



Dedalus: A Software Platform for Performing Wireless Networking Experiments

E. Aliaj^{*}, G. Dimaki[§], P. Getsopoulos^{*}, Y. Thomas^{*}, N. Fotiou^{*}, S. Toumpis^{*}

^{*}Mobile Multimedia Laboratory, Athens University of Economics and Business, Greece

[§]Massachusetts Institute of Technology, MA

20/11/2018

1. Introduction

In this document we provide a high-level description of Dedalus, a software platform that can be used for performing experiments with wireless experimental networks, which was developed in the UNSURPASSED project [UN].

Dedalus was developed with the aim of accelerating experimentation with wireless networks, by automating, centralizing, and streamlining necessary network control procedures, by enabling experimenters to have a continuous view of the topology, and by allowing them to remotely control, in an immediate and efficient manner, the different nodes in the network.

This document is targeted to experimenters interested in using, rather than further developing, Dedalus. Therefore, its aims are, firstly, to allow wireless experimenters to decide if Dedalus is suitable for their wireless experimenting needs and, secondly, to provide them with information on how to install and run it. We note that, for experimenters interested in *expanding* the functionality of Dedalus, the complete source code and detailed documentation, describing individual functions, etc., are available on gitlab through the project website [UN].

The rest of this document is organized as follows: In Section 2 we discuss the RAWFIE and UNSURPASSED projects, in the context of which Dedalus was developed. In Section 3 we present the architecture and capabilities of Dedalus. In Section 4 we discuss the wireless networking protocols already integrated with it, and future plans on adding more. In Section 5 we provide details on installing Dedalus, and in Section 6 we describe the Graphical User Interface (GUI) developed for use with it. We proceed in Section 7 to discuss briefly our own experiments using Dedalus, and we conclude in Section 8 mentioning other documentation that is available on Dedalus and related documents.

2. The UNSURPASSED and RAWFIE projects

Dedalus was developed as part of the UNSURPASSED (Unmanned Surface Vehicles as Primary Assets for the Coast Guard) project [UN], a one-year project (1/10/2017-31/12/2018) operated by the Mobile Multimedia Laboratory (MMLab) of AUEB. The UNSURPASSED project itself is part of the larger RAWFIE project [RAW] (2014-2018), which, in turn, is operated by an extensive consortium of partners operating across Europe and coordinated by the University of Athens. Members of that consortium involve a variety of



operators of experimental testbeds. Notably (for the purposes of UNSURPASSED) the Hellenic Navy operates a wireless testbed at an extensive naval base located in the town of Skaramagas near Athens.

The RAWFIE project aims at providing a federation of testbeds used for performing experiments with Unmanned Air Vehicles (UAVs), Unmanned Surface Vehicles (USVs), i.e., autonomous boats, and Unmanned Ground Vehicles (UGVs), collectively referred to as UxVs. A fundamental goal of RAWFIE is that experimenters should be able to use the resources of the testbeds remotely, notably describing experiments through a dedicated experiment authoring tool called the Experiment Description Language (EDL).

The UNSURPASSED project is a third-party project of RAWFIE (the third party being AUEB) which was introduced with the goal of setting the RAWFIE USV testbeds in the service of the coast guard, showcasing the potential of USVs to perform surveillance and search-and-rescue tasks. The project’s goals are twofold. The first involves the integration (in RAWFIE) of networking and security mechanisms that are based on distributed paradigms, i.e., ad hoc, delay-tolerant, and information-centric networking and identity-based encryption, which will turn USVs into real assets for the coast guard. The second goal involves using these mechanisms to conduct experiments of escalating complexity in order to understand the current technological challenges and thus bring large maritime wireless networks closer to realization. Our experiments were conducted by placing Raspberry Pi nodes on USVs, thus forming experimental mobile wireless USV networks. Dedalus was developed for controlling these experiments, in a manner that would allow quick debugging of the various obstacles and a resulting acceleration in the conducted research.

3. Architecture of Dedalus

In short, Dedalus is a self-healing, distributed system developed for controlling experimental wireless networks. Controlling means monitoring the connectivity between nodes, gathering real-time statistics on, e.g., connectivity and traffic, instructing nodes to run specific protocols and create and transport specific traffic profiles, maintaining and distributing event logs, and collecting experimental results.

Dedalus is based on the Majordomo protocol [MD], and so is designed for resiliency and can operate in unstable networks. In particular, its various entities can be created in any order without any constraints imposed by connectivity. Also, if a connection is lost, messages traveling to their destinations will persist until they are delivered or a timeout occurs. Therefore, different types of nodes can be initialized with any order without any issues regarding connectivity. If a connection is lost, the message will persist until it is delivered or a timeout occurs.

As shown in Fig. 1, Dedalus creates a distributed system comprised of three kinds of entities, i.e., Clients, Workers, and Brokers, running at the nodes of the network.

Firstly, workers create, receive, and forward traffic, and report their status to Brokers. Typically, there is one worker installed in each node. Workers are responsible for interfacing with the various wireless networking protocols installed on the nodes and converting instructions into commands understood by them. Workers query the host system for a number of useful metrics, such as CPU or memory load, network traffic, etc. In short, workers are responsible for performing all of the required operations on each node. As soon as they find a running instance of a broker, they will connect to it and make available one or more services that they can offer. Once the initial installation phase has concluded, they will be available to answer requests and monitor

the node they are running on. They can be implemented in any language, as long as the communication constraints are respected.

Secondly, brokers assign tasks to workers according to instructions received from the clients, making sure that instructions are eventually received when there are gaps in the connectivity, and so on. Therefore, a Dedalus broker is responsible for passing messages back and forth from all clients and workers connected to him as well as implementing the needed queues for ensuring that all requests will eventually be served. Brokers are also tasked with basic bookkeeping operations since they have to maintain lists of all active workers as well as test their connectivity and status. Finally, brokers have their own service (which provides statistics that can be requested by any client) registered along with all other worker services and can operate as an asynchronous backend server.

Finally, clients monitor the conditions of the network and assign tasks to workers by communicating with them through the brokers. For example, they interact with the nodes for collecting statistics, and instruct them to change between the available routing protocols or execute some other experiment scenario. This can be done by writing scripts that will inherit from a base worker class that is provided by Dedalus, allowing for a consistent and easy way to communicate with a broker instance. Another, easier, way is by using one of the existing graphical interfaces, or in the case that the experimenter is using the Kafka server of RAWFIE (as described later on), interface with the nodes through an EDL script. Clients only have to respect the specific message formats and otherwise are not limited in their implementation details in any way. They can be created in most available programming languages and from any operating system. We are also not limited to the design of the client since it can be anything from a desktop application with a complete user interface, a mobile app, a terminal application, or even a simple logger.

