A Partition-tolerant Manycast Algorithm for Disaster Area Networks

Mikael Asplund, Simin Nadjm-Tehrani
Department of Computer and Information Science, Linköping University
SE-581 83 Linköping, Sweden
{mikas,simin}@ida.liu.se

***Abstract*—Information dissemination in disaster scenarios requires timely and energy-efficient communication in intermittently connected networks. When the existing infrastructure is damaged or overloaded, we suggest the use of a manycast algorithm that runs over a wireless mobile ad hoc network, and overcomes partitions using a store-and-forward mechanism. This paper presents a random walk gossip protocol that uses an efficient data structure to keep track of already informed nodes with minimal signalling. Avoiding unnecessary transmissions also makes it less prone to overloads. Experimental evaluation shows higher delivery ratio, lower latency, and lower overhead compared to a recently published algorithm.**

I. Introduction

When a natural disaster strikes, the critical infrastructure that supports our society can be completely disabled for long periods of time. After an earthquake or flood, we cannot expect telecommunication services such as GSM or UMTS to function. Moreover, due to the interdependencies between different types of infrastructures [17], disruptions can also be caused by more commonly occurring incidents such as heavy storms.

Currently, many types of communication media are used by first-responder communities, ranging over specialised equipment, standard telecommunication devices and the Internet. While the specialised equipment (satellite, TETRA-based systems, JTRS, UTF radio, etc) may resist a setback during disasters, the most likely scenario is that GSM, 3G, etc will be severely overloaded, if not destroyed. Experience has shown that when large number of actors are involved in major disasters, the specialised equipment can beneficially be complemented with ad hoc communication over commodity devices. These networks can be set up quickly without central management, where no infrastructure exists, and allow communication at a very low cost. An experience report from the Katrina hurricane [20] demonstrates the usefulness of being able to set up spontaneous networks in disaster areas.

Disaster area networking brings together the two extremes of ad hoc networking challenges: pockets of intense activity which create locally overloaded areas, and collapse of infrastructure together with large geographical areas creates sparse networks with intermittent connectivity [16], [8]. In addition, lack of energy resources and inherent bandwidth restrictions in wireless communication create the need for resource-efficient algorithms. In this paper we address the need for an information dissemination algorithm with the following characteristics:

- Can deal with intermittent connectivity by partition tolerance.
- Assumes no knowledge about network topology or identity of actors.
- Is bandwidth- and energy-efficient in the sense of few transmissions per data packet.
- Has a reasonable average latency.

Note that mobility and potential for partitions implies occurrence of partial transmissions which corresponds to lost messages in connected networks.

Manycast algorithms [4] have the benefit of spreading information across a network without the high cost associated with broadcasts for reaching the last few in a dispersed network. We present a manycast algorithm which enables a packet to reach k nodes with the characteristics needed in a disaster area network. The algorithm keeps track of the already informed nodes with a hash structure in the message in a bandwidth-efficient manner. Thus, the protocol does not need to disseminate messages unnecessarily. Furthermore, no more transmissions take place when partitions are encountered. Finally, when there is an opportunity for a message to spread (e.g., a partition has healed due to node movement), the protocol will actively disseminate the message.

The contributions of this paper are as follows:

- a novel resource-efficient and partition-tolerant manycast algorithm that is random-walk and gossip-based (RWG)
- an experimental evaluation of the algorithm with
  - sparse and non-uniform mobility traces based on a large training manoeuvre from FIFA world cup in Germany [1]
  - careful reconstruction of a recently published partition-tolerant algorithm (Hypergossiping) as a baseline [12]
- preliminary model for optimising protocol parameters to reduce transmission energy

The rest of this paper is organised as follows: We proceed by presenting the related work in Section II. The algorithm

is described in Section III and optimisation of its main parameter is discussed in Section IV. Section V contains the experimental evaluation. Finally, Section VI concludes the paper.

II. Related Work

Research on broadcast schemes can be broadly divided into three categories, (1) reliable multicast where all-or-none semantics is required, (2) high-throughput multicast streams (e.g., video streaming) and (3) epidemic algorithms. Most of the research concerns wired networks and fixed topology. This allows keeping track of exactly which node has received which message, and to construct trees for efficient streaming. There is also an extensive amount of research on multicasts in MANETs taking into account varying degrees of mobility and disruptions.

With the disaster scenario in mind, we are mainly interested in intermittently connected mobile ad hoc networks which constitute some kind of worst-case scenario for message dissemination. Research work representing all of the three categories above are found also in IC-MANETS (e.g., [24], [23], [10], [21]). However, we further restrict our attention to protocols that do not assume a-priori knowledge about node mobility and contact patterns.

Such an assumption is reasonable for disaster scenarios, but can result in more expensive protocols. Karp et al. [11] show that there is a large cost associated with performing address-oblivious epidemics. That is, if the state of each node is only dependent on how many messages it has received, and not *from which* nodes it has received messages, then more messages have to be sent in order to guarantee high probability of delivery.

The work by Luo et al [13] uses a combination of push-based gossip and pull-based anti-entropy (with limited buffer sizes, so delivery is not guaranteed). They use a simple group membership protocol to provide nodes with node knowledge and assume the existence of a unicast routing protocol. The authors evaluate their protocols using both an analytical model and by simulation.

Khelil et al. [12] use a two-stage approach called hyper-gossipping to achieve efficient broadcasts in partitioned networks. A message is first broadcasted within the current partition (using gossip). When a node encounters a partition that has not heard the message, it is rebroadcasted in that partition. This protocol is the one we have found to be closest to ours in trying to stay silent in an isolated partition and thus conserving resources until uninformed nodes appear. Therefore, we have used this protocol as a baseline in our experimental evaluation.

Vollset and Ezhilchelvan [22] present a manycast algorithm called Scribble that is in some respects similar to ours. It is designed to be partition-tolerant and uses a node signature to keep track of informed nodes (several different signature types are suggested). However, their approach can be significantly more resource-demanding in a disconnected network. In Scribble, a subset of nodes (termed responsible) periodically send messages until the message is considered to be delivered by a sufficient number of nodes (or another node takes over that responsibility). In contrast, in our algorithm nodes keep silent until they discover an uninformed node, and only then send the message. Another difference is that our algorithm has a small fixed header size whereas the Scribble algorithm requires headers that grow with a number of bits (8 bits [22]) for every informed node.

