Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 56 (2015) 217 – 224

Procedia Computer Science

The 10th International Conference on Future Networks and Communications (FNC 2015)

Adding support for delay tolerance to IPv6 networks

Tyler Ward[a], Kirk Martinez[a], Tim Chown[a]

[a]*Web and Internet Science,Electronics and Computer Science, University of Southampton, United Kingdom*

Abstract

As we continue to connect ever lower power and more power constrained devices to the Internet of Things the problem of maintaining constant end to end connectivity becomes harder. Accepting that continuous end to end connectivity cannot be maintained, we are forced to seek solutions to allow good operating function. Delay Tolerant Networking, an evolution of existing store and forward systems is a candidate for resolving this issue, however, current implementations are not ideal for use in constrained Internet of Things environments. We propose a solution to this by integrating the capabilities of Delay Tolerant Networking into the IP layer, in such a way as to maintain compatibility with existing and future systems and minimising additional overhead. This has been achieved by developing a new IPv6 Hop by Hop option header which contains the information required for messages to be delayed. This solution is then demonstrated to be implementable within the limitations of current Internet of Things hardware.

Keywords: Delay Tolerant Networking ; IPv6 ; Sensor Networks

1. Introduction

Maintaining continuous connectivity with Internet of Things devices is not always possible. This causes issues as continuous connectivity is the expected behaviour for internet connected devices. The reasons for the lack of connectivity can be due to a range of factors both deliberate and incidental.

The main reason for deliberately causing a lack of connectivity is to save power. Keeping communication hardware in a receive state is a significant drain on the energy resources of small battery powered devices. A method to reduce the power use of devices in sensor networks is to put the device into a low power sleep state for as long as possible. In this state the device has its communications powered down so is unable to be contacted but will consume substantially less energy. An alternative solution to this is to provide more energy resource to the device. Doing this limits the options for placement of these devices as they will either require access to a power source or be physically larger to accommodate the additional batteries. As a result this has become a trade off between connectivity and power consumption.

* Corresponding author. Tel.: +44 (0)23-8059-4583.
E-mail address: tw16g08@ecs.soton.ac.uk

doi:10.1016/j.procs.2015.07.202

With regard to incidental loss of connectivity, the environment in which the devices operate can cause breakdowns in communication. As an example, in remote sensing systems the weather can cause communications links to become unavailable. In the Glacsweb network, the winter snowfall often buries the equipment and prevents surface radio communication and in the summer, increased water levels within the ice can interrupt probe communications[1]. These issues are difficult to resolve and must be anticipated as a part of the deployment challenges.

Rather than trying to solve these issues individually the lack of low level connectivity can be addressed at a higher level. Currently many internet connected devices that have the issue of limited power reserves rely on the device initiating all communication rather than listening for any incoming communications. This relies on having an always available route to an endpoint which they communicate with, and requires all communication with the device go through that endpoint. While this has worked for current devices, this solution reimposes many of the limitations that had been removed by using direct internet connectivity rather than proprietary systems. Through previous research, Delay Tolerant Networking (DTN) had been shown to have potential but required additional research to make it suitable for IP based networks[2]. This paper covers the implementation of IP extension headers as a means of implementing DTN and discusses the initial testing of the system in Cooja simulated nodes then on Zolertia Z1 hardware.

2. Delay Tolerant Networking

With Internet Protocol (IP) networking technologies a message is either immediately delivered to its destination or is discarded. This is inconvenient when there may be many interruptions in connectivity as the originating device will need to keep retrying the transmission. While some link layers support retransmissions, these are intended to protect against packet corruption and occasional packet loss rather than loss of connectivity. In many cases the contents of the packet do not need to be delivered immediately and are still useful if they are delivered at a later time. The one solution to this is to allow nodes on the route to buffer packets and send them on when connectivity to the next hop has been restored. This is similar to existing store and forward technologies used in current sensor networks[3].

Store and forward is often used in sensor networks to allow packets to travel through the network one hop at a time. These existing systems are usually limited to a specific link layer protocol or application layer protocol. This is a workable technology when the data is constrained to a single network or the data will always be sent with a specific protocol. Removing these limitations, however, is one of the main benefits of connecting such systems to the internet.