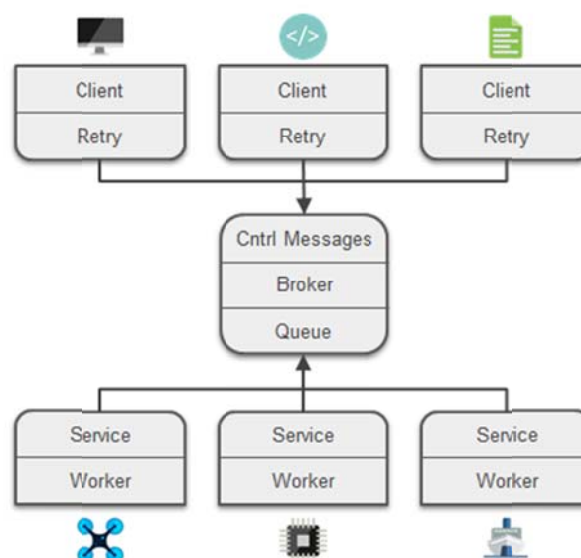


Figure 1: The Dedalus architecture comprises three kinds of entities (Clients, Brokers, and Workers) (taken from [ED18a]).



The Dedalus application itself can be run in four main modes, one mode for each kind of entity and a mode combining two entities.

Dedalus has the ability to work with one or more network interfaces, some of them wired. Therefore, if only one (necessarily wireless) network interface is available, then that interface will be used both by the experimental wireless network (for the transport of experimental data) and also by Dedalus (for transporting control information, such as instructions for starting and ending data traffic, etc.). On the other hand, if multiple network interfaces are available, then one of them will be taken over by the wireless nodes, for use by the experimental wireless networking protocols, thus forming the data plane, and the other(s) will be used for carrying control traffic, e.g. answering queries about the status of nodes, or pushing network messages for storing to a remote log file, etc., thus forming the control plane. As an example, in our own experiments, we equipped Raspberry Pi nodes with two wireless network interfaces, one interface for use by the experimental routing protocols and one interface for connecting to the Wireless LAN available throughout the building where our laboratory is located (when performing experiments in our lab), or the Wireless LAN available to the RAWFIE testbed (when performing experiments there). A third, Ethernet network interface was also used by nodes that did not have to be mobile.

Dedalus was designed to be well integrated with the rest of the RAWFIE infrastructure, but can also be used independently of it. When used with the RAWFIE infrastructure, it can take advantage of it. For example, it can use the wireless LAN available to the testbeds to form the network control plane, as described above. Also, the wireless nodes can communicate among themselves and also with other entities in the testbed through the Apache Kafka technology on which the RAWFIE testbeds are based, which leads to a number of features. Firstly, Dedalus entities can communicate with GPS transmitters installed on USVs, thus learning the physical location of the wireless nodes attached to the USVs, thereby avoiding the integration of GPS receivers on the nodes themselves. Secondly, experimenters can specify the traffic to be carried by the nodes in the same EDL script that also specifies the node mobility; after that point, a Dedalus broker can take over and perform the experiment, as far as the wireless traffic is concerned. Thirdly, traffic measurements (throughout, latency, jitter, etc.) by the wireless nodes pass through the Kafka server, and are therefore immediately available to other entities, including remote experimenters.

4. Wireless Networking Protocols integrated with Dedalus

In this section we discuss wireless networking protocols already integrated with Dedalus. These protocols are located in each node, and receive instructions by each node’s worker. As described earlier, in turn, the workers receive instructions from the clients, through the brokers. We stress that this three-tier architecture of Dedalus was chosen so that other wireless networking protocols can be added, with minimum overhead, by other researchers. In particular, adding more protocols requires adding interfaces between these protocols and workers, and possibly the addition of new high-level instructions by the clients. In this section we also provide some information on our future work in this direction.

A. Ad Hoc Routing Protocols

Ad hoc routing protocols are responsible for discovering and maintaining multihop routes between distant nodes that do not have a direct link between them. We have integrated so far two ad hoc routing protocols with Dedalus, Babel [ED18b], BATMAN, and its extension, BMX7 [BMX7]. These protocols have many



similarities, but a comparison of their relative performance would still be interesting, as it would highlight the effects of their specific differences on the performance of the network.

As specified in the Babel RFC [ED18b], Babel is based on the Bellman-Ford algorithm with additional features aiming at avoiding the creation of loops. A Babel node periodically broadcasts Hello messages to all of its neighbors; when a node receives a Hello message, it replies with an IHU ("I Heard You") message. From the information derived from Hello and IHU messages received from its neighbor B, a node A computes the cost $C(A,B)$ of the link from A to B. Based on this information, the Bellman-Ford algorithm is performed in every node, and routes are formed. It should be noted that, because of its proactive operation, in large, stable networks Babel generates more traffic than protocols that only send updates when the network topology changes.

For our experimentation with Babel, we have installed `babeld`, which is available in the Babel git repository [BA], and configured the wireless interface of each Raspberry so as to run in ad hoc mode and have a static IP. Finally the `babeld` daemon is configured to automatically start after booting and keep a log of its operation in `babeld.log` file.

Dedalus, in order to retrieve Babel-specific information, like the current state of the network, makes use of the `babeld` configuration interface. To this effect, `babeld` is invoked with a `-g` flag so it can accept TCP connections from localhost. What the protocol scanning module of Dedalus does is to request a dump of `babeld` information. Then, a TCP connection is established, the request is sent, and the reply is then parsed so as to provide us with all the information we need. Specifically, we keep track of routing information accompanied by metrics for each route as well as information about neighbor nodes (i.e., nodes in the range of the specific node's WiFi).

On the other hand, BatMan-eXperimental version 7 (BMX7) [BMX7], the third protocol we have implemented, is focused on security and has parts of the protocol cryptographically signed so as to allow only authorized nodes to take part in the network. It too is a proactive distance-vector routing protocol, unfortunately, though, documentation on it is very sparse. In BMX every node sends a HELLO message as a multicast packet every 0.5 seconds as well as REPORT messages that contain various types of information, the most important of which being the number of HELLO messages they received on that interval from all neighbors. In this manner, i.e., by a node knowing the number of messages it should have received versus the number it did receive, a link cost can be computed for any particular link with a neighbor.

Interfacing with BMX can be done by opening a second instance of the application in client mode; an example might look like the following:

```
>bm7 -c show=status
```

This command will display status information for a node. This is a good way for retrieving information, but is not always reliable. Until these issues are addressed, at a later stage of BMX’s development, we have opted to directly retrieving the information by streaming it from the protocol’s runtime files located at `‘/var/run/bmx7’`.

