Service for MANET node self-configuration and communication

Eduardo Soares, Ana Aguiar, Pedro Brandão, Rui Prior

Service for MANET node self-configuration and communication

Eduardo Soares*‡, Ana Aguiar*‡, Pedro Brandão†‡, Rui Prior†‡
*Faculty of Engineering and †Faculty of Sciences, University of Porto,
‡Instituto de Telecomunicações, Porto
esoares@dcc.fc.up.pt, aana@fe.up.pt, pbrandao@dcc.fc.up.pt, rprior@dcc.fc.up.pt

***Abstract*—Setting up a MANET is a complex task, involving the configuration of a routing protocol and the network parameters of the nodes, since there is no centralized point for distribution of IP addresses. This complexity is further compounded by the absence of a name resolution system. We describe NetService, a service that frees application developers and deployers from this burden. NetService runs on the nodes and auto-configures a predefined MANET at the request of applications. It handles IP address assignment and includes a name resolution mechanism that provides a hierarchical naming scheme and is compatible with a standard resolver. The service provides an API to configure the nodes and another one for exchanging messages, abstracting applications from the details of network access.**

***Keywords*—MANET, auto-configuration, middleware.**

I. Introduction

A Mobile Ad-Hoc Network (MANET) is a particular type of network where nodes, which may be mobile, form ad-hoc connections between them without any centralized coordination or *a priori* knowledge of topology or even existing nodes.

The need for MANETs arises in many scenarios where a network infrastructure is not available, is infeasible or not practical to deploy, or cannot be trusted. Such scenarios include poorly connected places in remote areas (forests, ocean, etc.), military operations, natural or man-originated catastrophes [1], [2], or social upraise events [3]. MANETs may also be useful for group communication in mass events, like conferences or concerts [4], where the infrastructure may be unable to support all communication demand. In our particular case, we need MANETs for communicating in first responder scenarios like fire events, where cellular networks are not always available and time constraints prevent the deployment of a network infrastructure. Nonetheless, the developed framework is entirely general and may be used in any MANET use case.

The firefighters' scenario addressed in the research project that funded this work[1] is a particularly good illustration of what could be described as a MANET. The nodes are grouped in small groups (firefighters) which form an initial network and then move to the operation field, finding and using other nodes to route packets and extend the network. There is no predefined traffic pattern, with communication happening from nodes within a group to a sink (sensors at each node sending data to the team-captain), duplex communication between nodes to check on their status, and even broadcast/communication to nearby nodes to check on them or ask for help. This also illustrates the use of unreliable and reliable communications, where some sensor data could be lost but packets to check on other nodes' health is highly important.

Configuring a MANET node is considerably more complex than connecting a node to an infrastructured wireless network. Address configuration is the first problem, as nodes that enter the network must be assigned an Internet Protocol (IP) address that does not collide with any other node in the MANET. The second problem is providing multi-hop communication. Ad hoc networks (e.g., IBSS mode of 802.11), by themselves, usually provide only single hop communication, where a node may communicate only with other nodes within its radio range. Multi-hop communication is made possible through the use of specific MANET routing protocols, allowing nodes to discover other nodes and routes to them outside their radio range.

The usual absence of hierarchy or structure in MANETs makes the use of a central coordinator undesirable. Protocols that elect a central coordinator can be used, but add extra traffic and delays, making the solution inefficient in terms of overhead. As nodes lose communication with each other, paths are torn down and node islands (partitions) emerge. This partitioning of the network and remerging when connectivity is reestablished causes issues with the election of a single centralized coordinator. Moreover, these mergers are delayed until an election protocol finishes, limiting communication between the groups in the meantime. Avoiding a centralized service can thus be more efficient and scalable, and should be preferred whenever possible.

In an effort to make it simpler to (1) configure and manage nodes for MANET communication and (2) develop applications that communicate over the network, we created *NetService*. NetService is responsible for setting up the MANET and ensuring all necessary bits and pieces (like the appropriate ad hoc routing protocol) are up and running. It offers applications an Application Programming Interface (API) that abstracts network communication (transport layer) to a rich and flexible message-based interface and also provides useful feedback and network information (like current topology).

The rest of the paper is organized as follows. In the next section we review the literature on some issues addressed by NetService. In section III we present NetService, describing the solutions used to deal with the different issues along with contextual aspects that motivated their choice. In section IV

[1] VR2Market http://vitalresponder.inesctec.pt/

ISBN 978-3-903176-03-4

we describe the API offered by NetService to the applications and the network packet format. Implementation aspects of NetService are discussed in section V. In section VI we analyse the solutions adopted for NetService, from both a qualitative standpoint, through comparison with alternative solutions, and from a quantitative standpoint, from experimental results. We conclude the paper with some remarks and the discussion of a few pointers for future improvement of this work.

II. Related Work

Creating a MANET demands coordination between all the nodes. They need to use the same network stack, physical layer, data link layer with the same network configuration (name, physical layer frequency, shared key if needed), an IP layer without conflicting IP addresses, compatible configurations, packet forwarding, and a common routing protocol.

