Introduction to Distributed Protocol Stacks for Wireless Sensor Networks

Peter Rothenpieler, Dennis Pfisterer
Institute of Telematics, University of Lübeck, Germany
Email: {rothenpieler,pfisterer}@itm.uni-luebeck.de

***Abstract*—In this paper, a model is proposed, in which neighbouring nodes share layers of their communications stack with each other in a cooperative fashion to enable the use of IPv6 on each node without the need to include the stack's full programming logic on each node, while still preserving transparent IP interconnectivity by forming a Distributed Protocol Stack (DPS). We give an overview on the concept behind DPS, a first protocol specification and evaluate our prototype implementation on the iSense WSN platform against a native IPv6 implementation regarding code size, memory usage and round-trip time.**

***Keywords*-6LoWPAN; WSN; RPC; distributed stack;**

I. Introduction

Since its beginnings, the field of Wireless Sensor Networks (WSNs) has been changing from custom-tailored, problem-specific solutions towards standardized hard- and software platforms. While miniaturization is used to produce even smaller nodes for even less money, the software of each node at the same time becomes more complex and requires more memory. Contemporary WSNs allow the direct integration of each sensor node into the Internet using full-featured IPv6-stacks and dynamic routing protocols. As an example, Table I shows that more feature complete implementations (including e.g. ND) require 21 to 50 KB of Flash and 3 to 5.9 KB of RAM [1]. While code sizes of up to 50 KB are supported by modern WSN platforms, the aforementioned size only includes the IPv6 stack and neither routing, security mechanisms, duty-cycling nor the application itself and the actual memory requirements will easily grow beyond the supported sizes.

To overcome the resource constraints of WSNs, cooperation between neighbouring nodes in a network is an essential step. A lot of research concentrated on cooperation on different levels like medium access [3], duty-cycling [4] or routing [5]. In this paper, we propose a technique to overcome the code size constraints of existing IP-Stacks for WSNs using techniques from the field of remote procedure calls (RPC). In this model, neighbouring nodes share parts of their communication stack with each other in a cooperative fashion to enable the use of IPv6 on every node without the need to include the full stack on each node. Through the evaluation of our prototype implementation, we show the feasibility of this approach by comparing it to a native IPv6 stack on the same platform.

Stack	Os	Platform	Flash	RAM
6lowpancli	TinyOs	MSP430	21.0 KB	3.0 KB
B6loWPAN/blip	TinyOs	MSP430	25.0 KB	3.2 KB
SICSlowpan	Contiki	MSP430	40.0 KB	4.5-5.0 KB
iSense6LoWPAN	iSense	JN5148	50.2 KB	4.4-5.9 KB

Table I
Comparison of Code Size (Flash) and RAM Usage [1] and [2].

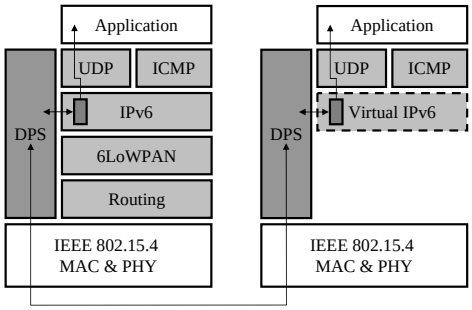

Figure 1. DPS IP stack

II. Concept

A protocol stack consists of a number of layers in a standardized order (cf. OSI and TCP/IP model). Each layer is designed with a single purpose, which – in combination with other layers – allows applications on different hosts to exchange data over a network. Data, which is sent from one application to another, traverses the stack downwards on source nodes and upwards on destination nodes, thus requiring each node to implement every layer.

Figure 1 shows the model proposed in this paper. In this example, the left node implements the full protocol stack, while the right node implements only part of it: application and transport layer (UDP and ICMP) followed by a virtual IPv6 layer. Both nodes have an additional stack element: the DPS (Distributed Protocol Stack), which spans all of the software-implemented protocol layers and which is used to share implementations of layers between nodes. The right node uses the IPv6 layer (and all subsequent layers) of the left node to send and receive IP packets to/from other hosts

in the network.

Both IP layers include a link to the DPS component, which forwards packets to the IP layer on a neighbouring node if necessary, or otherwise uses the remaining elements of the local stack. In the following, we use the terms *server* and *client* for the nodes which are connected via the DPS, where servers are nodes offering the implementation of a specific layer to clients. The DPS thus acts as a bridge between clients and servers and allows clients to use the required layers of servers.

