata that must be carried with each message. Secondly, we run experiments to test the proposed algorithm by using the MobEmu [8] emulator, together with three mobility traces and Wikipedia pages to simulate insertion and removals.

The rest of this paper is organized as follows: Section 2 contains the related work, while in Section 3 we propose a causal order algorithm. Next, in Sections 4 and 5 we present the details about the total order algorithm we propose for ONs. In Section 6 we present the experimental results, and Section 7 contains conclusions.

## 2. RELATED WORK

With mobile devices becoming almost ubiquitous, today we witness the advent of several new types of mobile networks. Such networks are composed almost entirely of mobile devices, and differ considerably from the classic wired networks, both in terms of structure, but also with regard to the protocols and algorithms used for routing and data dissemination.

One type of such mobile networks that have been deeply researched in recent years is represented by opportunistic networks (ONs). However, up-to-now research in ONs focused mostly on improving delivery performance, through routing or dissemination algorithms for choosing next hop [9], adequate mobility models, or approaches to motivate nodes to participate [7]. We previously walk the same path, and proposed the SPRINT ONs routing protocol [6], or the ONSIDE algorithm [5] for dissemination. ONSIDE, for example, uses social information to disseminate data by using nodes that are not necessarily interested in a given message, but they have a high chance of encountering nodes that are.

When it comes to ensuring an order for message delivery, Operational Transformation (OT) [12] and Commutative Replicated Data Types (CRDT) [34] are well-known algorithms designed for causal order. They represent an interesting starting point for us as well, for several reasons. First, any algorithm that requires either a central node or synchronization between nodes cannot simply work in ONs. Apart from the CAP theorem [15] (i.e., consistency, availability and partition tolerance cannot be simultaneously provided in a distributed system), in ONs it is impossible to assign a node to assist with total ordering, because, at any moment, it can get out of direct contact with any other node or it might leave the network and never return. The use of a synchronization protocol between nodes, such as the three-phase commit protocol [36] or Paxos [20], is also extremely hard to implement in ONs, simply because in just networks it is possible that messages never reach their destination.

### 2.1. *Operational Transformation*

OT algorithms promise to achieve convergence, causality and intention preservation in distributed systems, without using locking and serialization [12]. They originate from research on shared editors, where users can edit the data at any time and in any part. The idea is to use operation transformation to achieve consistency, without waiting for messages from other nodes before applying user changes [12]. The earliest OT algorithm is dOPT [12], designed by Ellis et al., that supports transformations for insertion or removal of a single character. Operations are performed immediately at the originating site, offering good responsiveness. But there are some special cases where concurrency control for operation transformation fails, and two nodes can end up with different views of the shared contents. Algorithms like NICE [35] and SOCT3/4 [38] require a



DOI: 10.1002/cpe

notification server or sequencer to maintain a global total order of execution, making them difficult to implement in ONs.

Although OT algorithms seem promising, they seem to share a set of common problems that made them unsuitable for our problem. Many of them include transformations for and work with only one character (i.e., developing apps with one message per one character, over ONs, is not suitable). We would like to maximize the actual data transported by each node in regards to metadata, as nodes in ONs must carry a message for as long as possible to ensure that it reaches its destination(s), without wasting space that can be used for storing other messages. In addition, decentralized OT algorithms require a garbage collection mechanism to reduce the operation log size, which is improbable to obtain in ONs. Another problem is that most of these algorithms are complex and hard to extend to other applications besides share editors. We would like to use algorithms that can be used for more application types with minimal changes. Lastly, the complexity of the OT algorithms requires high computation for executing the operation transformations, while ONs nodes are usually mobile devices with limited resources in terms of processing power, memory and battery.

### 2.2. *Commutative Replicated Data Types*

Another solution we considered for total ordering in ONs consists in the use of CRDT based algorithms. A CRDT is a data structure designed to achieve strong eventual consistency (SEC) [34] in distributed systems. SEC is a subset of eventually-consistent systems where nodes that have received and delivered the same set of messages will immediately reach an equivalent state. As conflicts are not present in SEC, there is no need for a special process to handle them, and nodes end up with the same view of the contents, making them ideal for ONs. Among the first CRDT algorithms being proposed is WOOT [25] by Gérald Oster et al. WOOT relies on the property that in a document every character is located between another two characters. By knowing the neighbors, WOOT is able to perform character $insertion$ and $removal$. In order to ensure consistency for concurrent edits, a deterministic algorithm based on the same rules at all nodes is used to order them. But there is a problem related to this algorithm: it requires $tombstones$ to correctly ensure consistency. $Tombstones$ are the elements removed from the document, and, when they are deleted, they are only made invisible in the user view, without actually being released from memory. A garbage collection is required to remove tombstones, which is impractical in ONs as executing synchronization algorithms such as Paxos is improbable, just as mentioned in a previous paragraph.

An important CRDT algorithm is Logoot [39] by Stéphane Weiss et al.. In Logoot, each character gets assigned to a unique identifier. Identifiers are generated from a $densely\ ordered$ set, where for a given set $S$ and any two elements $x$ and $y$ in $S$, there is another element $z$ in $S$ such that $x < z < y$, and, by ensuring causal order, consistency can be maintained at all participant nodes. Logoot is able to create unique identifiers between any two characters by using a clock value that is incremented with every insert operation, and a list of pairs of integers made of a random value $x$ and a site ID $s$ such that the previous identifier $x$ value is smaller than the new $x$, which itself is smaller than the next identifier $x$ value. In case $x$ is equal for both previous and next identifiers, then the site IDs are used to order the characters. It is possible for the IDs to not respect the "$<$" relation, – in this case, a new pair of integers made of another random value $x$ and the site ID is appended to the pairs of the previous identifier in order to create a new identifier that can be positioned between its neighbors. Because of this, the Logoot identifiers can grow infinitely. If we add the fact that an identifier is required for every different character, we will end up with a high overhead associated with every message, which is undesirable in ONs (as mentioned previously). But an important fact related to Logoot is that it does **not** require tombstones. Characters can be removed at any time, and they do not need to be kept in memory in order to maintain consistency, eliminating the need for a garbage collection. The fact that a garbage collection is not necessary makes it an ideal candidate for ONs in case the previous limitation related to message overhead can be overcome.

As an alternative to Logoot, Mihai Letia et al. present the Treedoc algorithm [21], where the document identifiers are represented by a binary tree. The tree nodes contain the document characters, while the path from the root of the tree to the node with the actual character represents the character identifier. The left child of a node represents the previous character, while the right child



DOI: 10.1002/cpe

represents the next character. By using this approach, Treedoc obtains a more compact space for the characters identifiers. But some tombstones are required because a node cannot be removed from memory unless it does not have any children, though it is safe to remove nodes with no children. Due to tombstones and the fact that the tree can become unbalanced in some scenarios, a garbage collection mechanism is unfortunately required.

A really promising CRDT algorithm where multiple characters can be supported by a single identifier is LogootSplit [1] by Luc André et al. The algorithm is based on Logoot, eliminating the need for garbage collection. It supports sequence of characters using a single identifier by adding a new field to the Logoot identifier. Let us consider the same example presented by the authors in their article, where a new string "HEY" has been created. In Logoot, the three characters would have identifiers similar to: $\{< daa, H >, < dab, E >, < dac, Y >\}$ (the authors use a notation different from the original Logoot algorithm, but the $daa$ string can be considered the equivalent of the Logoot identifier, while "H" is the first character in "HEY"). In the new algorithm, the "HEY" string would have the identifier $\{< da[a.c], HEY >\}$. In case of a further append of the characters "WO", then these characters will have the identifier $\{< da[d.e], WO >\}$, and once its corresponding message is received by other nodes, the "WO" string is merged with "HEY", resulting in a single string with the unique ID $\{< da[a.e], HEYWO >\}$. Instead of having 5 different identifiers that would have a size of $3 * 5 = 15$ characters, the new string will have an identifier of only $4$ characters. Inserting new characters in a string with a new identifier is as simple as splitting the existing string and creating a new identifier using the same rule as the Logoot algorithm. Removing characters is done in the same manner as Logoot, though reusing some identifiers is not permitted.

### *2.3. Algorithms for DTNs and ONs*

In [27], the authors present how HTTP can be achieved over DTN. Although the work focus on end-to-end communication over a DTN, theoretically the approach could work for total order applications by piggybacking the contents of a message over HTTP in a DTN or ONs. However, the overhead will be too high for small messages. For example, if a user writes a few words every few minutes without having any contact with another node, then any total order algorithm we are aware that runs over reliable protocols will generate different messages for those contents as they are written at different times. The algorithm will merge those message in a single message as no contact occurred, reducing the metadata size that travels with every node. Also, as we target mobile devices such as smartphones that have limited memory, we prefer to avoid any unnecessary overhead, and utilizing protocols such as HTTP when they are not necessary would incur a space overhead that could be used for actual content. A similar situation is described in [4].

*All of the previously presented algorithms for total order were created for environments different from ONs*. As far as we know, we are pioneers in this directions in ONs. Most of the presented algorithms offer real-time group editing with multiple users in either a distributed systems or a peer-to-peer network (i.e., they assume either end-to-end reliable communication, nodes having enough resources, or mechanisms such as Paxos or garbage collection be already available). A result approaching tangentially our problem is presented in [18], by Teemu Kärkkäinen et al.. The authors present the construction of a shared editor over ONs, but instead of enforcing total order, they use revision control mechanism (merging), and adopt or discard whole versions of the documents. To make a better idea of the difference, the authors propose the equivalent of a distributed file system, while we target the construction of a distributed shared doc editor (among others). Although the idea might look promising, it poses a series of limitations which we consider to be too restrictive for ONs. One of their assumptions is that every node in the ON carries a copy of the document contents in order to be able to modify it. This ensures that, at any contact between two nodes, changes can be propagated from a node to another and the contents of the document can be updated in order to display a consistent view to the user. The problem is that not every node in an ON might be interested in the contents of that document (and yes, there might be multiple documents simultaneous being carried). Requiring nodes that have no interest in the document to hold a copy of it represents an overhead that cannot be ignored in ONs with limited resources, including memory. Nodes should carry only the contents they are interested in and a limited amount of messages that



DOI: 10.1002/cpe

are not intended for them, and which are routed using a dissemination algorithm as in the first part of this section. These messages not intended for the user can be removed at any time in case there is no more free memory, being replaced with new data or new messages, which leads to scenarios where the removed messages might never reach their destination. Another problem is that some sort of merging is needed in the case of merge conflicts, leading to cases where useful data might be discarded or user intervention is required in order to avoid the data loss. Both of these scenarios are undesirable.