The Ad-Hoc Network Autoconfiguration (AUTOCONF) working group of the Internet Engineering Task Force (IETF)[2] provided guidelines for IP address configuration in ad hoc networks in RFC 5889 [5]. Other relevant RFCs concerning IP address assignment applicable to MANETs is RFC 4193 [6], which defines local unique unicast addresses for IPv6, with a well-defined prefix, a configured global ID and a part defined by the interface identifier. RFC 4862 [7] defines auto-configuration for IPv6 stateless address, also having a part defined by an interface identifier and a well-known prefix.

Gilabert and Herrero [8] explored the problem of IP address allocation in MANETs, and used the IP range 169.254/16 despite RFC 3927 [9] stating that this address range should not be routable, which may create problems for multi-hop communication in the MANET.

MANETconf [10] is a proposed solution to the IP address allocation, where when a node enters a MANET it asks another node in its vicinity for an IP address. To ensure uniqueness, all nodes maintain a list of IP addresses they know are already allocated or in process of allocation. A node requesting an IP address (*requester*) uses a neighbour with an already allocated address (*initiator*) to flood the network with a message asking for known conflicts with the tentative address. All nodes must respond to the *initiator* after checking their list. If the address is already in use, a new one must be selected. If it is in process of allocation by other node, the *initiator* with lower IP address keeps the allocation. Network partitions are not a problem. Mergers are resolved by comparing the list of allocations between the two nodes that notice the merger and then asking the conflicting nodes to change their IP address.

This solution works perfectly with multiple nodes requesting the same IP address and multiple nodes doing IP address requests in the network because of the conflict resolution strategy, but is quite intensive in terms of network traffic and takes a long time to assign an address. Even without conflict, it comprises a broadcast from the *requester* to its neighbours to find the *initiator*, the flooding of the tentative address, responses from all nodes to the *initiator*, and the confirmation to the *requester* that the address may be used. This aspect limits its ability to scale to larger-sized MANETs.

[2]https://datatracker.ietf.org/wg/autoconf/about/

A. Transmission Control Protocol (TCP) in MANET

Communication between machines is highly standardized, with a layered organization that abstracts the underling details, like channel access or framing. Each layer provides a set of services that layers above can rely on. The most common transport protocols are TCP and User Datagram Protocol (UDP). TCP is a connection-oriented protocol that provides reliable transfer, and is used by most application protocols that need reliability, such as HTTP, FTP or SSH. TCP also implements flow and congestion control, and offers a byte stream abstraction. By contrast, UDP offers an unreliable datagram abstraction, and is used when reliability is not the main concern, like in interactive multimedia applications where timely delivery is more important than an occasional loss, or when the overhead of connection establishment cannot be justified. There are other transport protocols for the Internet stack, like Stream Control Transmission Protocol (SCTP), but they are less frequently used and not so widely deployed.

The wireless medium typically has much higher error rates than cabled media, an issue that is further compounded in MANETs by the use of multiple wireless hops and by the ever-changing topology due to mobility. The resulting packet losses are troublesome for TCP, since it interprets them as a sign of network congestion. The problem of packet loss in wireless networks has been approached in previous works. Balakrishnan et al. [11] proposed a solution for the simpler case of a wireless link between a base-station and a mobile host in an infrastructured scenario, where the base-station caches the TCP packets and performs local retransmission of lost packets.

A similar approach was also proposed by Dunkels et al. [12] for Wireless Sensor Networks (WSN). This proposal shares more characteristics with the MANETs scenario, such as the ad-hoc nature of the network, battery powered nodes and multiple wireless hops. The main difference is that packets are cached in nodes along the path between source and destination, and the retransmission of a missing packet, identified from ACKs, is done from the node closest to the destination that has the missing packet cached.

SCTP is an alternative to TCP supporting multi-homing and multi-streaming, but several studies [13], [14] point out that it provides no better performance in a MANET environment than TCP. Aydin etal [13] show that SCTP has more overhead in headers for data and the Selective ACKs (SACKs) than TCP.

III. Architecture

We designed a service (NetService) that makes it easier for applications to work in MANETs by automating the network configuration and providing name resolution and a simpler communication API. The developed software stack configures nodes to connect to a predefined Independent Basic Service Set (IBSS) (configurable) and handles the IP address distribution and name resolution.

A. Node naming

To help applications communicate between nodes, names are a must-have feature to address other nodes. We assume that each node has a unique name when it enters the network. For our firefighter scenario we took advantage of the natural hierarchical structure that groups firefighters, i.e., our nodes. Firefighters are organized in teams, each with a leader, and with several teams from each fire station. Fire stations belong to country regions, which are then grouped by country. This structure maps very naturally to a hierarchical scheme for node naming, each part of the name denoting one level in the hierarchy. For example, `johndoe.team1.firestation1.porto.pt.manet` would be the name assigned to the node held by firefighter John Doe of team 1 of the first firestation in Porto, Portugal.

With this naming scheme, it is trivial to ensure that each node has a unique name by having some control at each level of hierarchy. Other name allocation techniques could be used and should not affect anything in NetService.

B. IP address auto-configuration

Given the uniqueness of the node names, we can use them to derive algorithmically an IP address for each node, with a very low probability of collision. IP address computation is made through a mechanism similar to that defined in RFC 3927 [9], but using the hierarchical node name instead of the Media Access Control (MAC) address. The algorithm consists on:

1) Initializing a pseudo-random number generator with the full node name as input (seed);
2) Extracting 24 bits from the random number generator;
3) Using those 24 bits as the host part to create a valid IPv4 address in the 10.0.0.0/8 private addressing block.

