ProtoMesh

A new approach to communication

Blechschmidt, Til
til@blechschmidt.de

Peeters, Noah
noah.peeters@icloud.com

Brandt, Merlin
merlin.brandt@hotmail.de

December 14, 2017

Abstract

Most modern communication is running on top of the TCP/IP stack. It is the building block for the Internet as we know it today but since the stack was created the type of devices that use it shifted from stationary, big computers that are directly wired together to mobile, small devices and smart-phones. Most manufacturers made use of the existing infrastructure to connect these devices with each other and labeled it the Internet of Things. The devices changed but the stack persisted mostly in its original form. This white-paper aims to provide a new solution which takes the building blocks from the existing stacks and builds upon them to provide a better way for portable and small devices to interact with each other. It is crafted with the principles like security, anonymity and high performance at its core.

This is *not* supposed to be replacing the TCP/IP stack since it aims at a drastically different crowd of devices that will utilize it but rather use the existing stack as one possible transmission medium for communication between devices. In a nutshell this provides a layer of abstraction for the communication between devices regardless of the medium and proposes a common way for devices to interact with each other in a unified language.

Contents

List of schemes

For the most up-to-date list of schemes please refer to the GitHub repository.

Section 1

Communication

On the current market there are countless solutions for communication like Ethernet, WiFi, Bluetooth, Z-Wave or ZigBee just to name a few. Most of the time one of the latter is used to cover the last mile to a device whereas a specialized intermediate device translates it into one of the former. This results in situations where unnecessary long and indirect routes with translations along the way are used between devices that are just metres apart. Since many devices are already capable of more than one of the aforementioned carriers it only makes sense to extend on those capabilities.
To remove the need for translation, inefficient routes and specialized bridging devices a protocol that is carrier independent and unified is required. Such a standard would bridge the gap of communication between devices with different capabilities and enable *any* device with more than one carrier to act as a bridging device. It provides the foundation for uniform interaction between different nodes from different manufacturers with different capabilities. Through such a solution the so called ”cloud” will be omnipresent. All devices shape one big cloud with localized communication.

1.1 Transmission medium

As previously mentioned there are many standards for data transmission between devices and each of them has its own API and caveats. The protocol discussed in this white paper is not meant to be a replacement for those but rather an additional layer which only requires the underlying medium to support at least one to one communication and optionally broadcast transmissions. It abstracts the foundation of interaction between devices away providing a stateless, decentralized network of arbitrary size in which devices can interact.

1.1.1 Ethernet

1.1.2 Bluetooth

1.2 Data format

1.2.0.1 Serialization

All messages transmitted are assembled and disassembled using binary serialization in the form of scheme based Flatbuffers [1]. Excluding the routing layer compression can be used if advertised in the message wrapper.

1.2.0.2 Recursion

All datagrams are processed recursively. For example a MessageDatagram (1.5) might contain a RouteDiscoveryDatagram (1.4.2) so the intended recipient unwraps the MessageDatagram and processes the DiscoveryDatagram as if it was received directly.

1.3 Node representation

Each participant in the network is represented by a Universally Unique Identifier which is directly linked to a asymmetric key pair. In order for the UUID to be unique and representative it is provided by the alliance to hardware manufacturers and hobbyists in the form of address spaces. When a new manufacturer wants to use the protocol he requests an address space with the approximate amount of addresses he will likely utilize. The alliance then assigns the manufacturer a specific prefix which has a variable length depending on the volume of devices due to be produced and the manufacturer can then assign UUIDs within this space at his own discretion. This ultimately makes the already very unlikely scenario of ID collisions even less probable.

1.4 Routing

With the foundation of a unified way of transmitting messages between devices regardless of the medium a network which previously consisted of a central gateway and clients enslaved to it became state- and shapeless. Nodes may come into range or connect and others may leave which poses a special challenge for routing messages between devices.

In order to cope with the dynamic architecture and possibly vast size of the network the Zone Routing Protocol [2] is used. As opposed to proactive routing which is common in most networks ZRP is a hybrid approach that utilizes both. The network is split in zones of a specific size where s_{zone} is the amount of hops the local zone spans. Take into consideration that the zones are defined for every node. Each node has his own zone which overlaps with the zones of other nodes.