Random walk is a basic technique for constructing randomised algorithms, and has been analysed and used in a large variety of areas, including membership services [5], [2], searching [6], [3], distributed computation [9], and routing [19]. Mian et al. [14] use the same low level mechanism for random walk to implement a unicast with no provision for partitions.

III. Random Walk Gossip Algorithm

In this section we describe the RWG manycast algorithm which consists of two elements: a random walk element (Section III-B) and an optimisation element to avoid unnecessary transmissions (Section III-C). The latter is used to avoid transmitting to already visited nodes without ending up in local dead ends.

We start by explaining the terminology. The goal of the algorithm is to deliver the message to k nodes, whereby the message becomes *k-delivered*. A node in the network can be in one of the following states, with regard to one particular message m:

- *Uninformed*, has not received m
- *Informed*, has received m, can be in one of the following disjoint subclasses:
  - *Active*, is the current custodian of m,
  - *Inactive*, silently holds m,
  - *Done*, knows that m is k-delivered.

When a node is active, it tries to forward the message to a randomly chosen node. Since the nodes cannot be expected to have an up to date view of their neighbours each such hop is preceded by a sequence of packet exchanges. First a "request forwarding" (REQF) packet is sent by the current custodian. The neighbour nodes will reply with an "acknowledge" (ACK) message. The custodian will then randomly choose one of these nodes and send an "OK to forward" (OKTF) message to that node. The receiver of the OKTF message then becomes active, and the sending node becomes inactive. The reason for inactive nodes to retain the message is that if an uninformed neighbour is discovered at some later point, then the inactive nodes will become active and thus initiate a new random walk. If an uninformed node hears a REQF message, but is not selected as the next forwarding node, it will go directly from being uninformed to being inactive. Finally, when k nodes have been informed the node becomes done. In this state the node does not need to retain the message any longer. Moreover, it sends a "be

Table I
RWG HEADER

0-7	8-15	16-23	24-31
packetLength		type	hops
groupSize		sequenceNumber	
origin			
target			
sender			
timeToLive			
informed			
...			

silent" (BS) message to all its neighbours so that they can also proceed to the done state. Thus, every node retains only copies of inactive messages (up to their expiry time).

A. Message Header

Every message sent by the RWG protocol contains a header as shown in Table I. The *packetLength* field indicates the total length of the packet. The packet *type* can be one of REQF, ACK, OKTF, and BS. The *hops* field is needed for energy optimisation and will be explained in Section III-C.

The *groupSize* field is used in order to know how many nodes the message should be delivered to (i.e., the parameter k in the protocol description). Each message is uniquely identified by the fields *sequenceNumber* and *origin*. All the address fields can be ordinary IP addresses or some other id that uniquely identifies each node. *target* is used when a specific node is intended to receive the message (used for OKTF messages), and *sender* is the node that sent out the packet, that is, the most recent custodian. The *timeToLive* indicates the time that the packet is allowed to remain in the network. This will be passed as a remaining time from one node to another, thus avoiding the need for time synchronisation.

The field *informed* is a bit-vector which is used to indicate which nodes have seen this message. The semantics of the vector is the following: if $hash(nodeId) = j$ where *nodeId* is informed about m, then $informed[j] = 1$, where hash() is a standard hash-function (e.g., modulo division). This field allows each message to keep track of informed nodes in a bandwidth-efficient manner, enabling recognition of uninformed neighbours and reactivation of inactive messages on a new encounter. The length of this vector must be at least k. We have fixed the length to 256 for a delivery with k ranging from 30 to 90. In Section IV we provide a formula for deciding a suitable length of the bit vector given k and the desired collision ratio.

B. The Random Walk Element

The random walk element of the algorithm description is divided in three parts: 1, 2 and 3. We begin by explaining a few general concepts. The algorithm is presented in an event-based manner by scheduling procedures to execute at certain times. This is denoted:

schedule <PROCEDURE> in $[T_1, T_2]$,

Table II
ALGORITHM PARAMETERS

Parameter	Symbol	Default value
Group size	k	30
Time to live	*TTL*	600 [s]
Minimum wakeup interval	T_{wmin}	4 [s]
Maximum wakeup interval	T_{wmax}	6 [s]
Max time before sending	T_s	0.01 [s]
Time to wait for acks	T_a	0.1 [s]
Limit on number of acks	L	3
Time to wait for forward	T_f	0.1 [s]
Length of the *informed* field	b	256 bits

meaning that <PROCEDURE> will be invoked at a time uniformly distributed between T_1 and T_2 seconds from the current time. It is also possible to cancel a scheduled process by the cancel keyword.

The parameters of the algorithm are summarised in Table II. Their meaning will be given during the description of the algorithm.

```
SEND(m):
  m.groupSize ← k
  m.origin ← nodeId
  m.seq ← unique sequence number
  m.timeToLive ← TTL
  SENDREQF(m)                          // originator forwarding m
RECEIVE(m):
  update m.informed
  WAKE(hash(m.sender))                 // wake inactive messages
                                       // not seen by this sender
  if m is not k-delivered:
    switch (m.type):
      case REQF:
        if new packet:
          DELIVER(m)
        schedule SENDACK(m) in [0, T_s]
        cancel SENDREQF(m)
      case ACK:
        store m.sender in receivedAcks[m]
      case OKTF:
        if m.target = nodeId:
          SENDREQF(m)                  // custodian forwarding m
        else:
          inactive ← inactive ∪ {m}
  else:                                // m is k-delivered
    cancel all procedures related to m
    inactive ← inactive \ {m}
    if (m.type ≠ BS):
      schedule SENDBS(m) in [0, T_s]
  cancel WAKEONEPACKET
  schedule WAKEONEPACKET in [T_wmin, T_wmax]
```

Figure 1. Random walk gossip, part 1

Now consider part 1 (Figure 1) showing the basic primitives SEND(m) and RECEIVE(m). The send procedure, which is invoked from the application layer, initialises the basic header fields and then calls the SENDREQF(m) procedure in order to spread the message to the nearest neighbours.