Though the 169.254.0.0/16 (link-local) prefix is used in RFC 3927, we used 10.0.0.0/8 for two reasons. The first is that link-local addresses are non-routable, which might cause issues in multi-hop MANETs. The second and most important reason is that 16 bits provide a relatively small addressing space for hosts ($2^{16} = 65536$ addresses). This is acceptable if combined with Duplicate Address Detection (DAD), which is easy to perform on a single link. However, DAD is difficult to execute sensibly in a multi-hop MANET, particularly when network partitioning and merging may occur. The larger addressing space provided by 24 bits ($2^{24} = 16777216$ addresses) reduces the collision probability to a value sufficiently low to be acceptable, as discussed in sec. VI-A.

The objective of having nodes with unique names is to enable applications to identify a remote node with which they need to communicate by name. Since the algorithm to generate the IP address from the node name is deterministic, name resolution is a simple matter of calculating the address locally.

IV. APIs

In addition to automating the network setup procedures (routing protocol, IP addresses, etc.) and ensuring all required resources are up and running, NetService also provides two APIs that make it easy for applications to interact with the networking subsystem, one for configuration and the other one for communication. The APIs are accessed by message exchange through a set of queues in a message broker.

Each application is assigned an ID id, a well-known unique integer that distinguishes it from other applications, akin to well-known port numbers. At startup, the application registers two queues in the broker, one for data and one for control messages, named id`_data` and id`_control`. After that, applications can send messages to the NetService using two pre-existing queues named `control_data` and `network_data`, one for controlling the service and the other one to communicate with other application instances in the same or different nodes. The queues are illustrated in figure 1.

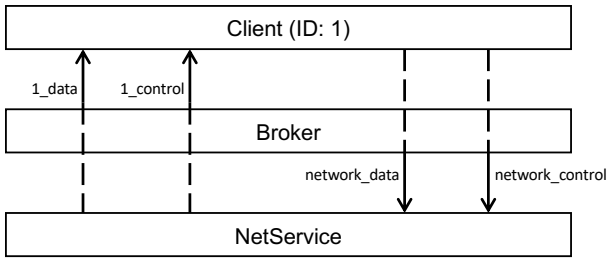

Figure 1. Queues created on the broker for an application and the NetService

A. Configuration API

The configuration API defines a set of messages to provide all the needed services to the applications: starting or stopping the network, resolving names to IP addresses, retrieving the current network status and other informations that important to applications, and to subscribe/unsubscribe from notifications of events.

Events are state changes that NetService detects and can be useful for the application to adapt itself. The events currently defined are changes to the network status (started/stopped), failure to send messages requesting reliability, and the arrival of new nodes in the neighbourhood.

Figure 2 illustrates a typical startup sequence. When the application (client of the service) is started, it creates the message queues for control and data (with `ID 1`, in this case) in the broker. It then registers itself by sending a message to the `network_control` queue. The registration process is necessary for the application to be able to receive data from the network. Afterwards, it checks whether the network is already started by another application. In this case, the network is not yet started, so the application invokes the network start command. This command requires the full name of the node (with all the previously defined hierarchical components) as a parameter. After the network is started, the application may use NetService to send and receive data to/from other nodes.

B. Communication API

The design of the Communication API was influenced by the traffic pattern to be more used in the firefighter scenario, gathering data from several sources to a sink. Nonetheless, other patterns can also be used under the same API such

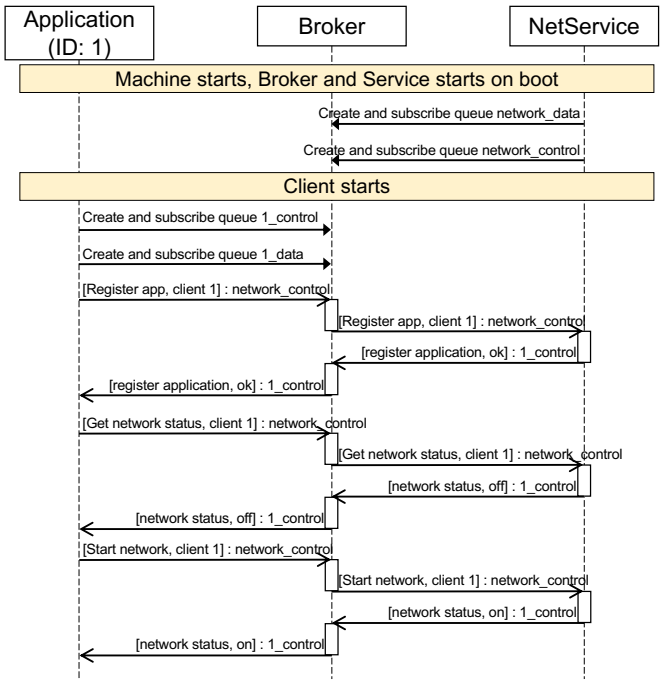

Figure 2. Interaction between API and Service

as bidirectional or broadcast messaging. The data from the wearable sensors in the nodes to the collector is sent in small chunks, as datagrams without reliability, though optionally the sender may request reliable delivery. To send a chunk of data, the application must specify the node name or IP address of the destination, the ID of the receiving application, and a service class (described below in section IV-C).