### *2.4. Summary*

Except for [18] that uses revision control mechanism for ensuring a total order in an ON, we are unaware of any attempts to ensure total order in ONs or DTNs by using the high inter-contact times present in these networks. Our proposed algorithm takes advantage of these high inter-contact times in order to reduce metadata size, by merging messages during contacts that can occur after a few minutes or hours or even days, and it is able to achieve total order even if messages are lost or duplicated, by relying on clocks and document view synchronization at direct contact between nodes. Also, the routing layer of the ON is unaware of our order algorithm and can work independently for applications that do not require any order.

The algorithm is based on Opportunistic Causal Barriers (presented next), and Logoot, and part of it is similar to LogootSplit in the regard that we also support a single identifier for a sequence of characters, but our algorithm is able to use the space identifier better than LogootSplit and reuse any deleted identifier. The algorithm is explained in section 4.

## 3. OPPORTUNISTIC CAUSAL BARRIERS - AN ALGORITHM FOR CAUSAL ORDER

In this section we first propose *Opportunistic Causal Barriers*, an algorithm designed to achieving causal order in opportunistic networks. In our system, causality is defined based on the *happened before* relation presented by Lamport [19]. There are two types of events: $generate(m)$, which is the moment a node generates a new message, and $download(m)$, which is the moment a node downloads a message from another node that holds that message. The causality relation "$\rightarrow$" is defined as:

*Definition 1*
For any two events $e_1$ and $e_2$ events, $e_1 \rightarrow e_2$ if:

1. $e_1$ and $e_2$ are two events on the same node, then $e_1$ occurs before $e_2$ or
2. $e_1$ is the generation of a message by node $n_i$ and $e_2$ is the download of that message at any other node $n_j$ or
3. there is an event $e_x$ such that $e_1 \rightarrow e_x$ and $e_x \rightarrow e_2$

In order for *causal ordering* of message delivery to be respected, for any two messages $m_1$ and $m_2$ such that $m_1 \rightarrow m_2$, then $m_2$ is delivered to the application only after $m_1$ has been delivered. Otherwise, if $m_2$ is downloaded before $m_1$, it must be delayed until $m_1$ is downloaded.

Instead of using the $send(m)$ and $receive(m)$ concepts, we use $generate(m)$ and $download(m)$. In an ON, the concepts of sending or receiving a message do not hold. A message is not sent or received, but downloaded by other nodes if they consider it relevant either for them or for another node with which an opportunistic contact might occur. As there are no send or receive events, we define causality in terms of the generation and downloading moments.

The algorithm we propose reduces the message overhead and ensures a deadlock free scenario in case nodes come in direct contact with any other node that is logically part of the same application, a.k.a. *thread* from now on (e.g. for a chat application, they are part of the same thread discussion) and does not discard prior content. We call this feature *direct download of missing messages*. In order to explain how message merging and direct download of missing messages work, we present our proposed ON system architecture in Figure 1.



DOI: 10.1002/cpe

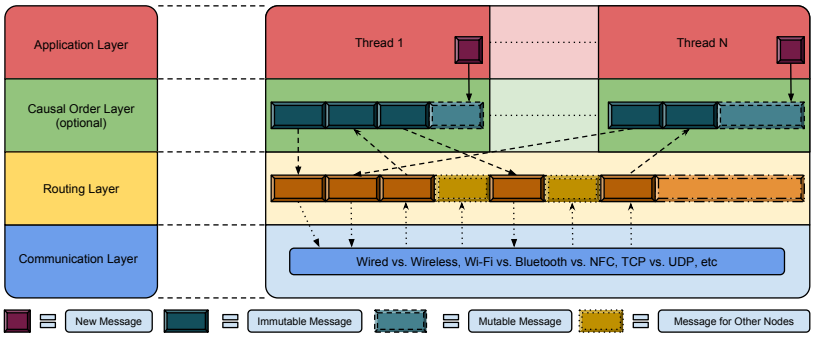



Figure 1. Opportunistic Networks System Architecture

At the lowest level, we have the *Communication Layer* for actually transferring bits of data from one node to another one when they are in a direct contact (i.e. they can reach each other using methods present in traditional networks). At the next level, we have the *Routing Layer*. The routing layer contains messages generated locally and messages generated at other nodes intended to him or for participant nodes in future opportunistic contacts. The order of messages at this layer is given by the routing algorithm, while the memory is split in two: one for messages generated by or intended to the local node, and one for messages meant for other users, which we call *data memory*. Afterwards, we have the *Causal Order Layer*, which is optional, depending if the application requires causality or not. This layer contains the own generated messages and messages from other nodes which have as destination the local node, while their order respects causality. The messages memory is **shared** with both the routing and the application layers, just that we store extra information for keeping track of causality. At the highest level, we have the *Application Layer*. Each application can be composed of multiple threads, and in every thread we ensure causal order between messages. The application layer memory is shared with the causal order layer, and only messages that respect causality are shown to the user, while downloaded message that do not respect it are hidden until obtaining the predecessors.

### *3.1. Messages Merging*

In order to reduce the metadata size, *Opportunistic Causal Barriers* proposes merging messages generated at the same node. Merged messages have a single $CB$ vector instead of one $CB$ per every message. Merging messages uses the concept of *mutable message*:

*Definition 2*
A **mutable message** represents the last part of the causal layer memory, where the application thread can only **append** newly generated messages. A mutable message can become **immutable** only when an opportunistic contact between two nodes occurs, and when one of the next rules applies:

1. the content of the mutable message must be downloaded, an operation that leads to the split of the mutable message in two parts to accommodate the network capacity, with the first part becoming **immutable** and being downloaded by the other node, while the last part remains mutable, and future appends are merged with it, **or**
2. the local node downloads and delivers a message from the other node that is meant for the application thread which contains the mutable message; the mutable message is split in multiple immutable parts with fixed or variable sizes that permits other nodes from future contacts to efficiently download any part

An immutable message cannot be converted back into a mutable message. The scope of the (im)mutable messages is to reduce the metadata of the application contents that travels with each message, and to preserve causality. Causality is preserved due to the two rules for making a message immutable. In the first case, any appended messages depend causally only on previously generated



DOI: 10.1002/cpe

messages at the local site, so it is fine to merge them with these messages in order to use a single $CB$. The second mutable part after the split will update its $CB$ in order to depend causally on the first immutable part. In the second case, when we download a message on which future generated messages must depend on, we split the existing mutable message in smaller immutable parts and add dependencies between them as in the original Causal Barriers algorithm. Any new message generated locally will depend on the last immutable part of the split message and on any downloaded message that has just been delivered.

Unsuccessful attempts of another node to download the contents of a message (e.g. the contact duration is too short to download the message, or the routing algorithm decides not to download it), or downloads of messages from other nodes that do not result in a delivery due to missing causal predecessors, will not change the state of the mutable message. Only successful downloads and deliveries can make a message immutable.

Another important aspect to mention is that the length of the mutable message is variable. That is, with every append, the mutable message grows, being limited only by the physical storage memory. A mutable message can have a length of zero in case the application thread has just been started, or in case the last part after the split operation is empty.

### *3.2. Direct Download of Missing Messages*

In contrast to traditional networks where lost messages are sent again, in an ON we can end up with messages that never reach their destinations, even though two nodes that run the same application might have a direct contact in the future. This scenario is possible because the data memory for the routing layer is limited. Once the data memory is full, and the routing algorithm decides to download a new message, it will first remove a message from the data memory to make space for it. Due to this reason, some messages might end up being lost forever. As this scenario is a normality in ONs, we have to design algorithms that can cope with messages lost forever.

Because messages might be lost forever, we can end up with the next deadlock scenario: we have three nodes $n_i$, $n_j$ and $n_k$, all participating in the same application thread. $n_i$ generates a message $m_1$, which after a short period is downloaded by $n_j$. Then $n_j$ generates a new message $m_2$, which now depends causally on message $m_1$. After a period of time and a multitude of contacts between $n_j$ and other nodes beside $n_k$, $n_j$ has to remove $m_1$ from its data memory in order to make room for other messages downloaded by the routing layer. After a short period, $n_j$ comes in contact with $n_k$, which downloads message $m_2$. But message $m_1$ is not present anymore in the data memory, though it is present in the application memory (or the causal order memory in our case), a fact not taken into consideration by routing algorithms. If $n_k$ never has a contact with $n_i$, and all other nodes have removed $m_1$ from their memory, then we have arrived at a **deadlock**, as any messages causally dependent on $m_2$, and $m_2$ itself, will never be delivered at $n_k$.

In order to cope with the previous scenario, we introduce the concept of *direct download of missing messages* at the causal layer, by bypassing the routing layer. Not only does direct downloading of missing messages avoid deadlocks for the above scenario, but it also improves the hit rate and delivery latency compared to routing algorithms that do not impose any order. We make the assumption that any participant node in an application thread will **eventually** get in direct contact with at least one node from the same application thread. This assumptions holds for the scenarios where total order in opportunistic networks is required, as proved by Samir Okasha in [23].

Downloading of missing messages is done only after the routing layer has downloaded its messages from the data memory, and in the limit of remaining resources (i.e. contact duration or battery). The routing layer has priority because it downloads messages for any application, including previous messages intended to us that have not been removed from its data memory. Plus, we aim to *eventually* get out of the deadlock scenario. The download algorithm works as follows:

1. Firstly, we download all the missing messages already delivered to the application thread based on the causality order (i.e. messages that depend causally on previous messages are download after these previous messages).



DOI: 10.1002/cpe

2. Secondly, we download all the missing messages in the delayed queue based on the same causality order.
3. Lastly, we insert any downloaded message in the data memory, so nodes not interested in our application thread can can carry these messages in their data memory to nodes that might be interested in them.

The first two steps can be halted at any moment, in case the network capacity limit has been reached. For the last part, we insert the messages in the data memory based on the causality order: firstly predecessors are inserted in the data memory, eliminating any previous messages present in the memory, and when we reach the full capacity of the memory such that only the messages that were just downloaded are present in it, *Opportunistic Causal Barriers* stops (we do not want to remove our own inserted messages from the data memory).