B. Delay-Tolerant Networking Protocols

We have also installed and tested the well-known IBR-DTN standard [IBR,BU] for multi-hop and data muling scenarios. Therefore, we can have the individual nodes operate autonomously using only the IBR-DTN protocol for routing, or have it operate in cooperation with another ad hoc routing protocol to help it discover



new neighbors. This gives us the ability to seamlessly support both delay-sensitive and delay-tolerant communication between disconnected networks, data muling operations, data streaming, and file transfer in the unsafe and unstable maritime environment.

We are able to use the bundle protocol as defined in RFC 5050 [BU], as well as the bundle security protocol as described in RFC 6257. Moreover we can limit bundle propagation based on age or hop count.

Currently we have experimented with various routing algorithms available in the IBR-DTN framework, including:

- Routing with static connections
- Epidemic routing
- Flooding routing
- PRoPHET routing

The implementation supports a number of different storage schemes:

- In-memory storage (RAM)
- Persistent storage in the file system
- Database Storage

A few additional issues related to DTN routing are still being investigated and require work before they can be ironed out and integrated in the RAWFIE platform; these include the time synchronization between nodes and fragmentation occurring to the network control messages sent by the DTN protocol.

In future work, we aim at supporting the following applications:

- Network performance monitoring for DTN-enabled networks
- Connection keep-alive and polling for DTN-enabled networks
- Path tracing for DTN-enabled networks
- File transfer functionality
- Automatic monitoring and synchronization of files (DTN-enabled distributed filesystem)
- Audio and Video streaming by leveraging the ad hoc layer for route discovery and the DTN layer for alleviating issues arising from channel quality fluctuations

Other frameworks investigated and under consideration for future implementation and use within the Dedalus framework are:

- ION (Interplanetary Overlay Network): This DTN architecture was described in RFC 4838 and is designed with space communications in mind; the ability to work on embedded devices makes it a viable choice for our network.
- DTN2: A less mature, but nonetheless promising implementation of a DTN routing protocol.

C. Information Centric Networking

The third layer integrated in our framework is Information Centric Networking (ICN). We have built, installed and tested CCN-Lite [CCN] in fully-connected networks and multi-hop scenarios successfully. Furthermore, ICN routing based on CCN-Lite has been tested and works equally well over ad hoc networks as it does over traditional networks.



The CCN-Lite protocol gives us the advantage of reducing the network hops and eliminating redundant requests for specific named content, as it handles the caching on every node and delivers requested content from the cache that is nearest to the node requesting it. This function is important in order to reduce the traffic footprint of the wireless mobile ad hoc network, where bandwidth resources are diminished compared to traditional wired network.

Other issues that need to be investigated further and will be solved soon include the automatic creation of Faces and Forwarding rules based on the list of neighbors the node has, as provide by the ad hoc layer.

After solving the aforementioned issues, we are interested in testing more ICN frameworks. The most notable of these is the Named Data Network (NDN) Project [ZL14]. This is a large project, with participants from universities around the world, which may provide us with tools for better management and monitoring of the network.

The overall target is to incorporate the ICN protocols in an as transparent as possible way with the rest of the layers in order to support file transferring and caching abilities with support for named content lookup among the nodes.

D. Identity-Based Proxy Re-Encryption

In Identity-Based Encryption (IBE) schemes [GA07], an arbitrary string can be used as a public key. An Identity-Based Proxy Re-Encryption (IB-PRE) scheme is a scheme in which a third party is allowed to alter a ciphertext, encrypted using an arbitrary string ID1, in a way that another user that owns a secret key SKID2 can decrypt it. Such a scheme requires two new algorithms, RKGen and Reencrypt, in addition to algorithms used in IBE schemes:

- **RKGen:** it is executed by a user that owns a secret key SKID1. It takes as input a set of public parameters PP, that are publicly available, the secret key SKID1 and an arbitrary string ID2 and generates a (public) re-encryption key RKID1→ID2.
- **ReEncrypt:** it is executed by a third party. It takes as input PP, the re-encryption key RKID1→ID2 and a ciphertext CID1 and outputs a new ciphertext CID2.

The ciphertext generated by the ReEncrypt algorithm can be decrypted only by using SKID2. The entity that performs the re-encryption learns nothing about the encrypted plaintext or about the secret keys of users ID1 and ID2.

We have developed the message encryption application using the Green and Ateniese IBE-PRE scheme [GA07] as implemented by the Charm-Crypto library [CHA]. The current implementation is composed of 6 python scripts.

5. Installation of Dedalus

Dedalus is available on Linux using Python 2.7, with a version compatible with Python 3 being on its way.

There are two main different versions of Dedalus that experimenters might need to run. The first one is a standalone implementation that will not use the RAWFIE project’s Kafka server but will instead provide all streaming and messaging capabilities on its own. The second version, named “Dedalus Kafka”, can integrate



with the rest of RAWFIE’s platform and interface with EDL. Both can also work in parallel with a client application, which is a useful feature for monitoring and executing experiments.

The installation of both versions is straightforward:

- Clone the git repository available through the project website [UN] for the standalone version of Dedalus, or for Dedalus Kafka, using *git*.
- Open a terminal and change directory to the newly cloned repository:

> cd “path to repository”/Dedalus

- Make sure you have all the needed requirements by executing

> pip install -r requirements.txt

- After this step is completed you can choose to either directly use it or install it by running

> python setup.py

A. Broker

Dedalus can be used to track and monitor a number of nodes, keep statistics and distribute messages to them by running the broker, as follows:

> python dedalus broker -a [endpoint]

In the above, the endpoint is the address of the interface you want to bind to and use as your control network. Note that for the version of Dedalus designed to work with Kafka, no such network is needed. Instead, what you must provide is the address of the Kafka server.

E. Worker

Worker mode can be invoked by simply typing

> python dedalus worker -a [address of broker]

The Kafka version of Dedalus will simply need, again, the address of the running Kafka instance that will serve it. All information about the commands and data passing through a worker or broker node are saved on a ‘Log’ directory that can be controlled by defining the ‘DEDALUS_LOG_CFG’ environmental variable. If the environmental variable is not set, then the default location of Dedalus on the host system will be used instead.

