Depth first forwarding for low power and lossy networks: Application and extension

Jiazi Yi, Thomas Heide Clausen, Ulrich Herberg

Depth First Forwarding for Low Power and Lossy Networks: Application and Extension

Jiazi Yi, Thomas Clausen
Laboratoire d'Informatique (LIX) – Ecole Polytechnique, France
Jiazi@JiaziYi.com, Thomas@ThomasClausen.org

Ulrich Herberg
Fujitsu Laboratories of America
ulrich@herberg.name

Abstract—**Data delivery across a multi-hop low-power and lossy networks (LLNs) is a challenging task: devices participating in such a network have strictly limited computational power and storage, and the communication channels are of low capacity, time-varying and with high loss rates. Consequently, routing protocols finding paths through such a network must be frugal in their control traffic and state requirements, as well as in algorithmic complexity – and even once paths have been found, these may be usable only intermittently, or for a very short time due to changes on the channel. Routing protocols exist for such networks, balancing reactivity to topology and channel variation with frugality in resource requirements. Complementary component to routing protocols for such LLNs exist, intended not to manage global topology, but to react rapidly to local data delivery failures and (attempt to) successfully deliver data while giving a routing protocol time to recover globally from such a failure. Specifically, this paper studies the "Depth-First Forwarding (DFF) in Unreliable Networks" protocol, standardised within the IETF in June 2013. Moreover, this paper proposes optimisations to that protocol, denoted DFF++, for improved performance and reactivity whilst remaining fully interoperable with DFF as standardised, and incurring neither additional data sets nor protocol signals to be generated.**

I. INTRODUCTION

Low-power and Lossy Networks (LLNs) are often deployed with constrained devices, under challenging and unreliable environments. Typical communication and data processing modules are equipped with CPU in several MHz processing power, and tens of KBytes of memory. In addition, the communication medium connecting all those devices is "lossy": the communication channels are of low capacity, time-varying and with high loss rates.

Transiting data across such a network, especially when multiple hops are present between the source and the destination, is a challenging task: routing protocols finding paths must be frugal in their control traffic and state requirements, as well as in algorithmic complexity. To discover appropriate paths in such networks, protocols such as Light-weight On-demand Ad hoc Distance-vector Routing Protocol – Next Generation (LOADng) [1] and IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) [2] have been developed.

Those routing protocols for LLNs are designed to limit the routing overhead imposed to networks as much as possible, and to be adapted to the varying nature of communication medium. However, even once paths were found, those paths might be unusable from time to time due to different reasons: presence of noise or interferences, low power supply in certain

devices, uni-directional links, etc. From a routing protocol point of view, when such link failure is detected, it needs some extra signalling and/or time to recover and discover new, valid routes. During this recovery phase, data packets being sent over the broken link, must either be buffered and wait for the route recovery, or be dropped because of lack of memory in constrained devices.

To alleviate the effects of inevitable random link failures in LLNs, a set of data forwarding mechanisms have been proposed [3]. Those mechanisms that work in the "forwarding plane" use data packet to detect loops, update routing tables, and reroute data packet through alternative paths when the primary routes are broken. By doing so, the packets that are originally forwarded through failed links can be recovered, instead of being dropped.

"Depth-First Forwarding in Unreliable Networks" (DFF) [4] is an experimental data forwarding standard which proposes a mechanism for rapid and localised recovery in case of link failure. Colloquially speaking, if a device fails in its attempt to forward a packet to its intended next-hop, then DFF suggests a heuristics for "trying another of that devices' neighbours", while keeping track of (and preventing) packet loops. While DFF can operate independently, *i.e.*, without a routing protocol (which amounts to simply doing a depth-first exploration of the network), it can also be used conjointly with a routing protocol: the routing protocol can provide an "order of priority" of the neighbours of a device, in which data delivery should be attempted – and DFF can also signal to a routing protocol when data delivery to a destination has (possibly repeatedly) failed via a neighbour but (possibly repeatedly) succeeded via another neighbour.

A. Statement of Purpose

This paper explores the application of DFF in LLNs. One key issue of DFF – selecting appropriate next-hop candidates for forwarding data packets, is specially discussed. Based on experiments and observations, an optimisation to the DFF standard [4], denoted DFF++, is proposed. It offers a procedure to choose next hops based on previously processed packets, and reduces unnecessary explorations in the depth-first search. The extension is of very low impact to DFF protocol: it does not introduce additional protocol signalling and data sets, while remaining completely interoperable with DFF as standardised in [4].