When a data packet is received by NetService, it first checks whether the application with the specified ID is registered. If it is not, the encapsulated message is discarded, otherwise it is delivered to the respective queue. If reliable delivery was requested by the sender, it also sends an acknowledgement back to the message source.

The application receiving the message is also informed of the IP address of its sender. The main reason to provide the IP address instead of the name is that it is not always possible to know the name of the source. Though the name of the source node could be added to every packet, it would add extra overhead that in general does not provide any benefit to the applications – if the receiver wants to reply to the sender, it may simply use the IP address.

Reverse name resolution is currently not implemented because we want the naming system to be lightweight and not require additional network traffic. This is not a problem in our scenario, where communication occurs mostly between known nodes (e.g. within a team). It would be, however, trivial to enrich the service with the functionality of asking the node with a given IP address for its name.

As mentioned, communication to a group of nodes (e.g. a team check request) is a traffic pattern with interest i.e. a scenario for the usage of multicast. Though a few applications would benefit from network multicast, like the previously mentioned team check, it is not yet implemented. However, group messaging is still possible by registering a group (e.g. `group.team1.firestation1.porto.pt.manet`) and specifying the list of members (e.g. `johndoe.team1.firestation1.porto.pt.manet`, `joana.team1.firestation1.porto.pt.manet`, `10.23.90.101`). Data messages are then sent by NetService in multiple unicasts to the members of the registered group.

C. Service class

Not all types of messages have the same requirements. Some need reliable delivery (e.g. team check), while for others best-effort delivery is more appropriate (e.g. readings from a thermometer), and should the network capacity be insufficient for all traffic, some might be prioritized. To cater for these diverse requirements, a service class identifier (an integer) is specified for messages sent to the network.

Currently, two service classes are supported, one for reliable and another for unreliable delivery. In the future, we intend to add more classes to support different priorities, which will be mapped to different traffic classes in the network layer and, preferably, also the link layer.

D. Data format

Messages exchanged between NetServices in different nodes conform to a specific format, which was designed to have minimal overhead, to make the most out of the available network capacity. This format is as follows: 6 bits to indicate the sender application ID, necessary for the receiving application to know where the message came from and, eventually, where to reply to; 1 bit indicating if it is a packet destined to a group; an optional 128 bits for a group hash; and the rest of the packet with variable length carrying the data.

Since group names can be lengthy, the group hash, a 128-bit hash (MD5), is used instead of the full group name to avoid excessive overhead in all group messages. When a node receives a group message, it checks this hash against the hashes of the groups of which it is a member. This optimization can provide considerable savings. In the example group mentioned in section III-B, `group.team1.firestation1.porto.pt.manet`, even if we omit the constant prefix ".group" and suffix ".manet", the name size is 216 bits (considering 8-bit ASCII names). Since the names have variable length, using a fixed size hash also saves a name size field or a terminator. Note that the MD5 hash collision probability is extremely low (low enough that MD5 hashes are used for file integrity checking), making this a reasonable tradeoff.

V. Implementation

NetService was tested in our target scenario in Raspberry Pis (RPis) 3 with Raspbian version 8 (jessie), TP-Link TL-WN722N 802.11 dongles for better coverage (the RPi's integrated 802.11 interface does not have an antenna), and sensors collecting electrocardiogram (ECG), CO_2 and GPS. The

existent applications for the firefighters collected the data from the sensors and made extensive use of RabbitMQ[3] as a broker. As such, the NetService was implemented using RabbitMQ in Python to provide the API described in section IV.

The MANET uses the IBSS 802.11 mode, with Optimized Link State Routing Protocol (OLSR) [15] version 2 as a routing protocol, given its active development, support for plugins and capability to export the network topology to NetJSON[4]. This gives extra information to the NetService that it can then provide to applications. For example, it can announce the discovery of new nodes, indicate the distance in hops of a node or the path chosen for a destination.

A. Service structure

When starting, the NetService has to change network details such as the routing protocol used, the type of network (currently only IBSS, but 802.11s mesh is to be supported), and configure the IP address, all transparently to the applications that use this service. It also has to abstract how the traffic is sent. We abstracted and modularized each network component in order to swap them according to a configuration provided at boot. The configuration has options to choose the IP address space to use, the network interface to use, the network name, the routing protocol and the level of logging that should be performed.

The message-based API provided to the applications (via RabbitMQ) was also isolated. RabbitMQ is only used for providing the message queues. The message format consists of structured JSON objects serialized as strings, and is independent from RabbitMQ, making it easy to migrate to a different broker in the future.

B. Service class

In the current implementation there are only two service classes, unreliable and reliable, as discussed in sec. IV-C.

Unreliable messages are sent using UDP datagrams without any check if the network is ready to send data or the destination is known by the routing protocol.