## 4. A TOTAL ORDER ALGORITHM FOR OPPORTUNISTIC NETWORKS

In the following Sections we propose the *Opportunistic Logoot* algorithm for achieving total order in ONs. The algorithm extends *Opportunistic Causal Barriers* with two new features: mutable messages and direct download of missing messages (features needed in order to increase the metadata reduction and ensure correctness in case of lost messages). More exactly, while a message is mutable, besides trying to use a single causal barrier for different messages lightredgenerated at the application layer, in the new algorithm we try to also merge the identifiers used for ensuring total order, while the download algorithm has been modified in order to automatically identify removed contents (i.e. deletion of text in a share editor), without keeping any tombstones.

We first define the *application content model* used by Opportunistic Logoot, the identifiers used for total ordering, and the operations supported by the content model. In Section 5 we present how Opportunistic Logoot merges messages in order to reduce metadata size, and how it handles lost messages.

### 4.1. Opportunistic Logoot Operations

Our algorithm supports two operations for modifying the contents of an application:

1. *insert(pos, base_element)*, which inserts a *base element* at position *pos*
2. *remove(pos)*, which removes a *base element* stored at position *pos*

A *base element* represents the smallest possible element in the contents of an application. For example, in a text editor, the base element is a character, while in a chat it is the whole message until the user presses *enter*. Applications will work only with this API. It is not up to the application layer to handle merges between multiple base elements, with this task being ensured by the order algorithm (i.e. our proposed algorithm). The reason for this approach is to provide a simple API which could be used for more types of applications than only text editors. Applications just have to provide base elements or split the contents in base elements (e.g. for a text editor, the string “abc” would be split into “a”, “b” and “c”).

### 4.2. Opportunistic Logoot Identifiers

In order to support a total order between all base elements, we use identifiers similar to the ones presented in Logoot. The original Logoot algorithm uses unique identifiers generated in a *densely ordered* set in order to ensure global order between all participant nodes (see [39]). At any moment, it is possible to create a new identifier between any two existing identifiers, and deleted elements are removed immediately from memory, without being transformed into tombstones.

Just as Logoot, we require only causality. Starting from existing causal order algorithms [33, 2, 31], we proposed the Opportunistic Causal Barriers algorithm in Section 3. For the identifier notation, we use the same convention as in the original Logoot proposal.



DOI: 10.1002/cpe

Our content model is identical to the Logoot document model, just that, instead of *lines*, we use a sequence of base elements, which we shall simply call *sequence*. The construction of sequences will be explained in a further paragraph. A sequence is defined by a pair <*id, sequence_content*>, where *"id"* represents the unique identifier and *"sequence_content"* is a concatenation of base elements representing the sequence itself. There are two default sequences: <*id_left_limit, null_sequence*> and <*id_right_limit, null_sequence*>. These sequences cannot be removed and they are used to make sure that we always create identifiers only between them.

The *id* field is represented by a list of pairs $< x_i, s_i, l_i, r_i >$ and a logical clock $clk_s$, where $x_i$ is the node ID (an integer), $s_i$ is the unique site ID, $l_i$ is the ID of the first base element in the sequence (or the range left limit), and $r_i$ is the ID of the last base element in the sequence (or the range right limit). A site represents any device that can exchange messages in the ON. Both $l_i$ and $r_i$ are short integers, $r_i >= l_i$ and a sequence has $r_i - l_i + 1$ base elements. The $clk_s$ is the logical clock of the site $s$ which created the ID and it is used to make sure that identifiers are unique in case of reusing the same list of pairs after removing an element, a behavior similar to the original Logoot. The difference from the original Logoot is that we added the fields $l_i$ and $r_i$ to support sequences of base elements, which is an addition similar to LogootSplit. In summary, an *id* is defined as:

*Definition 3*
$id =$ <$x_0, s_0, l_0, r_0$>, <$x_1, s_1, l_1, r_1$>, .... <$x_{n-1}, s_{n-1}, l_{n-1}, r_{n-1}$>, <$clk_s$>, where $n$ is the number of pairs <$x_i, s_i, l_i, r_i$>

For example, the sequence "This is an example." created at node $4$ can have the identifier $< 4, 3, 1, 19 >$, where $x_i$ (i.e the node ID) is $4$, $3$ is the site identifier, $1$ is the range left limit and $19$ is the range right limit. Every character can be uniquely identified using the numbers between left and range limits, i.e. $1$ for "T", $2$ for "h", and so on.

In order to obtain a total order between identifiers, we define the comparison relation between them. First we define the relation $<_{pair}$ that compares two pairs, and then we define the relation $<_{id}$ that compares two identifiers.

*Definition 4*
Let $pair_1 =$ <$x_1, s_1, l_1, r_1$> and $pair_2 =$ <$x_2, s_2, l_2, r_2$>.
$pair_1 <_{pair} pair_2 \iff x_1 < x_2 \vee (x_1 = x_2 \wedge s_1 < s_2) \vee (x_1 = x_2 \wedge s_1 = s_2 \wedge l_1 < l_2)$

*Definition 5*
Let $id_1 = p_0, p_1, ...p_{n-1}, clk_1$ and $id_2 = q_0, q_1, ...q_{m-1}, clk_2$ be two identifiers, where $p_i$ or $q_i =$ <$x_i, s_i, l_i, r_i$>.
$id_1 <_{id} id_2 \iff \exists k \leq m \wedge k <= n, (\forall l < k, p_l = q_l) \wedge (k = n \vee p_k <_{pair} q_k)$

For example $< 4, 3, 4, 5 >$ is less than $< 4, 6, 2, 4 >$ due to the site IDs, while $< 4, 3, 4, 5 >$ is greater than $< 4, 3, 1, 2 >$ due to the range left limit.

It is important to mention that, for any two consecutive identifiers $id_1$ and $id_2$, if all pairs until the last pair are equal (i.e. same site identifier generation), then $r_1 < l_2$, where $r_1$ is the range right limit for $id_1$ and $l_2$ is the range left limit for $id_2$ for the last pairs. It is not possible for the same site to have superposition for two different sequences as this would mean that different base elements are on the same positions. More exactly, it is impossible to have $< 4, 3, 2, 6 >$ and $< 4, 3, 4, 8 >$ as identifiers for different sequences as base elements between $4 - 6$ would superposition.

Another important observation is that our algorithm can reuse identifiers deleted after a remove operation, just as the original Logoot. The reused identifier is slightly different from the old identifier, as it has a higher clock value $clk$ compared to the initial identifier.

### *4.3. Insertion*

In order to execute an insertion *insert(pos, base element)*, we need to find either the sequence where to insert the new base element, or the previous and next sequences in case base element must be inserted between two sequences or merged to one of them, search that depends on the value of $pos$. In order to find the correct place for base element, we first explain how we calculate the number of base elements in the application contents.



DOI: 10.1002/cpe

*Definition 6*

$$num\_of\_base\_elements = \sum_{i=0}^{n-1} last\_pair\_range_{id_i}$$

where $n$ is the total number of sequences,
$last\_pair\_range_{id_i} = r_{m-1} - l_{m-1} + 1$,
$m$ is the total number of pairs for $id_i$

For example, the total number of elements for $< 2, 3, 4, 8 >$, $< 4, 4, 11, 16 >$, and $< 5, 3, 4, 4 >$ is 12.

Finding the correct sequence(s) where to insert the new base element is done in a linear time in terms of the number of sequences, i.e. $\mathcal{O}(n)$, which is different from the number of base elements. Depending on how long each sequence is and the value $pos$, we can end up with two scenarios:

1) insertion between two sequences
2) insertion in the middle of a sequence

*1) Insertion between two sequences* – Let us assume we have two sequences $seq_1$ and $seq_2$ between which we have to insert a new base element $elem$ at a site with the ID $s$. When inserting a new element $elem$ between two sequences $seq_1$ and $seq_2$, we can distinguish four cases:

a. both $seq_1$ and $seq_2$ identifiers where generated at sites with IDs different from $s$
b. $seq_1$ identifier was generated at the current site with ID $s$ and $seq_2$ identifier was generated at a different site
c. $seq_1$ identifier was generated at a different site, while $seq_1$ identifier was generated at the current site with ID $s$
d. both $seq_1$ and $seq_2$ identifiers where generated at the current site with ID $s$

In the first case, we will create a new sequence $new\_seq$ with a single base element $elem$ that will have an identifier between $seq_1$ and $seq_2$ identifiers. The rule to generate this identifier is similar to Logoot:

*Definition 7*
A new identifier $new\_id$ inserted between $id_1 = p_0, p_1, ...p_{n-1}, clk_1$ and $id_2 = q_0, q_1, ...q_{m-1}, clk_2$, with $p_i$ or $q_i =$ <$x_i, s_i, l_i, r_i$>, must have a shortest number of pairs $p_0, p_1, ...p_i, < x, s, l, r >, i \leq n$ such that $id_1 <_{id} new\_id <_{id} id_2$

The $x$ value can be generated using multiple rules, but we have used only three. The first rule is to generate it in increments. That is, $x = px_i + 1$, where $px_i$ is the first integer of the previous sequence $id_1$ as mentioned above and $i < n$, or $x = 1$ if $i = n$ (i.e. new pair). This rule is useful for applications that usually append elements at the content end, just as chat applications. The second rule is useful for share text editors, and it creates a random $x$ between $[px_i; qx_i]$ if all pairs until $i$ are either equal or there is no space between $px_{i-1}$ and $qx_{i-1}$. $px_i$ is the $x$ value of a pair in $id_1$, while $qx_i$ is the $x$ value of a pair in $id_2$. If $i$ has passed over $n$, we use the default left limit 1 instead of $px_i$, while if $i$ has passed over $m$, we use the default right limit $max\_value$ instead of $qx_i$. The third rule is also useful for share text editors, but it produces deterministic results beneficial for the experiments section, and it chooses $x = (px_i + qx_i)/2$ in case all pairs until $i$ are either equal or there is no space for between $px_{i-1}$ and $qx_{i-1}$. If $i$ passed over $n$ or $m$, we use the same rules as for the random method. We call this method the *half method*.