Now lets turn to what happens when a packet is received by a node (i.e., the RECEIVE procedure). First, the node

updates the *m.informed* field by flipping the hash(*nodeId*) cell to 1. Next, irrespective of what type of packet the node n receives, it could be an indication of a neighbour n' that some of n's inactive packets have not traversed (i.e., for some messages held by n the informed field for n' is zero). The WAKE procedure, explained later, will deal with activating those messages.

Consider the first case where the received packet is a request to forward (see line *case REQF:*). If it is the first time that this packet visits the node, then it is delivered to the application layer. Moreover, the node will reply to the sender by sending an ACK message. This is done by scheduling the SENDACK(m) procedure to run within T_s. If the node intended to forward this packet itself, then this activity (SENDREQF) is cancelled. Cancelling the forwarding prevents a flood of REQF messages every time an uninformed node comes into contact with a number of inactive nodes, which will otherwise all try to send a message to the uninformed node.

If the message received was an ACK message, then this is simply stored locally to be further processed. When an OKTF message is received the node checks whether it was the intended receiver (*m.target*) and if so, forwards the packet using the SENDREQF(m) procedure. If the node was not the intended receiver the packet is stored as an inactive packet in that node (until the TTL expires). If a message with the same message ID already exists in *inactive* it is overwritten. This way, every node who has received a packet can potentially start spreading it if it happens to encounter an uninformed node in the future. This causes the random walk to branch when there is an opportunity to spread to uninformed nodes, but not otherwise.

When the number of 1's in the *m.informed* field is k or higher, then the message has been successfully k-delivered. When this happens, it is just a waste of resources to continue spreading the message. Thus, when a node detects that a message has reached the threshold, a BS message is sent out to its neighbours informing them to stop all activities related to this message (using the SENDBS procedure).

The last two lines in RECEIVE reschedules the WAKEONEPACKET procedure which ensures that the network does not become permanently silent (thus preventing nodes from being able to detect uninformed neighbours). We will come back to this in part 3.

We now proceed to describe the primitive functions used above in the algorithm, described in Figure 2. First, we see the code for SENDREQF(m), which also takes care of expunging old messages. Before a message is forwarded the node makes sure that time-to-live for message m has not expired. The actual sending of the packet is done by invoking TRANSMIT(m), which represents the MAC-layer. The HANDLEACKS(m) procedure is scheduled to run T_a seconds later (when hopefully a number of acks have been received for this message).

After the T_a seconds have elapsed, the HANDLEACKS(m) procedure randomly chooses one of the acknowledging nodes and sends a "OK to forward" message to it. If no acknowledgements have arrived, the message becomes inactive. Note that the TRANSMITHEADER procedure does not transmit the payload, thereby reducing bandwidth and energy consumption.

Any node that intends to send an ACK message, by invoking SENDACK, will only do so if not more than L such messages have been sent by other nodes. This is to avoid a flood of acknowledgement messages in response to a REQF message.

The purpose of SENDBS has already been explained.

```
SENDREQF(m):
  decrease m.timeToLive with time spent in the node
  if m.timeToLive < 0
    cancel all procedures related to m
    inactive ← inactive \ {m}
  else
    m.type ← REQF
    m.sender ← nodeId
    TRANSMIT(m)
    schedule HANDLEACKS(m) in [T_a, T_a]

HANDLEACKS(m):
  if receivedAcks ≠ ∅
    randomly select t from receivedAcks[m]
    m.target ← t
    m.type ← OKTF
    TRANSMITHEADER(m)
  else
    inactive ← inactive ∪ {m}
  receivedAcks[m] ← ∅

SENDACK(m):
  if (|receivedAcks[m]| < L):        // default L = 3
    m.type ← ACK
    TRANSMITHEADER(m)
  receivedAcks[m] ← ∅

SENDBS(m):
  m.type ← BS
  TRANSMITHEADER(m)
```

Figure 2. Random walk gossip, part 2

We now proceed to describe how the WAKE procedure, invoked from the RECEIVE procedure, reactivates the inactive messages not seen by the sender of any received message (Figure 3). Every time a message m is received, the address of the sender is hashed and all the inactive packets m' at the receiving node are checked to see if the sender is uninformed of m'. Although this information is possibly stale in most cases, the sender of m has probably not seen m'. Hence, the SENDREQF(m') is scheduled to run. The elegance of this mechanism is that the sender node does not have to provide information about which packets it has seen, this information is already in each message.

Finally a procedure is needed for the special case where all nodes are silent, waiting for another node to activate their packets. This can in principle happen if the application does

```
WAKE(h):
  for all m' such that
          m'.informed(h) = 0 AND m' ∈ inactive:
    schedule SENDREQF(m')
    inactive ← inactive \ {m'}
WAKEONEPACKET:
  choose m_i so that:
    m_i ∈ inactive ∧ ∀m_j : | m_i.informed | ≤ |m_j.informed|
  schedule SENDREQF(m_i) in [0, T_s]
  inactive ← inactive \ {m_i}
  schedule WAKEONEPACKET in [T_wmin, T_wmax]
```

Figure 3. Random walk gossip, part 3

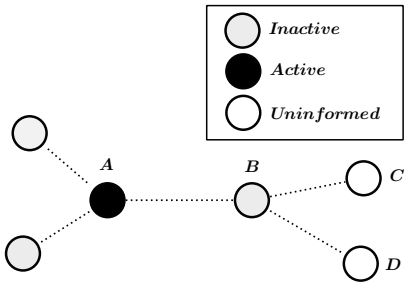

Figure 4. Motvation for resetting toAvoid

not generate new messages within the system for a while. To counteract, there is a procedure WAKEONEPACKET which is run at least every T_{wmin} (and at most every T_{wmax}), except when there is any other activity going on. When the procedure is run, it will activate the message with the least number of informed nodes. Just in case activation of m_i does not lead to any responses, the WAKEONEPACKET reschedules itself. This is harmless since the last two lines of part 1 ensures that such scheduled events are postponed.