F. Client

Clients present a few major differences with the rest of the running modes. This is due to the fact that Dedalus exposes an API that works on nodes that operate over an ad hoc and DTN network. There is no limit as to what those clients might do or how will they be structured. Two clients are provided. The first is a simple terminal application that inherits a client base class, offering timeout and on-reply hooks so the creator can react in the way he/she finds appropriate for their needs. The second client is a graphical application exposed to the web through an http server. There is, however, one convention that all clients must follow: Dedalus will only accept messages that have a specific structure, in binary string format, and ignore all the rest.

A typical message accepted by a Dedalus broker looks like the one below:



[", dedalus_protocol, dedalus_service_a, wid, command_id, args]

The first argument is an empty space used by the broker to distinguish the sender id from the rest of the message. This is followed by the Dedalus protocol version we want to use, then the specific worker service we want to operate on, the worker id, if it is required, and any specific command id and any command arguments as required.

6. Running the Dedalus Graphical Interface

For the purposes of viewing real-time statistics, log-keeping, and interfacing with the nodes of a Dedalus network, a simple graphical interface has been created.

The interface is available in the repository. Installing it is as simple as running

> npm install -r

in the directory of the client application. After all the required software has been installed, the client is executed by using the command

> node server.js

This will run an http server on the localhost at the default port 9092, so if the user opens a browser and points it at the location <http://localhost:9092> she will be greeted by the login screen of the client application (Fig. 2). It is also possible to run this app on a public IP and use it to open multiple clients from various machines. Note, though, that they will all have access to the same nodes that are currently connected to the network they are monitoring, and no effort to create some scheduling or reservation mechanism has been made. In order to proceed, the user has to provide the address and port of a running Dedalus broker.

The graphical interface consists of a dashboard that neatly organizes the available functionality into small sections, and displays information focusing on what is most important. A view of all the sections in their current state of development is shown in Fig. 3. Depending on the available size on the screen that is being used, the ordering and structure of those sections might change, but the functionality remains the same. For the purposes of this exposition, each different section is annotated with a small bubble containing a number, and its region is highlighted with a unique color. What follows is a small description of each one of the six sections, going over their functionality, and, where needed, a step-by step-guide on how to use them.

A. Header

The first section of the dashboard (denoted by 1 in Fig. 3) is simply a welcome message coupled with an indicator of the connection status with the Dedalus broker. This indicator has three available modes: an ‘online’ mode displayed on a green background, a ‘connecting’ mode displayed on a deep orange background, and, lastly, a ‘lost’ mode displayed on a red background indicating that connection with the server is lost. Each time the client does not receive a reply to a predetermined amount of messages from the broker, it will drop its current connection and try from the beginning. If this procedure happens more than three consecutive times, then the client will give up and stop trying to reconnect. Each of these operations is displayed in real time on the header panel.

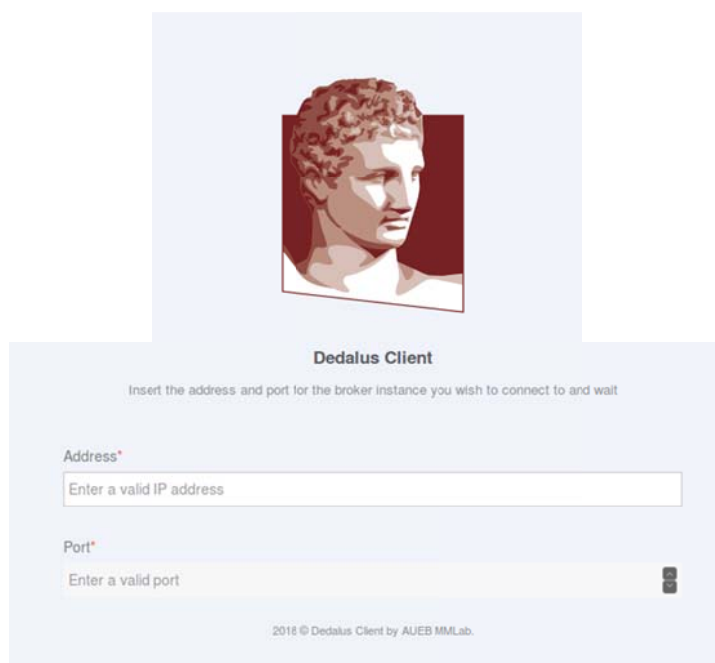


Figure 2: The welcome screen of the Dedalus Graphical Interface

B. Statistics Panel

This section, also shown separately in Fig. 4, is used for displaying a number of metrics, such as the number of connected users to this instance of the server and the number of nodes that have reported a given service. For now, this section is under development and more relevant information will eventually be displayed.

C. Protocols

Continuing in the dashboard, we note a set of selection boxes (denoted by 3 in Fig. 3). These offer a convenient way to define the running protocols throughout the entirety of the network. Thus, it is possible to set the specific combination of DTN, ad hoc, and CCN protocols that are to be used by all currently connected nodes. This section is useful as a quick way to create new network settings and hot-swap the running protocols. See a snapshot of its usage in Fig. 5. At the time of the creation of this document, the available set of protocols were the following:

- Ad hoc protocols: BABEL, BATMAN, BMX
- DTN protocols: EPIDEMIC, PROPHET
- ICN protocols: CCN