Connection interruptions are also an issue in other networks such as interplanetary communications. In the interplanetary use case, connection interruptions are mainly caused by alignment issues between nodes. Planetoids might be in the way or dishes not pointed in the right direction for immediate communication. These extreme distance communications also need to support the delays while a packet is in flight. There is currently an effort to create a solution to these issues known as Delay Tolerant Networking[4,5].

Delay Tolerant Networking aims to improve the connectivity between devices by providing a consistent methodology for managing delays that can be used over multiple link layers and containing any sort of data. While in most store and forward applications the communications system needs to be a specific type, in Delay Tolerant Networking this is no longer a requirement.

As well as improving the reliability of communications DTN can also improve network efficiency. With DTN networks, attempts at retransmitting messages that were lost can be achieved far more efficiently than is possible with end to end retransmissions. There are several reasons for this, as the DTN node is closer to the problem it has a better knowledge of when it is appropriate to retry. This avoids retrying when the message would still be unable to make it through; in addition the retries do not need to travel the successful part of the journey again. These combine to make retransmission far more efficient than would previously be possible, also as the origin does not have to be involved in the retransmissions it can perform other tasks or enter a sleep state itself. While the retransmissions will be more efficient, this comes at the cost of additional overhead of transmitting and processing the delay tolerance of the packet. For more reliable networks there will be less of an advantage as retries will be required less frequently.

While Delay Tolerant Networking is intended to be compatible with existing protocols and networks it is not simply a drop-in solution. As well as supporting the delay tolerant packets, the software on the endpoints will need to be able to deal with packets that have been delayed. Software which has been designed to wait for an immediate response from a device before continuing might be challenging to retrofit DTN compatibility into. While any upper

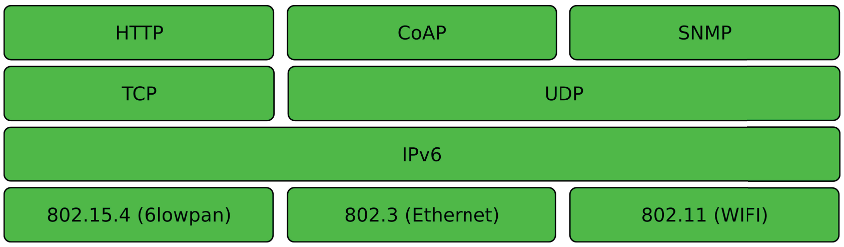

Fig. 1. IP as the common layer in the internet of things.

layer protocol can be sent with Delay Tolerant Networking enabled, the protocol might not be capable of handling the potential delay or require careful thought on its use. The Transmission Control Protocol (TCP) is one example of this, while there is no inherent problem with using TCP in a DTN environment the time-outs need to be at least as long as the DTN packet lifetime to avoid resending a packet that is still being transfered through the network.

Using Delay Tolerant Networking opens up several security and quality of service considerations mainly related to denial of service. As packets are buffered for a node, a large amount of packets can be built up, which can cause several issues. Intermediate routers can become saturated with messages that will be held waiting for other devices to reappear, this can prevent other users getting access to them. Alternatively this can overwhelm devices when they regain connectivity as they receive the entire buffer of packets waiting for them. These situations can occur either maliciously or as an unexpected side effect of the intended use.

2.1. The Bundle Protocol

Most of the current work on implementing Delay Tolerant Networking has been on the Bundle Protocol[6,7] which is an implementation of DTN for the deep space network use case. The Bundle Protocol works as an overlay network implementing its own system on top of existing networks and encapsulating the data within it. This makes it possible to transmit packets over multiple network types along its route, and allowing it to use the existing deep space networks which use a mixture of different network stacks as well as using the internet. This significantly increases the size of the packet headers, as information must be contained in the encapsulated protocol as well as the networks own headers.

The encapsulating nature of the Bundle Protocol puts additional demands on the nodes that need to access data within the encapsulated packet. These demands come as both additional code to decode the protocol and the resulting code space requirement on the device, there will also be the processing required to decode an additional protocol. These demands would be required on any node which would be able to delay the packet, but would not be required for nodes that just route the underlying network.