C. Avoiding Unnecessary Transmissions

The random walk described so far would waste a lot of transmissions by visiting already informed nodes. The natural solution to this problem is to make the random walk avoid nodes that are already informed via the never-go-back policy. In principle, we could have used the *informed* data structure to make the random walk avoid already informed nodes. However, at some point the walk may end up in local dead ends due to all neighbouring nodes being informed. Our mechanism to deal with such cases is to introduce regular rejuvenations. Since the *informed* vector cannot be reset, we introduce a new vector for this purpose by adding a field *toAvoid* to the message header. This data structure is in principle copy of *informed*, except that it is now and then rejuvenated to contain only zeroes.

To illustrate the need for optimisation consider the situation in Figure 4. Node A is actively trying to forward a message, but all its neighbours are already informed. Thus, with the never-go-back policy, no acks will be received by A and the message will become inactive. However, just two hops away, there are uninformed nodes (C and D), which will not receive the message until B discovers that they are there and activates the message. This may take a significant amount of time, thereby causing high latency and possible low success ratio due to TTL timeout.

Rejuvenation allows the message to break the barrier of informed nodes and reach new (uninformed) nodes. We thereby find a middle way between a completely undirected random walk that wastes resources and a never-go-back random walk with high latency.

This results in a small change to the algorithm description in part 1 as follows. The first line in RECEIVE (Figure 1) is changed so that when *m.informed* is updated, so is *m.toAvoid*. In addition, in the same procedure, the line:

```
schedule SENDACK(m) in [0, T_s]
```

is changed to:

```
if m.toAvoid[hash(nodeId)] = 0:
  schedule SENDACK(m) in [0, T_s]
```

The rejuvenation is triggered by counting the number of hops since the last rejuvenation. Thus, the message header also needs to include a *hops* field (see Table I). If the number of hops is larger than the parameter H, then the *toAvoid* field and the hop count is reset. So the procedure HANDLEACKS is changed to:

```
HANDLEACKS(m):
  randomly select t from receivedAcks[m]
  m.target ← t
  m.type ← OKTF
  if m.hops > H:
    m.toAvoid ← [0, ..., 0]
    m.hops ← 0
  TRANSMIT(m)
  receivedAcks[m] ← ∅
```

In Figure 4, if B is not in the *toAvoid* vector of the message from A it is a candidate for becoming active. In our simulation studies we have found the value $H = 10$ to provide the best performance. Thus, in the simulations, every 10 hops, the message is rejuvenated. Since the *toAvoid* vector is rejuvenated after H hops it is a sparse vector[1] that can be compacted in the implementation, saving bandwidth.

IV. BIT VECTOR LENGTH

There is a non-zero probability that the algorithm will not terminate before the message TTL even if k nodes have been informed by the message. The reason for this is that even if $r > k$ nodes have been informed by a given message, there is a non-zero probability that only $s < k$ indices have been marked in the *informed* vector. Since hashing is used to determine the index of a given node, collisions of the

[1]Every hop will increase the number of 1's in the *toAvoid* vector by at most L, so the vector will never contain more than $L \cdot H$ 1's.

hash function will lead to s being less than r even for small r^2. In our case , two clashing nodes B and C who ACK a message sent by A will increase A's count only by 1. Thus, A might continue to disseminate unnecessarily. A second problem arises when A is inactive and B earlier informed. An arriving node C which clashes with B will not even be informed (A will not wake up).

We proceed to quantify the risk of the algorithm continuing to send despite k successful receipts. This means that we want to find the probability that for a given size of the vector b, a number of informed nodes $r \leq k$, the number of marked indices s is less than the desired k.

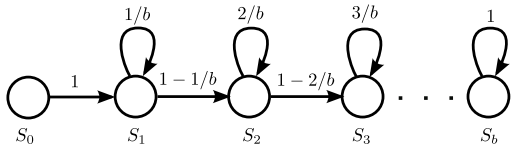

Figure 5. Markov chain representing the number of marked nodes

The result of the hash function can be modelled as an integer random variable evenly distributed in the range $[1, b]$. Thus when s nodes have been marked and a new node i is discovered, then the probability that the index of i is already marked is s/b. Based on this we can set up a Markov chain as in Figure 5 where the state S_s means that s indices have been marked. The probability of being in state S_s after r steps (starting in S_0) is then identical to the probability of s indices being marked after r nodes have been informed.

We can express this in matrix notation with the transition matrix P of size $(b+1)\times(b+1)$ where $P_{i,j}$ is the probability of a transition from state S_i to state S_j. We have that $P_{i,i} = i/b$ for all $i \in \{0, \ldots, b\}$, $P_{i,i+1} = 1 - i/b$ for all $i \neq b$, and $P_{i,j} = 0$ for all other i, j. The transition matrix can be multiplied with itself so that $P^r_{i,j}$ represents the probability of being in state S_j after r steps starting from state S_i. Now we can express the desired probability:

$$Prob(s < k) = \sum_{i=0}^{k-1} P^r_{0,i} \tag{1}$$

Clearly the size of the vector b has a huge impact on this probability. A large b gives a low probability of collision. Unfortunately this means larger message headers resulting in unwanted overhead. Obviously, it would be useful to be able to derive a reasonable value for b. To do this we need to go to the average case since the above equation cannot easily be transformed to solve for b.

