POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Application-gateway in a DTN environment

Supervisor
prof. Fulvio Risso

Candidate
Saverio Passeri S265606

March 2021

Perseverance and resilience were the fuel of my engine.

Acknowledgements

Is it with a great pleasure that I offer my sincerest gratitude to professor Fulvio Risso who gave me the opportunity to seize this project.
I would like to thank Dr. R.Loti who allowed me to work inside Tierra.
A great recognition goes to goes to Calogero Carrabbotta who has extensively supported me, in every needed moment.

Infinite recognition to my family who always believed in me, my friends, especially Francesco, that is a milestone in my life and my girlfriend Marta, that was always present particularly in the difficulty moments and in the moments of swinging mood.

Abstract

The aim of this project is to provide a proxy service to the Internet for any nodes in the DTN network. As a starting infrastructure we used Aether that is an application that tries to reproduce a DTN (Delay tolerant network) and that implements the bundle protocol which was designed for networks such as challenged networks. Basically Aether provides all the functionality via API, so any device can interact with the DTN using some calls provided by Aether. We started by analyzing the existing proxy projects, then we chose one of them to make it compliant with the bundle protocol and "attach" it transparently to Aether. We then studied the types of proxies that exist and three cases emerged: the forwarding proxy, the reverse proxy and the open proxy. The forwarding proxy has the objective of providing greater security within a network locare ensuring a single point of access to the Internet and controlling access and traffic passing through it. Reverse proxy allows users to connect indirectly to a remote server. Open proxy is a forwarding proxy that is accessible by any user in the network. In general, proxies are used as firewalls between the user and the Internet, as in the case of forwarding proxy, which replace the user's IP with a unique IP that is the same for all, which "hides" the local network. Proxies that act as firewalls are distinguished in two cases, according to where they are in the OSI protocol stack; in fact we speak of application-firewall proxy in the case in which each application protocol requires its own dedicated firewall. Much more efficient is the firewall that is working in the session layer, which therefore has a blurry view of which application is being used but which manages connections as sessions and therefore has a view that can optimally manage each TCP/UDP connection. So you could have an SMPT request on port 80 without any further control on the application protocol used. Today one of the most reliable and used protocols to create a session-layer firewall is SOCKS. SOCKS is referred to as a generic proxy protocol for TCP/IP based network applications. SOCKS, short for "SOCKETS", consists of two components: A SOCKS client and a SOCKS server. The fundamental purpose of the protocol is to enable hosts on one side of a SOCKS server to access hosts on the other side of a SOCKS server, without requiring direct "IP reachability". The socks protocol performs some action, such as making connection requests, setting up proxy circuits, relaying application data and performing user authentication. The proxy keeps a table of all sessions and connections. It maps the IP addresses and port numbers from inside to a single IP address and the corresponding port number. This function is called NAT (Network Address Translation). The protocol has some phases, firstly the Client contacts

