An Efficient Synchronization Method for Wireless Networks

Cem Karan
U.S. Army Research Laboratory
2800 Powder Mill Rd.
Adelphi, MD 20783
tel: (301) 394-0667
email: cem.f.karan.civ@mail.mil

Proceedings of the 18th ICCRTS conference

BibTeX:

```
@inproceedings{Karan:2013fk,
Author = {Cem Karan},
Booktitle = {18th ICCRTS},
Keywords = {delay tolerant networking, information dissemination, rsync, distributed rsync},
Title = {An Efficient Synchronization Method for Wireless Networks},
Year = {2013}}
```

Suggested Topics:

1. Topic 8: Networks and Networking
2. Topic 11: Autonomy
3. Topic 7: Architectures, Technologies, and Tools

An Efficient Synchronization Method for Wireless Networks

Cem Karan[a]

[a]US Army Research Laboratory, 2800 Powder Mill Rd., Adelphi, MD 20783-1197, USA

ABSTRACT

When a group of independent actors wish to coordinate their actions with one another, each actor uses internal state information combined with information known about all other actors' states to decide what to do. This generally requires some form of communication to synchronize state information between all actors. Since communication is often costly and not all parts of an actor's state change between synchronization attempts, using a method such as `rsync` can reduce the number of bits transmitted, and thus, the communications cost. However, `rsync` is designed for pair-wise interactions. In wireless communications, group-wise synchronization, which is more efficient than `rsync`, is possible. This paper describes DANDELION, an algorithm that builds on the ideas of the `rsync` algorithm to efficiently distribute information to all actors in a group over a shared broadcast medium. The algorithm is described in detail, as well as some experimental results using the algorithm on robots. Finally, theoretical comparisons to `rsync` and generic multicast trees are provided, as well as experimental comparisons between an implementation and a current version of the `rsync` program, showing that the costs of transmission are as low as a multicast tree, without the associated cost of building or maintaining a tree.

Keywords: delay tolerant networking, information dissemination, rsync, distributed rsync

1. INTRODUCTION

Coordinating the actions of multiple actors, be they human, robotic, or otherwise, involves some amount of communication between each actor. Communications generally have costs, which are usually measuring some critical resource, such as power, bandwidth, opportunity cost, time, etc. If any of these resources are exhausted, then further progress on any task is impossible unless the resources are replenished. If the resource cannot be replenished, then the objective is unachievable and the mission is a failure. For this reason, it is generally desirable to reduce costs as far as possible.

Unfortunately, if we decide to optimize over all possible costs at the same time, we may end up with an unworkable system. In the worst case, it may fail to meet hard requirements for any message. For this reason, the system described in this paper was designed to solve only one goal optimally. That goal is to ensure that every node in a mobile ad-hoc wireless network has a copy of a byte string b at as low a cost as possible[1]. Cost is measured as the sum of all bits transferred over the network to ensure the goal. The number of bits transferred is used as the metric for several reasons:

- Transmitting bits wirelessly requires power. This paper assumes that transmitting fewer bits will reduce the power consumed, conserving it for other tasks.
- Nodes may be hidden from one another, which prevents nodes from coordinating their transmission strategies. This means that transmissions may unintentionally collide with one another. By reducing the number of bits that are transmitted, the likelihood of collision is reduced, effectively improving communications.
- Radios that are broadcasting are susceptible to detection and destruction by an adversary.

[0]Further author information: (Send correspondence to Cem Karan)
Cem Karan: E-mail: cem.f.karan.civ@mail.mil, Telephone: 1 301 394 0667

[1]The terms *bit string*, *byte string*, and *string* are used interchangeably throughout this paper.

To aid in meeting this goal, this paper defines the Total Cost Ratio (TCR) as follows:

$$\text{TCR} = \frac{\text{Total number of bytes transmitted over the air for byte string } b}{|b| \times \text{Number of nodes updated}} \tag{1}$$

where $|b|$ is the number of bits encoding string b. The numerator includes the cost of encoding b for transmission[2], the cost of each node communicating b to any other node including retransmission costs, and any other costs measured in bits that are incurred to transmit b to every other node in the network.

This paper assumes that the lower this ratio is, the lower the costs are, and therefore, the method that lowers this ratio the most is the best communications method to use. All of the methods analyzed in this paper are compared using this metric.

To meet this goal, this paper defines an epidemic-like algorithm called Dandelion that is based on an offer-request-response system, where the offer and request messages describe ranges of indices of bytes in byte strings, and the response is only sent when a request for a particular substring is made. This is analogous to what the `rsync` algorithm (see [1]) does, with one caveat: an arbitrary number of silent nodes may listen in on the broadcast, updating their own information cheaply. This idea is the same as the "opportunistic listening" idea of [2, 3].

Note that Dandelion is explicitly unconcerned with latency; it assumes that all messages it transports may be delayed an arbitrary period and still be valuable. Because of this, Dandelion is designed as a delay-tolerant networking system. Thus, while both `rsync` and multicast systems require more support to handle network partitions, Dandelion assumes that network partitions are a normal occurrence and silently handles them. Moreover, Dandelion turns the potential liability of latency into an advantage; instead of trying to transmit messages as rapidly as possible, each node in a Dandelion network will delay transmitting messages for a random period of time. By doing so, actors can determine if some other node is already transmitting the information, which means that this node can avoid transmitting it. This technique can significantly improve upon the costs incurred by other methods and can, in theory, approach what a multicast tree can provide. This technique is the primary contribution of this paper.