Let the function $y(r)$ be the expected number of marked indices when r nodes have been informed (i.e., $y(r) = \sum_i i \cdot P^r_{0,i}$). Since the expected probability of marking a new index is $1 - y(r)/b$ we can express the rate (derivative) of y accordingly:

$$y'(r) = 1 - \frac{y(r)}{b}$$

Solving this differential equation with the boundary condition $y(0) = 0$ gives:

$$y = b(1 - e^{r/b})$$

We now introduce a constant $c = r/y$, which corresponds to the proportion of extra nodes the message needs to visit. Thus, if $c = 1.05$, then (on average) a message will need to visit $1.05k$ nodes before terminating. Due to space limitations, we cannot provide the entire derivation, but given this substitution, and expanding using the McLaurin series and finally approximating, we get:

$$b \approx \frac{2c^2 y}{3c - \sqrt{24c - 15c^2}} \tag{2}$$

Given the redundancy factor c and the expected number of marked indices $y = k$ (the desired level), we can derive the size of the vector b. For example, for $k = 100$ and $c = 1.2$ the bit vector should be 314 bits long. Finally, using this setting for b we can use equation (1) to calculate the probability of a message not terminating after having informed $r = 150$ nodes to be $Prob(s < 100) = 4.1 \cdot 10^{-7}$

This approximation assumes that $r < b$ which means that the constant c should be fairly small (i.e. in the range of 1.0 to 1.5). Note that $c = 1.5$ corresponds to an expectancy of 50% more nodes than desired being informed, not counting the effects of branching.

V. Experimental Evaluation

In order to evaluate the protocol we have performed simulations using the Network Simulator 3 (ns-3) [7] which is a successor of the popular ns-2 simulator. The baseline which we compare our protocol against is the Hypergossiping protocol by Khelil et al.[12]. We believe this to be the protocol published for the kind of scenarios we are considering. However, it is designed as a broadcast protocol (deliver to all) rather than a manycast (deliver to some), so the comparison should be interpreted with this in mind. We have used the original ns-2 implementation of Hypergossiping and ported it to ns-3. We verified the correctness of the port by reproducing a number of results from [12]. In order to speed up the simulations we changed (to the better) the hello interval from 1s to 10s (average). See Section V-E for a more detailed discussion on this.

We have used the 802.11 protocol, with 6 Mb/s data rate. The normal packet size (including header) was 100 byte for RWG and 60 byte for Hypergossiping. The other simulation parameters are listed in Table III (except those already shown in Table II) . Some of the parameters are varied in the experiments below (clearly indicated). Every data point was averaged over 10 runs (95% confidence intervals are indicated in each graph).

[2]The famous birthday paradox tells us that if the number of indices is 365 and $r = 23$ then there is a 50% probability that two nodes will have the same index.

A. Scenario and Mobility Model

In recent years there have been a number of reports (e.g., [15], [1]) on the drawbacks of using random waypoint (RWP) as the mobility model for evaluating the performance of protocols for ad hoc networks. One of the main problems related to evaluation of partition-tolerant protocols is the fact that RWP tends to create networks with very good connectivity properties in the sense that nodes tends to concentrate in the center of the simulation area. Moreover, even if the network is sparse, all nodes move over the entire area making it very probable that a node will meet a number of uninformed nodes in a short time.

While access to real disaster mobility models is practically impossible, we have used synthetic models that are sparse and non-uniform, based on traces from Aschenbruck et al.[1]. Their mobility generator is based on the analysis of a large training manoeuvre in preparation of the FIFA world cup in Germany. We have used 100 nodes out of the 150 described in [1] (thus excluding ambulances and one of the clearing stations). However, at the time of this work the mobility model was not publically available so we had to resort to prepared traces.

Since we used a given scenario for mobility patterns there are only two ways to modify the density of the network, changing the number of nodes, or changing the radio range. When using the default radio model in ns-3 together with 802.11a running at 6 Mb/s data rate, the resulting network is very well-connected. Since both the RWG and the Hypergossiping protocols are intended for very disconnected networks we needed to reduce the density of the network considerably, but we did not want to drastically reduce the number of nodes (and thereby making the simulations uninteresting). Therefore we modified the system loss (we use the term signal loss since it is more intuitive) parameter in ns-3 which controls how much the radio signals are dampened. The default value in ns-3 is 1 (no dampening). In the simulations we used the value 500 corresponding to a radio range of approximately 17 meters. The speed of the units varied in the range 1-2 m/s (corresponding to walking speed), moving in an area of 200mx350m. This results in a coverage factor of 1.3 which is an indicator of sparseness. In addition we know that the traces create partitions due to non-uniform distribution over the area.

The load was generated with the following procedure: at a given interval (e.g. every 0.1 seconds when the load is 10 packets/second), send a message originating from a randomly chosen node. Thus, all 100 nodes are potential senders, and every node will send a message on average every 10 seconds if the load is 10 packets/second. The simulation time for each run was 500 seconds (i.e. just over 8 minutes). Table III summarises the simulation parameters.

We will discuss the performance of the protocol by considering three different performance measures: success ratio (akin to delivery ratio), latency and overhead. There is an inherent tradeoff between these three metrics in intermittently connected networks. Most protocols aim for the first two metrics, whereas our objective was to achieve acceptable delivery properties with low energy consumption.

Table III
DEFAULT SIMULATION PARAMETERS

Parameter	Value
Number of nodes	100
System load	10 packets/s
Simulation time	500 [s]
Data rate	6 Mb/s
Signal loss	500 (radio range $\approx$ 17m)
Time to live	600
Number of runs	10

B. Success Ratio

The success of a manycast protocol is not based on the number of nodes that have been informed by a message, but that a minimum number have been informed. Therefore, we define the success ratio as the proportion of packets that reach at least k nodes (i.e., 30 nodes unless otherwise stated) during the simulation.