proxy instead of the originally addressed server, the the proxy evaluates client request and decides which to pass or which to reject. If the proxy accepted the request, it transfers request to originally addressed server on behalf of client. Then the Server answers to proxy without knowing about client behind At the end the proxy passes answer to requesting client. The client, on the other hand, has to run socksified, that is it has to send the requests via SOCKS protocol. In a nutshell, the socks client wraps all network-related system calls made by a host with their own calls to the socket so that the host network calls are sent to the socks server on a designated port (as said before, the port 1080). After a careful review of some projects, 3Proxy was chosen. 3Proxy is a combination of many specific application proxies, then in general it can be considered a general server proxy. It is possible to use every proxy as a standalone program (socks, proxy, tcppm, udppm, pop3p) or use combined program. In fact, 3Proxy is a shim-layer proxy that accepts connections on different port. In our case of study, 3Proxy will be mostly used as a SOCKS server and will wait on port 1080 for incoming connections and will use only this port to communicate with the client applications. 3proxy is implemented as multithread application. Server model is implemented as "one connection – one thread". It means, for every client connection new thread is created. The program starts with the main thread, that has the aim to read and parse the configuration file. Each command of the configuration file, starts a new thread, called service thread. The service thread loops waiting for requests from the client, and spawn a new thread, the client thread. The last one, has the to perform the SOCKS handshake and to relay the connection. One focus on the fact that SOCKS can handle whatever application, while the other services are specialized and oriented only to one application. To hack into 3Proxy was used a plugin. 3Proxy plugin is any dynamic/shared library. There is no specific requirement for plugin, actually you can load any dynamic library with 'plugin' command. No linking with any libraries are required. However, to interoperate with 3proxy dynamic library must have an export function 3proxy may call to pass the structure with required information. Because there is no linking between 3proxy and plugin, all 3proxy functions and structures are passed with a special structure called Pluginlink. Pluginlink is actually a collection of pointers to 3proxy internal structures and functions. One insight was to put all the pointers to functions in that structures to allow the plugin to interact with the original workflow of the project. With a simple call of this functions inside 3Proxy was possible to switch in the bundle context and execute commands and actions that interact with Aether in such a way that 3Proxy was not involved and was not aware of this context switch. Besides not only pointers were stored inside that collection but also data structures and variables useful for the bundle plugin. There are three actors involved: the client node making the request, the intermediate nodes and the gateway node that has the router role. As we have seen, there is the DTN layer that allow the connection between the two sides. It is worth to know that the Aether boxes are the application that runs over each device and that there are more than one connection instantiated between the nodes and Aether. The task of Aether is to route packets to the correct destination, send and receive data in bundle format and provide the application data to the proxy. From now on we call 3Proxy2Bundle the original 3Proxy project plus all the modifications and plugins, that are essential to the proxy to "talk" the bundle protocol. Basically, the original architecture of 3Proxy is modified in such a way that the socks server is splitted in two sides, client and server's. The client side provides the function to accept connections of the clients and to send the requests to the gateway. Therefore, it relays the connection between the client and the gateway. As we can see in the figure below, the client application establishes a connection with 3Proxy2Bundle listening on port 1080. Since on the server side, all connections were flowing to a single node listening on the DTN, it was necessary to create a nating table to map all connections. In this case, the remote EID was used to trace back to whom the packets had to be delivered once the response from the server was obtained. Various tests were performed to verify the performance of the solution. The protocol used to test the application were FTP and HTTP. The former, using FileZilla gives a more detailed look at the real performance of the proxy service. It is possible to obtain the average speed of the downloads and uploads and so on. In the latter case it is possible to see how many connections are really established and closed correctly. Following a particular study is carried on the former which a focus was taken on the throughput and performance. It was noticed that to the high number of requests, it is possible that Aether cannot keep up with all of them. This brings the system, on both side, in a state that is not guaranteed stable downloads. As soon as the load becames less, the downloads rocket up to the maximum speed possible. This behaviour is justified from the fact that, on the overload situation, there is a congestion in delivering the packets; this results in slowing down the speed of the download and in some cases also the interruption of it. Generally the performance have suffered a drop due to the computational expanse of Aether intermediation. It was possible to notice a decrease of 5 times with respect to the normal speed of the proxy running alone. It is acceptable considering that the plugin has to listen on 2 sockets more, generate packets and send them. Moreover, in the case that there is a lot of packets to deliver, there is a congestion phase that tends to drastically reduce the overall throughput. In general, 3Proxy2Bundle works correctly and efficently. In some cases, there might be problems with the reordering of packages or security issues. Future work could focus on understanding and improving the fact that sometimes the connection goes down and the download doesn't terminate correctly. Performance could also be improved analysing the congestion problem in Aether.