The features of the Bundle Protocol make it a highly versatile protocol at the cost of overhead. While this reduces its suitability for use in constrained networks, the principles of Delay Tolerant Networking can still be applied.

3. Integrating Delay Tolerant Networking into IP based networks

In order to use DTN on a network, it needs to be added to a layer common across all network types that the data will flow though. With store and forward this would typically require the use of a specific application protocol or low level network, which as previously discussed prevents flexibility. Previous attempts at implementing Delay Tolerant Networking technologies on IP, such as the Bundle Protocol, operate by adding another networking layer above the current IP framework and using it as if it was a link layer[8]. This encapsulating layer includes the packet data and routing information but cannot be processed by devices that do not support the protocol. In addition this requires duplication of data already contained in the packet as it must be included in the encapsulated version as well as the packet headers. To allow for seamless integration with existing systems both end devices and routers without DTN support need to be able to handle DTN packets in a predictable manner. DTN nodes will also need to be capable of handling non DTN enabled traffic. For these reasons DTN can not be implemented as an encapsulating layer or as a new lower layer protocol.

The alternative is to add the DTN functionality to a common layer across the network, for internet communications IP is the only layer which meets these requirements, this can be seen in Figure 1. IPv6 was created with the ability to accept optional extension headers which can add additional features or information to packets[9], examples of this include the routing header which specifies the route through the network or the authentication header which provides cryptographic authentication of the packet. These extension headers can be used to add the required information for using DTN to packets traveling across the network. Adding a new type of header can cause unpredictable responses from devices that are unable to process it, there are, however, two existing expansible option headers, the hop by hop options and destination options headers which can be used. These headers explicitly state how packets should be handled by devices which are unable to recognise one or more of the options. This guarantees a predictable response from devices which do not have support for delay tolerance, however, some parts of the internet infrastructure just drop packets with these headers[10] which could cause problems for protocols like this. The hop by hop options header is evaluated by each router along the route whereas the destination option is only evaluated by the final destination so the hop by hop options header is the one to use.

Adding a hop by hop option to a packet incurs a two byte overhead, one for the option type and one for the option length. There is an additional 2 byte overhead to add the hop by hop header itself if it is not already present. The actual data of the packet will contain a set of control flags and an expiry timestamp. This will provide the features required for the processing and handling of delayed packets. The total size of the DTN hop by hop options including type and length details is 6 bytes. While other DTN systems include additional features within the DTN data format these would be best served by a separate extension header. Many of the additional features of the Bundle Protocol already have equivalent IP headers available to perform those functions such as the fragmentation header. Figure 2 shows how the new DTN header fits in with the existing IP headers.

3.1. Packet expiry timestamps

The most important piece of data required for DTN operation is the expiry time at which a packet should be discarded. Without this, a packet could remain held in the network consuming storage resources indefinitely, or delivered too late to be of any use. In order to maintain a small packet overhead an efficient method of storing the time is required.

Storing a time as a fixed Unix style timestamp will require 64 bits, for small packets on constrained networks this is a noticeable amount of overhead. While 32 bit timestamp values could be used, this would make the protocol obsolete in the near future so is not a sustainable option. Using a fixed timestamp would also require every DTN node to maintain a reasonably accurate clock in order to determine packet expiry, which will not be possible for low power devices without an on-board real time clock.

Instead of using a fixed timestamp, a relative timestamp could be used. By doing this the value would be the amount of time remaining before the packet expired. This would require each node to update the remaining lifetime on route but avoids the need to know the exact time, just the difference in time between reception and transmission. Using a relative timestamp, the use of a 32 bit timestamp variable becomes feasible and reduces the packet size compared to a fixed timestamp by half.

The size of a relative timestamp can be decreased further by reducing the accuracy of the timestamps for long duration delays. As the delay lifetime extends into the order of hours or days then defining the lifetime down to the exact second is no longer required. By using this property an exponent based timestamp was developed, this allows for high precision for small lifetimes while still providing far longer timeouts. A 4 bit exponent combined with a 12 bit multiplier was chosen as this gives a high level of accuracy whilst still allowing packet lifetimes up to eight and a half years which should be plenty of time for any application. This solution reduces the amount of timing information that needs to be transmitted to 2 bytes, a 4 fold improvement over the fixed timestamps.