B. Paper Outline

The remainder of this paper is organised as follows. Section II provides a brief overview of DFF, as defined in [4], and section III studies, by way of an example, one of the cardinal points of the performance of DFF: the ordering of the elements of Candidate Next Hop List. Section III also identifies a set of inconveniences in a "naive" (but, perfectly valid, according to [4]) ordering. [4] stipulates some basic constraints on how the Candidate Next Hop List is to be ordered, but otherwise leaves the exact approach and order unspecified. This paper, therefore, proceeds by proposing a simple and no-overhead optimisation to DFF, denoted DFF++, in section IV. This optimisation remains fully interoperable with DFF. Both DFF and DFF++ are evaluated in section V. As DFF, and as a consequence DFF++, are able to operate both independently and conjointly with a unicast routing protocol, this evaluation will study the performance of DFF/DFF++ alone and when used with LOADng [5], [1]. Finally, this paper is concluded in section VI.

II. DEPTH FIRST FORWARDING

DFF [4] is a forwarding mechanism for improving the data delivery success ratio across unreliable multi-hop networks. It operates solely on the forwarding plane, *i.e.*, does not assume any specific routing protocol to be in operation (or, indeed, that any routing protocol is in operation) – but can, as appropriate and as indicated in section I of this paper , interact with a routing protocol. DFF relies on an external mechanism providing each router with a list of its neighbours.

In order to support the loop detection and duplicate detection, each router running DFF maintains a Processed Set, which lists sequence numbers of previously received packets, as well as a list of Next Hops to which the packet has been sent successively as part of the depth-first forwarding mechanism.

Schematically, the basic operation of DFF is as follows, when a data packet for a destination arrives at the forwarding plane of a router:

1) The router temporarily creates an ordered Candidate Next Hop list for that packet, which does not contain the neighbour from which the data packet was received (if any), from among the neighbours in the router's neighbour list.

2) The router attempts to forward the data packet to the first neighbour in the resulting Candidate Next Hop list.

3) There are five possible outcomes from this attempt:

   a) The Candidate Next Hop list is empty, in which case the data packet is returned to the neighbour from which it was initially received, and the process for this router stops.

   b) Delivery to that neighbour succeeds (*e.g.,* as confirmed by an L2 acknowledgement), and that neighbour is the destination for the data packet. The L2 acknowledgement indicates successful data packet delivery to the destination. The process for this router stops.

   c) Delivery to that neighbour fails (*e.g.,* detected by lack of an L2 acknowledgement), in which case that neighbour is removed from the Candidate Next Hop list, and the process resumes at step 2 above.

   d) Delivery to that neighbour succeeds (*e.g.,* as confirmed by an L2 acknowledgement), but the data packet is returned from the neighbour as "undeliverable", in which case that neighbour is removed from the Candidate Next Hop list, and the process resumes at step 2 above, with the resulting Candidate Next Hop list.

   e) Delivery to that neighbour succeeds (*e.g.,* as confirmed by an L2 acknowledgement), the neighbour is not the destination for the data packet. That neighbour will, now, execute this very same procedure (create its own Candidate Next Hop list, and execute this process starting at step 1).

The initial Candidate Next Hop list for a data packet, by default, contains all the neighbours of a router, except for the neighbour from which the data packet was received, but may be smaller. The list is ordered, section 11 in [4] suggests several criteria to take into account when ordering that list, including that if a routing protocol is in operation, then the neighbour on the shortest path (as indicated by that routing protocol) must be part of the initial Candidate Next Hop list – and is recommended to be first in that initial Candidate Next Hop List. Link quality, historical information on "good and bad neighbours as next hop" is suggested to be used for ordering remaining neighbours.

DFF contains mechanisms for detecting looping data packets, encoded as flags and sequence numbers in IPv6 Hop-by-Hop header options, carried in each data packet, and specifies processing here. This incurs a small, but fixed, per-data-packet overhead of 8 octets. This paper does not discuss this signalling and processing in further details.