Contents

List of Figures

Listings

Chapter 1

Introduction

The increasing spread of IoT devices, in conjunction with the rapid evolution of wireless communication technologies, introduces a number of new networking challenges. It is increasingly common to encounter contexts in which entities involved in communication are constantly on the move and characterised by limited hardware resources (Industrial IoT). In these contexts, the devices involved may be fleets of vehicles, machinery, or even sensors for precision agriculture, which may not always rely on stable WAN connections (e.g. via 3G network) and traditional communication paradigms. [13] These scenarios are identified as "challenged networks". On the Internet, communications are based on the assumption that, at all times, at least one end-to-end route between the source and destination of the traffic is guaranteed. Moreover, connections between two network devices tend to be stable. However, this cannot be guaranteed in the context of a challenged network, where devices have only short range wireless communication channels (e.g., WiFi, Bluetooth), characterised by continuous interruptions or even lack of connectivity for indefinite periods. This does not ensure stable paths between source and destination. In addition to device mobility, intermittent connectivity may result from factors intrinsic to the environment in which the challenged network is located, such as the presence of obstacles between devices, or the occurrence of adverse atmospheric and environmental phenomena. Intermittent connectivity brings with it a number of additional problems, such as network partitioning, long and/or variable delays, high information loss rates, which make communication even more complex. The scenarios subject to such problems are many and range from military to interplanetary, sensor networks in areas without any telecommunication infrastructure, such as building sites, agricultural or livestock farming, in mountainous or rural areas. Most existing IoT applications assume that they operate on connected networks with minimal or at least negligible delays. Modifying such applications to allow them to operate on challenging networks would require considerable effort. A more viable alternative approach is to introduce a network framework common to all nodes in the network, acting as a sort of interface, to meet all the typical needs of challenged networks in a transparent manner.

The Delay-Tolerant Networking paradigm represents a potential solution to the problem, able to guarantee interoperability within and between different challenged networks, by defining an abstraction from the underlying protocols. DTN architectures aim at the creation of an "overlay" network, able to operate on existing protocol stacks, within the most varied application contexts. In the case of the Internet, for example, a DTN could operate on the TCP/IP suite, while, in the case of sensor networks, it could favour the interconnection of devices using Bluetooth technology, or even communication protocols not yet standardised. For this purpose, it is necessary to extend the network stack of the devices involved in the communication, and therefore, in other words, introduce a new protocol layer: the Bundle Layer. This layer, common to all nodes participating in the DTN network, in combination with the Bundle Protocol, defines how and in which message format they can communicate with each other. Delay-Tolerant Networking, therefore, meets the need to guarantee interoperability between heterogeneous networks, each characterised by its own assumptions and protocol architectures. However, its main objective is to guarantee, with a good probability, that a packet will arrive from source to destination, despite the temporary lack of a complete path between them. The pursuit of this objective is achieved through the use of an asynchronous message forwarding mechanism, which uses an approach very similar to that adopted for e-mail, known as store-and-forward message switching. According to this approach, a message is kept locally until it is possible to deliver it directly to its destination, or forward it to some other intermediate node, considered a potential next-hop to the destination.

The aim of this project is to provide a proxy service to the Internet for any nodes in the DTN network. First of all, will be presented the current solution for forwarding data and packets within the DTN. Secondly, it is described all the kinds of proxies presently used and a focus is addressed to the **SOCKS protocol**, that is the most common solution to design a **general proxy server.** Thirdly, it will be presented the initial project over all the work spawned from, the workflow and the structures whose the project is made of. The core of the work is the implementation of the project and all the artifacts that were done to let the original project accomplish the requirements of **communicating with the Aether** and the DTN. Lastly, some test were performed to analyze the performance of the final project and also is discussed some room for improvements of the project.

Chapter 2

IBR-DTN

2.1 Delay/Disruption Tolerant Network

.