With relative timestamps, the time taken while the packet is in flight needs to be considered. In most cases this will be less than a second and can be ignored, however in some cases the time between transmission and reception will be significant. While this is not the case with almost all current IP carrying systems, the ability to accept delayed packets opens up such options to protocol designers. In cases where there could be an unknown time between transmission and reception it will be necessary to use a fixed timestamp as part of that link layer's headers in order to determine the transmission time.

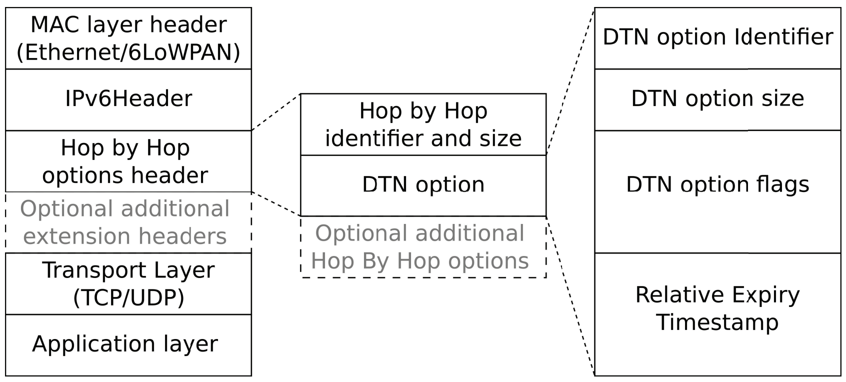
MAC layer header (Ethernet/6LoWPAN)
IPv6Header
Hop by Hop options header
Optional additional extension headers
Transport Layer (TCP/UDP)
Application layer
Hop by Hop identifier and size
DTN option
Optional additional Hop By Hop options
DTN option Identifier
DTN option size
DTN option flags
Relative Expiry Timestamp

Fig. 2. How the DTN option header fits into the network layer structure.

3.2. Custody transfer

To pass packets from one DTN enabled device to another, a process known as custody transfer is required. To do this the sender of a DTN packet needs to know that the next DTN router has received the packet and has taken responsibility for its onwards transmission. This is achieved by the recipient sending a custody response to the current custody holder. There may be several non DTN hops between two DTN routers so the custody acceptance response cannot be sent to the last link layer hop as that might not be the previous custody holder. While it would be possible to add the IP address of the last DTN node to the packet to give a place to return the custody report to, this adds additional overhead. This can be avoided by sending packets back towards the message's origin during which they will travel back through the previous DTN hop. To allow DTN nodes to identify these packets they need to be tagged, which will allow the router to know that they need to evaluate their contents rather than just letting them pass though. This tag can be implemented as another bit of the flag byte in order to avoid additional overhead.

As well as ensuring that the DTN routers can detect the custody packets it is necessary to send the details of the custody transfer in some format. Internet Control Message Protocol (ICMP6) makes a good candidate to transfer status information around as it is already widely used for that purpose on the internet.

Depending on the situation there will need to be different types of responses. In the case where the custody has been accepted and can be transfered an acceptance message will be sent. The acceptance message completes the handover process at which point the transmitting node can delete its copy of the packet, it is then the job of the receiving node to ensure forward transmission. As this message is intended for the DTN router holding the custody this message should not be passed beyond that router.

In some cases the next DTN router will not be able to accept custody of the packet for the entire remaining lifetime in which case a temporary custody acceptance can be used. It indicates that a node has taken custody over the packet for a given period of time but the original custody holder should maintain custody. The original custody holder can then retry after the temporary custody has expired, should a full custody transfer not be received in the meantime.

The case where the next DTN router cannot accept custody is an important one. This situation could occur for many reasons, such as running out of storage space or only accepting custody in certain conditions. The way to deal with this is to send the packet onwards if it is able, just like a normal router would, otherwise the packet should be silently dropped. The lack of a custody response for the dropped packet will cause the current custody holder to perform a retransmission at a later time when it might be possible to accept custody again.