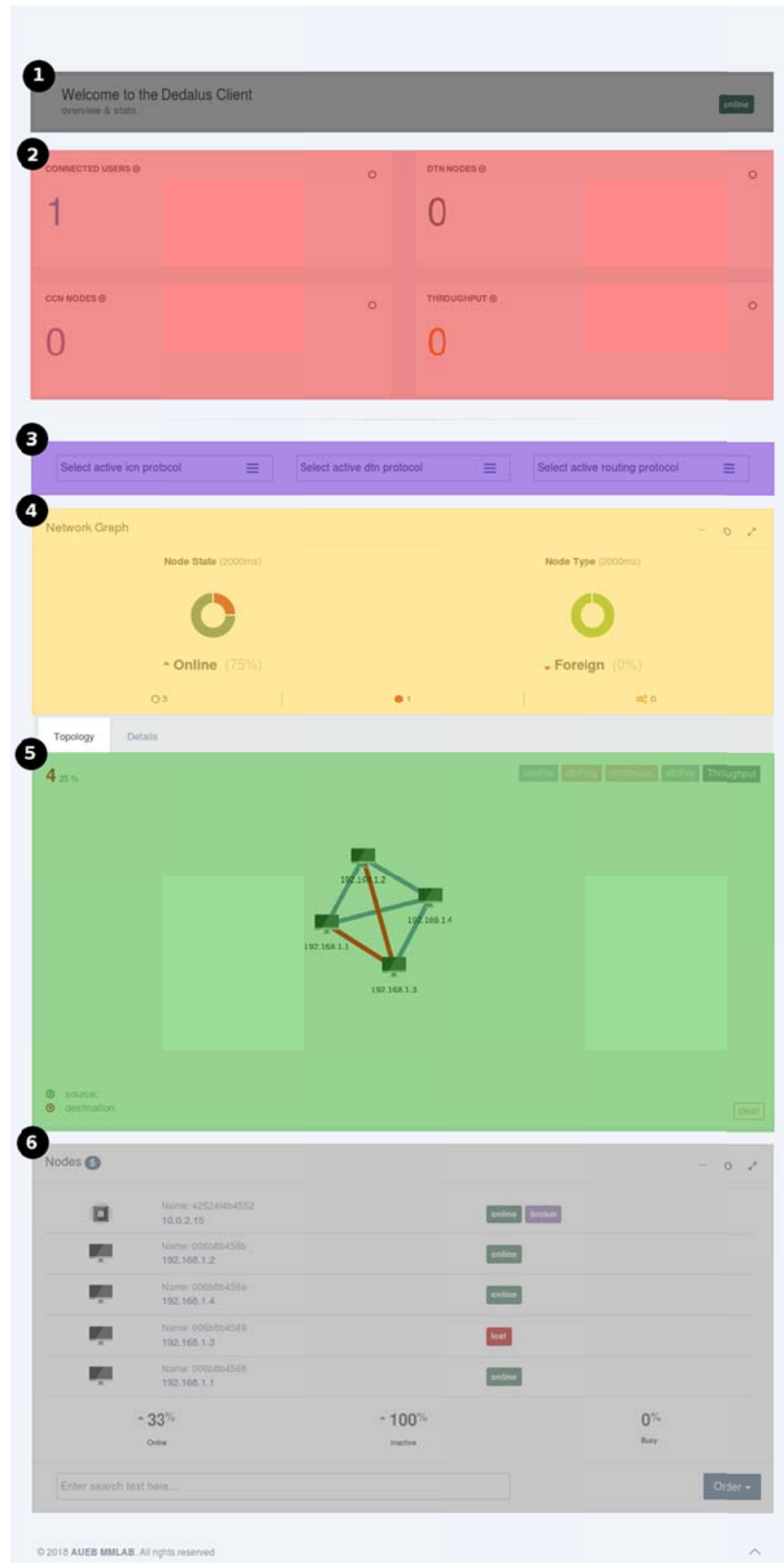


Figure 3: The main dashboard of the Dedalus Graphical Interface

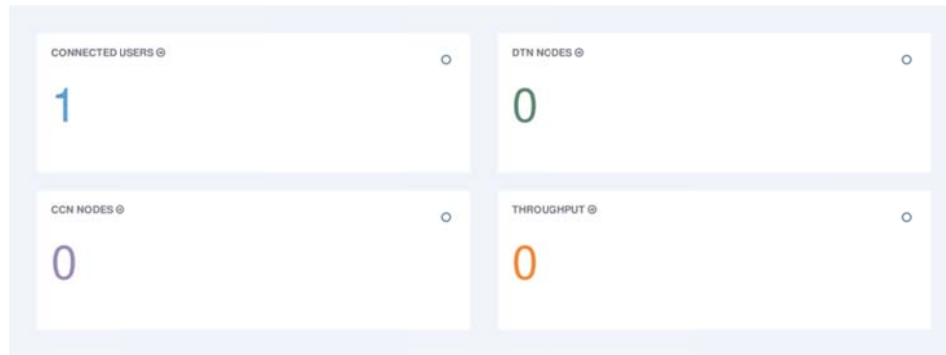


Figure 4: The statistics panel of the dashboard

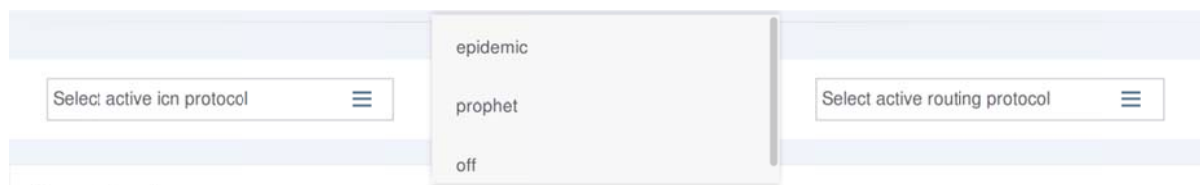


Figure 5: An example of selecting the routing protocols used by the nodes

D. Pie Charts

The fourth section of the dashboard, also shown in the upper part of Figs. 6 and 7, contains two charts:

- The first chart offers a quick glance at the size of our network, including the number of nodes that are currently on-line, lost, or busy performing a task. The chart will display the percentage of nodes that are in each of the three possible states with green, orange and red color respectively. (Note that this choice of colors is followed throughout the dashboard’s panels.)
- The second chart contains the number of nodes that have successfully managed to prove their identity with the broker and are thus marked as known and safe nodes.

The interval at which the information is updated for every chart is displayed in a light gray colored text located next to the chart name. In the case of Fig. 6, for example, we can see that the information is updated every 2 seconds. Notice that, for both charts, a number is given as a percentage for each of the possible node states. This simply displays the last change in the size of each group, accompanied by a green, upward looking arrow if the change is positive, and a red downward looking arrow if it is negative.

E. Topology Graph and Details List

This part of the dashboard, denoted by 5 in Fig. 3, is the most important section of the dashboard as it allows for direct control over each registered node. It is comprised of two tabs, the topology tab and the details tab, shown, respectively, in Figs 6 and 7.

Regarding the topology tab, we note that it is the most complex in design and thus a certain amount of care must be taken in order to familiarize oneself with its functionalities. The central component, and the one occupying most of the view, is a graph of the network in its current state, displaying in real-time information about the link from each node to each other node as a straight line of varying colors ranging, again, as in most of the program, from blue, denoting a good connection, to orange, denoting a somewhat more costly



connection, to, finally, red denoting a connection that is either too bad to be used or not even operational at this time. See the bottom of Fig. 6 for a snapshot of the graph.

Measurements regarding throughput, DTN file transfers, or CCN data requests, can be executed either through the ‘Actions’ menu (as discussed later on), or directly from the topology graph, by selecting the appropriate option from the list on the top right and then clicking on the nodes that you want to perform this action. As an example, in Fig. 8 we perform a throughput measurement between nodes 192.168.1.1 and 192.168.1.2. The first node that was clicked after selecting throughput as the preferred action was marked as a source, while the next one was immediately marked as the destination node and the measurement started. This can be seen from the legend on the left down corner and the loading indicator. As soon as the experiment is performed, it will be logged in both the nodes that took place in the experiments, the broker, if there is any, and the client that is being currently used.