Reliable messages are currently sent over TCP, but additional mechanisms were implemented to handle the MANET characteristics and the extra requirements of the applications. We added upper layer retransmissions because, while TCP retries to connect, these retrials are performed within a short interval (max. 5 retries over a period of 180 s, by default[5]). This is adequate for packet losses caused by network congestion, but not for the disruption that exists in MANETs, caused by node mobility, routing protocols having outdated information leading to wrong paths, or the higher error rate of the wireless channel [12], [16]. We use TCP with a fixed ten seconds delay between retries to a maximum of five attempts in order to handle temporary disruptions on the network. The limitation of retries is meant to avoid cases where some nodes can simply disappear forever of the network (e.g. battery dies).

The correct reception and decoding of the data at the destination node is acknowledged by NetService by sending back a small packet, providing an application level confirmation of message reception that is relevant in some scenarios (e.g. team check).

There could be some advantages of following the approaches discussed in sec. II, but it would imply re-implementing most of TCP at application layer. There could be some gains from focusing on message sending as opposed to streaming data. Also, acknowledgement at the application layer while using TCP adds extra unnecessary overhead that could be removed in an implementation of a similar protocol at application level.

C. Debugging and logging data

To enable collection of data for debugging and off-line analysis we developed multiple levels of debugging. Several levels of timestamped events are available for discovering nodes entering and leaving the network, transmitted and received messages (with source/destination name and application identifier), size of packets and the service class. This also allows to perceive network changes and evolution.

Network changes were collected through the Linux state monitoring tool $ip\ monitor$. It provides information regarding routes changes, and thus, of neighbour changes on a MANET. The structure of the network known by OLSR is also exported and stored for later analysis using the NetJSON plugin.

VI. Solution Analysis

In this section we will analyze NetService both from a qualitative standpoint, making some considerations about different aspects of the solution, and a quantitative standpoint, using experimental results.

A. IP Address Auto-Configuration

Our approach to IP address assignment has the advantages of simplicity and requiring no network traffic. However, even though node addresses are derived from their guaranteed unique names, there is no guarantee that addresses will not collide, because the mapping from larger names to only 24 bits of addressing is necessarily not injective. To analyse the probability of collision of addresses independently generated according to our scheme, we first note that it is similar to the well-known birthday paradox, only with a domain of 2^{24} different addresses instead of 365 different days in a year. Equation 1 gives the collision probability $p(n; H)$ for the birthday problem[6] with H picks from a domain with n possible choices, where p is the probability that at least one value is chosen more than once during a single experiment.

$$p(n; H) \approx 1 - e^{-n(n-1)/(2H)} \approx 1 - e^{-n^2/(2H)} \qquad (1)$$

In our scenario, the number of picks from the domain is the number of firefighters simultaneously combating the same fire

[3]See https://www.rabbitmq.com/.

[4]http://www.olsr.org/mediawiki/index.php/NetJson_Info_Plugin

[5]Section *tcp_syn_retries* in http://man7.org/linux/man-pages/man7/tcp.7.html

[6]This formulation is normally used in hash collision, and described as an attack to find a collision in an hash function. It is referred to as birthday attack.

incident. Based on the largest fires that happened in Portugal in 2017, we deemed 1000 firefighters as a reasonable upper bound for this number[7]. Using equation 1 for the birthday problem, n is, in our case, the 1000 picks of IP addresses, and H is the possible number of choices, in our case the total number of addresses available to choose from. We find that while with 16-bit addresses the probability of collision of two or more IP addresses in a given MANET is nearly 100%, with 24-bit addresses that probability decreases to approximately 2.9%, an upper bound we consider acceptable.

In any case, measures could be taken to address the collisions. When using a proactive routing protocol, as OLSR, information of the routing protocol may be used to detect IP address collisions. Clausen et al. [17] describe a passive address collision detection method for OLSR using the messages exchanged by the protocol.Another solution to detect IP address collisions would be generating extra broadcast messages announcing a unique ID of the node, as proposed in [18].

An alternative that would share the good properties of our current solution would be moving to IPv6 and using the unique local IPv6 unicast addresses [6]. However, this would lead to extra overhead in every packet in the network, since the IPv6 header is twice as large as the IPv4 header. The routing protocol would also have to accommodate the larger IPv6 addresses, specially bearing for proactive protocols. In a quick test using OLSR in a network with 5 nodes, and no mobility and only one address per node, using IPv6 made the traffic generated by the protocol increase 1.35 times compared to using IPv4.

B. Name Resolution

The name resolution mechanism contrasts with the common approach in MANET of resorting to flooding the network for resolving the names or for finding a name server that can perform the resolution. Unlike mDNS [19] or Engelstad's proposal [20], our solution does not generate any traffic in the network. However, it implies a static mapping between name and IP address, which becomes infeasible should we add duplicate address detection and reassignment to eliminate the possibility of address collisions.

C. Routing protocol

Though in the current implementation we use OLSR as routing protocol, other options exist. The NetService enables configuring the routing protocol to use according to the requirements needed.

OLSR tries to keep all nodes informed of existing routes in the network. Although the MultiPoint Relay (MPR) mechanism tries to keep the packets that flood the entire network relatively low, it can still be a higher overhead than necessary for a proactive protocol when compared with for example Better Approach To Mobile Adhoc Networking (BATMAN) [21].