It performs a similar task as an Object Request Broker (ORB, cf. CORBA) or a message broker in message-oriented middleware systems. The virtual IP Layer in the example above can also be seen as the *client stub* of the IP Layer, which communicates the *server skeleton* over the network just like RPCs or Remote Method Invocations (RMI) in distributed computing environments. We will describe the DPS protocol in the following section in more detail.

III. Protocol

This section gives an overview on the protocol used by the DPS components on the server and the client. We begin by describing the packet format, followed by the process used by clients to discover the protocols offered by neighbouring servers, the binding protocol and how RPC messages are exchanged once a connection has been established.

A. Packet Format

The general packet format of the DPS protocol consists of the DPS header and payload. The header includes fields for the source and destination of the message, message type, message counter and protocol identifier. The message type indicates which of the different DPS message types (e.g. Discovery) is contained in the payload and contains flags that indicate whether the message uses fragmentation or acknowledgements. The counter is used to map a request to a response, to identify fragments of the same packet, and to match acknowledgements to specific packets. The protocol identifier is chosen according to the protocol numbers from the IPv6 next header field as defined by the Internet Assigned Numbers Authority (IANA). In the case of fragmented messages, the header is followed by a fragmentation header, which consists of two fields indicating the length of the non-fragmented complete message the offset of the current fragment in the message. Each fragment contains the full DPS and fragmentation header.

B. Discovery and Binding

When a node initializes its protocol stack, it checks which skeletons and stubs it is using. For each skeleton, nodes listen for incoming *DISCOVERY* messages and respond to them with *ADVERTISE* messages. For every stub, nodes send out *DISCOVERY* messages via broadcast to their one hop neighbourhood. To establish a connection, the client starts a three-way handshake by sending a *CONNECT* message to the server. The server may then answer with an *ALLOW* or an *ABORT* message, depending on its decision to allow the connection to be established or not (e.g., if the server does not have enough resources to serve another connection). The client may then reply to an incoming *ALLOW* message with either a *FINISH* or with an *ABORT*, e.g., if the handshake has taken too long and the client has meanwhile established a connection to another server. Once the handshake has completed, both server and client may start by exchanging DPS messages.

C. RPC messages

The RPC calls in the DPS protocol follow a request-response pattern, in which a RPC request is answered with an RPC response by its communication partner. The RPC message is sent using one or more fragments , where the first byte of the first fragment encodes the destination function addressed by this RPC call, selecting the target function inside the stub or skeleton (identified by the protocol identifier), which shall be called at the receiver and which will un-marshal the payload.

If reliability in the message exchange is requested through the ACK-bit in the TYPE field of the header, each fragment must be answered by the receiver with a RPC ACK, which includes the same message counter and fragmentation header as the request, while the payload remains empty. If the function call of the request produces a return value other than void, the receiver must answer with a corresponding RPC response message, including the same connection counter.

IV. Evaluation

As a first evaluation scenario, we have implemented the DPS protocol for using a virtual IPv6 layer for the iSense WSN platform. The client stub offers the RPC functions $send$ and $receive$, each accepting an IPv6 packet as the only parameter and the $setIPAddr$ function which accepts an IPv6 address and the networks prefix lengths, while the server skeleton implements the corresponding functions. After the client has successfully established a DPS connection to a server, the server determines the clients IPv6 Address by using the client MAC Address and afterwards calls the $setIPAddr$ function of the client, telling the client its IPv6 Address and the network prefix.

The server treats all IPv6 packets received by its skeleton like regular IP packets and decides to either process them if its IPv6 matches the address in the destination field or forwards it otherwise. If the server has no routing entry to the destination in its forwarding table, he calls the $createRoute$ function of its routing layer as if he would be the originator of the packet. If the server receives an IPv6 packet, whose destination address matches one of his clients, he forwards the Packet to the client. If the server receives a routing message which tries to establish a route to the client, he

	Native IPv6 (JN5139)	DPS client (JN5139)	DPS server (JN5148)
Application	5.5 KB	5.5 KB	3.8 KB
Os	43.9 KB	43.9 KB	36.4 KB
IPv6 Stack	50.2 KB	10.1 KB	27.6 KB
DPS	–	13.6 KB	8.0 KB
$\sum$ Program	99.6 KB	73.1 KB	75.8 KB
Heap (used)	–	3.3 KB	5.1 KB
Heap (free)	–	8.8 KB	39.8 KB

Table II
CODE SIZE AND RAM USAGE FOR THE DPS SERVER AND CLIENT

answers on behalf of the client and acts as if the routing messages got forwarded by him to the client.