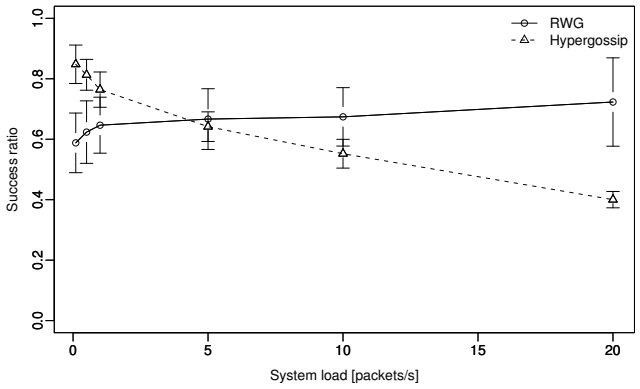

Figure 6. Success ratio vs. System load

Figure 6 shows the delivery ratio as a function of system load for the two compared protocols. We see that RWG performs much better for loads higher than 5 packets/s. Whereas for loads less than 1 packet/s Hypergossiping is better. We believe that one of the main explanations for this fact is that RWG can discard messages that have informed a certain number of nodes, whereas Hypergossiping continues to spread messages until all nodes have received it, resulting in an overloaded network. Moreover, the message is not discarded until the time-to-live has expired. The increasing performance of RWG can be explained by the fact that an increased activity in the network will lead to more inactive packets being woken up by the presence of uninformed nodes (see part 3 of the description), thereby increasing the chance of delivery for those packets.

Note that this result partially contradicts the one provided by Khelil et al. [12] in their performance analysis of Hypergossiping. In their setting, load had no negative

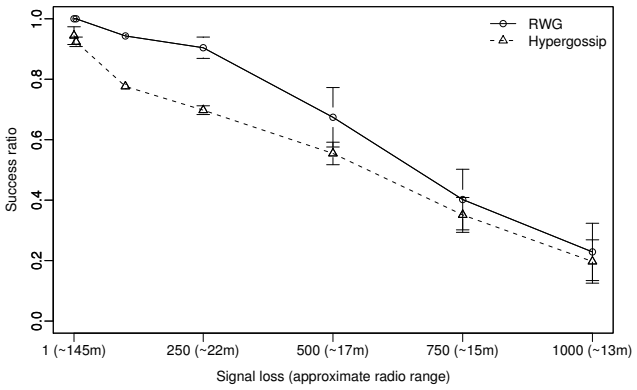

Figure 7. Success ratio vs. signal loss (reduced density)

effect on the delivery ratio. We were able to repeat this experiment as well, and could conclude that the different results are due to the fact that we count the delivery for all packets released during the entire simulation (i.e. 500 seconds), whereas in [12] only packets sent during the first 50 seconds were counted. Since the full overload effect does not occur until after 200-300 seconds, their results were subject to boundary effects.

In Figure 7 the delivery ratio is shown as a function of signal loss. The result is fairly intuitive, both algorithms suffer as the dampening increases. A similar graph is achieved when varying the group size k (not shown here): the higher the parameter k, the lower the success ratio.

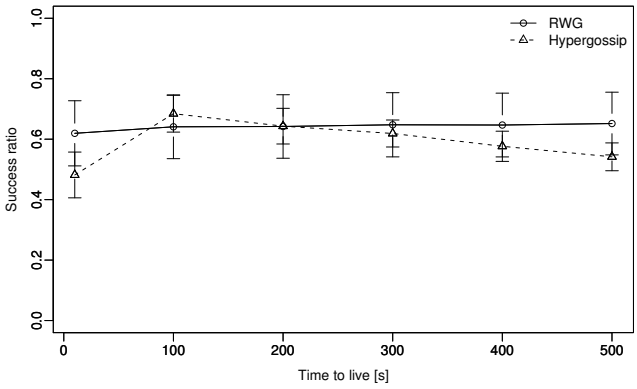

Figure 8. Success ratio vs. Time to live

In the above simulations, the time-to-live (TTL) has been longer than the simulation time (thus emulating an infinite TTL). In Figure 8, this was varied to see the effects of a shorter TTL. The positive effects of a shorter TTL is to reduce the effects of the system load. Since RWG did not suffer any overload issues for the default load, a shorter TTL only had a negative impact on delivery. For Hypergossiping, on the other hand, there seems to be an optimal TTL of approximately 100 seconds.

C. Latency

In an intermittently connected network, latency is measured on a very different scale compared to normal systems. Still, latency can be very important even in such networks. Although RWG is designed to have reasonable latency properties, we were surprised that it performed so well in simulations.

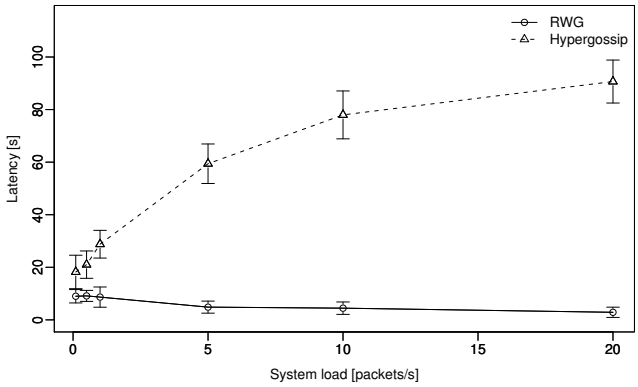

Figure 9. Average latency vs. System load

Figure 9 shows the average latency versus load reveals surprisingly that the latency of RWG is constantly decreasing with higher load. For RWG this can be explained by the fact that higher load means more activity. This will result in inactive packets waiting to hear from an uninformed node will be woken up much sooner compared to when there is little activity. For Hypergossiping, the overload inherently produced by itself causes some messages to be delivered very late, thus pulling up the average latency.

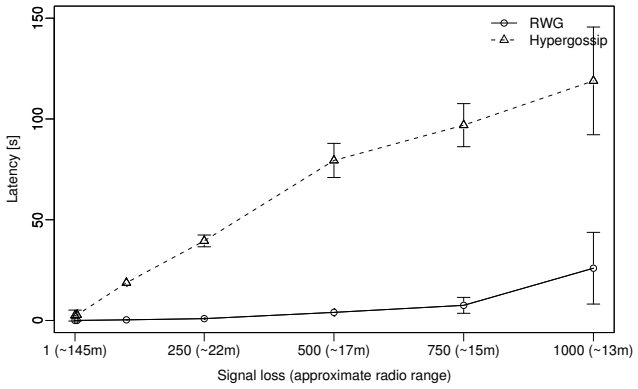

Figure 10. Average latency vs. signal loss (reduced density)

The latency shown in Figure 10 is plotted against the signal loss. Again both algorithms suffer. However, RWG manages to keep the latency fairly low until the radio range is down to 15m whereas Hypergossiping starts increasing rapidly from start. However, in a well-connected network with lower load, Hypergossiping outperforms RWG with several orders of magnitude (not shown here). This is due to the fact that Hypergossiping works as a standard localised

gossip algorithm in such cases whereas RWG uses the comparatively much slower random walk.