The *Delay/Disruption Tolerant Network* (DTN) define an end-to-end architecture capable of providing connectivity in the so called Challenged networks". These networks are characterised by intermittent connectivity, heterogeneous nodes and very different network conditions. The concept of Delay Tolerant Networking was born in the field of interplanetary communications, but currently finds many applications in the commercial, scientific, military and public service fields. Traditional Internet protocols are not able to provide communication efficiently, because the assumptions on which they are based are not valid for this particular type of network. Nowadays, in fact, it is increasingly common to come up against application scenarios where the devices that have to communicate are in motion and operate at limited power, this can lead to the interruption of a connection due to the presence of an obstacle, or, in certain situations, the interruption of the physical link in order to preserve energy. The consequence of these phenomena of intermittent connectivity is a natural partitioning of the network.

In such scenarios, communication via IP-based protocols is particularly inefficient. The IP protocol is based on the idea that at any given moment there is an end-to-end path linking source and destination of a packet. This is not at all conceivable in a "challenged network", which is instead characterised by intermittent connectivity, long and/or variable delays, high error rate and transmission asymmetry. Just think of TCP/IP, its use to communicate within an unstable network would cause a significant amount of lost data. In fact, in the case of a packet that cannot be forwarded immediately, TCP will assume network congestion, discard the packet and try to retransmit it by gradually lowering the retransmission speed until the session is closed if the intermittence is too high.

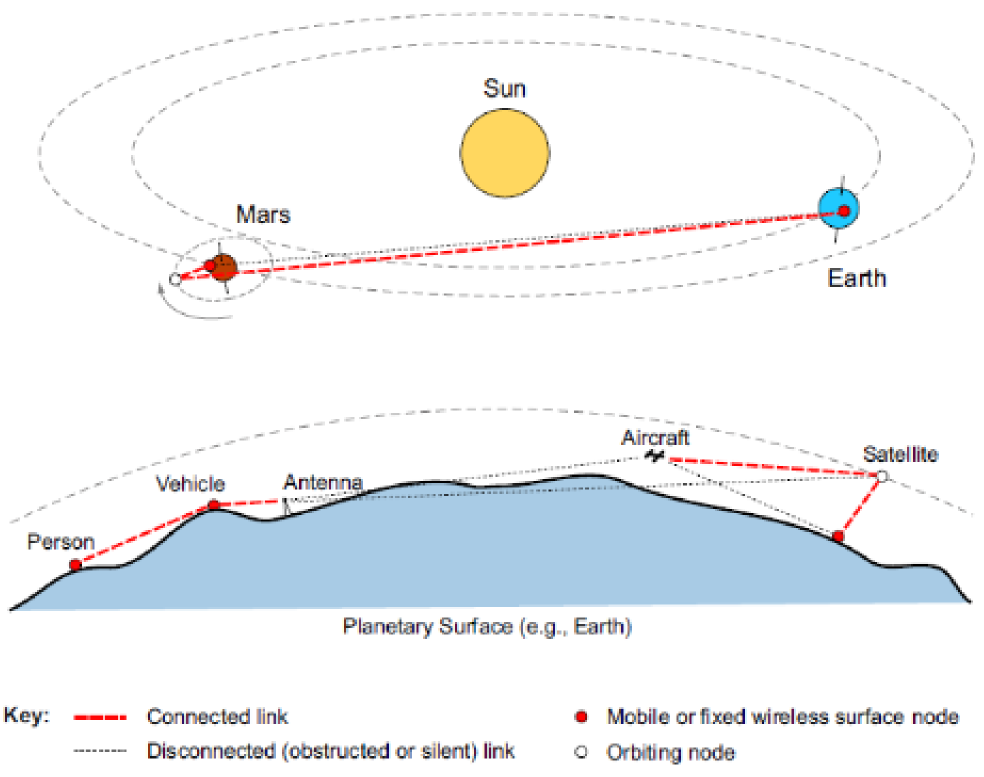

Sun
Mars
Earth
Aircraft
Satellite
Vehicle
Antenna
Person
Planetary Surface (e.g., Earth)
Key:
Connected link
Disconnected (obstructed or silent) link
Mobile or fixed wireless surface node
Orbiting node

Figure 2.1: Examples of planned contacts (interplanetary communications) and opportunistic contacts (communications on the earth's surface).