A. Code size and memory usage

The data in Table II shows that the native IPv6 application on the iSense JN5139 platform would require 99.6 KB of program memory and therefore does not meet the available resources of only 92 KB on this platform. The DPS client on the other hand, only consumes 73.1 KB of program memory, since we were able to reduce the size of the IPv6 stack to 10.1 KB by only including ICMP, UDP and the IPv6 stub plus 13.6 KB needed for the DPS protocol. This adds to a total size of 23.7 KB for the IP/DPS stack, thus lowering the code size requirements for this component by 52.8%.

On the DPS server side, the total code size requirements only add up to 75.8 KB, caused by the updated compiler used for the JN5149 platform. The DPS protocol increases the code size requirements of the stack by 8.0 KB to a total of 35.6 KB which is a moderate increase by only 28.9%, compared to the 52.8% reduction for the DPS client. The memory usage on the heap for the DPS client is low enough to provide 8.8 KB of free memory for the application and during sending and receiving of IP message using the DPS protocol. This shows that although the DPS protocol takes up a considerable amount of memory, it is still beneficial to use it on the target platform. We will continue to optimize our implementation of the DPS protocol to further decrease its code size and memory consumption to increase its advantage in this field.

B. Round trip time

To evaluate the round trip time (RTT), we use the server node to send ICMP Echo Request messages to one of its neighbouring nodes on the network.

First, we sent these packets to a neighbouring server node, using the native IPv6 stack via 6LoWPAN and then sent echo requests to one of the attached client nodes using the DPS with and without acknowledgements. Figure 2 shows the result of the experiments for different ICMP payload sizes and the resulting RTT. The results were obtained by sending 100 ICMP echo request message for each payload size from the server to the target node. The curve in Figure 2 shows the median value of the evaluated RTT, while the colored margins span from the corresponding upper quartile to the lower quartile. The results show that the RTT linearly increases with increasing ICMP payload sizes and additionally increases in periodic intervals. These periodic increases are caused every time the maximum fragment size of the underlying fragmentation algorithm is reached, resulting in an additional fragment to be sent.

In the native IPv6 case, the average time for sending a packet increases by ≈0.5 ms every 8 byte due to the available bandwidth and increases by an additional amount of ≈8 ms for every fragment. As depicted in Figure 2, the use of the DPS protocol increases the RTT by an offset of ≈5 ms for shorter payload lengths, which increases to up to ≈14 ms for 512 byte. Whereas the absolute offset increases for longer payload lengths, the relative offset halves from 30% down to 15%. The initial offset between the native and DPS results is caused by the missing compression of the 6LowPAN Layer in the DPS protocol, which adds 40 byte for the uncompressed IPv6 header and 4 byte of the uncompressed ICMP header. The RTT increase for each additional fragment for the DPS messages is caused by the fragmentation overhead of the DPS messages, which amounts for 14 byte per packet, while the overhead of the 6LoWPAN fragmentation header is only 5 byte per additional fragment.

Since neither IP nor ICMP offers any reliability features, this would be the default mode of operation for the DPS protocol when exchanging IP packets, but since reliability is an important feature for remote procedure calls such as the abovementioned $setIPAddr$ function, we have evaluated the impact of acknowledgements on the RTT. As shown in Figure 2, the use of acknowledgements has a stronger effect on RTT than the use of DPS alone and adds another 15 ms per fragment, effectively doubling the RTT for longer payload sizes in relation to the native IPv6 case. This increase is caused by the acknowledgement mechanisms, in which the originator of the DPS message waits for the corresponding ACK to arrive from the destination before sending the next fragment. Each ACK has a length of 14 byte and causes one additional clear channel assessment to be performed.

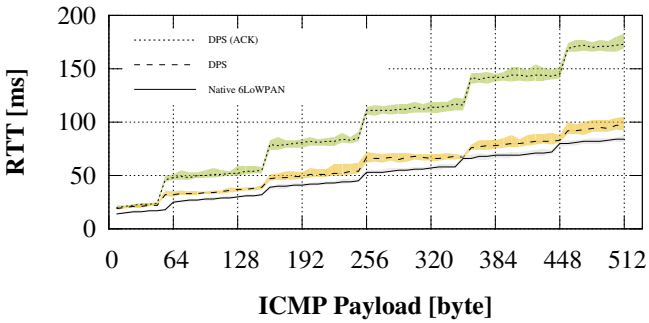

Figure 2. Singlehop RTT: Native 6LoWPAN (bottom) vs. DPS (top)

V. Related Work

There are existing general RPC frameworks for WSN, most notably the TinyRPC [6] and SpartanRPC [7] for TinyOS. Both RPC designs use extensions for the syntax of the nesC programming language, which enable the use of asynchronous remote interfaces and automatically perform the steps of marshalling and unmarshalling. While TinyRPC does not feature any security mechanisms, SpartanRPC uses AES for authentication. Other RPC designs for WSN include Marionette [8] for remote analysis and debugging and the (sec)Fleck [9] operating system.