In the second case, we try to append $elem$ to $seq_1$ and increment the last pair range right limit $r$ of $seq_1$ identifier. This is possible only if $r < max\_value$. Otherwise, instead of merging, we will create a new sequence with only $elem$ inside, just as in the first case.

The third case is similar to the second case, just that instead of appending we will try to place $elem$ at the beginning of $seq_2$ and decrement the last pair range left limit $l$, as long as $l > 1$.



DOI: 10.1002/cpe

In the last case we will try to either place $elem$ at the end of $seq_1$ or at the beginning of $seq_2$. It is important to notice that, in case all pairs of $seq_1$ and $seq_2$ are equal, with the except of last pair that is different only in terms of range left and right limits, then we are allowed to merge the new element with $seq_1$ or $seq_2$ only if $r_1 + 1 < l_2$, where $r_1$ is the range right limit of $seq_1$ last pair and $l_2$ is the left right limit of $seq_2$. We are not allowed to have base elements that might end on the same position.

*2) Insertion in the middle of a sequence* – Let us assume we have a sequence $seq$ and an element $elem$ which we try to insert in the middle of $seq$ (i.e. between the $l$ and $r$ value of the last pair of the $seq$ identifier). We will first try to directly insert $elem$ at its right position inside $seq$ without splitting it and by decrementing $l$ or incrementing $r$ to take in consideration $elem$, but only if we do not end up with a superposition with the previous or next sequences. Otherwise, we split the sequence in two, and, where $elem$ must be inserted, we create a new sequence with a single element $elem$ using the same mechanism as above.

It is important to mention that merging is allowed only for **mutable messages**, otherwise we will treat insertion as in the first case presented in Section 4.3. The concept of mutable messages will be explained in a further paragraph. Also, with every insertion that does not end in a merging operation, we will increment the $clk_s$ value of the site $s$.

### *4.4. Removal*

Removal is simpler. In case a sequence has only one element, we just delete it. In case the sequence has more than one element, we have two cases:

a. removal at the beginning or the end of the sequence, which means that we just accordingly update the range left or right limit of the last pair
b. removal in the middle of the sequence, which means that we split the sequence in two, create two identifiers that are similar to the first one, and update the range right limit of the left sequence and the range left limit of the right sequence to not include the removed element

In case the site ID $s$ is different from the site ID that generated the sequence, or the message is immutable, then we will create a new mutable removal message that will be placed on the routing layer to be disseminated to the other nodes in the ONs. Consecutive removals are merged together until the message becomes immutable. In case the message is mutable, we just remove the element from the corresponding sequence, without generating a new removal message. This is one of the **key aspects** of our algorithm, as base elements that are just inserted, and after a few moments they are removed (i.e. grammatical corrections), will never generate messages that waste resources in the ON.

The complexity to execute the remove operation is identical to the insert operation: $\mathcal{O}(n)$ in terms of the number of sequences.

## 5. OPPORTUNISTIC LOGOOT FEATURES FOR OPPORTUNISTIC NETWORKS

Identifiers for a sequence of characters, together with *mutable messages*, lead to a lower metadata size compared to the original Logoot that uses one identifier per every character. Mutable messages not only reduce metadata size, but also increase hit rates by generating fewer messages at the routing layer. The hit rates are also increased by the *direct download of missing messages*, though it is mainly used for ensuring correctness of our algorithm in case of lost messages.

It is important to mention that the routing layer of the ON is unaware of our algorithm and any application that uses it. Our algorithm is allowed to modify a message contents and the data structures it controls until a message is downloaded for the first time by another node, and afterwards that message cannot be modified anymore. Once a message is downloaded for the first time, it becomes immutable and the routing layer handles it without any knowledge of the message contents. Applications are unware of how the other layer work. They will only work with the API mentioned at 4.1, and it is up to the total order and causal order layers to combine messages and identifiers



DOI: 10.1002/cpe

until a message is downloaded. Once the message is downloaded, the contents become immutable and the routing layer takes over.

As shown in Figure 1, *mutable messages* represent the last part of the memory per application thread situated at the causal order layer. Any new message is appended at the end of the corresponding mutable message, having the same Causal Barrier vector ($CB$) as all previous messages merged together inside the mutable message (for more details about $CB$ see [31]). If we do not merge messages, then we would have to use one $CB$ per every new message, resulting in an overall higher metadata size. A mutable message becomes *immutable* either when it is downloaded by another node, or when downloading messages from other nodes, as any future message to generate will depend causally on these downloaded messages. When making a message immutable, if required, we split the original mutable message in smaller parts in order to accommodate the network capacity, with every part having its own $CB$ and depending causally on the precedent part which was generated earlier, as proved by the append operation. This step is required as a mutable message can have a significant size in case a lot of messages have been generated from the last interaction with another node, which represents the only moment when a mutable message is made immutable. But even in this case the remaining number of merged messages per every part is still high enough to produce significant savings, as shown in section 6. After that, once a message becomes immutable, we are not allowed to modify it anymore because messages generated afterwards will depend on its $CB$, $CB$ that should not be changed as it ensures causality.

### *5.1. Mutable Messages*

In order to maximize the metadata size reduction, we have extended the *mutable messages* feature to Opportunistic Logoot. The new system architecture is presented in figure 2, and it contains a new layer called *Total Order Layer* for applications that require total order. Just as the causal order layer, the total order layer is optional and can be ignored by application that do not require it.

A difference from the causal order layer consists in the fact that a mutable message at the total order layer can be situated at any part in the content model, and not only at the end. The causal order layer supports only appends to correctly ensure causality, but Opportunistic Logoot needs to support insertions and removals at any position in the document. To cope with this requirement and increase the metadata savings, we have slightly relax causality. More exactly, we ensure causality only between immutable messages, and base elements present in mutable messages can be arranged in any way before becoming part of a mutable message. Of course, this causal relaxation might be too much for some application types, but without it we would have an overall higher metadata size and in some cases even result in the requirement of tombstones as we would have to keep the messages for the removed elements, something we strongly tried to avoid. For a shared editor, we think that user intention is still preserved, as we shall see in the next paragraph.

Though we have relax causality, we still need to discuss how base elements will be ordered once a message becomes immutable. The conditions to insert or remove elements from a mutable message are the same as the conditions to merge or remove a base element from an existing sequence, as explained in the previous paragraphs. Let us consider the following example where the contents of the application present at user $u$ are made only of one immutable message: "*This is a fact.*". Every base element is represented by a character, and what value the identifier of the immutable message has is not important for this example. Before coming in contact with another node, user $u$ makes the next transformations:

1. This *interstng awesome fact* is a ~~fact~~*real ebent*.
2. This inter*est*i~~b~~*v*ent.
3. This interesting ~~awesome~~ fact is a real event.

For this example, we will have three mutable messages at the total order layer, each with a single Opportunistic Logoot identifier. In the first step, the first mutable message contains the text "*interstng awesome fact* " with a newly generated identifier, the second mutable message contains the corresponding identifier for "*fact*", but no text as this is a remove operation, and the third mutable



DOI: 10.1002/cpe

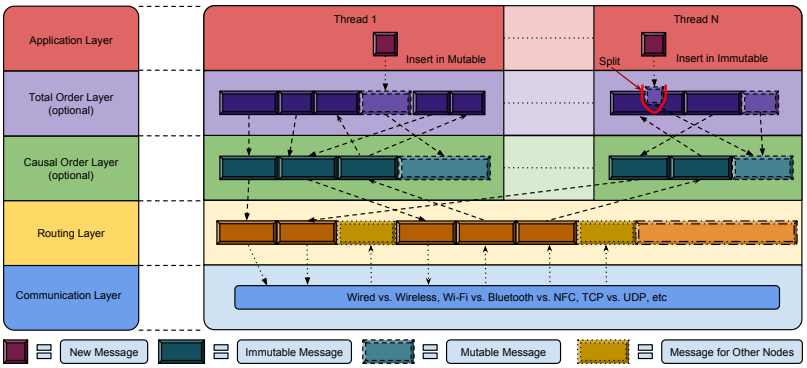



Figure 2. New Opportunistic Networks System Architecture

message contains "*real ebent*" and its new identifier. In the second step, the user corrects the text in the first mutable message to "*interesting awesome fact* " and its identifier is updated to reflect the change, and then he corrects the third mutable message contents to "*real event*" without modifying the identifier as the number of elements is the same. In the last step he removes "*awesome*", but the first message is mutable, so there is no need to generate a remove message, but only to update the corresponding identifier. When a contact occurs with another node that is interested in the above contents, these three mutable messages are merged and ordered consecutively in a message at the causal order layer, using a single $CB$ vector for all of them. Of course, we still have three Opportunistic Logoot identifiers, but there is no need to use three $CB$ vectors. This merge of total order messages at the causal order layer can also be seen in Figure 2, where multiple total order messages are part of the same causal order message (for Thread 1, messages 2 and 5 from the total order layer are part of message 2 at the causal order layer, and messages 3 and 6 are part of message 3, while for Thread N, the message generated in the middle of an immutable message that leads to a split, together with the last message, is part of the mutable message at the causal order layer).

In case we had strict causality for all messages as they are generated at the application layer, then, instead of three mutable messages with only three identifiers, we would have had seven messages requiring seven identifiers. The reasons are that we could support only append, and any remove or insert at a later time would have to use a new identifier as it has been generated later at the application layer.

As a final observation regarding mutable messages, the total order layer messages can be merged together only with messages of the same operation type (i.e. we can merge only consecutive removals from the same identifier or consecutive inserts). The reason is that insertion generates a new identifier, while removal contains the identifier of the removed element. At the causal order layer the situation is different and we can merge messages with different types, as the causal order algorithm has no knowledge of operation types.

### *5.2. Downloading Missing Messages*

In contrast to traditional networks where lost messages are sent again, in an ON we can end up with messages that are never received, even though two nodes that run the same application might have a direct contact in the future. This scenario is possible because the data memory for the routing layer is limited, so old messages are sometimes removed to make room for new messages even though these old messages have not yet been received by all nodes that are interested in them. As this scenario is a normality in ONs, we have to design algorithms that can cope with messages lost forever. It is important to note that even though messages are lost forever, the contents are not, as they are still present at the nodes interested in them (i.e. for a shared document that is all of its characters, from where we can generate new messages if another node requires them). In order to cope with this scenario, we introduce the concept of *download of missing messages on direct contact*.