Furthermore, when talking about intermittent connectivity, it is appropriate to distinguish between planned and opportunistic contacts (3.2). The typical scenario of planned or *scheduled* contacts is that of space, in which nodes move on predictable orbital paths, so much so that it is possible to predict or receive the instants in which they will occupy their future positions and therefore organise future communication sessions. Planned contacts, therefore, require the time synchronization of the entire DTN. By opportunistic contacts, on the other hand, we mean contacts between a transmitter and a receiver in unplanned instants. This is the case of people, vehicles, planes or satellites who may want to exchange information when they are close enough to communicate using their power, albeit limited.

To deal with the typical problems of "challenged networks" and benefit from planned and/or opportunistic contacts, DTNs use the *store-and-forward message switching*. According to this paradigm, similar to the mechanism used for e-mail, entire messages or fragments of them are moved from the storage of one node to another, along a path that potentially leads to the destination. When a node receives a packet, it is forwarded immediately if possible, or stored locally for future transmission. For this reason, each DTN router must have a medium that allows messages to be stored indefinitely (a hard disk, for example), guaranteeing the persistence of the information. This is in contrast to what happens in IP routers, which use memory buffers to queue packets waiting to be forwarded, guaranteeing a persistence in the order of milliseconds. Storage must be persistent because some communication links may not be available for long periods of time, in situations where retransmission of a message is required or in the case of a node that transmits and/or receives data much faster than a neighbour.

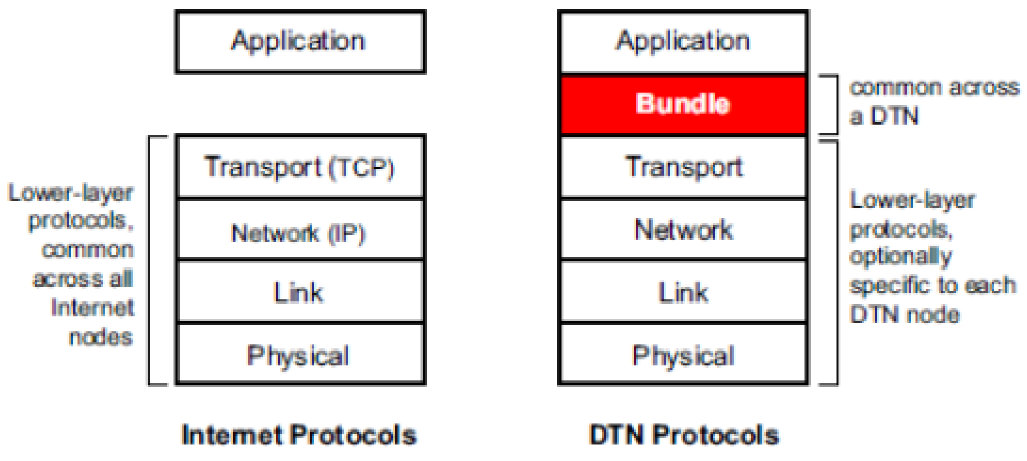


Figure 2.2: Comparison between an Internet stack (left) and a DTN stack (right).

The DTN creates an overlay network by introducing a new abstraction layer, the *Bundle Layer*, which extends the network stack of nodes participating in the DTN by placing itself between the application layer and the transport layer. The main objective of this layer is to make application programs agnostic with respect to the transport layers used, favouring the creation of heterogeneous networks. Two nodes that want to establish communication will interact with the Bundle Layer, without worrying about the nature of the protocols used in the lower layers. The bundle layer will be responsible for routing these messages, called Bundle, from source to destination. DTNs use an asynchronous non-conversational model, in contrast to the typical TCP/IP family's request/response communication mechanism. Conversational protocols, such as TCP, involve long RTTs and often fail. The Bundle Layer communicates via a non-conversational protocol that minimizes the round trips needed to confirm transmissions, making acknowledgments optional.