There is another framework, also called Distributed Protocol Stacks [10], aimed at detaching atomic functional blocks from a particular layer and moving it to a different location in the network. The DPS from [10] thus does not detach the complete layer of one node to one of the neighbouring nodes in the network, but detaches a single atomic function (e.g. the ACK mechanisms of TCP) of all nodes in the network to one or more specific positions in the network (e.g. the border router/gateway). This is done to enhance the TCP performance between an external node and nodes in a wireless network by generating the ACKs at the border router, whereas using network-specific forwarding and reliability features in the wireless network, thus increasing both RTT and goodput.

VI. Conclusion

In this paper we propose the concept of Distributed Protocol Stacks, a novel method of using cooperation between neighbouring nodes. The DPS enables the use of IPv6 on devices without the need of implementing the full protocol stack on every node while preserving transparent IP interconnectivity. We present a specification of the DPS protocol (including discovery, handshake, and request/response-based RPC with fragmentation and reliability mechanisms). We have evaluated our first implementation using an experimental deployment regarding code size, memory usage and round-trip time and have compared the proposed method against a native IPv6 implementation on the same hardware and software platform. Our evaluation shows the feasibility of our approach as it reduces the communication stack code size for the Client node by 52.8% at the cost of an increase of the RTT by 15% to 30%.

Future work will include additional evaluations and optimizations (e.g., better fragmentation mechanism with less overhead, compression and security mechanisms). We will implement additional stubs and skeletons (e.g. routing protocols). We will further evaluate which additional benefits can be generated by the use of DPS, which may lead to an increase of fault tolerance and load balancing by distributing the layers over multiple nodes, which limits the effect of programming errors and security vulnerabilities to the nodes implementing the vulnerable layers. Furthermore, surrounding servers which implement the same layer may identify malicious behaviour or high load of servers and advertise themselves as an alternative to the affected client.

References

[1] Ricardo Silva, Jorge Sá Silva and Fernando Boavida, "Evaluating 6LoWPAN implementations in WSNs," Department of Informatics Engineering – University of Coimbra, Tech. Rep., 2009.

[2] Peter Rothenpieler, "Poster: Distributed Protocol Stacks for Wireless Sensor Networks," in *Wireless Sensor Networks - 9th European Conference, (EWSN)*, Trento, Italy, 2012.

[3] M. R. Ahmad, E. Dutkiewicz, and X. Huang, "Performance evaluation of mac protocols for cooperative mimo transmissions in sensor networks," in *Proceedings of the 5th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, ser. PE-WASUN '08. New York, NY, USA: ACM, 2008, pp. 54–62. [Online]. Available: http://doi.acm.org/10.1145/1454609.1454623

[4] C. Gui and P. Mohapatra, "Power conservation and quality of surveillance in target tracking sensor networks," in *Proceedings of the 10th annual international conference on Mobile computing and networking*, ser. MobiCom '04. New York, NY, USA: ACM, 2004, pp. 129–143. [Online]. Available: http://doi.acm.org/10.1145/1023720.1023734

[5] C. Dominguez and N. Cruz-Cortés, "Energy-efficient and location-aware ant colony based routing algorithms for wireless sensor networks," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 117–124. [Online]. Available: http://doi.acm.org/10.1145/2001576.2001593

[6] T. May, S. Dunning, and J. Hallstrom, "An rpc design for wireless sensor networks," *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, vol. 0, pp. 8–138, 2005.

[7] P. C. Chapin and C. Skalka, "Spartanrpc: Secure wsn middleware for cooperating domains," in *MASS*, 2010, pp. 61–70.

[8] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: using rpc for interactive development and debugging of wireless embedded networks," in *In IPSN 06*, 2006.

[9] W. Hu, P. Corke, W. S. Shi, and L. Overs, "secfleck: A public key technology platform for wireless sensor networks," in *Wireless Sensor Networks, 6th European Conference (EWSN)*, Corke, Ireland, February 2009, pp. 296–311. [Online]. Available: http://eprints.qut.edu.au/33731/

[10] D. Kliazovich and F. Granelli, "Distributed protocol stacks: A framework for balancing interoperability and optimization," in *Proceedings of the IEEE International Communications Workshops, 2008*, ser. ICC Workshops '08. IEEE Press, 2008, pp. 241–245.