DOI: 10.1002/cpe

Because messages might be lost forever, we can end up with different views for different users, even if the total order is still respected. Unfortunately, this means we are in an inconsistent state. The condition under which our algorithm, with the downloading missing messages feature, ensures consistency, becomes: *In an opportunistic network, consistency will happen after nodes stay in direct contact for a sufficient time for them to synchronize their contents views.* In other words, this is a case of eventual consistency. In order to ensure consistency, instead of only downloading messages at the routing layer, we shall go up to the application layer and download the actual missing parts. This is possible only on direct contact between two nodes participating in the same application, because, even if the routing layer has removed the message from its data memory, the application will keep the contents as long as the user wants to have them.

Downloading the missing parts of the contents is done after the routing layer has downloaded its messages and in the limit of remaining resources (i.e. contact duration or battery). So it is possible for the download process to be interrupted at any time, a case that is handled differently from the case where all missing parts are downloaded.

For now, let us assume that we have enough resources/memory to download all missing messages, and then we shall extend our algorithm to the case of limited resources. Before presenting the download algorithm, we have to mention that every node $n_i$ stores a map $\{n_j, last\_insert\}$ containing the Opportunistic Logoot identifier for the last insert operation received from $n_j$. The space complexity is $\mathcal{O}(n)$ in terms of the number of nodes, though for some nodes there will be no entry until we find out about them. This situation with nodes that we do not know about has shortly been presented in the *Oppostunistic Causal Order* algorithm, and it does not affect the download operation.

The algorithm works as follows:

1. we make all messages in the corresponding application thread immutable, as downloading them will affect causality for future new messages
2. we download all identifiers from the other node
3. we compare the identifiers one by one to see which base elements are missing **or** which base elements have been removed; figuring if a base element is missing or has been removed is done by using the map $\{n_j, last\_insert\}$, and it will be presented shortly by using an example
4. we generate new messages for the missing base elements and we download them, but **without** yet delivering them to the causal and total order layer; in case the two nodes finish their direct contact due to any reason before downloading the missing parts, then we keep the contents of the application just as before the direct contact
5. after downloading all the missing base elements and by knowing which base elements were removed, we download the internal data structures regarding the causal order algorithm from the other node and use it to advance the local causal order data structures; for example, for CBCAST [2] we download the vector clock of the other node and select the highest value per every node from the downloaded vector clock and the local vector clock, while for Causal Barriers [31] and Opportunistic Causal Barriers, we download the delivered vector (see [31]), and select the highest value per every node from the downloaded delivered vector and the local delivered vector
6. we deliver the downloaded base elements in ascending order of the Opportunistic Logoot identifier **clock** value and insert them at the correct position, while at the same time removing the base elements as figured at step 2

The algorithm has the complexity $\Theta((N + M) * K)$, where $N$ is the number of identifiers present at the first node, $M$ is the number of identifiers present at the second node, and $K$ is the average number of pairs per identifier. The identifiers are compared and advanced one by one in ascending order, avoiding useless comparisons, so this is why we have considered an average value for $K$. Though, in our experiments, $K$ was usually between 1 and 2, but for really large documents with a lot of insertions in the middle of existing identifiers, this might change. As a performance improvement, in case of termination of direct contact in the middle of step 4, we can keep the



DOI: 10.1002/cpe

already downloaded messages in a cache, saving resources in case of a future contact, but this does not affect correctness. This is how we have extended the direct download to the case with limited resources.

As we have seen, we can execute steps 5 and 6 only if all messages have been downloaded from the other node. We do this in order to correctly handle causality for future messages received from other nodes. The reason is that a user can insert base elements in the middle of the application contents, and if the messages of the base elements at the contents end are immutable, then we should deliver the new generated elements only after delivering the elements at the contents end. If we would not download all messages, then step 5 will be executed incorrectly, as the $delivered$ vector (or vector clock) only knows how many messages were delivered from each node, but not which were the actual delivered messages.

It is important to mention that causality is relaxed during the download of missing messages. In case of messages for insert operations generated after messages for remove operations that were lost forever, we will process the insertion on direct contact without knowing if the remove was generated before or after the insertion. The motive we do not know this information is that we do not keep tombstones. After completing the download algorithm, the rules for causality return to normal.

We shall now explain with an example how we identify removed elements when executing step 3 of the download algorithm. For the sake of simplicity, let us consider that we are working with the original Logoot identifiers and no merged messages. A Logoot identifier is similar to our algorithm, just that it is only made of $<< x_i, s_i >>$ without the left and right range limits. For an explanation of $x_i$ and $s_i$ see 4.2. Extending the process to the Opportunistic Logoot identifiers is a trivial matter. Also, in order to simplify our explanation even more, let us consider three nodes $n_1$, $n_2$ and $n_3$ with only $n_3$ generating insertions, while all of the nodes can generate removals. The general case where any node can generate insertions is handled anyway separately, based on the clock value of each identifier and the corresponding value to the identifier site generator in the $\{n_j, last\_insert\}$ map we have mentioned in step 2. Let us consider the example shown in Figure 3, containing only the Logoot identifiers (the text is not important), where $n_3$ has generated 10 messages, but not all of them have been received by $n_2$. For further clarity, instead of writing $< 1, 3 >, 1$ for the first message at site 3 with timestamp 1, we will simply write $m_1@t_1$ (since the site is the same in this example). Thus, we have the initial scenario shown in Figure 3a.

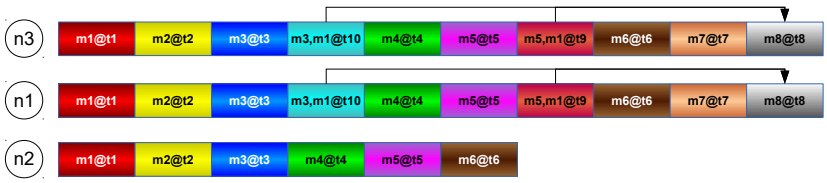
We can observe that $m_3, m_1@t_{10}$ and $m_5, m_1@t_9$ are **not** shown at $n_2$ even if messages at later positions are shown. The reason is that $m_3, m_1@t_{10}$ and $m_5, m_1@t_9$ are generated after $m_8@t_8$, as the clock value suggests. As they are generated after $m_8@t_8$, then they depend causally on it, so they cannot be delivered to the application thread until $m_8@t_8$ is received. The map $\{n_j, last\_insert\}$ has the entry $\{n_3, m_3, m_1@t_{10}\}$ at node $n_1$, while at node $n_2$ it has the entry $\{n_3, m_6@t_6\}$.

Afterwards, $n_1$ deletes the elements with $m_2@t_2$, $m_3, m_1@t_{10}$ and $m_4@t_4$ as identifiers, while $n_2$ deletes $m_3@t_3 3$ and $m_4@t_4$. The remaining messages for each node are shown in Figure 3b.

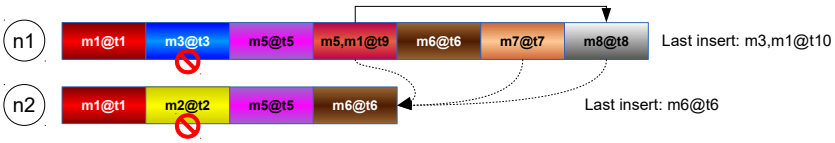
Let us assume that $n_1$ and $n_2$, due to interactions with other nodes, eliminate the corresponding "remove" messages from the data memory, and those messages never arrive at the other node. Afterwards, they have a direct contact. As shown in Figure 3b, node $n_1$ downloads all identifiers from node $n_2$. Node $n_1$ sees that only $m_1@t_1$, $m_2@t_2$, $m_5@t_5$ and $m_6@t_6$ are present at $n_2$, but from those, $m_2@t_2$ is not present at node $n_1$, while $m_3@t_3$ and all identifiers after $m_6@t_6$ are present. So node $n_1$ looks at the map $\{n_j, last\_insert\}$, and sees that, for the identifiers generated from $n_3$, the entry is $m_3, m_1@t_{10}$ for node $n_1$'s map, while for $n_2$ it is $m_6@t_6$. As the clock value is 10, it means that at some point all messages until $m_3, m_1@t_{10}$ were present locally. So any identifier not present at node $n_1$, but present at node $n_2$, which has the clock value $\leq 10$, has been removed locally. This is a given fact because causality does not allow us to deliver messages with a higher clock before delivering messages with a smaller clock if they were generated at the same site (messages generated at different sites have independent identifier clock values and they are processed separately). So node $n_1$ knows that it does not need to download $m_2@t_2$ as it was removed locally, but it has to remove $m_3@t_3$, because node $n_2$ no longer has it, and the message has a lower clock than 10, which means it was removed remotely. $m_4@t_4$ was removed at both nodes, something neither of them can know (as they do not keep tombstones), and they do not even have to know to display the same text correctly



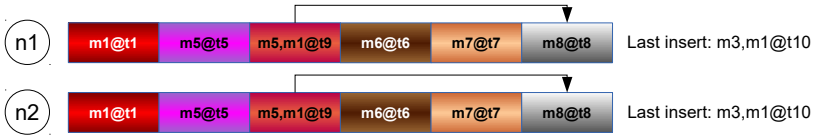
DOI: 10.1002/cpe




(a) Step 1 (nodes $n_1$ and $n_2$ performed removals)




(b) Step 2 (nodes $n_1$ and $n_2$ meet)




(c) Step 3 (after the data exchanged between nodes $n_1$ and $n_2$)

Figure 3. Identifying removed elements (dotted arrows represent data transfers, while normal arrows represent temporal message dependencies).

(if they come in contact with a node that has $m_4@t_4$ present locally, then they would known they have removed it). But $n_1$ still sees that it has identifiers present locally but not present at $n_2$, and those are the identifiers with values greater than $m_6@t_6$. But $m_6@t_6$ is the last insert at $n_2$, so $n_1$ knows that $n_2$ could not have removed them as they were never delivered at $n_2$, which means it has to keep them.