4. Implementations

In order to investigate the potential of adding DTN technologies into the IP layer, two versions of the handler were created, one to run on the Linux based border routers and one to run on the constrained sensor network hardware itself. These implementations can be used together to provide a working delay tolerant system which is capable of handling delays both on and off the low power network.

4.1. Implementation for Linux systems

In order to provide delay tolerance on the 6LoWPAN border router a Linux implementation of the delay tolerant header processing is required. While handling such a protocol is much better suited to a kernel module this would be time consuming for initial development. To avoid this a userspace program was created, Python was chosen for this in order to leverage existing packet analysis libraries and tools for faster development.

As the Linux kernel provides an abstraction from the low level network when using network sockets a different solution was required to get access to the raw option headers. This is done by capturing complete packets and performing any required actions in userspace, the packet capture library (py-pcap) was used to perform this. These raw packets then need to be decoded in order to extract the required information from the packet. The dpkt library was used to decode the packets, allowing the packed binary data to be separated into individual parameters.

With the packets decoded, they are then filtered to remove those which did not contain a delay tolerant header. At this point the DTN header was evaluated and the packet passed to the packet store. As the usual Linux routing has been interrupted to allow the analysis of the DTN packets, the packet's next hop needs to be calculated manually so it can be sent over the correct interface. Initially the Linux code was implemented without custody transfer so Ethernet ARP tables were used to provide a guess as to whether it was possible to deliver the packet or not. Later on custody transfers were added to provide a guarantee that the packet had been received. The use of ARP tables provides a good example of where additional local network knowledge of a DTN node can avoid unnecessary transmissions.

In order to send the packets onwards with a DTN option header the standard sockets library could not be used. Raw sockets were used to send the packets, this allows the userland code to control the DTN header but also requires the software to calculate the packet routing and link layer addressing as well.

The Python Linux implementation provides a functional demonstration of DTN working in a significantly less constrained environment. In order to realise the full benefits of Delay Tolerant Networking the protocol would also need to be supported on the low power network as well.

4.2. Implementation for constrained hardware

With a Linux implementation created, the next step was to implement the protocol on a constrained 6LoWPAN system. To develop this the Contiki operating system running on Zolertia Z1 nodes was used. The code was also tested on Tmote sky nodes, however, due to the code space limitations some of the Contiki example programs would no longer fit when the DTN processing was also present. The DTN code requirements came to approximately 1Kib of code space memory, however, with some optimisation this could be reduced, The increased amount of ram and flash on newer chips are likely to make this a non issue. A test DTN network was created using the Cooja emulation tool for Contiki to allow for faster development and easier access to the contents of in-flight packets. The Cooja tool uses mspsim to emulate the node hardware, which will provide realistic results without needing to manually retrieve and program hardware for each test.

The Contiki implementation was separated into two sections. Part of the implementation would be integrated into the network stack to provide fast processing of incoming packets and acknowledgements. The other part would be implemented as a separate process that would manage the stored packet records and perform any retransmissions required. An overview of the DTN implementation can be found in Figure 3.

Contiki's IP networking uses the uIP network stack to handle the processing of the IP layer frames. The modularity of this code meant that the DTN code could be directly integrated into Contiki's network stack. This still allows for nodes that did not support DTN to be built with the same codebase without incurring any processing or code space overhead. The RPL routing system used by Contiki uses the hop by hop option for some of its information exchanges, this meant that Contiki already had support for detecting hop by hop options and processing them. This code was expanded to include the DTN header, allowing the DTN options to be inspected as part of the packets reception.

The additional code in the network stack decodes and validates the DTN option included in the packet. If the packet is a new DTN enabled packet the packet is stored, if possible, in a new record in the DTN custody store and timers are set for the packet expiry. The packet then continues through the Contiki uIP networking stack as usual and is forwarded onto the next node or passed to the receive code as required. This allows packets to be passed through the network without additional delay if the network is up. As this is done regardless of whether the packet could be stored or not, it also allows packets to bypass nodes that have filled up the custody packet buffer.