III. ORDERING THE CANDIDATE NEXT HOP LIST

Section II, has introduced the basic operations of DFF, indicating that a key operational parameter is the ordering of elements in the Candidate Next Hop List for a data packet. To elaborate on this parameter, this section will consider the example in figure 1, where device *A* sends a data packet to device *D*, and which arrives at *B* – the sole neighbour of *A*.

A. Ideal Ordering and Default Ordering

The ideal ordering of the elements in the Candidate Next Hop List for that data packet in *B* would list *F* and *G* first, followed by *E*. The ideal list would not include *C* at all. Absent topological information (maintained by some external process), however, *B* would by default populate the initial Candidate Next Hop List with all of its bi-directional neighbours, except for *A*, would not be able to determine that *E* should be listed after *F* and *G*, nor that *C* should be excluded from the list. This, would lead to a "blind" search for all the DFF forwarded data packets. Indeed, in this example

constructing the Candidate Next Hop List lexicographically – $\{C, E, F, G\}$ – would lead to the worst possible search order.

B. Ordering with Unicast Routing Protocol

If a unicast routing protocol is in place, as suggested by section 11 in [4], that routing protocol would provide *B* with information that *F* (or *G*) is the next hop identified on the shortest path to *D*, and therefore allow *B* to ensure that the first element in the Candidate Next Hop List for a data packet for *D* would be *F*. Unless if the routing protocol provides multiple paths or a complete topology in each device – unlikely, given that the application space is constrained devices with limited memory – the remainder of the list would have to be constructed without any additional guidance from the routing protocol as to a specific order of elements. A simple lexicographical order of the remaining elements, as in the above, would result in $\{F, C, E, G\}$.

C. When Links Break

The Candidate Next Hop List order, obtained when using a unicast routing protocol as illustrated above, is better than without – but is still not ideal. Consider that with the Candidate Next Hop List being $\{F, C, E, G\}$, and data packets being successfully transmitted from *A* to *D* along the path *A-B-F-D*. Now, the link between device *B* and *F* breaks – which would be detected by *B* when trying to forward a data packet to *F*. *B* would then remove *F* from the Candidate Next Hop List, and forward it to the next entry – *C*. As *C* is not on any route route to device *D*, the packet would eventually be returned to device *B*, after having traversed the network indicated in the "cloud" in figure 1, and *B* would be able to remove *C* from the Candidate Next Hop List for that data packet.

D. Candidate Next Hop List Per Packet

[4] specifies that the Candidate Next Hop List is constructed per data packet. [4] also specifies that DFF may signal to the routing protocol when delivery to a next hop, indicated by the routing protocol, fails such that the routing protocol can take corrective action (*e.g.,* remove the entries from the routing table, corresponding to the failed next hop, and initiate recovery as specified by the routing protocol). In the example above, if DFF signals to the routing protocol that *F* has failed, and if the routing protocol then removes routing table entries indicating *F* as next hop, then for *all* subsequent data packets to *D* (until the routing protocol recovers and establishes a new entry in the routing table), the initial Candidate Next Hop List will be $\{C, E, G\}$ – back to the "worst case" default ordering.

IV. DFF++: THE DESTINATION FIELD EXTENSION

As introduced in section II, when data delivery of a data packet fails, DFF removes the failed "next hop entry" from the Candidate Next Hop List for that data packet, and forwards the data packet to the next entry (if any) in that list. Section III illustrated, by way of an example, that while DFF thus may eventually succeed in delivering data packets to the intended destination, the efficiency of that operation – the path-length
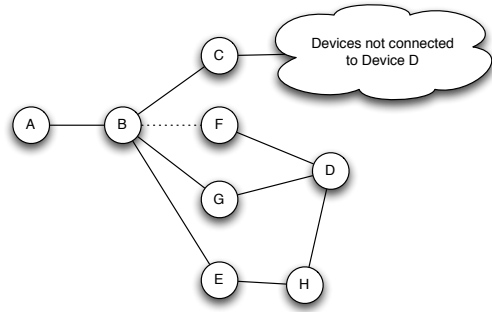

Figure 1. An example of DFF. device *A* sends packets to device *D*. Dahsed line is broken link.

and number of forwards of a data packet – depends on the ordering of entries in that Candidate Next Hop List. As the Candidate Next Hop List is constructed *per-packet*, several subsequent data packets to the same destination may take the same "detour" through the network (or, in the example in figure 1, all explore the same "blind alley" in the network by way of *C*) – either persistently, or, if a unicast routing protocol is also operating in the network, until such time that that unicast routing protocol has recovered from the failure and provided a new entry for the destination in the routing table.