In addition, Dandelion's lower layer is fully anonymous. Actors are never given identities; only messages are. This is useful in rapidly changing topologies where knowledge of the set of neighbors may be out of date by the time the knowledge needs to be used. This is different from [2, 3], which at a minimum requires knowledge of how many neighbors a node has.

Finally, Dandelion does not require, nor does it use, a routing table. All operations are purely between one-hop neighbors. Since `rsync` is not designed to operate in such an environment, some changes had to be made in the experimental setup that benefitted `rsync` to the maximum extent possible. These are outlined in §4.

The cost of this flexibility is that Dandelion's lower layer protocol, which is the focus of this paper, is not a reliable protocol. Although under the appropriate assumptions all nodes will have identical state at time t_∞, the lower layer is not able to prove this fact to any broadcaster. That said, it should be possible to build a reliability layer on top of Dandelion that will prove to all actors that a message has been successfully received by all other actors in the network.

1.1 Organization and Scope of this Paper

§2 discusses several other methods of disseminating data across a wireless network in a robust and efficient manner. §3 then describes the Dandelion protocol in detail, proving its correctness and runtime, and then comparing it against several other protocols from a theoretical point of view. §4 goes from the theoretical to the experimental, comparing an implementation of Dandelion against what idealized versions of several other protocols would do if built on top of `rsync` as a communications protocol. §5 describes further work needed to make Dandelion practical, and §6 finishes with final conclusions that can be drawn from this work.

[2]This may involve error correction codes, packet overhead, etc.

2. RELATED WORK

One of the simplest methods of ensuring that all nodes in a network have a copy of a byte string is by flooding the string throughout the network. Unfortunately, flooding carries a high cost as nodes may receive the same information over and over again. In wireless networks, this has the added cost of effectively jamming the network. For this reason, pure flooding is not examined in this paper. Instead, epidemic and gossip protocols are examined.

Epidemic protocols (see [4]), which were built on the base that flooding provided, were designed to reduce the overhead of flooding. In this case, messages are modeled as a disease and nodes as disease carriers. Nodes communicate in a pair-wise fashion, determining if a node has been "infected" with a message or not. If not, then the carrier passes the message along. This process continues in a pair-wise fashion until all nodes have been infected. [5] extended this simple model to include the idea of immunization. Once a message has been delivered, an immunization message is broadcast, causing all receivers to be "immune" from further infestation by the same message.

Gossip protocols (analyzed in §3.1.3) are built upon epidemic protocols, delivering messages to randomly chosen subsets of the possible set of recipients. As a further optimization, some gossip protocols choose to drop messages at random. This further reduced the number of copies of messages, at the risk of losing some messages entirely.

A completely different line of research attempted to build multicast trees, ensuring that every node in the tree received a message exactly once and at the least total cost. This line of research depends not only on the ability to quickly form the tree initially, but also to keep the tree up to date as the network's topology changes. However, as discussed in §3.1.4, finding a minimal-cost multicast tree is NP-complete, which forces tradeoffs between the quality of the approximation of the minimal tree and amount of runtime used to find it. Moreover, many current algorithms require a centralized approach to build the tree, which can become a bottleneck.

3. DANDELION'S LOWER LAYER

Dandelion's fundamental view of data is that the universe of data can be partitioned into three sets: the set of information we know, the set that we know exists but don't have, and the set of information that we know nothing about. Algorithm 1 describes how Dandelion converts elements of the second and third sets into elements of the first set.

To support conversion of elements of the second set into the first, Dandelion gives each string a unique identifier, which allows different nodes to reference the string without actually passing the string along. It also gives each string a monotonically increasing version number, which allows Dandelion to replace old strings with newer strings. When a node wants to transmit some string, it first constructs an Offer message for it. The message contains the string's identifier, version, and the compressed set of indices of the bytes in the string. For example, if the string under discussion is `abcdefgh`, then the Offer would have the string's identifier and a compressed version of the set of indices $\{0, 1, 2, 3, 4, 5, 6, 7\}$. One possible compression scheme would only include the maximum and minimum in the set, resulting in a message that only contained the set $\{0, 7\}$. If a receiving node requires any of the offered bytes, it broadcasts a Request message with the compressed set of indices of the bytes it needs. Any node that receives the Request that has the given data can respond with a Data message, transmitting the actual bytes requested. This transmission of metadata prior to broadcast of actual data is what Dandelion borrows from `rsync`. Note that once a string is created, it, its identifier, and the version number for that version of the string are immutable. This is why the protocol works; if a node sees an (identifier, version) pair, then it is guaranteed that that pair refers to one and only one string.