Replacing OLSR with BATMAN would cause the loss of data been collected about the network structure and connections between nodes, since BATMAN only keeps information about the next hop for each node in the network, and not all the existing connections in the network. A possible solution would be using Batadv-vis[8] to collect data, but that would cause extra overhead in the network and possibly cancel the improvements of using BATMAN.

D. APIs

The provided APIs have no counterpart in the existing APIs to configure the network. It is possible to use NetworkManager[9] to configure the network, but it is not as simple as calling a method to start, and does not solve either IP address distribution or name resolution.

The communication API simplifies the usage of either UDP or TCP, enabling the use one or the other in the same application. For reliable transmission it handles retries from temporary network disruption and makes the process of sending a packet only a single method call, in contrast, around 10 lines of code are needed in the POSIX sockets' API to start a TCP server and a client, connect, transmit and end the connection on both sides.

E. Performance and overhead

The proposed solution adds some costs on the machine for each service that it provides. There are costs on the CPU and memory usage in general, but also overhead on the traffic that is sent using the communication's API. Generally, to provide the APIs, a message broker (RabbitMQ in our case) needs to be running. For the configuration API one thread consuming from the broker and another thread to keep up to date the network and neighbourhood status are spawn. The network API encompasses one thread also for consuming from the broker and one thread to keep receiving incoming packets per application registered.

To compare the costs of using the NetService we made a set of experiments in a Virtual Machine (VM) running the network emulator Mininet-Wifi [22] (version 2.2.0d1). Mininet-WiFi is a fork of Mininet adding wireless communication to the emulator. It uses network namespaces and a wireless emulator to emulate multiple wireless nodes (terminals and access points) in a single machine.

For the emulation of the environment we used a radio range for the antennas of 100 meters in an area of 400 x 400 meters, using the available Random Waypoint mobility model with a minimum speed of 0.5 m/s and maximum of 2 m/s corresponding to a person walking [23]. To simulate the sensors, we used a traffic generator developed within our group that was modelled by analysing the data pattern of the sensors (GPS, ECG and CO_2) according to their output format and the application needs. They followed a normal distribution for the inter-arrival time of each packet, where the mean and standard deviation where set to model the sensors' data production.

Each time a new block of data arrives from a sensor it is sent via unreliable transport to its team-captain. We assume

[7]From data reported in local news http://sicnoticias.sapo.pt/especiais/tragedia-em-pedrogao-grande/2017-06-20-Dezoito-aldeias-evacuadas-em-Gois.

[8]Batadv-vis : https://www.open-mesh.org/projects/alfred/wiki/Batadv-vis.

[9]https://wiki.gnome.org/Projects/NetworkManager

3 teams, with each team having 4 nodes generating data and one being its team-captain collecting, a total of 15 nodes. The team-captain sends a team-check each two minutes to verify the status of the team's members. The team-check is a small packet, but needs to be reliably delivered. When a node receives a team-check it replies with a "status:ok", also using the reliable service class. For the described traffic, reliable and unreliable service classes types were used for the NetService. When the NetService was not used, TCP and UDP were respectively directly used for comparison.

We collected traffic, CPU and memory consumption. Battery usage was not directly measured, although being relevant in a MANET. We assumed that energy expenditure is related to the usage of the wireless connection and computation. Thus, more traffic and CPU used will imply extra battery drain.

Tests where performed with an initial start of all the software followed by 10 minutes of traffic. The initial phase was removed as a non-stable initiation of the components. From all the tests we collected, we determined that after 100 seconds, the initialisation process ended in every test. In all the graphics only the stable phase (after 100 s up to 600 s) is shown.

For comparison, we assumed a base setting where the network emulator was running with 15 nodes and OLSRv2, and RabbitMQ was running in each node (*Baseline*). On the second setting we added the traffic from the sensors and the periodic team-checks via NetService (*Traffic via NetService*). For evaluating the impact of RabbitMQ, we tested a setting with only the network emulator running with 15 nodes and OLSRv2, with no traffic (*Baseline without RabittMQ*). The last setting has the network emulator running with 15 nodes with OLSRv2 and uses directly TCP or UDP connections (*Traffic without NetService*).

CPU and memory were measured globally in the VM, i.e., values pertain to the usage on the full VM, using *psutil* Python library. All services were reduced to the minimum and the graphic environment was not active. The measurements were taken at each second. For the CPU calculation we deducted the cost due to the wireless emulator (`wmediumd`). This was done for two reasons. First the emulator is software that is substituting the wireless card work. As such, its impact on the solution's performance should not be accounted. Secondly, we encountered some discrepancies on its performance with the mobility setting. `wmediumd` was consuming less CPU when sensor traffic was being sent (*Traffic with NetService*) than when only OLSR was running (*Baseline*), with the same issue with *Traffic Without NetService* and *Baseline without RabbitMQ*. Because of this extra CPU measure that we deducted from the total CPU, we considered the mean of 5 second intervals to avoid problems caused by the small time differences between obtaining the total CPU usage and the CPU usage of *wmediumd*.

Figure 3 shows a boxplot of CPU consumption with mobile and static nodes. The whiskers are the minimum and maximum measured values. All mobility tests used the same seed to have the same mobility pattern in the nodes. As expected, there is an increase in CPU consumption when sending traffic, which is larger when the NetService is used. The largest increase in CPU consumption, however, stems from having RabbitMQ running, from 4.65% to 9.24% on average when comparing both baselines with and without RabbitMQ.