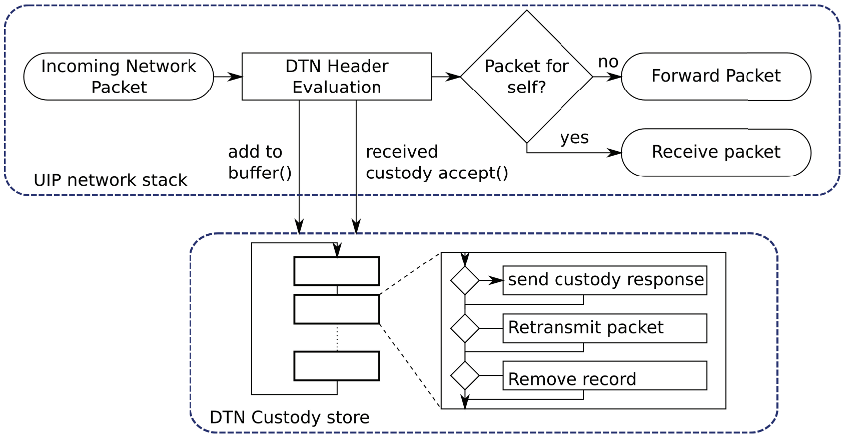

Fig. 3. DTN processing within the Contiki operating system.

If the packet is a custody acknowledgement it is matched to the appropriate record in the custody store, that record is then marked for removal in the next store cleanup, the packet is then dropped to prevent it being forwarded any further. If there is no matching record the packet is sent onwards as a previous node might be holding custody of the packet.

The custody store management process iterates though each of the records in the custody store and performs any actions required. The data storage in the custody store is implemented in two parts, the index and the packet store itself. The index stores the state of the packet and basic information about the packet, adding this information to the index avoids the need to access the packet except when performing retransmissions. The packet store, stores the actual packet, although this was implemented in ram it does not have to be, use of the coffee file system within Contiki would allow for packets to be stored in flash allowing for the space saved to be allocated to a larger index.

The main actions of the custody store are to send custody acknowledgements, perform retransmissions, and remove packets that have been delivered or expired. Due to limitations in the network stack of Contiki it is not possible to generate a custody response while processing the message in the network so this is done as one of the actions of the store. An ICMP6 message is built and sent out in the normal way, the addition of the DTN header and flagging as an informational DTN packet is implemented as part of the network send routine.

Where a custody message has not been received the custody store process will retransmit the message. In the trial network a retry was attempted after a fixed time if no response had been received. In order to perform a retransmission the packet was copied from the packet store into the network buffer. As the packet has been in storage the expiry timestamp was updated using the information in the index to give the new time until expiry. The network buffer with the updated packet was then transmitted.

In order to prevent the custody store filling up with unused packets, completed records need to be removed. Records that have been flagged for deletion by a custody response having been received are deleted as part of the stores operation. In addition the expiry timeout is checked to see if the packet has passed its maximum lifetime.

4.3. *Preliminary Testing and Evaluation*

Initial testing was carried out using the Cooja[11] tool for Contiki. A multi hop network was created comprising of a mix of Z1 and sky nodes with and without support for delay tolerance. This network was connected using a SLIP interface to a Linux border router which was running the Python DTN code. Several of the nodes were programmed with a DTN enabled UDP responder which would be used as the target for the test communications. The network traffic was then extracted for analysis by feeding the cooja radio logs into the wireshark packet capture and analysis tool.

Using this testbed framework, several experiments were carried out in order to see how the DTN nodes would behave. Nodes along the packets route were deliberately disconnected from the network in order to represent a loss of connectivity. Using the results from wireshark, it was possible to see the retransmissions of packets and the custody acknowledgements after a successful transmission. With the DTN features added to the network, packets were successfully retransmitted and delivered when the disconnected devices were reconnected to the network.