Unfortunately, the above algorithm is insufficient to guarantee messages are broadcast to all recipients. If a node is unaware of the existence of a message (data in the third set), then it cannot participate in the protocol because it cannot reference the byte string's identifier. Dandelion has two mechanisms that solve this problem. First, every node periodically generates Offer messages for all the data they contain and Request messages for all the data they know exists but which the node is missing. Second, Dandelion defines the Query message type. When a node receives a Query message, it immediately generates Offer and Request messages for all data in its possession. In essence, this is identical to the periodic generation of those messages, except that it

Algorithm 1: This algorithm describes how DANDELION's lower layer processes incoming messages.

```
Input: $maxDelayTime$ A number in $\mathbb{R}^+$
Data: knownStringsDict A key-value datastore
Data: delayQueue A minimum priority queue keyed by time
Loop forever:
    Let $msg$ ← the next incoming message;
    if type($msg$) is QUERY then
        foreach partial string $ps$ in knownStringsDict do
            Let $delay$ ← random($(1 -$ receptionQuality($msg$)$) \times maxDelayTime$);
            updateOrAddIndicesToDelayQueue(OFFER, ID($ps$), indexSet($ps$), $delay$);

            Let $delay$ ← random($(1 -$ receptionQuality($msg$)$) \times maxDelayTime$);
            updateOrAddIndicesToDelayQueue(REQUEST, ID($ps$), missingIndicesSet($ps$), $delay$);
    else
        /* getPartialStringForMsg() will create a new, empty partial string if one doesn't
           exist for $msg$ in knownStringsDict if and only if $msg$'s version is later than the
           version of the latest complete partial string in knownStringsDict and one doesn't
           already exist for it.  Otherwise, it will either return an existing partial
           string or $None$ if $msg$ is older than the latest complete partial string in
           knownStringsDict.                                                                */
        Let $ps$ ← getPartialStringForMsg(knownStringsDict, $msg$);
        Let $delay$ ← random($(1 -$ receptionQuality($msg$)$) \times maxDelayTime$);

        if $ps \neq None$ then
            if type($msg$) is OFFER then
                Let $cancelSet$ ← indexSet($ps$) ∩ indexSet($msg$);
                subtractIndicesFromDelayQueue(OFFER, ID($msg$), $cancelSet$);

                Let $missingSet$ ← missingIndicesSet($ps$) ∩ indexSet($msg$);
                updateOrAddIndicesToDelayQueue(REQUEST, ID($msg$), $missingSet$, $delay$);
            else if type($msg$) is REQUEST then
                Let $cancelSet$ ← missingIndicesSet($ps$) ∩ indexSet($msg$);
                subtractIndicesFromDelayQueue(REQUEST, ID($msg$), $cancelSet$);

                Let $availableSet$ ← indexSet($ps$) ∩ indexSet($msg$);
                updateOrAddIndicesToDelayQueue(DATA, ID($msg$), $availableSet$, $delay$);
            else if type($msg$) is DATA then
                Paste all data in $msg$ into $ps$;
                Let $msgSet$ ← be a set consisting of all and only the indices of all known substrings of $msg$;
                subtractIndicesFromDelayQueue(REQUEST, ID($msg$), $msgSet$);
                updateOrAddIndicesToDelayQueue(OFFER, ID($msg$), $msgSet$, $delay$);
                if $ps$ is now complete then
                    foreach partial string $oldPS$ in knownStringsDict[ID($msg$)] older than $ps$ do
                        subtractIndicesFromDelayQueue(REQUEST, ID($oldPS$), allIndices($oldPS$));
                        subtractIndicesFromDelayQueue(OFFER, ID($oldPS$), allIndices($oldPS$));
                        subtractIndicesFromDelayQueue(DATA, ID($oldPS$), allIndices($oldPS$));
                        Delete $oldPS$ from knownStringsDict;

        else
            Let $availableSet$ ← indexSet(Latest complete version of $ps$ we have);
            updateOrAddIndicesToDelayQueue(OFFER, ID($msg$), $availableSet$, $delay$);
```

occurs immediately. Between both of these mechanisms, as time goes to infinity, all messages will be disseminated to all nodes.

If this were all there was to DANDELION's protocol, then it could be replaced by pairwise `rsync` updates. DANDELION's contribution is that it uses delay to its own advantage. Every message that a node plans on transmitting is put into a delay queue[3], which is a minimum priority queue that is sorted by time. While the message is in the delay queue waiting to be transmitted, the local node is processing messages that are coming in. If a remote node has transmitted a message that overlaps one that the local node is planning on transmitting, then the local node squashes that portion of its own message. This behavior solves the scalability problem that flooding causes.

3.1 Analysis of Dandelion and Other Protocols

In this section, proofs of correctness are provided as well as expected costs for using DANDELION. These costs are compared with several other broadcast algorithms, and it is shown that, in the best case the single-round costs of using DANDELION can be driven arbitrarily low. Also described is the worst-case behavior, which is shown to scale linearly with the number of nodes in the network. The proofs are correct based on the following assumptions:

- The network connectivity graph may become partitioned at any time.
- The link quality between two nodes may not be symmetric. That is, while the link quality from A to B might be excellent, the link quality from B to A might be poor.
- For any particular link, failures are distributed uniformly at random over the packets that are sent over the link. In particular, this means that there does not exist a malicious agent that is choosing to jam a particular packet over and over again.
- Once a string is created, it, its version, and the string form an immutable triple.
- Any actor may fail at any time, either permanently or temporarily.
- Actors are never malicious; they will always try to execute the protocol correctly.

The last point is especially important. DANDELION is only resistant to random failures; it has no ability to withstand concerted effort to damage the network. It may be possible to extend DANDELION to improve its security, but such work is beyond the scope of this paper.

For mathematical completeness, the following assumptions are also made:

- The number of nodes is finite.
- All transmitted information (identifiers, version numbers, bit strings, etc.) are finite in length, and there are only a finite number of each.
- After some time t_{end}, no new strings are created.
- For every piece of information and every pair of nodes n_i and n_j, at some point in time, there exists a path (which may be temporally disconnected) from n_i to n_j.