This section proposes a simple extension to DFF, henceforth denoted DFF++, for establishing "memory" across several data packets for the same destination. In the interest of being frugal with required state, this extension (i) piggy-bags off information already maintained by DFF, and (ii) maintains information only temporarily, for as long as DFF otherwise maintains information pertaining to forwarded packets.

A. State

In order to support loop and duplicate detection, each device running DFF maintains a Processed Set, which records sequence numbers of previously received data packets, as well as a list of next hops to, which each data packet has been successively sent, as part of the depth-first forwarding mechanism. Without going into the details of the loop and duplicate detection mechanisms in DFF (refer to [4]), the "Processed Set" consists of "Processed Tuples", of the form:

```
(P_orig_address, P_seq_number,
P_prev_hop, P_next_hop_neighbor_list,
P_time)
```

where:

- `P_orig_address` is the originator address of the received packet;
- `P_seq_number` is the sequence number of the received packet;
- `P_prev_hop` is the address of the previous hop of the packet;
- `P_next_hop_neighbor_list` is a list of addresses of next hops to which the packet has been sent previously,

as part of the depth-first forwarding mechanism;

- `P_time` specifies when this tuple expires and must be removed.

The proposed DFF++ extensions adds an element to each such tuple, thus:

```
(P_orig_address, P_seq_number,
P_prev_hop, P_next_hop_neighbor_list,
P_time, P_dest_address)
```

where:

- `P_dest_address` indicates the destination address of the received packet.

The proposed DFF++ extension also imposes an additional constraint on P_next_hop_neighbor, which is that:

- `P_next_hop_neighbor` must be ordered such that the last element (`P_next_hop_neighbor_list[LAST]`) of that list contains the last neighbour, to which delivery to P_dest_address was attempted (and all previous entries in that list contain successively earlier attempts, with the first element of the list containing the first neighbour, to which delivery was attempted).

B. Processing

On receiving a data packet, not destined to itself, DFF++ defines the following process for selecting an ordered Candidate Next Hop List (CNHL), within the constraints and guidelines from section 11 in [4].

Find the (unique) Processed Tuple, where:

- `P_dest_address` == the destination address of the data packet; AND
- which has the greatest `P_time`.

Using that tuple, the CHNL is constructed thus (where $\oplus$ indicates list concatenation, \ indicates list exclusion, RT(address) is the next hop on the shortest path to the destination from the routing table – if any, and NS indicates the set of neighbours of the device):

1) CNHL = RT(P_dest_address)
2) CHNL = CHNL $\oplus$ P_next_hop_neighbor_list[LAST]
3) CHNL = CHNL $\oplus$ {NS\ {P_prev_hop} \ P_next_hop_neighbor_list}
4) CHNL = CHNL $\oplus$ P_next_hop_neighbor_list

Where 1) satisfies the requirement from [4] that first element in the CNHL is the next hop, indicated by a routing table (if present). Items 2) and 3) capture "pick up where the most recent data packet delivery to the same destination left off". Specifically, 2) is the neighbour, last tried for the most recent packet to the same destination, and which is not yet confirmed as having failed (in which case there would be a subsequent entry in the list, except if all neighbours had been tried and failed), 3) includes all other so far untried (by the most recent data packet delivery for this destination) neighbours. Finally, 4) – which is an optional step in DFF++ – includes all previously (by the most recent data packet delivery) tried neighbours – excluding, of course, the one from which the

data packet was received – capturing the fact that a previous failure may have been due to transient losses.

C. Impact

Adding and using P_dest_address, as described above, allows construction of the Candidate Next Hop List to make use of information on previous data packet forwards to the same destination.