Now let us consider $n_2$. The download process works exactly as for $n_1$, just that it will do an extra step. All of the identifiers less than or equal to the last insert $m_6@t_6$ are processed as above (i.e. $m_6@t_6$ is removed), while elements with greater identifiers will be downloaded. But the last insert $m_3, m_1@t_{10}$ at $n_1$ has been removed, which means it will not be downloaded. At a later time it is possible for the message with $m_3, m_1@t_{10}$ to be downloaded from another node $n_k$ that did not remove it from its data memory, but $n_2$'s newest last insert would be $m_5, m_1@t_9$, resulting in an invalid operation. In order to cope with this problem we simply update $n_2$'s last insert to the last insert of $n_1$, as $m_3, m_1@t_{10}$ is still present in the map $\{n_j, last\_insert\}$, even if it has been removed from the application thread. This is not a tombstone, as once a newer element from $n_3$ is received, we discard the previous last insert and just memorize the new insert in its place. The download algorithm will still work correctly due to causality, as the node figures out it has removed the discarded last insert if it is present at the other node with which it came in contact. Of course, if the other node does not have the discarded element present, the local node will not know it discarded the previous last insert, but it does not have to know, as the contents are still displayed correctly. The messages that each node ends up with are shown in Figure 3c.

As for extending the above algorithm to Opportunistic Logoot identifiers, the process is really simple. Let us have the next identifier generated at node $n_1$: $< 1, 1, 0, 9 >$,0. This identifier is split into 10 parts: $< 1, 1, 0, 0 >$,0 | $< 1, 1, 1, 1 >$,0 | $< 1, 1, 2, 2 >$,0 | ... | $< 1, 1, 9, 9 >$,0, and afterwards node $n_1$ executes the download algorithm just as explained above for the original Logoot identifiers. After the download algorithm ends, node $n_1$ merges back all consecutive identifiers that were generated at the same site and have the same clock value and range values of the last pair consecutive (i.e. $< 1, 1, 0, 0 >$,0 and $< 1, 1, 1, 1 >$,0 can be merged into $< 1, 1, 0, 1 >$,0, but $< 1, 1, 3, 3 >$,0 and $< 1, 1, 5, 5 >$,0 cannot be merged if $< 1, 1, 4, 4 >$,0 was removed, as the range values are not consecutive).



DOI: 10.1002/cpe

As a final observation, our algorithm is able to handle duplicate messages by using the clock value. In case of receiving a new messages from a node, we check if the clock value of at the causal order is greater or not from the clock value of the last delivered message from that node. As delivered messages are shown at the application layer, then the last delivered message must have the highest clock value as imposed by the logic of the causal order algorithm. In case we receive a message with a smaller or equal clock value, then we drop it as it is a duplicate. This approach applies only at a destination node. For messages at intermediate nodes that only forward them towards destinations, it is up to the routing layer algorithm to take care of duplicates, and not our algorithm. Also, it is impossible to have duplicate mutable messages as a mutable message is present only at its creator until it is downloaded, when it becomes immutable and starts to be available in the network.

### *5.3. A Case Study*

In this subsection, we present two scenarios where we believe opportunistic networks can be useful, and where total order is a requirement. Firstly, let us imagine a scenario where a stadium is full with supporters attending a match. It is well-known that, in such situations, download and upload speeds are very small, because tens of thousands of people access the infrastructure (i.e., 3G, 4G, WiFi) simultaneously. Therefore, opportunistic networking is extremely helpful (and suitable), since the network being formed is very dense, so messages can spread quickly. Thus, only a small number of nodes would connect to the wireless infrastructure, and act as relays for the other nodes. Let us now assume that a team of reporters from the television network that broadcasts the match is spread throughout the stadium, gathering information from various sources (e.g., from the players and coaching staff on the benches, from the supporters, from the commentary cabin, etc.). In order to offer a more immersive experience for the fans at home, they can aggregate all this information in a single live commentary feed, that would then be used by the TV commentators or published as a live analysis of the match on a website. This can be done using all the devices inside the stadium as ON nodes, and transmitting data opportunistically. However, since events on the pitch are causally related (for example, an injury observed by the sideline reporter can be the cause of a substitution), the live commentary feed would require the events to be in the correct order. Furthermore, an ad-hoc forum can be created for the fans in the stadium, where they can share their opinions and discuss the match live, without the need to connect to the wireless infrastructure. This too requires a total order of messages, as specified in Section 1.

Another scenario that would benefit from total order in opportunistic networks is a distributed crowd-sourced review application. Users would install an app on their mobile devices which, after visiting a tourist attraction or having dinner at a restaurant, would require them to perform a review. Users would also attach pictures or any kind of media to the review, which would be specific to the time that they visited the place. Decentralizing such an applications through ONs means cutting the costs of keeping all the data on servers, and would permit the information in the reviews to age. The application would allow reviews to be amended, which would require chronological sorting of the messages in order for other users to see the most recent version of the review. Furthermore, other users would have the possibility of adding additional information to a review, or discussing their opinions, which would also require the content to be displayed in the correct order.

These are only two scenarios which show that total order in opportunistic networks is something that can bring benefits in real-life situations. In the future, we wish to expand on these use cases and highlight even more advantages that ONs can bring in terms of costs and effectiveness, when compared to infrastructure-based networks.

## 6. EXPERIMENTS

In this section we present our experimental results. We prove that our algorithm obtains good results compared to the hit rate and delivery latency of bare routing / dissemination algorithms, while in some cases even surpasses them. Also, due to merging messages, our algorithm is able to reduce the metadata size by a significant magnitude when compared to the original Logoot.



DOI: 10.1002/cpe

Table I. Information about the Mobility Traces

| Trace | Nodes | Duration | Type | Topics |
|---|---|---|---|---|
| Sigcomm | 76 | 4 days | Conference | 720 |
| UPB | 24 | 64 days | Academic | 5 |
| Infocom | 99 | 4 days | Conference | 35 |

### *6.1. Experimental Setup*

The experiments were run using MobEmu [8], an opportunistic network emulator that uses real mobility traces to simulate the behavior of different ONs routing algorithms. The mobility traces used in the tests are Infocom [17], Sigcomm [30], and UPB [22], and they contain both the node contacts and information about their interests. Details can be found in table I. Sigcomm and UPB also contain information about social connections between nodes, while for Infocom, the social aspect needed by some algorithms had to be disabled.

We have two models for generating messages: one based on a series of Wikipedia pages, and one similar to the experimental setup presented at [5]. The data is modeled as messages that are generated through channels to which nodes subscribe to. Each channel is represented by a topic of interest. When a node subscribes to a channel, it is interested in any message that is generated on that channel. Nodes can generate data only in the channels they are subscribed to, and data intended for a channel cannot be downloaded on another channel. Every node that has at least one interest generates messages either from the Wikipedia page, or by using a predefined rule if the channel does not have a page assigned to it. For UPB and Infocom, some participant nodes did not have any interests (Sigcomm has interests for all participants). Those nodes do not generate any messages, but they can carry messages for other nodes.

We have used four Wikipedia pages [42, 44, 41, 43], each with 18, 14, 14 and 7 contributors at the moment we have accessed the pages to obtain their corresponding XML (see bibliography for access times). All of the pages are short in comparison with most Wikipedia pages, but unfortunately we where restricted by the mobility traces to search for pages with a few number of contributors (i.e. UPB has 18 nodes for the topic with the most number of users interested in it). Pages that were longer, and which we tried to consider as alternatives, had more contributors than what the traces supported. This is a limitation, but unfortunately we could not go around it.

To replay the Wikipedia pages history we used the corresponding XML file containing the page revisions. The XML can be obtained using [40]. Then we used a diff library [14] to compute modifications performed between each two revisions at the level of characters. In order to have concurrent edits, we have considered that each line in the Wikipedia page is independent from any other line, but, to also have causality, modifications to the same line depend on previous modifications to that line. That is, for insertions, we can generate the first element of a line without waiting for any other elements from other contributors, but starting with the second element of a line and so on, we have to wait for the previous element to be delivered to its application thread. As for removals, we issue them only after the elements to remove have arrived at the contributors that want to remove them. Of course, this means that we will not arrive at an identical page as the last revision in the XML file, but as we do not have any traces for ONs, we had to create a parallel model even if Wikipedia pages were obtained in a client-server architecture. But as long as we can correctly obtain the same page for all contributors if we force them in direct contact, then we think this trade-off is not that costly.

To make sure that all of the messages generated from the Wikipedia revisions are evenly spread, they are generated at fixed times: a whole revision is processed at the time

$$\frac{i * (traceEndTime - traceStartTime)}{maxNumOfRevisionsPerContributor}$$

, where $i$ is the index for a revision. In case some elements of a revision are not processed due to missing dependencies (i.e. for insertions starting with the second element in a line or for removals



DOI: 10.1002/cpe

for which we have not received the element they are dependent on), we try to process them again at the next $i$.

Not all messages are generated using the above model, as not all channels have been assigned to Wikipedia pages. For our second model, represented by the remaining channels and that simulates extra network traffic, we have used a similar generation method as in the experimental setup from [5]. Each node generates 80 messages per day, with the exception of the last day, in the two-hour interval when most of the contacts happen (around midday in all three traces), by randomly selecting one of its interests as a channel for a message.

The actual number of messages that can be stored at a node for routing or dissemination is finite, in order to simulate limited resources in ONs. In case the data memory is full, on downloading a new message that is or not intended for that node, but that might be required by another node in the future, the first downloaded message in the data memory is removed to make space. As data memory values, we have used 50, 250, 1000, 3000 and 10000 messages / node memory.

### *6.2. Results*

We have run our experiments using ONSIDE [5] and Epidemic [37], first without ensuring any order, and then with the original Logoot and our proposed algorithm Opportunistic Logoot on top of it, though any other routing algorithm should work. As an observation, the original Logoot is not able to cope with lost messages as causality will stop it from delivering new messages to the application layer, but it is still important for a comparison when it comes to metadata size. ONSIDE is a ON routing algorithm that uses a node's online social connections, its interests and the history of contacts to decrease congestion and required bandwidth, without affecting the overall network's hit rate and delivery latency. The Epidemic algorithm floods the network by downloading every available message from other nodes.