A temporally disconnected path is one where not all nodes in the graph are in the same partition at the same time, but there is still a path over time that a message can traverse to reach its destination. As an example, if a graph contains nodes $\{n_i, n_j, n_k\}$, it may be that at time t_0 the graph is partitioned into $\{\{n_i, n_j\}, \{n_k\}\}$, while at time t_1, where $t_1 > t_0$ the set of partitions is $\{\{n_i\}, \{n_j, n_k\}\}$. While at no time are all nodes in the same partition, it is still possible for a message to travel from n_i to n_k by being transmitted through n_j.

[3]Delay queues are not a new concept. Thread schedulers in operating systems do something similar, as does Java's `DelayQueue` (`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html`).

3.1.1 Correctness

Before considering using a new protocol, one should be convinced that the protocol is correct. If Dandelion achieves the goal of distributing all byte strings to all nodes, then we can consider it to be correct. Let the knowledge that each node has about the set of byte strings in the universe be that node's state. In this case, if there is a point in time after which no byte strings are created or modified and the states of all nodes are identical, then all nodes must have copies of all byte strings. Theorem 1 proves that under reasonable assumptions at time t_∞ all nodes will have identical states.

Theorem 1 *If for all time t, $t_{end} < t < t_\infty$, the set of nodes N are described by a connected communications graph, then with probability 1, $state(n_i) = state(n_j)$, $\forall n_i, n_j \in N$. Note that "connected" in this case means that a path between nodes n_i and n_j always exists such that the probability that a packet will successfully be transmitted from n_i to n_j is greater than 0.*

Proof If this is false, then a message m must exist that node n_i has and node n_j will never receive. Dandelion nodes periodically generate Offer messages for all data in their possession. Since this is periodic, it is clear that in an infinite amount of time, there will be an infinite number of attempts. If p is the probability of successfully receiving the message, then the probability of failing for all of time is $(1-p)^\infty$. Since $p > 0$, $\forall n_i, n_j \in N$, we know the probability that the message fails to get across the network for all of time is $(1-p)^\infty = 0$. This implies that the probability a message will be successfully transmitted from n_i to n_j is 1. We can make similar arguments for the Request and Data messages that make up the rest of the protocol. Thus, m must be transmitted from n_i to n_j with a probability of 1. This contradicts the statement that it is not transmitted, proving that Dandelion is correct. ■

Unfortunately, Theorem 1 is only true if we wait an infinite amount of time and if the network is connected for that time period. If there isn't a temporally connected path, then information cannot disseminate. If there are only a finite number of finite-sized windows of opportunity for a message to be transmitted, then the probability that a message will make it to all nodes will be less than 1, and in sufficiently pathological cases, the probability will fall to 0. Determining what the exact probability is for a given topology requires a great deal more knowledge about the network topology, the environment, etc., and is therefore beyond the scope of this paper.

3.1.2 Cost Effectiveness

Theorem 1 proves Dandelion is correct, but it doesn't answer why anyone should consider using Dandelion. After all, Dandelion makes no particular effort to reduce latency; messages may disseminate throughout the network slowly. What Dandelion offers is a cost savings as measured by the TCR (see equation 1) over what other protocols can provide. In this section, Dandelion is compared with other protocols to show that it is at least as efficient as other protocols at disseminating information.

Calculating the cost of using Dandelion is not easy for three reasons. First, the network is assumed to be constantly in flux, resulting in a set of probabilities that are constantly changing. Second, if the network becomes partitioned and is never rejoined, then determining the costs requires knowledge of what data were synchronized before the partition event occurred. To simplify matters, in this section, the probability of any message of any type (Offer, Request, Data, or Query) being successfully received by any node is assumed to be p, and it is assumed that $p > 0$ for all time. Note that p is the worst possible probability at any point in time for any message type; this ensures that the analysis is an upper bound on the actual costs. This is also why all calculations are done using Big-O notation, represented by the function $\mathcal{O}()$. The final reason why calculating costs for Dandelion is difficult is because Dandelion periodically sends Query messages to see if anything has changed. The Query acts like a beacon and causes all listeners to send Offer and Request messages out. This means that there is a minimum amount of traffic on the network at all times. Once all byte strings have been synchronized, this traffic becomes overhead, reducing the TCR as defined in equation 1 to zero. That said, this is true of any communications system where network partitions are possible and where new byte strings can be generated at any time. The only question is how much overhead does a particular communications system incur.

The final point outlined above makes comparisons to other methods more difficult. `rsync` assumes a connected network where the endpoints can communicate with one another, and multicast methods require additional protocol support to be able to decide if the network has been partitioned since the last time a byte string has been updated. Thus, to directly compare `rsync` and multicast methods with DANDELION would require adding a beaconing protocol to them. Since this would then raise the question of what beaconing methods were used, this paper avoids the problem by choosing to remove any beaconing portions of any protocol from the analysis, concentrating only on those portions of the protocol that are necessary to communicate data. Note that the implicit assumption is that the costs of beaconing are similar for all protocols, which may not be true.