1.4.1 Intrazone Routing Protocol

For all nodes within the local zone - reachable within s_{zone} hops - proactive routing is used based on advertisements. Each devices advertises its presence to neighboring devices by broadcasting a datagram according to Listing 1 at a regular interval. When a Node receives an advertisement it stores the path the

```
table AdvertisementDatagram {
    // ***
    // * Identification of a device
    // * Consisting of the UUID and the public key of the advertiser.
    // ***
    uuid: UUID;
    pubKey: PublicKey;

    // ***
    // * List of nodes this particular datagram passed through.
    // * Utilized to build a routing table for the local zone.
    // * Datagram is discarded if the list exceeds the zone size.
    // ***
    route: [UUID];

    // ***
    // * Signature
    // * Created by calculating the SHA512 hashes of:
    // * - uuid
    // * - public key
    // * and applying SHA512 on the concatenated hashes which then gets signed.
    // ***
    signature: [ubyte];

    // ***
    // * Advertising interval
    // * Used to detect stale routes.
    // ***
    interval: uint;
}
```

Listing 1: IARP advertisement

datagram has taken (as stored in the route property) in a local routing table and thus knows a route to the advertising node. Since routes might go stale the datagram also contains the maximum interval at which the devices advertises. Is no advertisement received within this time period of the last one the route will be considered stale.
In addition to storing the datagram locally it is broadcasted to all neighbors. If the underlying transmission medium permits it the origin of the advertisement is excluded from the broadcast. Upon broadcasting the node appends its UUID to the route property.

1.4.2 Interzone Routing Protocol

In order to discover routes to nodes not within the same zone ($hopCount > s_{zone}$) the already existing routing tables of nodes outside the own zone can be utilized. In order to discover a route the source dispatches a specific datagram designed for this task as seen in Listing 2. The datagram will be sent to all nodes that are exactly s_{zone} hops away and thus on the border of the local zone. This process is known as Bordercasting. To prevent loops the datagram

```
table RouteDiscoveryDatagram {
    // ***
    // * Nodes the datagram has already covered.
    // * Upon relaying all target nodes are added to this list prior to dispatch.
    // * During dispatch these targets are skipped since they received it already.
    // ***
    coveredNodes: [UUID];

    // ***
    // * Identification of origin/destination
    // * Used to determine where the datagram should go.
    // ***
    origin: PublicKey;
    destination: UUID;

    // ***
    // * List of nodes traversed
    // * Keeps track of the route to be used.
    // ***
    route: [UUID];

    // ***
    // * Unix timestamp
    // * Used to calculate the travel time this route takes.
    // * Utilized to determine a timeout value for this route.
    // ***
    sentTimestamp: long;
}
```

Listing 2: IERP route discovery

contains all nodes it was dispatched to. During dispatch targets that are in this list are skipped and prior to dispatch all pending targets are added. Once a relaying node has the destination in its routing table it will be directly sent to it and further Bordercastings is stopped. Should the length of the route exceed a maximum value of r_{max} then the datagram will be discarded and a DeliveryFailureDatagram (4) is sent back along the route to the origin.

1.5 Message transmission

In order for IERP to work the RouteDiscoveryDatagram needs to be relayed via multiple nodes to the next border node according to the IARP routing table. This and further use of the discovered route require a special datagram type (refer Listing 3) designed to deliver a payload along a previously discovered route to a destination. This datagram contains the route to take where each

```
table MessageDatagram {
    // ***
    // * List of nodes
    // * Path this datagram should traverse.
    // * Contains origin at the beginning and destination at the end.
    // ***
    route: [UUID];

    // ***
    // * Payload data
    // * To be delivered to the destination.
    // * Encrypted with the shared secret between the two involved parties.
    // * Non-encrypted payloads get rejected.
    // ***
    payload: [ubyte];

    // ***
    // * Signature
    // * SHA512 hash of the payload which then gets signed.
    // ***
    signature: [ubyte];
}
```

Listing 3: Message delivery

consecutive node it should traverse through is s_{zone} from the previous and the traversal to the next border node is at the discretion of the relaying node and its routing table. The payload is encrypted with the shared secret of the two involved parties. Note that non-encrypted payloads are to be rejected. Additionally the message is signed with the private key of the sender as a proof of integrity and to prevent tampering along the route.