On the upper left corner of the panel, one can see the number of all of the known nodes in the network, as well as the percentage of the most recent change in that number, displayed as a negative value if nodes unregister or get lost, and as a positive change if new nodes register.

The upper right corner, on the other hand, contains a set of buttons that become available depending on the particular combination of protocols chosen. So, for example, the button ‘dtnStream’, as its name suggest, will create a stream from a source node, to a destination node. These nodes can be chosen directly from the graph by clicking at the desired nodes in the correct order, or by the ‘detail view’ of each node, as will be shown below. In both cases, the nodes currently set as source and destination will be displayed in the lower left corner of the panel.

Regarding the ‘Details’ tab, as shown in Fig. 7, it is comprised of a list containing routing and general statistics for each node. These include things like memory usage, network traffic and node type. The last column contains an ‘Actions’ button that will display a list of node-specific available actions. These include performing a throughput measurements or other DTN/CCN related experiments (as shown in Fig. 8) and, very importantly, viewing a card containing the detailed information of each node which also gives the user the ability to individually change the protocols that are being used, or restart the node (Fig. 9).

F. Node List

The last panel (Fig. 10) contains a list of all of the known nodes, alongside some real-time information about their current status and IP. The list can be ordered according to various criteria or filtered using a user defined string (as shown in Fig. 11). Again some more information about the network trends is given by displaying the percentile change of all the nodes, this time depending on the various status groups. So, for example, we have the three status groups ‘online’, ‘busy’, ‘lost’. Each node might enter or leave any of those groups causing it to alter its size as these changes happen. This difference in size is displayed in the list and persists until the next change in the group. The user also has the ability to filter or search for a particular node as well as change the list order based on a desired criterion.