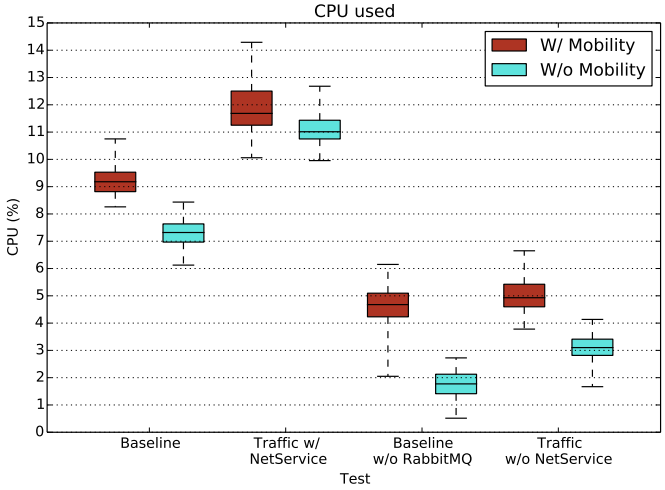

Figure 3. CPU usage in the VM running the emulation, with and without mobility

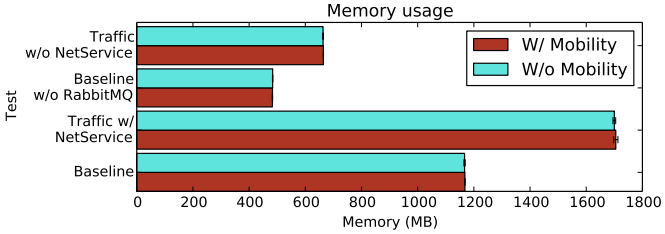

Figure 4. Memory usage in the VM running the emulation, with and without mobility

The total memory used in the VM for the scenario with mobility at each second was on average 1167 ± 2.3 MB (*Baseline*), 1700 ± 4.9 MB (*Traffic w/ NetService*), 483.8 ± 0.8 MB (*Baseline w/o RabbitMQ*), and 663.2 ± 1.3 MB (*Traffic w/o NetService*). These values are stable along the experiments, the largest standard deviations occurred in the tests where RabbitMQ was used (*Baseline* and *Traffic w/ NetService*). The behaviour was similar in the static environment.

The traffic was measured via a virtual radio that was capturing all the medium data using `tcpdump` for posterior offline analysis. In figure 5, we can see the different bandwidth consumption of each element in each test in a mobile scenario. The OLSR traffic accounts for most of the traffic in all settings, due to its periodic beacons and MPR flooding. The UDP traffic is all the traffic from sensors generated during the lifetime of the test. The TCP traffic is the team-checks and replies that amount to 15.9KB and 11.4 KB for the case of using the NetService and without it respectively. The other traffic is traffic created by other protocols like ARP and ICMP.

VII. Conclusions

NetService makes it much easier to develop and deploy applications that run on MANETs. It handles all necessary

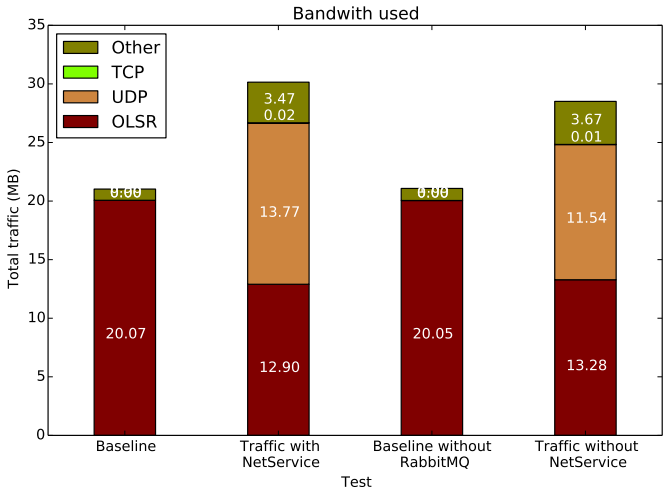

Figure 5. Break down of bandwidth used in each test by each traffic source

tasks to setup and run the network (including the ad-hoc routing protocol) and provides a simple interface for applications to start/stop the network and specify the name of the ad-hoc network to connect to. This configuration interface is based on messaging, using a control queue on a message broker. Different types of feedback to applications are also provided through messages on a queue. Over similar message queues a network communication API is also offered, unifying APIs for reliable and unreliable communication.

From the experimental results, we conclude that the routing protocol (OLSR) consumes a fair share of all network traffic, and so do other network protocols not directly related with data communication (ARP, ICMP, etc.). The combination of both accounted for more than half of all network traffic in all our experiments.

One other conclusion we draw from the experimental results is that RabbitMQ is very resource-consuming, and was probably not the best choice for moderately resourced nodes running on battery. In the future, we want to replace it with a lighter, more resource-conscious message broker.