3.1.2.1 Dandelion's Cost Assume that we wish to guarantee that all nodes in network N of size $|N|$ have a copy of some byte string b. By the DANDELION protocol, we know that if all members of N already have a copy of b, then no node will ever request it or issue DATA messages containing it. Thus, with the exception of periodic beaconing, which as stated above is ignored, no communication will occur. On the other hand, if only one node has a copy of b, then some amount of communication must occur. Unfortunately, without beaconing, the probability that a byte string will be transferred from one node to its nearest neighbor is at most p. This is because outside of beaconing, OFFER messages are generated exactly once. If we accept this and continue with the analysis, then we know that at a minimum, we must have one OFFER, one REQUEST, and one DATA message for any message that is being transmitted. However, once the byte string has been successfully copied over, it will not be requested a second time.

Because we wish to concentrate on a worst-case analysis, we must also search for a worst-case topology. Since all nodes in a DANDELION network engage in "opportunistic listening," there is a chance that some nodes will participate in the protocol and receive a copy of b without ever transmitting any bytes. However, this can only work if there are nodes that can listen in on an exchange without communicating. The only topology where opportunistic listening might not be possible is a simple line. There are two types of nodes on a line, interior nodes and the two end nodes. Interior nodes always have two neighbors, which means one neighbor could listen in on an exchange between the interior node and the other neighbor. This means that interior nodes are not guaranteed to be the worst-case for DANDELION. On the other hand, the end points do not suffer from this problem. Thus, for the purposes of worst-case analysis, this paper assumes that the network topology is linear and that one of the endpoints of the network is the originator for byte string b.

Fortunately, a linear topology is also the simplest to analyze. Every pair of neighboring nodes is identical to any other pair of neighboring nodes, which means that the total cost for transmitting a byte string to all nodes is simply the cost of transmitting a message between a pair of nodes multiplied by the total number of connected pairs in the lines. It is easily seen that the total number of pairs of nodes in a straight line is $|N|-1$. This leaves the intra-pair cost.

Since messages are assumed to fail at random, we cannot calculate an exact transmission cost, but must instead calculate an expected cost for a given level of reliability. This paper assumes that the reliability must be maximized, and so the inter-node costs are analyzed on this basis. Given this, we can assume that node n_i is trying to transmit b to its immediate neighbor n_j. First, n_i must send an OFFER, then n_j must send a REQUEST, and finally n_i must send a DATA message. Since we are ignoring the beaconing part, n_i will send the OFFER message exactly once. We know that this is successfully received with probability p, which means that the rest of the protocol will be followed with probability p. However, if n_j receives the OFFER, it will generate REQUEST messages until it receives the data. The probability n_i will not receive the REQUEST is $1-p$ the first time, $(1-p)^2$ the second, and so forth. However, the probability that n_j will send the REQUEST given that it received the OFFER is $(1-p)^0 = 1$ the first time, $(1-p)^1$ the second time, and so forth. Carried forward M times yields the geometric sum:

$$\text{Expected cost} = \sum_{i=0}^{M-1} (1-p)^i = \frac{1-(1-p)^M}{1-(1-p)} \tag{2}$$

$$= \frac{1-(1-p)^M}{p} \tag{3}$$

If we wish to guarantee that the REQUEST is received, it must be transmitted an infinite number of times. Thus, we expect to transmit $\frac{1}{p}$ messages to guarantee completion of this phase of the protocol.

Once n_i receives the REQUEST message, it replies back with a DATA message, and it does so each and every time it receives a REQUEST. In a manner similar to the calculation above, this means the probability that n_i sends a DATA message the first time is $(1-p)^0 = 1$, etc., which means that the costs of sending DATA messages is also $\frac{1}{p}$.

If we assume that the cost of sending a message is a constant, then the cost for sending b from n_i to n_j is at most:

$$\mathcal{O}\left(1+\frac{1}{p}\right) \tag{4}$$

However, as stated earlier, without beaconing, the OFFER message is sent once, which means that the costs actually go as:

$$\mathcal{O}\left(1+p\cdot\frac{1}{p}\right) = \mathcal{O}\left(1+1\right) = \mathcal{O}\left(1\right) \tag{5}$$

where all the usual reductions due to $\mathcal{O}\left(\right)$ notation have been done. The cost across the entire line easily follows as:

$$\mathcal{O}\left(|N|-1\right)\cdot\mathcal{O}\left(1\right) = \mathcal{O}\left(|N|\right) \tag{6}$$

with a probability of $p^{|N|-1}$ of all nodes receiving the message.

3.1.3 Gossip Protocols and Their Costs

Gossip protocols build on epidemic protocols. Because of the great similarity between the two, and because gossip protocols are designed to be more efficient than epidemic protocols, this paper does not analyze epidemic protocols.

There are many different gossip protocols defined, but they all share similar characteristics (see [6–10] for various background papers). In general, if a node n_i is able to communicate with some set of nodes S, it will choose some subset of nodes $T \subseteq S$ at random with which to perform pair-wise synchronizations. Some protocols build routing tables first, using gossip protocols solely to bootstrap to a multicast tree. In this analysis, only the gossip protocol itself is considered, which means that S is all and only the one-hop neighbors of n_i.

The worst-case scenario for a gossip protocol is a clique[4], because this maximizes the probability that two nodes will choose to update the same node[5]. Let N be the set of all nodes in the network. If we assume that for any node $n_i \in N$, n_i will choose k nodes at random to perform a pair-wise update with, then we know that $(|N|-1)k$ messages will be sent. Since we only require at most $(|N|-1)$ messages to be sent to guarantee that every node in N receives the message, we waste $(|N|-1)(k-1)$ messages. If we make assumptions similar to §3.1.2.1 about costs and transmission probabilities, then we can immediately state that the cost of this protocol is:

$$\mathcal{O}\left(\frac{1}{p}\cdot(|N|-1)(k-1)\right) = \mathcal{O}\left(|N|\,k\right) \tag{7}$$

Comparing equation 7 to the worst case for DANDELION, which is equation 6, we can see that DANDELION is a significant improvement over ordinary gossip protocols.