1.5.1 Delivery failures

It might happen that delivery to a destination is not possible due to a stale route or an unresponsive target. In this case the node which notices discards the original datagram and transmits a DeliveryFailureDatagram as described in Listing 4 back along its path encrypted for the original author of the datagram that couldn't be delivered. Note that the failure of delivering such a datagram may not result in the dispatch of another DeliveryFailureDatagram.

```
table DeliveryFailureDatagram {
    // ***
    // * List of nodes
    // * Path this datagram should traverse.
    // * Contains origin at the beginning and destination at the end.
    // ***
    route: [UUID];

    // ***
    // * ID of the target to which delivery failed
    // ***
    originalRecipient: UUID;
}
```

Listing 4: Delivery failure

1.6 Communication workflow

In order for two parties to establish secured communication through a network of intermediate relays (if required) the following steps are required:

1. Origin checks whether or not the destination is within its zone. If that is the case bordercasting is skipped and the flow continues with (B)
2. Origin bordercasts a RouteDiscoveryDatagram which propagates through the network according to section 1.4.2 until either:
   (A) Its route exceeds the maximum route length and gets discarded along the way
   (B) It reaches the target zone where it is directly forwarded to the destination
3. Destination generates a shared secret from the public key inside the datagram and dispatches a symmetrically encrypted response along the route determined by the RouteDiscoveryDatagram.
4. Once the origin receives it the route is stored and further encrypted communication may start.

Section 2

Interaction

With the solid foundation built by the abstracted communication layer a unified and flexible way for devices to interact is necessary to preclude the emergence of proprietary ways for communication between devices which would result in a bulk of entities enclosed in their own environment only capable to talk to their siblings commonly known as IoT segregation.

2.1 How to communicate

In the following sections there are multiple datagrams described for devices to interact. Since this describes a new layer to the stack - the interaction layer - all the following datagrams are wrapped inside a MessageDatagram which contains the data necessary to be delivered to the other party where it is unpacked and processed. Devices (2.2) never directly interact with MessageDatagrams but rather pass their interaction-level datagrams together with their destination device's UUID on to the communication layer which then wraps and dispatches it possibly discovering a route in the process.

2.2 Devices

One node on the communication layer may represent a singular device on the interaction level and both share the same UUID.
Nodes are aware of each other through advertisements which enables routing but in order for interaction to take place it is required for a node to know the interface of a device in order to talk to it. To provide meta data of a device including its name and the available endpoints each device is required to have a special endpoint of the *Metadata* type at ID **0**.

2.2.1 Endpoints

Each device has an interface through which other devices can interact with it. It is comprised of multiple endpoints which are defined in Listing 5. Every endpoint has a type which is defined through a magic number listed in the *EndpointTypeID* enum. Those numbers can be looked up in the online type database where each type identifier correlates to a specific interface.

```
// ***
// * Magic numbers for endpoint types
// * Meaning is defined in online database.
// ***
enum EndpointTypeID : long {
    Metadata = 0,
    Color = 1,
    Temperature = 2,
    Brightness = 3,
    Authorization = 4,
    ...
}

table Endpoint {
    // ***
    // * Available functions
    // * IDs of the functions from the type this endpoint supports
    // ***
    availableFunctions: [ubyte];

    // ***
    // * Type identifier
    // * Uniquely identifies the endpoints type.
    // * Implicitly defines the available functions.
    // ***
    type: EndpointTypeID;

    // ***
    // * Identification for the endpoint
    // * Unique within the device.
    // ***
    identifier: uint;

    // ***
    // * Name of endpoint
    // * Can be set through the special set_name(name: string) function.
    // ***
    name: string;
}
```