Returning to the example in figure 1, one of the issues raised in section III-C, and detailed in section III-D, is alleviated:

1) The initial Candidate Next Hop List for the first data packet arriving at *B* for destination *D* will – using the same ordering (routing table entry first, then the the "worst-case" lexicographical order) be {*F*, *C*, *E*, *G*}.
2) Initial delivery is attempted via *F* (which is added to the end of P_next_hop_neighbor_list) and fails, and delivery via *C* is attempted (which is added to the end P_next_hop_neighbor_list).
3) Delivery via *C* also fails (no path via *C* to *D*), and delivery is now attempted via *E* (which is added to the end of P_next_hop_neighbor_list) – as there is a valid path to *D* via *E*, delivery succeeds, and the P_next_hop_neighbor[LAST] for that processing tuple now contains *E*.
4) Other data packet for *D*, arriving at *B*, *before* the routing protocol (if any) has recovered and provided an entry in the routing table for *D*, will, using the DFF++ Candidate Next Hop List construction rules given in section IV-B, result in a Candidate Next Hop List of:
   - If they arrive after step 3), {*E*, *G*} – thus avoiding the "broken link" to *F*, as well as the "blind alley" that would be attempting delivery via *C*.
   - If they arrive after step 2) but before step 3), {*C*, *E*, *G*} – thus avoiding the "broken link" to *F*, but not the "blind alley" that would be attempting delivery via *C*
   - If they arrive before step 2), {*F*, *C*, *E*, *G*} – thus offering no improvement over DFF, but also no additional penalty.

Note that DFF++ avoids the problem of repeatedly attempting delivery to a given destination via "blind alleys" and over "recently detected broken links", but does not attempt at offering "shortest paths" – that remains under the auspices of a routing protocol (if any) in the network. Also, DFF++ does not affect interoperability: the extension does not introduce any new signals or any new external behaviours, but simply offers guidance for how to order the Candidate Next Hop List for a data packet. The specification of DFF [4] specifically encourages an intelligent ordering, and DFF++ does just that. As that ordering of the Candidate Next Hop List for a data packet concerns only internal processing of a device, DFF and DFF++ remain interoperable. DFF++ can furthermore be deployed with exactly the same (or no) unicast routing protocols as DFF.

V. SIMULATION AND ANALYSES

In order to evaluate the performance of DFF++, and compare its performance to that of DFF, network simulations by way of NS2 are employed. While network simulations are, at best, an approximation of real-world performance (particularly due to the fidelity of their lower layers to reality), they do provide a baseline for comparison and, generally, best-case results, *i.e.,* real-world performance is expected to be no better than that which is obtained through simulations. The reason for using network simulations is that it allows running experiments with different protocols under identical conditions and parameters (MAC layer, distribution, number of nodes, etc.).

Simulations were conducted using the TwoRayGround propagation model and the IEEE 802.11 MAC. Although there are various low-layer technologies more commonly (and, perhaps, more viably) used for LLNs (power line communication, 802.15.4, low-power wifi, bluetooth low energy, etc.), general behaviours of a protocol can be inferred from simulations using 802.11.

DFF and DFF++ were evaluated both in isolation (without a concurrently operating unicast routing protocol) and when used in conjunction with the Lightweight On-Demand Ad hoc Distance-vector Next Generation (LOADng) routing protocol, standardised for use in, *e.g.,* SmartGrid/SmartMeter Automatic Metering Infrastructure (AMI) networks [5], [1] – and, compared with LOADng operating alone in the same networks, so as to evaluate the benefits of DFF and DFF++, respectively. In total, five different protocol combinations were evaluated:

- *DFFonly*: DFF according to [4], with Neighbourhood Discovery Protocol (NHDP) [6] used for bi-directional neighbour discovery, suggested by [4].
- *DFF++only*: As described in section IV, with [6].
- *LOADng*: LOADng, according to [1].
- *LOADngDFF*: LOADng with DFF ([4], [6] and [1]).
- *LOADngDFF++*: LOADng with DFF++ ([1] + the process described in section IV and with [6]).

A. Network Topology and Traffic Characteristics

The general network topology of a scenario is as follows: n (from 63 to 500) devices are placed randomly (while ensuring that the network is still connected) in a square field, such that to maintain a constant device density. There are $n-1$ Constant Bit Rate (CBR) streams in the network. The original node of the CBR stream (chosen randomly) sends one packet of 512 octets every 5 seconds to a destination (chosen randomly). As DFF is supposed to be particularly beneficial in lossy networks the simulations enforce that a packet is lost with a probability of 20%. Simulations were run for 100s each, and for each datapoint 20 different and randomly generate scenarios – all corresponding to the same abstract parameters – were simulated, with the results presented below representing averages from among these.