That said, gossip protocols have one significant advantage over DANDELION; as long as the network is connected, they can eventually reach an equilibrium state and cease communicating. On the other hand, if the network becomes partitioned for some time window, then there is some possibility that nodes within the two separate partitions will attempt to synchronize with one another, which will cause a failure and retry until the synchronization succeeds. These retry attempts have a cost in a manner similar to what DANDELION has. Thus, if the network is guaranteed to be connected long enough for all nodes to fully synchronize with all other nodes and no more byte strings will be generated after that point, gossip protocols will be more efficient than DANDELION. However, if byte strings are generated continuously or if the network can become partitioned, DANDELION may be a better choice.

[4] A *clique* is a complete graph; every node is neighbors with every other node in the graph.

[5] In this case, protocols that are able to talk to their one-hop neighbors are being analyzed. If the topology is a line, then this guarantees that nodes that are three hops or further apart can never choose to update the same node.

Before finishing this section though, the costs of gossip protocols that choose to drop packets at random need to be considered. Assume that the probability that a packet will be dropped is p_d, where p_d is chosen to ensure that exactly as many messages are generated as is needed to ensure all nodes receive the message. This means that instead of $(|N|-1)k$ messages being generated and sent, $(|N|-1)k(1-p_d)$ are generated and sent. As long as $k > \frac{1}{1-p_d}$, these gossip protocols will still generate more messages than they strictly require, and more messages than DANDELION generates. However, if the drop rate is such that $k < \frac{1}{1-p_d}$, messages will tend to die out before they are propagated to all nodes. Moreover, unless each node is able to choose its own value for p_d and change it dynamically, it is likely that for some nodes p_d will be incorrect. It may even be true that for some topologies no value of p_d is appropriate for at least some nodes.

3.1.4 Multicast's Cost

Because building a multicast tree in the field would require some amount of beaconing, we must assume that the multicast tree has already been built and that we are simply transmitting messages through it. Using a similar approach to §3.1.2.1, we let $p_{i,j} = p$ for all nodes n_i, n_j in the graph, and we assume the cost of sending byte string b is a constant. Also, assuming that a message is transmitted until it is received, we can calculate the cost of transmission as

$$\begin{aligned}\text{Multicast Data cost} &= \mathcal{O}\left(\sum_{i=0}^{\infty}(1-p)^i\right) &(8)\\ &= \mathcal{O}\left(\frac{1}{p}\right) &(9)\\ &= \mathcal{O}(1) &(10)\end{aligned}$$

which is the cost of sending a message across a single link. If we assume that the multicast system has been designed to use the "wireless multicast advantage" described in [11], then its behavior will be identical to that of DANDELION. The advantage is destroyed if a communications system is forced into using (or simulating) unicast. The set of situations that cause these problems for multicast systems are the same as those for DANDELION, and vice-versa. The easiest such situation to identify is when the tree is a single long line, in which case the total worst-case multicast data transmission cost is the same as the worst-case cost for DANDELION (see equation 6):

$$\text{Total worst case multicast data cost} = \mathcal{O}(|N|) \tag{11}$$

Unfortunately, there are several problems with multicast approaches. First, they always require that the multicast tree be built or specified before any data can be transmitted. The ideal tree would be one that minimizes the total costs to transmit to all nodes. This is the definition of the Directed Steiner Tree problem. The regular Steiner Tree Problem is known to be NP-complete (see [12, ND12]). It is easy to show that if the Directed Steiner Tree problem could be solved in polynomial time, then so could the Steiner Tree Problem, proving that the Directed Steiner Tree problem is also NP-complete. There are a number of approximation methods (see [13–17]), but each suffers from its own problems. Some of those problems include the requirement that the tree be built in a centralized fashion, the time it takes to build the tree, and the amount of communication necessary to build the tree.

The second major problem is that the tree must be rebuilt whenever the network topology changes. Since robotic networks can be highly dynamic, the costs of rebuilding the tree each time the network changes can quickly exceed the cost of transmission.

Finally, algorithm designers must decide if they want a reliable network or not. If they decide on a reliable network, then every node that receives a message must acknowledge receipt of a message, raising the overall costs of the network and forcing decisions on what to do if the network becomes partitioned. Even if the designer decides an unreliable network is acceptable, the tree must still be reliably built, otherwise the nodes in the tree may not agree on what the topology of the tree actually is, which may lead to network partitions even when nodes are within communications distance of one another.

3.1.5 An Undecidable Problem

As mentioned in §3.1.2, all protocols face the same problem; unless they communicate their presence, it is impossible for neighboring nodes to determine if they have neighbors or if their set of neighbors has changed. In a purely static network, the cost of determining a node's set of neighbors can be paid for once, at network start up. Provided the cost of determining the topology is finite, this cost can be amortized over the lifetime of the network, which if the network is alive for an infinite period of time, means the cost amortizes to 0.