After testing on a simulated system, the same code was loaded onto physical sky and Z1 nodes which were deployed in our research lab. Instead of trying to access the radio messages in the network, wireshark was used on the link between the border router and the network. While this does not give hop by hop information on the packet's progress it still provides enough information to confirm the successful delivery of the packets even with artificially injected connectivity breaks, such as removing antennas. From this deployment the DTN implementation was shown to work in a real world deployed network.

5. Conclusion

Meeting the current expectation of continuous connectivity for all internet connected devices is infeasible with the new generation of ultra low power Internet of Things devices. Previous attempts at working around those issues leave major limitations that counteract many of the benefits gained from using the Internet of Things over other connection technologies. Delay Tolerant Networking makes it possible to maintain reliable communications without the burden of providing constant low level connectivity or limitations on what can be sent.

Previous delay tolerant technologies have required significant overhead making them unsuitable for constrained devices. It is, however, possible to provide delay tolerance with a far lower overhead and without breaking compatibility with existing systems. This has been solved by adding delay tolerance to packets using the IPv6 hop by hop extension header. A protocol has been created around this and implemented on Linux and Contiki operating systems. Through these implementations this technology has been demonstrated to be viable for use on constrained hardware systems.

Bringing Delay Tolerant Networking to IP, will allow a new wave of ultra low power internet connected devices to be created. This opens many opportunities that were previously unavailable to such devices.

This protocol demonstrates that IP layer DTN is viable even on constrained Internet of Things hardware. To make the most from the protocol some additional refinement will be required to resolve any edge cases that may exist, but care must be taken to avoid bloating the protocol with additional features as these will increase the overhead of the protocol. With this complete the huge potential from using DTN in the internet of things can be unlocked.

References

1. Martinez, K., Padhy, P., Elsaify, A., Zou, G., Riddoch, A., Hart, J., et al. Deploying a sensor network in an extreme environment. In: *Sensor Networks, Ubiquitous and Trustworthy Computing*. IEEE Computer Society; 2006, p. 186–193. URL: `http://eprints.soton.ac.uk/262701/`; event Dates: 5-7 June 2006.
2. Ward, T., Martinez, K., Chown, T.. Simulated Analysis of Connectivity Issues for Sleeping Sensor Nodes in the Internet of Things. In: *The Eleventh ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Network*. 2014, .
3. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.. A survey on sensor networks. *Communications magazine, IEEE* 2002; **40**(8):102–114.
4. Cerf, V., Burleigh, S., Hooke, A., L.Torgerson, , R.Durst, , K.Scott, , et al. Interplanetary Internet (IPN): Architectural Definition. Active Internet-Draft; 2001. URL: `http://www.ietf.org/id/draft-irtf-ipnrg-arch-00.txt`.
5. Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., et al. Delay-Tolerant Networking Architecture. RFC 4838 (Informational); 2007. URL: `http://www.ietf.org/rfc/rfc4838.txt`.
6. Scott, K., Burleigh, S.. Bundle Protocol Specification. RFC 5050 (Experimental); 2007. URL: `http://www.ietf.org/rfc/rfc5050.txt`.
7. Fall, K.. A delay-tolerant network architecture for challenged internets. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*; SIGCOMM '03. New York, NY, USA: ACM. ISBN 1-58113-735-4; 2003, p. 27–34. URL: `http://doi.acm.org/10.1145/863955.863960`. doi:10.1145/863955.863960.
8. Bruno, L., Franceschinis, M., Pastrone, C., Tomasi, R., Spirito, M.. 6lowdtn: Ipv6-enabled delay-tolerant wsns for contiki. In: *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*. 2011, p. 1–6. doi:10.1109/DCOSS.2011.5982154.
9. Deering, S., Hinden, R.. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard); 1998. URL: `http://www.ietf.org/rfc/rfc2460.txt`; updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946.
10. Gont, F., Linkova, J., Chown, T., Liu, W.. Observations on IPv6 EH Filtering in the Real World. Active Internet-Draft; 2015. URL: `http://www.ietf.org/id/draft-gont-v6ops-ipv6-ehs-in-real-world-02.txt`.
11. Osterlind, F., Dunkels, A., Eriksson, J., Finne, N., Voigt, T.. Cross-level sensor network simulation with cooja. In: *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*. 2006, p. 641–648.