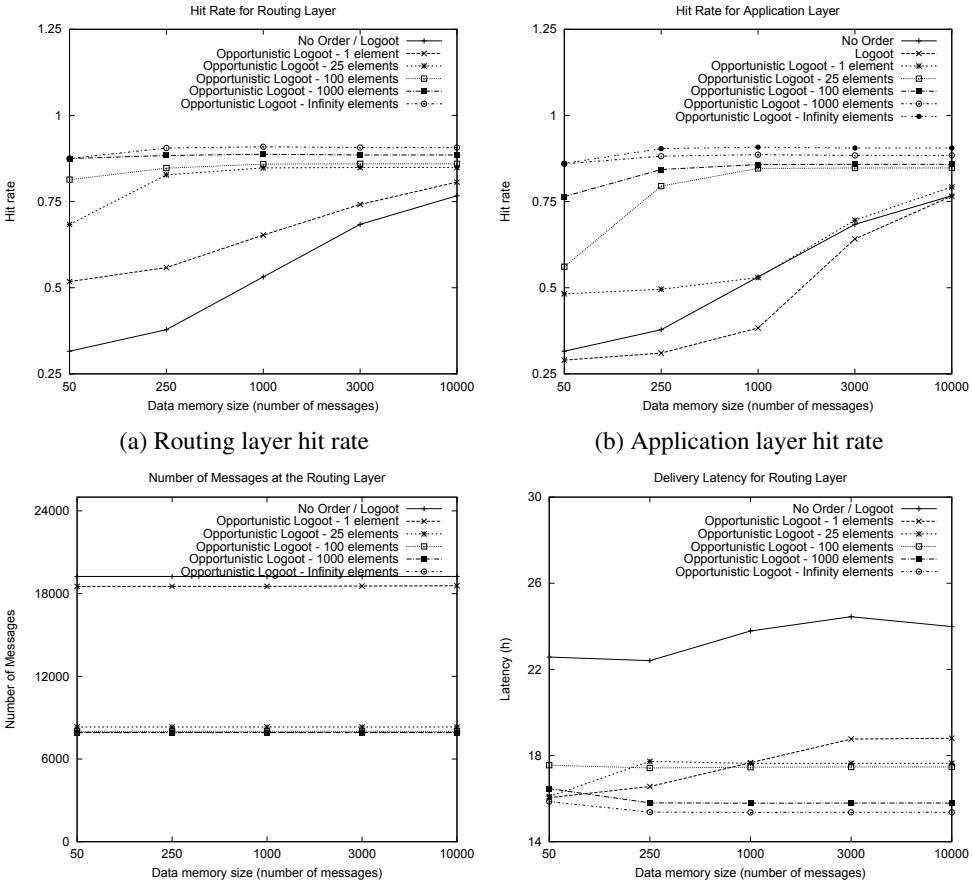
As metrics, we have considered the metadata size, hit rate and delivery latency. Hit rate and delivery latency are defined similarly to [5], with hit rate representing the ratio between the number of messages successfully delivered at interested nodes, just that separately for both routing and application layers, while delivery latency is the average amount of time passed between message generation and delivery to the corresponding layer. We have also used hop count, which is the average number of nodes that carried a message until the destination, and delivery cost, which is the ratio between the total number of exchanged messages over the total number of generated messages, in order to present how the routing algorithm behaves, even if they are mostly related to the routing algorithm and not to a total order algorithm.

Opposed to Logoot, Opportunistic Logoot has been run for every experiment five times, with 1, 25, 100, 1000 and $\infty$ as the maximum number of Wikipedia elements allowed to exist in an immutable message at the causal order layer. A Wikipedia element is either an insert or a removal operation of a single character. MobEmu considers a message as the basic unit exchanged between nodes, not the size of the message, so, in our model, immutable messages are the basic unit for messages exchanged between nodes. In order to simulate the network as a limited resource, we have chosen 1, 25, 100 and 1000 as limits because the payload of a Wikipedia element is the character itself for insert operations, or no payload at all for removal operations. The position where the insertion or the removal is performed is not sent with the message, as the Logoot identifiers are used to correctly insert or remove elements, and Logoot identifiers are part of the metadata. Also, the type of the operation is part of the metadata and not part of the payload, because the Opportunistic Logoot identifiers require a single entry for the type of its whole identifier (merging base elements is done based on the operation type), while original Logoot needs a type entry for every identifier. Having 1, 25, 100 and 1000 as limits is realistic, because, if we consider every character of the insert operation as having two bytes (Unicode characters), then, for 1000 as limit, we would have in the worst case around 2 KB of payload, which is still not that much, but it might be close to reality as users might not generate very long paragraphs until they come in contact with other nodes. $\infty$ was chosen as a theoretical limit.

As for the number of messages that can be exchanged during a contact, we have chosen a limit of 1 message per second. This means that the average amount of data exchanged per second is only



DOI: 10.1002/cpe




(a) Routing layer hit rate

(b) Application layer hit rate

(c) Routing layer num. of messages

(d) Routing layer delivery latency

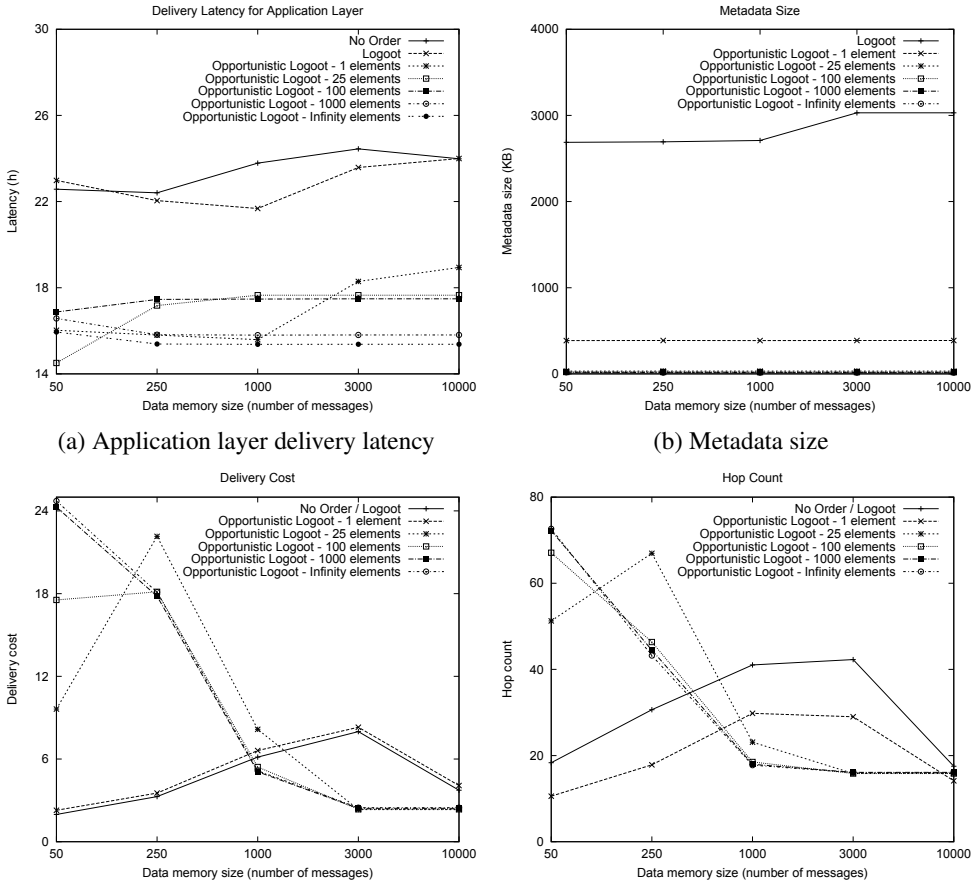
Figure 4. Sigcomm trace results, ONSIDE routing algorithm.

a few KB, as we shall see in the experiments. Exchanging a few KB per second using the current technologies can be considered pessimistic in some scenarios, as 802.11n Wi-Fi can reach around 100 Mbps, but we have chosen this value because not only Wi-Fi can be used for communication, but also Bluetooth, NFC and so on, and interferences can appear.

Figures 4 and 5 contain the results for the Sigcomm mobility trace. The total number of Wikipedia elements is 11406, and the number of extra messages is 7840, resulting in a maximum of 19246 generated messages. As we can see, both the hit rate and the delivery latency at the routing and application layers are better for our algorithm when compared to Logoot and No Order in almost all cases. For a merging limit of 25 and more, the main reason our algorithm performs well is given by the merging itself, as it reduces the number of routing layer messages to around 50%. But the direct download of missing messages also helps with this behavior, as proved by the case where we run Opportunistic Logoot with a limit of only 1 element. For this case, the number of messages at the routing layer is similar to Logoot and No Order, but not equal, due to the fact that our algorithm will *not* generate a message for elements that are removed while a message is mutable. More exactly, for every remove operation in a mutable message, we avoid the generation of not one, but two messages, as both the insert operation and the corresponding remove operation do not generate messages. This saving in routing messages, coupled with direct download, leads to better results for our algorithm even if we have a limit of 1 element for merging. Logoot and No Order have the same hit rate for routing layer because Logoot is just run on top of the routing algorithm without doing anything in plus, while the hit rate at the application layer is less for Logoot because the causal order layer withholds messages in case they do not respect causality.



DOI: 10.1002/cpe




(a) Application layer delivery latency

(b) Metadata size

(c) Delivary cost

(d) Hop count size

Figure 5. Sigcomm trace results, ONSIDE routing algorithm.

As for the metadata size, our algorithm significantly reduces the overhead, even in the case of a limit of 1 element where our identifiers are longer than the Logoot identifiers. Logoot requires around 2700 KB of metadata size for a data memory of 50 messages, which increases up to 3100 KB when the data memory increase to $\infty$. This behavior is normal, because, when we increase the data memory, we reach higher hit rates and deliver more messages to the application layer, which means we fulfill more dependencies for the Wikipedia elements and generate more elements at the routing layer. But having 2700 - 3100 KB of metadata is a lot, because, if we consider that the elements sent with every message are Unicode characters on two bytes, then we can have around 22 KB of payload as we have 11406 Wikipedia elements. 2700 KB of metadata for only 22 KB of payload (i.e. the payload is less than 1% of the metadata) means that the original Logoot cannot even be considered for our case scenario of shared editors with character granularity.