However, if a network is dynamically changing in an unpredictable manner, then the only way for nodes to know if they have neighbors, let alone what their set of neighbors are, requires periodic attempts at message exchange. Different heuristics can be used to determine how often a node needs to check its neighbor set, but it will always need to do so. This places a lower bound on any cost calculation. This was not addressed in any of the cost calculations described above, nor can it be. Although DANDELION explicitly states that it tests for neighbors via QUERY messages, the other protocols either don't address this issue or, in the case of flooding, ignore the issue entirely. Because of this, this paper does not to try to address the costs of determining the existence of neighbors in this paper.

4. EXPERIMENTS AND RESULTS

Although the results outlined in §3.1 make a strong argument in favor of DANDELION, they are just theoretical results. By comparing implementations with one another, we can gain further insight into actual behaviors and gain greater confidence in the correctness of any theoretical claims.

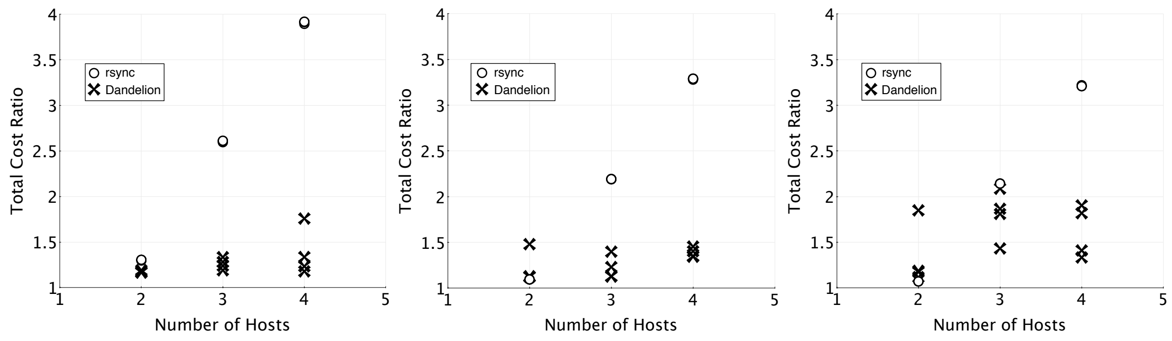
4.1 Experimental Setup

All testing was done using the built-in 802.11 wireless capabilities of the set of test laptops. The size of the sets were varied from 2 to 4 laptops in the set. Each laptop was given a string that it was to distribute to all other laptops. For each test, the size of the string was kept the same for each laptop; thus, for a test consisting of a 10,000 byte string, each laptop had its own 10,000 byte string that it had to distribute to every other laptop in the set. The test was not considered complete until every laptop had a copy of every string. Once a test was complete, the strings that each laptop had received were collected to a central location to compare the strings. This verified that every laptop not only had a copy of every string, but that every string on every laptop was identical, thus proving the correctness of the protocol.

Because DANDELION is designed to improve communications when many actors can listen in on a conversation, the tests were constructed to ensure that all laptops were able to communicate directly with one another. In short, the communications graph was a complete graph. Furthermore, care was taken to ensure that compression was disabled for all tests.

For all tests, Wireshark [18] was used to capture all packets. Because this implementation of DANDELION operates over UDP on port 5000, it was possible to write a display filter that displayed only those packets for particular IP address ranges. Wireshark could then be used to display summary information, which showed the number of bytes each host transmitted. From this, totals were generated, which allowed for the calculation of the TCR in equation 1. Each set of tests was repeated four times to check for consistency. Once the tests were started, all test results were kept; this ensured that the tests were fair, showing when DANDELION performed poorly as well as when it performed well.

Due to time constraints, creating a complete multicast protocol or a gossip algorithm was not possible. For this reason, the experiments are designed to tilt in the favor of both multicast and gossip protocols, with the reasoning that if the implementation of DANDELION performed well against an idealized form of multicast or gossip protocols, then it will perform even better against real implementations. Because of this, all experiments with `rsync` were done with one pair of laptops alone so as to minimize interference. If Wireshark did detect interference, then those runs were discarded. Once the data were captured, they were mathematically extended to the best case for both multicast trees and gossip algorithms. Mathematically, in the best case, every node in a network is updated exactly once. For this reason, both multicast trees and gossip algorithms reduce to trees, which is why the results shown in Figure 1 only reflect one set of `rsync` results.



(a) 10,000 byte-long tests (b) 100,000 byte-long tests (c) 1,000,000 byte-long tests

Figure 1: These plots show the Total Cost Ratio (as defined in equation 1) to synchronize different length strings between different numbers of hosts using different methods.

4.2 Results

Figure 1 shows the real results of Dandelion against the idealized, extended results of `rsync`. As can be seen, even when the results have been carefully tilted in favor of both multicast protocols and gossip protocols, Dandelion is still an improvement.

5. FURTHER WORK

The Dandelion algorithm requires significant work before it can be deployed on practical networks. The following sections outline some of the work that must be done before Dandelion could be deployed in the real world.

5.1 Efficiency

One concerning feature of Dandelion is the large variance between the number of bytes different nodes transmit to achieve similar results. While it is expected that as the number of nodes increases, so does the interference, one would expect that all nodes would experience similar amounts of interference, thus keeping the variance small. However, during the worst-case test that was recorded, the worst node had to send approximately 3.5 bytes for every byte successfully received by the rest of the laptops in the test, while the best node in the same test only had to transmit approximately 1.2 bytes. Moreover, as is clear from figure 1, as the number of nodes increases, so did the variance.