The unique feature of positioning between the transport layer and the application layer allows DTN to be used to create application proxies.

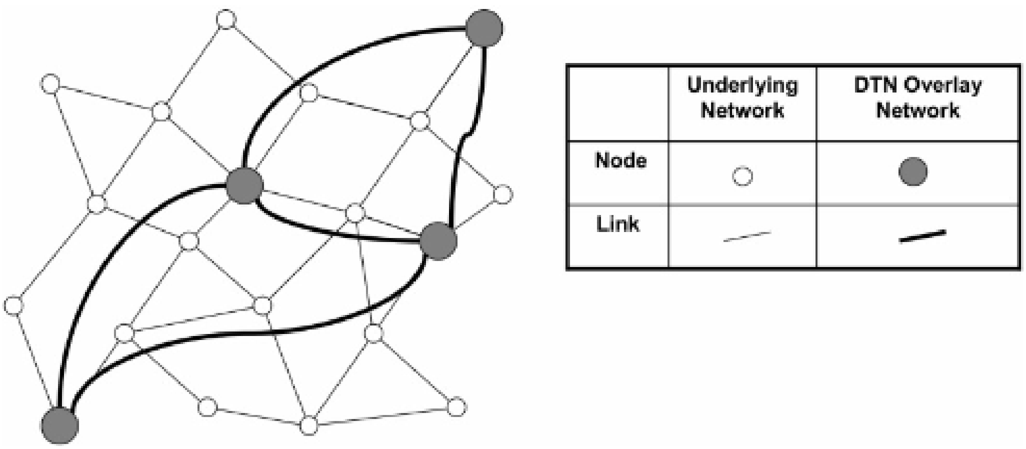


Figure 2.3: DTN network overlay on another type of network.

Take as an example applications running on TCP/IP, they typically use the Berkeley socket API, and have no access to DTN services. Furthermore, if they wanted to use them, they would have to be written in a way that would be tolerant of interruptions and delays, and they may need numerous message exchanges to perform their operations, such as SMTP. Rewriting applications to take advantage of the API would require changes to all applications. The other use we can assume of DTN is to create an Application Layer Gateway. It would be a protocol terminator, and it would take the information needed to recreate the same dialogue as the client, so that it could be re-submitted to the server and get the desired response.[16].

2.2 Bundle Protocol

The *Bundle Protocol*[17] is an experimental protocol, corresponding to the DTN architecture BundleLayer, developed within the Delay Tolerant Networking Research Group (DTNRG) of the IRTF.

2.2.1 Architecture

In the context of DTNs, the term *bundle node* refers to an entity capable of receiving and transmitting bundles. According to the specifications of the Bundle Protocol, a *bundle node* is conceptually made up of three basic components:

- **Bundle Protocol Agent (BPA)**: is the service provider of the bundle protocol. The way in which these services are offered depends on its implementation. In fact, the BPA can be implemented in hardware, as a shared library between multiple nodes on a single machine, as a process (a daemon) with which nodes on one or more machines can interact through communication mechanisms between processes or network communication (e.g. Socket Berkeley).

- **Application Agent (AA)**: uses the bundle layer services to communicate. The AA is generally composed of two elements, an administrative one and an application one. The administrative element builds and requires the transmission of administrative records (status reports and custody signals) and processes the custody signals received from the node. Typically it is integrated in the implementation of the BPA. The application element, instead, builds, transmits and processes the actual application data and can be implemented in software or hardware. Communication between the application element of the AA and the BPA takes place via the service interface displayed by the latter. A node that only has a "router" function may not have any application element.