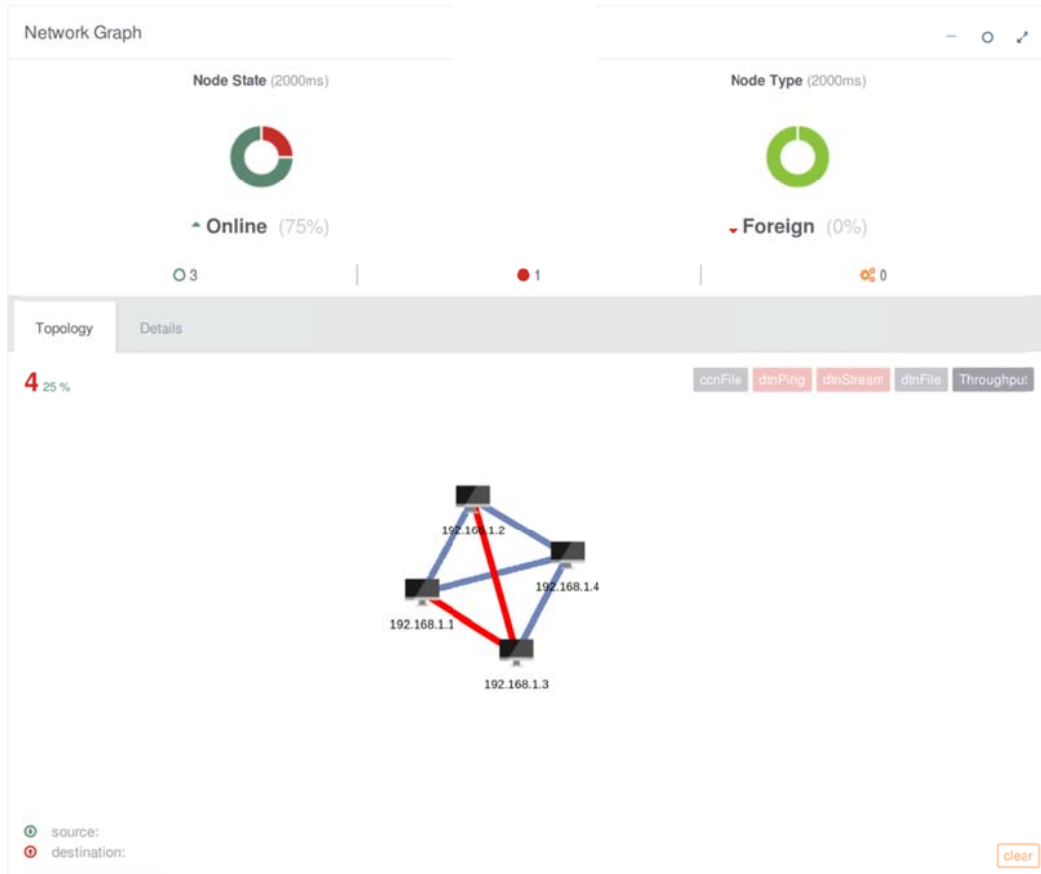


Figure 6: The topology tab

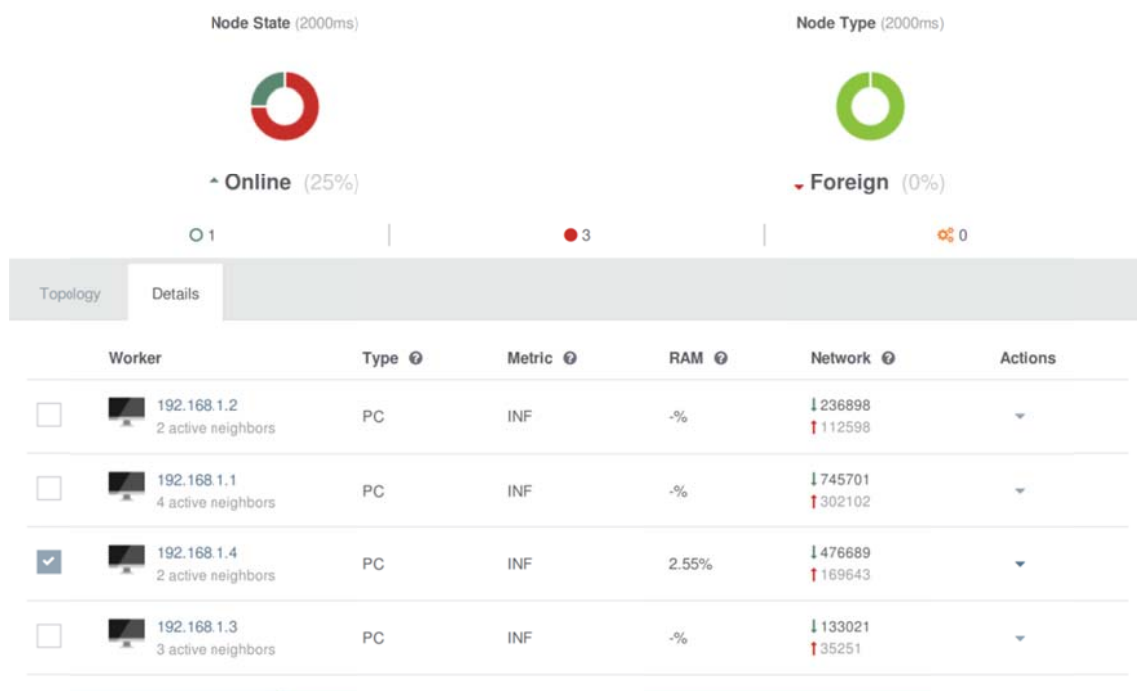


Figure 7: The details tab

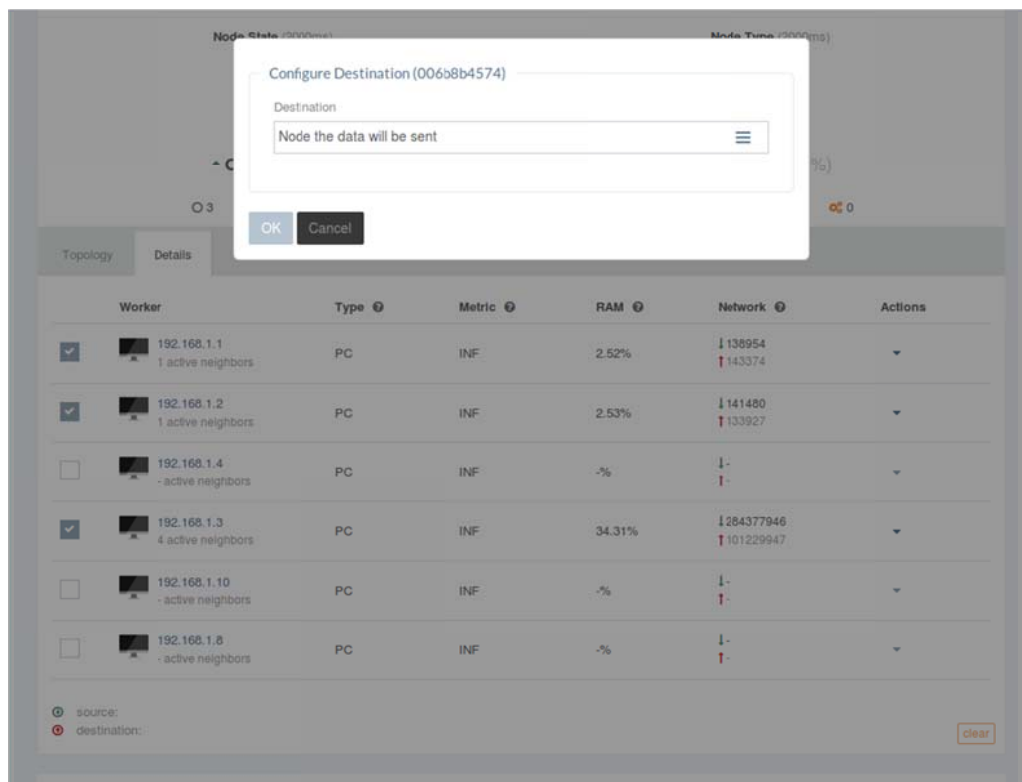


Figure 8: Performing throughput experiments through the details tab

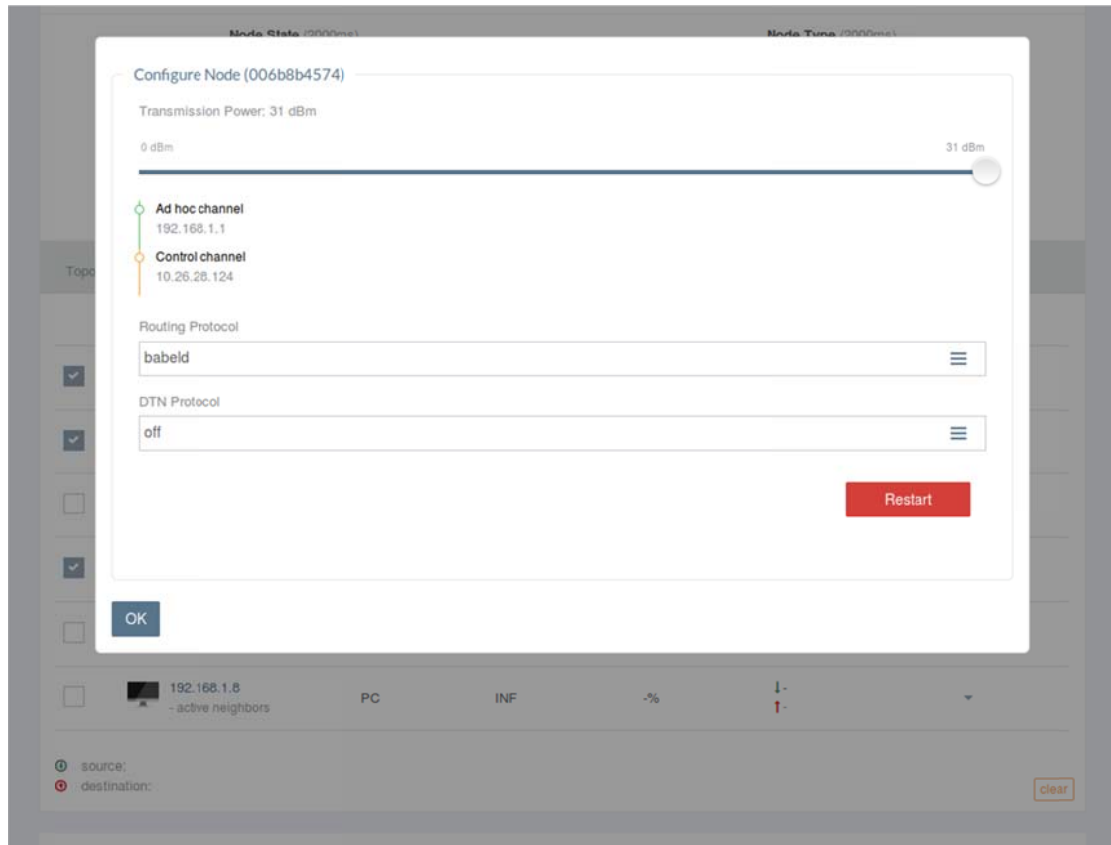







Figure 9: Configuring or restarting a node through the details tab

Nodes 5

	Name: 4252414b4552 10.0.2.15	online broker
	Name: 006b8b456b 192.168.1.2	online
	Name: 006b8b456a 192.168.1.4	online
	Name: 006b8b4569 192.168.1.3	lost
	Name: 006b8b4568 192.168.1.1	online