Comparing this to `rsync`, it is clear that the variance is too high, as are the number of bytes used. This must be solved for Dandelion to make the protocol generally useful.

In addition, the current protocol will not stop transmitting a given byte string, even if all nodes in a network have received it. It would be useful if it were possible to decide that a given string could be dropped. The current protocol can only decide if a string has been superseded by a later string, but it cannot decide to drop a string entirely.

5.2 Latency and Throughput

Although Dandelion is not explicitly concerned with latency, in general, the less latency a network has, the more useful operations it can support. It would be useful if nodes could desynchronize from their neighbors in such a way that the probability of packet collision was minimized while the use of the network was maximized. This would tend to decrease latency while increasing throughput. For a practical system, this would need to be done automatically and dynamically, without user interaction.

5.3 Security

As it is currently defined, the Dandelion protocol does not provide any security; any eavesdropper can join the network and participate in the protocol. Adversaries that are able to inject data into the network can break the protocol in any number of ways. Encryption and authentication can help, provided nodes cannot be physically compromised to become malicious nodes, but if any node is compromised, then the result is identical to a malicious node joining the network. This is because the current protocol cannot decide if a node should be removed from the network, much less actually do so. Unless this is solved, Dandelion will not be generally useful.

6. CONCLUSIONS

This paper introduced the Dandelion protocol, proving that it is both correct and efficient. It also showed that by operating in the time domain, it is possible to build networks that are as efficient as multicast trees can be, while avoiding the problems associated with having to build multicast trees. This manipulation of networks in the time domain is the primary contribution of this paper. By extending this idea, it may be possible to design better protocols in the future that are able to efficiently use wireless resources.

ACKNOWLEDGMENTS

Many thanks to Drs. Jon Fink and Ethan Stump for answering questions and acting as sounding boards for the more unusual ideas. Many more thanks to my advisors Vijay Kumar and Kostas Daniilidis for being there to answer questions and address concerns that no one else could answer. Last, but by no means least, thanks to my Branch Chief Stuart Young, who has had to hear time and again that something was failing and the exact cause had yet to be tracked down. None of this would have been possible without all of his support on this. Thanks.

REFERENCES

1. A. Tridgell, *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, February 1999.
2. N. Suri, G. Benincasa, M. Tortonesi, C. Stefanelli, J. Kovach, R. Winkler, R. Kohler, J. Hanna, L. Pochet, and S. Watson, "Peer-to-peer communications for tactical environments: Observations, requirements, and experiences," *Communications Magazine, IEEE* **48**, pp. 60 –69, October 2010.
3. N. Suri, G. Benincasa, M. Tortonesi, E. Casini, and A. Rossi, "Peer-to-peer cooperative networking for cellular mobile devices," in *Mobile Wireless Middleware, Operating Systems, and Applications*, N. Venkatasubramanian, V. Getov, and S. Steglich, eds., *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* **93**, pp. 85–97, Springer Berlin Heidelberg, 2012.
4. A. Vahdat and D. Becker, "Epidemic Routing for Partially-Connected Ad Hoc Networks," tech. rep., Duke University, 2000.
5. Z. J. Haas and T. Small, "A new networking model for biological applications of ad hoc sensor networks," *IEEE/ACM Trans. Netw.* **14**, pp. 27–40, February 2006.
6. K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.* **41**, pp. 8–13, October 2007.
7. R. Chandra, V. Ramasubramanian, and K. Birman, "Anonymous gossip: improving multicast reliability in mobile ad-hoc networks," in *Distributed Computing Systems, 2001. 21st International Conference on.*, pp. 275 –283, April 2001.
8. J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pp. 301 –310, October 2007.
9. A. Lee, I. Ra, and H. Kim, "Performance study of ad hoc routing protocols with gossip-based approach," in *Proceedings of the 2009 Spring Simulation Multiconference*, *SpringSim '09*, pp. 89:1–89:8, Society for Computer Simulation International, (San Diego, CA, USA), 2009.
10. Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," pp. 1707–1716, 2002.

11. J. Wieselthier, G. Nguyen, and A. Ephremides, “On the construction of energy-efficient broadcast and multicast trees in wireless networks,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, **2**, pp. 585 –594 vol.2, 2000.
12. Garrey, Michael R. and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 22 ed., 2000.
13. L. Zosin and S. Khuller, “On directed steiner trees,” in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, *SODA '02*, pp. 59–63, Society for Industrial and Applied Mathematics, (Philadelphia, PA, USA), 2002.
14. M. Charikar, C. Chekuri, T.-y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li, “Approximation algorithms for directed steiner problems,” in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, *SODA '98*, pp. 192–200, Society for Industrial and Applied Mathematics, (Philadelphia, PA, USA), 1998.
15. E. MING-IHsieh and M. TSAI, “Fasterdsp: A faster approximation algorithm for directed steiner tree problem,” *Journal of information science and engineering* **22**, pp. 1409–1425, 2006.
16. D. S. Lun, N. Ratnakar, R. Koetter, M. Médard, E. Ahmed, and H. Lee, “Achieving minimum-cost multicast: A decentralized approach based on network coding,” in *Proceedings of IEEE INFOCOM*, pp. 1607–1617, 2005.
17. R. B. Muhammad, “Distributed steiner tree algorithm and its application in ad hoc wireless networks,” in *Proceedings of The 2006 International Conference on Wireless Networks (ICWN'06)*, pp. 173–178, 2006.
18. A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*, Syngress Publishing, 2006.