For NHDP [6], a HELLO message interval must be chosen. The shorter the HELLO message interval, the more accurate a list of neighbours can be acquired (and so, the better can DFF and DFF++ do their jobs) – but at the expense of increased control traffic overhead. For the purpose of these simulations, a HELLO interval of 1s was (arbitrarily) chosen as it represents a "very frequent" HELLO message exchange and therefore a good "worst case" example. In a deployment, the HELLO interval should be selected so as to correspond to the expected local network topology change rate.

B. Simulation Results

Figure 2 depicts the packet deliver ratio of the different protocols combination. When DFF (DFFonly and DFF++only) is running without an external routing protocol, DFF++ offers a significant improvement of the packet delivery ratio over DFF. The lower performance, experienced when running without an external routing protocol is due to depth first searching being inefficient, worst case causing a complete transversal of the network graph for each data packet. DFF, used with LOADng, yields about 20 percentage points improvement of the delivery ratio, as compared to LOADng alone, and DFF++ used with LOADng further improves the data delivery ratio – albeit marginally so.
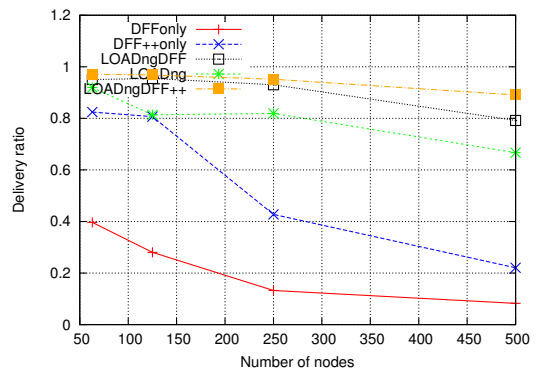

Figure 2.   Average data packet delivery ratio

Figure 3 depicts the average end-to-end delays – and it should be noted, particularly for this figure, that only data packets that successfully arrive the destination are accounted in the statistics. DFF and DFF++ alone exhibit significantly greater delays than when running with LOADng – balance with the data delivery ratio in figure 2, this is noteworthy: combining DFF/DFF++ yields lower delays and better data delivery. LOADng alone exhibits a slightly lower delay than when compared with DFF and DFF++ – which is compensated by the fact that inclusion of DFF/DFF++ increases the data delivery ratios obtained by approximately 20 percentage points.

Figure 4 depicts the average path length (of successfully delivered packets), *i.e.,* the number of hops required for a packet to reach its destination. When running without an external routing protocol, the "blind" depth-first search of DFF causes 6-7 times as long paths as LOADng – with DFF++ offering shorter path lengths than DFF. When combined with LOADng, both DFF and DFF++ yield significantly shorter
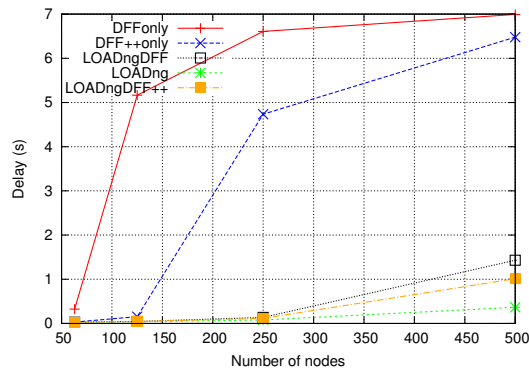
Figure 3. Everage end-to-end delay

path lengths, as compared to DFF/DFF++ alone – and slightly longer path-lengths than when running LOADng alone. This, again, is explained by the fact that LOADng with DFF/DFF++ increases the data delivery ratio.
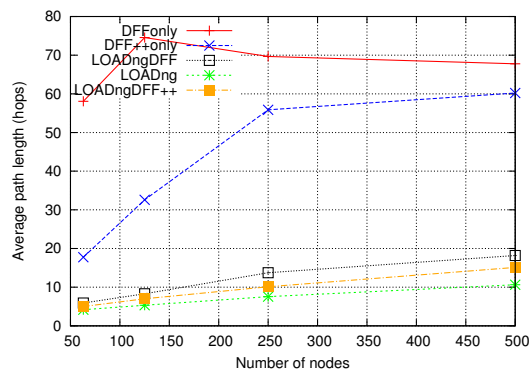