33% Online
100% Inactive
0% Busy

Order ▾

Figure 10: The node list panel

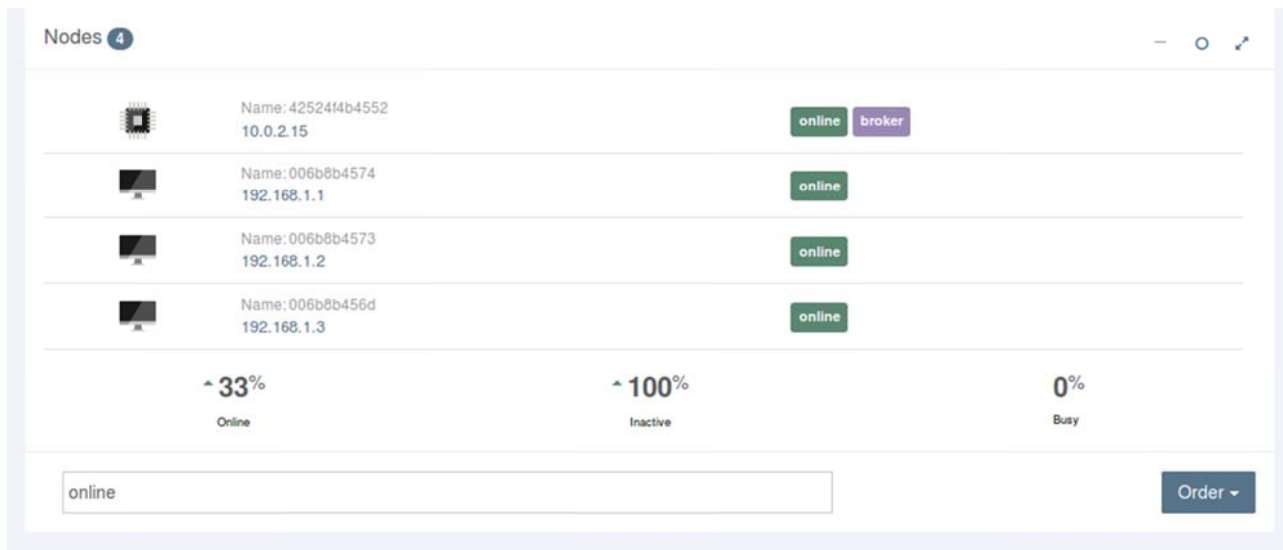


Figure 11: Filtering the node list

6. Experiments with Dedalus

We briefly note that we have performed experiments using Dedalus with an experimental network of 15 Raspberry Pis, in three different settings:

- Our laboratory, which is located in a four-story building with many walls made of plywood. Experiments there were mostly used for testing.
- The Skaramagas RAWFIE testbed. In this case, the Pis were placed in zip-lock bags and fastened to USVs, as shown at the left of Fig. 9. The USVs were spread over a sea area roughly 200 meters by 200 meters, as shown at the right of Fig. 9.
- Some experiments were also conducted in the main building of AUEB, a 100-year-old four-story building of thick brick-and-mortar walls occupying almost a complete city block and comprised of large auditoriums spread in the ground and basement floors and smaller rooms and halls spread in the other stories.

Reporting results of these experiments is beyond the scope of this work. The interested reader can refer to the other available documentation (see the next section).



Figure 12. Experiments at the Skaramagas RAWFIE Testbed

7. Other Documentation

For developers interested in extending the functionality of Dedalus, another document is available in the project website [UN], providing a detailed documentation of the Dedalus source code. A demonstration of Dedalus took place in WinTECH 2018; some details of that demonstration appear in [ED18b]. Dedalus was presented on the Athens Researcher’s night in and a poster is available in the project website [UN]. Finally, the UNSURPASSED project, including a description of Dedalus, was presented in [ED18a].

Acknowledgment

This work has received funding from the European Union’s Horizon 2020 Research and Innovation programme under grant agreement No. 645220 (Road-, Air- and Water-based Future Internet Experimentation - RAWFIE) through the National and Kapodistrian University of Athens. Administration support was provided by the Research Centre of the Athens University of Economics and Business.

We would like to gratefully acknowledge the crucial help of Kostas Kolomvatsos, Kakia Panagidi, Kostis Gerakos, Spyros Bolis, and the Hellenic Navy personnel for helping us with our debugging, providing information about the RAWFIE testbed, and helping us with conducting our experiments.

Finally, we would like to gratefully acknowledge the support of the senior MMLAB personnel M. Karaliopoulos, V. Siris, I. Koutsopoulos, and G. Polyzos, for their hard work in shaping and overseeing the conducted experiments and their numerous comments and suggestions.



References

- [BA] <https://github.com/jech/babeld>
- [BMX7] <https://lists.bmx6.net/pipermail/bmxd/2016-March/000034.html>
- [BU] <https://tools.ietf.org/html/rfc5050>
- [CCN] <http://ccn-lite.net/>
- [CHA] <https://github.com/JHUISI/charm>
- [ED18a] E. Aliaj, G. Dimaki, P. Getsopoulos, Y. Thomas, N. Fotiou, S. Toumpis, I. Koutsopoulos, V. Siris, G. C. Polyzos, “A platform for wireless maritime network experimentation”, in GIIS 2018, Thessaloniki, Greece, 2018.
- [ED18b] E. Aliaj, G. Dimaki, P. Getsopoulos, Y. Thomas, N. Fotiou, S. Toumpis, I. Koutsopoulos, V. Siris, G. C. Polyzos, “Wireless Maritime Networking Experiments with Dedalus”, in WiNTECH 2018, New Delhi, India, 2018.
- [GA07] M. Green, G. Ateniese, "Identity-Based Proxy Re-Encryption," in Applied Cryptography and Network Security. Springer Berlin/Heidelberg, 2007
- [IBR] RFC 6126, <https://tools.ietf.org/html/rfc6126>
- [MD] <https://rfc.zeromq.org/spec:7/MDP/>
- [ND] <https://named-data.net/project/>
- [RAW] <https://www.rawfie.eu>
- [UN] <https://mm.aueb.gr/projects/unsurpassed>
- [ZL14] Zhang, Lixia, et al. "Named data networking." ACM SIGCOMM Computer Communication Review 44.3 (2014): 66-73.