D. Overhead

To quantify the overhead, we measure the total number of transmissions sent out by the MAC-layer divided with the total number of data packets generated by the application layer. In our simulations, the packet payload is very small. Thus, due to the energy overhead associated with sending one packet irrespective of its size [18], we believe that this is a better indication of energy-efficiency than to measure the total number of bytes sent.

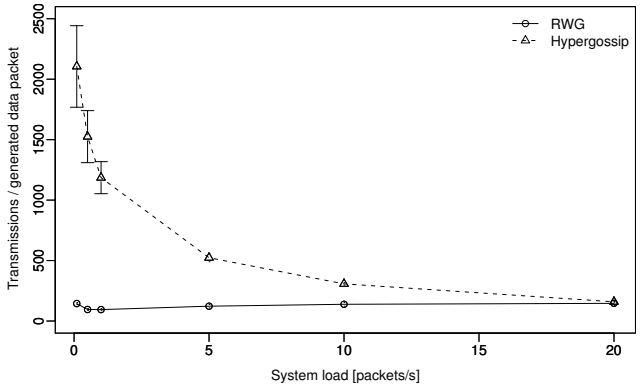

Figure 11. Transmissions/packet vs. System load

When considering the overhead as a function of load we see that RWG is hardly affected. This makes sense, since each packet behaves more or less independently from all other packets. Hypergossiping on the other hand is based on periodic beacons causing nodes to exchange information on their respective messages. If the number of normal packets increases, then a smaller portion of the total number of messages will be meta-information, resulting in lower overhead. However, note that around this point the success ratio and latency are prohibitive making this area less interesting.

As Figure 12 shows, for Hypergossiping, the shorter TTL, the lower overhead. RWG overhead is, on the other hand, not sensitive to the choice of TTL.

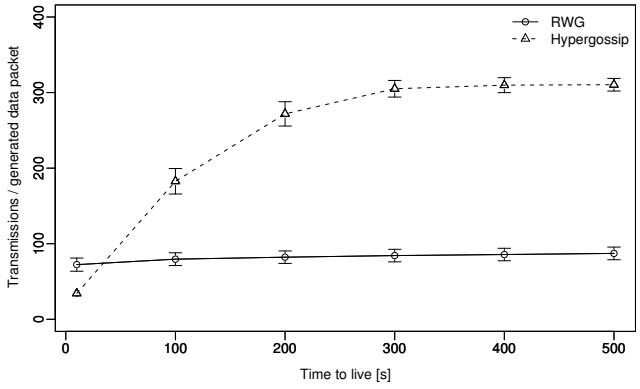

Figure 12. Transmissions/packet vs. Time to live

The final parameter we have investigated is the group size. This parameter serves two purposes, first of all, it is the basis of the definition of delivery ratio and latency used in the evaluation (i.e. a message is deemed to be delivered when it has informed at least "group size" nodes). Secondly it is a parameter to the RWG algorithm which uses this number to decide when a message is k-delivered and thus stops spreading it.

We have already alluded to the fact that RWG has an advantage in the fact that it can stop spreading messages when this condition is satisfied. From this we would expect that as the group size increases the advantage is lost and the performance is more like Hypergossiping. This is corroborated in Figure 13. Since Hypergossiping does not use the group size parameter, its overhead remains constant independently of the group size. For RWG, on the other hand, increasing the group size will mean that every message will need to visit more nodes before it can be discarded, this will increase the overhead as the group size increases. Still it is 70%-20% below Hypergossiping for group sizes above 20.

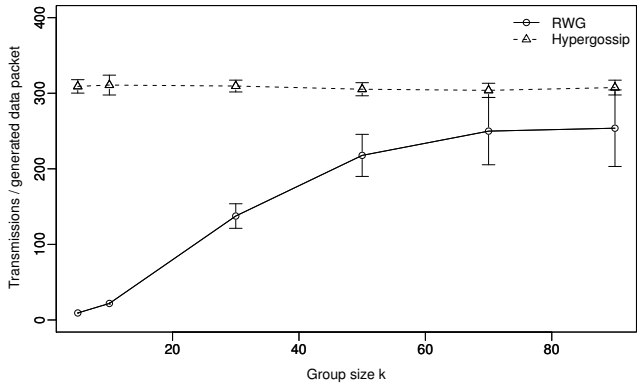

Figure 13. Transmissions/packet vs. group size k

E. Optimising Hypergossiping

We end the experimental section by returning to the rationale for changing the default hello-interval for Hypergossiping. This interval will affect all three metrics, delivery-ratio, latency, and overhead. Intuitively, shorter intervals should result in lower latency, higher overhead, and higher success ratio. However, for high loads, it is not that clear. As can be seen in Figure 14, longer hello intervals can be beneficial also for latency and delivery. Based on these results we concluded that 10s was a reasonable tradeoff between overhead and latency (without sacrificing delivery ratio). Unfortunately, this graph lacks confidence intervals (data is still based on an average over 10 runs).

VI. Conclusions and Future Work

We have presented RWG, a manycast protocol designed for intermittently connected networks with scarce energy

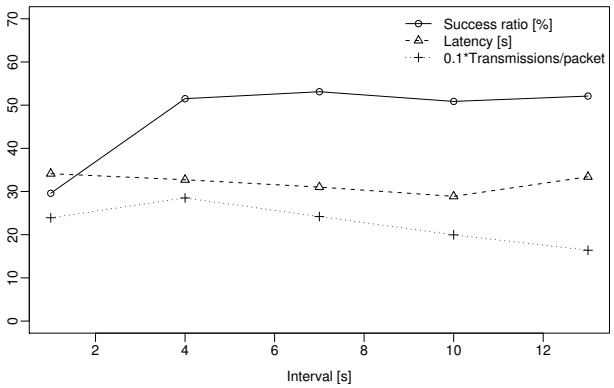

Figure 14. Varying hello interval for Hypergossiping

resources. Such a protocol could be useful in a disaster scenario, where the infrastructure is destroyed or disabled requiring robust and efficient communication strategies. The experimental evaluation has shown that RWG outperforms one of the most relevant existing solutions considerably; e.g. decreasing overhead by an order of magnitude in some situations. One of the main strengths of the protocol is being able to handle much higher loads. Moreover, by being able to keep track of informed nodes in a space-efficient way, the protocol can reduce the number of redundant transmissions.