Figure 4. Average path length

Figure 5 illustrate the control packet overhead. For DFF and DFF++ without an external routing protocol, the overhead is constituted from locally exchanged HELLO messages, generated by NHDP to discover the bi-directional neighbours. When introducing LOADng (either alone, or in conjunction with DFF/DFF++), the protocol overhead of that routing protocol for route discovery (see [7] for details) is also imposed on the network, and causes additional MAC layer collisions.

VI. CONCLUSION

This paper studies the application of depth-first forwarding in LLNs, and presented a minimal-impact optimisation to DFF [4], denoted DFF++. DFF++ is fully interoperable with DFF; it offers a procedure for ordering the Candidate Next Hop List for a data packet to be forwarded at a device. Using DFF++ alleviates some problems of DFF, such as repeatedly trying to forward traffic down "blind alleys" and across recently detected broken links. Performance studies comparing DFF and DFF++ alone (without a concurrently

operating unicast routing protocol) have revealed the benefits of this optimisation to be significant: DFF++ attains a higher
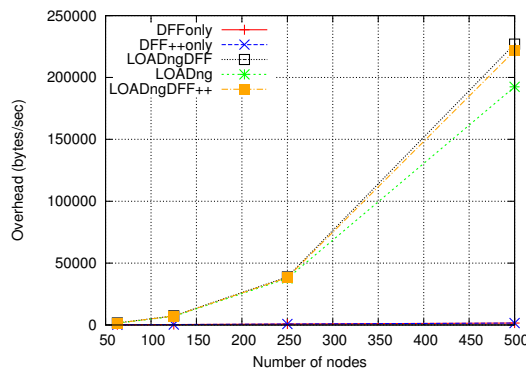

Figure 5. Control packet overhead

data delivery rate, shorter paths and lower data delivery delays than DFF.

Neither DFF nor DFF++ attempt to offer "shortest paths" – that remains under the auspices of a routing protocol, and both DFF and DFF++ are intended to operate concurrently with a unicast routing protocol. For the purpose of this study, the unicast routing protocol LOADng [5] has been tested with and without DFF/DFF++. A first observation is that the performance of both DFF and DFF++ is significantly improved by operating conjointly with LOADng. A second observation is, that the data delivery ration of LOADng also is significantly improved by the use of either of DFF and DFF++ – but that LOADng with DFF++ offers a moderate, but consistently better, performance when compared to LOADng with DFF.

DFF++ attains these performance improvements without introducing new control signals, minimal additional state (a single IP address added to an existing data set) and low implementation complexity – and, remains completely interoperable with DFF, as specified in [4].

REFERENCES

[1] T. Clausen, A. C. de Verdiere, J. Yi, A. Niktash, Y. Igarashi, H. Satoh, and U. Herberg, "The lln on-demand ad hoc distance-vector routing protocol - next generation," The Internet Engineering Task Force, July 2013, internet Draft, work in progress, draft-clausen-lln-loadng.

[2] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and J. Vasseur, "RPL: IPv6 Routing Protocol for Low power and Lossy Networks," March 2012, IETF RFC 6550.

[3] S. Cespedes, A. Cardenas, and T. Iwao, "Comparison of data forwarding mechanisms for ami networks." Proceedings of 2012 IEEE Innovative Smart Grid Technologies Conference (ISGT), January 2012.

[4] U. Herberg, A. Cardenas, T. Iwao, M. Dow, and S. Cespedes, "Depth-first forwarding (dff) in unreliable networks," Experimental RFC 6971, June 2013.

[5] "ITU-T G.9903: Narrow-band orthogonal frequency division multiplexing power line communication transceivers for G3-PLC networks: Amendment 1," May 2013.

[6] T. Clausen, C. Dearlove, and J. Dean, "Mobile Ad Hoc Network Neighborhood Discovery Protocol," Std. Track RFC 6130, April 2010.

[7] T. Clausen, J. Yi, and A. C. de Verdiere, "Loadng: Towards aodv version 2." in *VTC Fall*. IEEE, 2012, pp. 1–5.