Listing 5: Endpoint definition

2.2.1.1 Endpoint type defintions

The communication between nodes is type safe. That means every function has predefined arguments and return values. Every type has a unique identifier which can be used to reference it and a scalar data type to define the serialization. Additionally a human readable explanation on how to interpret the value is provided.

```
brightness_t: float // 1.0 = full on, 0.0 = off
temperature_t: float // temperature in Kelvin
state_t: bool // true = on, false = off
RGB: {uint8, uint8, uint8} // r g b; 255 = full on, 0 = off
HSV: {uint8, uint8, uint8} // h s v; 255 = full on, 0 = off
```

Endpoint Type		**Function**		**Parameters**	**Return Type**
ID	Name	ID	Name		
2	Temperature	0	set_temperature	temperature_t	void
		1	get_temperature		temperature_t
3	Brightness	0	set_brightness	brightness_t	void
		1	get_brightness		brightness_t

2.2.1.2 Remote Procedure Call

In order to invoke a endpoint it is necessary to make a RPC. To start such a process the callee issues a function call as defined in Listing 6. In case the

```
table FunctionCall {
    // ***
    // * Endpoint identifier
    // * Unique to the device.
    // ***
    endpointID: ushort;

    // ***
    // * Function identifier
    // * Unique to the endpoint.
    // * Implicitly defines the parameter and return value serialization.
    // ***
    function: ubyte;

    // ***
    // * Transaction identifier
    // * Used to correlate to the response.
    // ***
    transactionID: ubyte;

    // ***
    // * Serialized parameters
    // ***
    parameter: [ubyte];

    // ***
    // * Signature
    // * Used to determine permission.
    // ***
    signature: Signature;
}
```

Listing 6: Function call

function signature contains a byte stream the parameter property of the call only contains the non-stream entries. After the initial dispatch of the function call the device may then send consecutive StreamData (Listing 7) upon reception of a corresponding ok-response as defined in Listing 8 with the same transactionID.

The receiving device may only answer to StreamData in case the status code contains an error value. Unless the sender received a response with a non-ok error code it is assumed that the StreamData got transmitted successfully.

Once the server received a function call it executes the function and dispatches a response with the same transaction ID as well as the status and the return

```
table StreamData {
    // ***
    // * Transaction identifier
    // * Used to correlate to the request.
    // ***
    transactionID: ubyte;

    // ***
    // * Payload
    // * Contains the actual data.
    // ***
    payload: [ubyte];
}
```

Listing 7: Streaming data

value.

```
table FunctionCallResponse {
    // ***
    // * Transaction identifier
    // * Used to correlate to the request.
    // ***
    transactionID: ubyte;

    // ***
    // * Status of the call
    // * Used to determine whether or not the call succeeded.
    // ***
    statusCode: ubyte;

    // ***
    // * Serialized return value
    // ***
    returnValue: [ubyte];
}
```

Listing 8: Function call response

Status Code		Description
Base 10	Base 5	
0	000	Successd sajdf asdkfj aksld fjaks fksdl
25	100	Success

2.3 Users

Since a user may interact with devices it is required to represent a user entity within the network. This is done by generating a private key for the user based on random data. The user should then copy the key possibly through a file sharing endpoint between applications that allow the user to interact with the network and act as a proxy between the user and the network. Note that like devices users are participants in the network. They may have their own communication layer which uses their ID and public key. Unlike devices though users usually don't have endpoints.

Acronyms

IARP Intrazone Routing Protocol. 4, 6

IERP Interzone Routing Protocol. 5, 6

RPC Remote Procedure Call. 12

UUID Universally Unique Identifier. 2, 4, 9

ZRP Zone Routing Protocol. 3

Glossary

Bordercasting Relaying a message to all nodes that are on the border of the local zone and thus s_{zone} hops away. 5

datagram Atomic unit of transmission with sufficient information to be routed to its destination through a network. 2, 4–7, 9

device Entity which consists of endpoints. 8, 9

node Physical device and participant in a network. 4, 9

Bibliography

[1] Google Inc. Flatbuffers, memory efficient serialization library.

[2] University of Leeds Nikos Drakos, Computer Based Learning Unit. Cs765 - mobile ad-hoc networks, analysis of the zone routing protocol.