Our algorithm is able to produce good results, as it requires only 400 KB for a limit of 1 element, while for a limit of 25 elements, the metadata size is 32 KB, and as we go to $\infty$, the metadata size even reaches 15 KB, which is less than the payload size. The significant savings in metadata for limits greater than 1 are normal, as this is an ON environment where users do not communicate in real time. Users have enough time to generate an acceptable amount of content in mutable messages where users might be out of contact for 10 - 30 minutes (though in ONs it is possible to be out of contact for hours or even days), and take advantage of the merge operation. As an acceptable amount of content we consider only 25 - 100 characters, which we think any user could write in about 10-30 minutes (we have to also take in account grammatical corrections, as Logoot would generate new messages even for those, while our mutable messages avoid this scenario). Those 25 - 100 characters



DOI: 10.1002/cpe

do not need to be consecutive to reduce metadata (though that would be the best case scenario), as merging is not done only at the total order layer for Opportunistic Logoot identifiers, but also at the causal order layer that uses the same $CB$ vector even for elements that are *not* consecutive.

But even if we do not merge messages, we still reduce the metadata, as proved by the fact that our algorithm uses only 400 KB for a limit of 1 element when compared to the 2700 - 3100 KB in the case of the original Logoot. The main reason for this behavior is given by the range limits in our identifiers, which are the motive our identifiers are longer than the original Logoot identifiers. This might look like a paradox, but, by using the range limits, our identifiers arrive at an average amount of pairs per identifier less than the average amount of pairs per the original Logoot identifier. In our experiments, we generate the $x$ value in the identifier using the *half method* (see section 4.3), so, in case of consecutive insertions, we run out of identifiers made of only one pair in $log(N)$ operations. But $N = 2147483647$ (the maximum integer in Java), so after $log(2147483647) \approx 31$ insertions, which is really fast, we have to use 2+ pair identifiers. We could have used next available instead, but that solution comes with even a higher cost as insertions in the middle of the contents will have longer identifier from first insertion. But this limitation of the Logoot identifiers does not have such a dramatic effect on our identifiers. Even with a limit of 1 element, if the same site generates consecutive elements, we choose as an identifier the next available value (i.e. for <10,5,2,2> we would use <10,5,3,3>). This is similar to the next available method, but it works differently. In case we would want to insert a new element between <10,5,2,2> and <10,5,3,3>, and <10,5,3,3> is mutable and there is no immutable message with identifier <10,5,4,4>, then we move the element with identifier <10,5,3,3> to <10,5,4,4>, and assign <10,5,3,3> to the new element. The original Logoot does not do this, and that is why we obtain better results even for a limit of 1 element. As for the reason our algorithm does not have an increase in metadata size as we increase the data memory, we have the fact that almost all dependencies for Wikipedia elements were received even in the case of 50 messages.

As mentioned earlier, we ran our algorithm also for UPB and Infocom traces, and we also used Epidemic instead of ONSIDE as a routing algorithm. The results have similar trends, with differences only given by the routing algorithm no matter the trace. For UPB we have higher inter-contact times, so our algorithm is able to save even more metadata as more messages are merged, which in turn means better hit rates. But for Infocom the inter-contact times are less than for Sigcomm and UPB, resulting in less metadata savings. As for using Epidemic as a routing algorithm, just as mentioned, the results have similar trends when it comes to metadata savings and better hit rates due to these savings. Due to space constraints and the fact that the results are similar, we have chosen not to present them here. For a detailed information about those differences see the ONSIDE article [5].

## 7. CONCLUSIONS

In this paper, we have presented an algorithm for total order in ONs called Opportunistic Logoot. The experiments showed that, by using Opportunistic Logoot identifiers with messages merging, we are able to significantly reduce the metadata size. This reduction results in more space for messages at the routing layer, which, together with direct download of missing messages, is able to produce comparable results for the hit rate at application layer in regard to the hit rate at the routing layer even when not enforcing any order, and in some cases obtain better results.

It is important to note that the experiments have limitations. The mobility traces we have used do not represent all the types of ONs as they have been obtained only in academic and conference environments. Another problem is that the Wikipedia pages used in Section 6 were created in a server-client architecture, and we have modeled them to support decentralized nodes, which is not the equivalent of how users will really use the shared documents in an ON. Unfortunately, we are not aware of any distributed shared documents created especially for ONs where messages are not exchanged in real time.



DOI: 10.1002/cpe

## ACKNOWLEDGMENT

This work was supported by the Romanian national project MobiWay, Project PN-II-PT-PCCA-2013-4-0321. The authors would like to thank reviewers for their constructive comments and valuable insights.

## References

1. L. Andre, S. Martin, G. Oster, and C.-L. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 50–59, Oct 2013.
2. Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.
3. R.M. Bloom and W.J. Dunn. *The Constitutional Infirmity of Warrantless NSA Surveillance: The Abuse of Presidential Power and the Injury to the Fourth Amendment*. Boston College Law School faculty papers. Boston College Law School, 2007.
4. Gang Cheng, Yong Zhang, Mei Song, Da Guo, Deyu Yuan, and Xuyan Bao. An novel message delivery mechanism in opportunistic networks. In *Human Centered Computing*, pages 900–909. Springer, 2014.
5. R.-I. Ciobanu, R.-C. Marin, C. Dobre, V. Cristea, and C.X. Mavromoustakis. ONSIDE: Socially-aware and Interest-based dissemination in opportunistic networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–6, May 2014.
6. Radu-Ioan Ciobanu, Ciprian Dobre, and Valentin Cristea. SPRINT: social prediction-based opportunistic routing. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a*, pages 1–7. IEEE, 2013.
7. Radu-Ioan Ciobanu, Ciprian Dobre, Mihai Dascălu, Ştefan Trăuşan-Matu, and Valentin Cristea. Sense: A collaborative selfish node detection and incentive mechanism for opportunistic networks. *Journal of Network and Computer Applications*, 41:240–249, 2014.
8. RaduIoan Ciobanu, Ciprian Dobre, and Valentin Cristea. Social aspects to support opportunistic networks in an academic environment. In *Ad-hoc, Mobile, and Wireless Networks*, volume 7363 of *Lecture Notes in Computer Science*, pages 69–82. Springer Berlin Heidelberg, 2012.
9. Marco Conti, Silvia Giordano, Martin May, and Andrea Passarella. From opportunistic networks to opportunistic computing. *Comm. Mag.*, 48(9):126–139, September 2010.
10. Colin Daileda. How People in Hong Kong Can Communicate if Cell Networks Go Down, 2016. http://mashable.com/2014/09/29/hong-kong-cell-network-chat/, Last access February 2, 2016.
11. Michael Demmer and Kevin Fall. The design and implementation of a session layer for delay-tolerant networks. *Computer Communications*, 32(16):1724–1730, 2009.
12. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.
13. Alex Fitzpatrick. Hong Kong's Protesters Don't Need the Internet to Chat With One Another, 2016. http://goo.gl/uYRmco, Last access Janary 10, 2016.
14. Neil Fraser. google-diff-match-patch, 2012. https://code.google.com/p/google-diff-match-patch/, Last access on January 12, 2016.
15. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
16. Google Inc. Android One, 2016. http://www.android.com/one/, Last access January 13, 2016.
17. Pan Hui, Jon Crowcroft, and Eiko Yoneki. BUBBLE Rap: social-based forwarding in delay tolerant networks. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, pages 241–250, New York, USA, 2008. ACM.
18. Teemu Kärkkäinen and Jörg Ott. Shared content editing in opportunistic networks. In *Proc. of the 9th ACM MobiCom Work. on Challenged Networks*, CHANTS '14, pages 61–64, New York, USA, 2014.
19. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
20. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
21. Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.
22. Radu-Corneliu Marin, Ciprian Dobre, and Fatos Xhafa. Exploring predictability in mobile interaction. In *EIDWT*, pages 133–139, 2012.
23. Samir Okasha. Altruism, Group Selection and Correlated Interaction. *The British Journal for the Philosophy of Science*, 56(4):703–725, December 2005.
24. Open Garden. FireChat, 2016. http://opengarden.com/firechat, Last access January 14, 2016.
25. Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for P2P Collaborative Editing. In *Proc. of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.
26. Jörg Ott and Jussi Kangasharju. Opportunistic content sharing applications. In *Proceedings of the 1st ACM workshop on Emerging Name-Oriented Mobile Networking Design-Architecture, Algorithms, and Applications*, pages 19–24. ACM, 2012.
27. Jörg Ott and Dirk Kutscher. Bundling the Web: HTTP over DTN. In *Proceedings of the ACM WNEPT Workshop*, 2006.



DOI: 10.1002/cpe

28. Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *Communications Magazine*, 44(11):134–141, 2006.
29. Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *Communications Magazine, IEEE*, 44(11):134–141, 2006.
30. Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese, and Christophe Diot. MobiClique: Middleware for Mobile Social Networking. In *Proc. of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 49–54, New York, NY, USA, 2009. ACM.
31. Ravi Prakash, Michel Raynal, and Mukesh Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41:190–204, 1997.
32. Christian Rohner, Fredrik Bjurefors, Per Gunningberg, Liam McNamara, and Erik Nordström. Making the most of your contacts: transfer ordering in data-centric opportunistic networks. In *Proceedings of the third ACM international workshop on Mobile Opportunistic Networks*, pages 53–60. ACM, 2012.
33. André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proc. of the 3rd International Work. on Distributed Algorithms*, pages 219–232, London, UK, UK, 1989. Springer-Verlag.
34. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proc. of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.
35. Haifeng Shen and Chengzheng Sun. Flexible notification for collaborative systems. In *Proc. of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, pages 77–86, New York, NY, USA, 2002. ACM.
36. D. Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *Software Engineering, IEEE Transactions on*, SE-9(3):219–228, May 1983.
37. Amin Vahdat and David Becker. Epidemic routing for partially-connected ad hoc networks. Technical report, UCSanDiego, 2000.
38. Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.
39. Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 404–412, June 2009.
40. Wikipedia. Help:export, 2015. http://en.wikipedia.org/wiki/Help:Export, Last access on January 12, 2016.
41. Wikipedia. Hoverbox, 2015. http://en.wikipedia.org/wiki/Hoverbox, Last access on February 1, 2016.
42. Wikipedia. Japanese in Mangaland, 2015. http://en.wikipedia.org/wiki/Japanese_in_Mangaland, Last access on January 30, 2016.
43. Wikipedia. Opportunistic mesh, 2015. http://en.wikipedia.org/wiki/Opportunistic_mesh, Last access on January 10, 2016.
44. Wikipedia. Sagas of the demonspawn, 2015. http://en.wikipedia.org/wiki/Sagas_of_the_Demonspawn, Last access on February 2, 2016.




DOI: 10.1002/cpe