Overall, the presented solution sets a good ground for future addition of functionality with backwards-compatibility, e.g. improvements on the routing and package processing without applications needing to be updated. The presented architecture is open to optimizations such as package grouping to a same destination (per hop or on the sender), different traffic prioritization or packets with deadlines. Other functionally could be added such as better IP allocation and DAD, improvements on name resolution and support to subscribe for traffic from multiple groups on a same hierarchical level.

Acknowledgements

This work was funded by National Funds through the FCT - Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) within the project "VR2Market", grant CMUP-ERI/FIA/0031/2013. The authors wish to thank José Maria Fernandes and Ilídio Oliveira for discussions regarding the applications' requirements, and Emanuel Lima for defining the models for the sensors' traffic.

References

[1] P. Mitra and C. Poellabauer, "Emergency response in smartphone-based Mobile Ad-Hoc Networks," in *IEEE Int. Conference on Communications (ICC)*. IEEE, Jun. 2012, pp. 6091–6095.

[2] D. Srikrishna and R. Krishnamoorthy, "SocialMesh: Can networks of meshed smartphones ensure public access to twitter during an attack?" *IEEE Communications Magazine*, vol. 50, no. 6, pp. 99–105, Jun. 2012.

[3] C. Reuter, T. Ludwig, M.-A. Kaufhold, and J. Hupertz, "Social media resilience during infrastructure breakdowns using mobile ad-hoc networks," in *Advances and New Trends in Environmental Informatics*. Springer, 2017, pp. 75–88.

[4] J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. H. Lim, and E. Upton, "Haggle: Seamless networking for mobile applications," in *Proc. of the 9th Int. Conference on Ubiquitous Computing*. Innsbruck, Austria: Springer-Verlag, 2007.

[5] E. Baccelli and M. Townsley, "IP Addressing Model in Ad Hoc Networks," Internet Requests for Comments, RFC Editor, RFC 5889, 2010.

[6] R. Hinden and B. Haberman, "Unique Local IPv6 Unicast Addresses," Internet Requests for Comments, RFC Editor, RFC 4193, 2005.

[7] S. Thomson, T. Narten, and T. Jinmei, "IPv6 Stateless Address Autoconfiguration," Internet Requests for Comments, RFC Editor, RFC 4862.

[8] R. L. Gilaberte and L. P. Herrero, "Automatic Configuration of Ad-Hoc Networks: Establishing unique IP Link-Local Addresses," in *The International Conference on Emerging Security Information, Systems, and Technologies (SECUREWARE 2007)*, Oct 2007, pp. 157–162.

[9] S. Cheshire, B. Aboba, and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses," Internet Requests for Comments, RFC Editor, RFC 3927, May 2005.

[10] S. Nesargi and R. Prakash, "MANETconf: configuration of hosts in a mobile ad hoc network," in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, 2002, pp. 1059–1068 vol.2.

[11] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP Performance over Wireless Networks," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '95. New York, NY, USA: ACM, 1995, pp. 2–11.

[12] A. Dunkels, J. Alonso, T. Voigt, and J. Alonso, "Making TCP/IP Viable for Wireless Sensor Networks," *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004)*, 2004.

[13] I. Aydin, R. Ge, P. Natarajan, and C.-C. Shen, "Performance Evaluation of SCTP in Mobile Ad Hoc Networks," 10 2017.

[14] A. Kumar, L. Jacob, and A. L. Ananda, "SCTP vs TCP : performance comparison in MANETs," in *29th Annual IEEE International Conference on Local Computer Networks*, Nov 2004, pp. 431–432.

[15] T. Clausen, C. Dearlove, P. Jacquet, and U. Herberg, "The Optimized Link State Routing Protocol Version 2," Internet Requests for Comments, IETF, RFC 7181, Apr. 2014. [Online]. Available: https://tools.ietf.org/html/rfc7181

[16] X. Chen, H. Zhai, J. Wang, and Y. Fang, "TCP performance over mobile ad hoc networks," *Canadian Journal of Electrical and Computer Engineering*, vol. 29, no. 1/2, pp. 129–134, Jan 2004.

[17] T. H. Clausen, E. Baccelli, and J. Garnier, "Duplicate address detection in OLSR networks," in *Proc. of WPMC*, 2005.

[18] S. Boudjit, C. Adjih, A. Laouiti, and P. Muhlethaler, *A Duplicate Address Detection and Autoconfiguration Mechanism for a Single-Interface OLSR Network*. Springer Berlin Heidelberg, 2005, pp. 128–142.

[19] S. Cheshire and M. Krochmal, "Multicast DNS," Internet Requests for Comments, RFC Editor, RFC 6762, 2013.

[20] P. Engelstad, D. V. Thanh, and T. E. Jonvik, "Name resolution in mobile ad-hoc networks," in *10th International Conference on Telecommunications, 2003. ICT 2003.*, vol. 1, Feb 2003, pp. 388–392 vol.1.

[21] D. Johnson, N. Ntlatlapa, and C. Aichele, "A simple pragmatic approach to mesh routing using BATMAN," in *In 2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries, Pretoria, South Africa*, 2008.

[22] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in *CNSM 2015 11th International Conference*. IEEE, 2015, pp. 384–389.

[23] R. W. Bohannon, "Comfortable and maximum walking speed of adults aged 20—79 years: reference values and determinants," *Age and Ageing*, vol. 26, no. 1, pp. 15–19, 1997.