Based on the encouraging results we believe that the RWG protocol is a good choice for disseminating data in disconnected wireless networks with energy restrictions (e.g., disaster response networks) when the message is intended to be received by a subset of the nodes. Quorum systems for data consistency is an example of an application that could build upon such a dissemination service.

As future work, we plan to further mathematically analyse the protocol to see if it is possible to derive bounds on latency, given certain mobility and network properties.

ACKNOWLEDGEMENT

This work was supported by the Swedish Civil Contingencies Agency (MSB) and the Swedish Research Council (VR). Thanks are due to the BonnMotion group for mobility traces and A. Khelil for sharing the code of the Hypergossiping protocol.

REFERENCES

[1] N. Aschenbruck, E. Gerhards-Padilla, M. Gerharz, M. Frank, and P. Martini. Modelling mobility in disaster area scenarios. In *Proc. 10th ACM Symposium on Modeling, analysis, and simulation of wireless and mobile systems (MSWiM'07)*, pages 4–12, 2007. ACM.

[2] Z. Bar-Yossef, R. Friedman, and G. Kliot. Rawms - random walk based lightweight membership service for wireless ad hoc networks. *ACM Trans. Comput. Syst.*, 26(2):1–66, 2008.

[3] R. Beraldi. Random walk with long jumps for wireless ad hoc networks. *Ad Hoc Networks*, 7(2):294 – 306, 2009.

[4] C. Carter, S. Yi, P. Ratanchandani, and R. Kravets. Manycast: exploring the space between anycast and multicast in ad hoc networks. In *Proc. 9th annual international conference on Mobile computing and networking (MobiCom)*, pages 273–285, 2003. ACM.

[5] S. Dolev, E. Schiller, and J. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mobile Comput.*, 5(7):893–905, July 2006.

[6] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Performance Evaluation*, 63(3):241–263, 2006.

[7] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13, 2006. ACM.

[8] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *Proc. 5th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom'99)*, pages 195–206, 1999. ACM.

[9] K. Jung and D. Shah. Fast gossip via non-reversible random walk. In *Proc. Punta del Este Information Theory Workshop (ITW)*, pages 67–71. IEEE, 13–17 Mar. 2006.

[10] G. Karlsson, V. Lenders, and M. May. Delay-tolerant broadcasting. In *Proc. 2006 SIGCOMM workshop on Challenged networks (CHANTS)*, pages 197–204, 2006. ACM.

[11] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 565–574, Nov. 2000.

[12] A. Khelil, P. J. Marrón, C. Becker, and K. Rothermelns. Hypergossiping: A generalized broadcast strategy for mobile ad hoc networks. *Ad Hoc Netw.*, 5(5):531–546, 2007.

[13] J. Luo, P. Eugster, and J.-P. Hubaux. Route driven gossip: probabilistic reliable multicast in ad hoc networks. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 2229–2239. IEEE, March-April 2003.

[14] A. Mian, R. Beraldi, and R. Baldoni. A robust and energy efficient protocol for random walk in ad hoc networks with ieee 802.11. In *Proc. IEEE International Symposium on Parallel and Distributed Processing IPDPS 2008*, pages 1–8, 14–18 April 2008.

[15] W. Navidi and T. Camp. Stationary distributions for the random waypoint mobility model. *IEEE Trans. Mobile Comput.*, 3(1):99–108, 2004.

[16] S. C. Nelson, I. Albert F. Harris, and R. Kravets. Event-driven, role-based mobility in disaster recovery networks. In *Proc. second workshop on Challenged networks (CHANTS)*, pages 27–34, 2007. ACM.

[17] S. Rinaldi, J. Peerenboom, and T. Kelly. Identifying, understanding, and analyzing critical infrastructure interdependencies. *IEEE Control Syst. Mag.*, 21(6):11–25, Dec. 2001.

[18] Y. Sankarasubramaniam, I. Akyildiz, and S. McLaughlin. Energy efficiency based packet size optimization in wireless sensor networks. In *Proc. First IEEE Sensor Network Protocols and Applications 2003 IEEE International Workshop on*, pages 1–8, 2003.

[19] S. D. Servetto and G. Barrenechea. Constrained random walks on random graphs: routing algorithms for large scale wireless sensor networks. In *Proc. 1st ACM international workshop on Wireless sensor networks and applications (WSNA'02)*, pages 12–21, 2002. ACM.

[20] B. Steckler, B. L. Bradford, and S. Urrea. Hastily formed networks for complex humanitarian disasters. http://www.hfncenter.org/cms/KatrinaAAR, Sept. 2005.

[21] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical report, Duke University, 2000.

[22] E. Vollset and P. Ezhilchelvan. Design and performance study of crash-tolerant protocols for broadcasting and reaching consensus in manets. In *Proc. 24th IEEE Symposium on Reliable Distributed Systems SRDS 2005*, pages 166–175, 2005.

[23] Q. Ye, L. Cheng, M. C. Chuah, and B. Davison. Os-multicast: On-demand situation-aware multicasting in disruption tolerant networks. In *Proc. VTC 2006-Spring Vehicular Technology Conference IEEE 63rd*, volume 1, pages 96–100, May 2006.

[24] W. Zhao, M. Ammar, and E. Zegura. Multicasting in delay tolerant networks: semantic models and routing algorithms. In *Proc. 2005 ACM SIGCOMM workshop on Delay-tolerant networking (WDTN)*, pages 268–275, 2005. ACM.