The main services that a BPA should provide to the AA of a node are the following:

- recording of a node at an endpoint;
- termination of the recording;
- transmission of a bundle to a specific endpoint;
- cancellation of transmission;
- delivery of a received bundle.

2.2.2 Encapsulation

The Bundle Protocol extends the hierarchy of encapsulation achieved by Internet protocols by simply encapsulating them without altering the data. Figure 2.4 shows an example of encapsulation of TCP/IP protocols. In the case of bundles that are too large, the bundle layer should be able to split messages into several fragments, quite similar to how the IP layer fragments its packets. In the case of fragmentation, it's up to the destination node to reassemble the fragments in the correct order to get the original bundle.

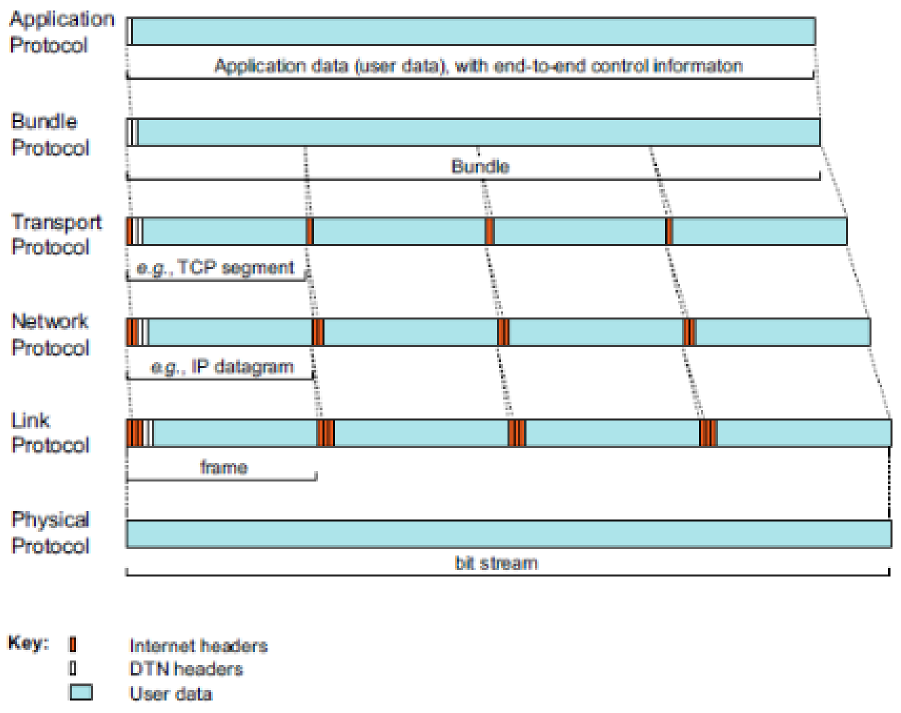


Figure 2.4: Encapsulation of TCP/IP protocols in the Bundle Protocol.

2.2.3 Fragmentation

To ensure that contact volumes are used to the full and to prevent retransmission of partially submitted bundles, DTN offers a fragmentation mechanism. Two types of fragmentation are provided by DTN: proactive and reactive. *proactive* fragmentation is arbitrarily chosen by a node forwarding the Bundle, then the node, or nodes, to which the fragments are sent will reassemble them. Reactive fragmentation occurs as a result of the incomplete transfer of a bundle to a node. The receiving node will decide to treat the received portion as if it were a fragment, and the sender to send the remaining portion as if it were a second fragment, either directly to the recipient or by passing through other nodes if the topology changes. Only *proactive* fragmentation is mandatory. Fragmentation at the Protocol Bundle level is supported using a header that indicates the length and offset of the fragment from the original bundle, using a mechanism similar to that used in IP. Fragments originating from the same bundle will be identified by source, destination, and creation time. For a bundle, you can also request non-fragmentation via one of the

Control Flags in the primary block. In addition, all blocks before the payload are placed in the minor offset fragment, and those after the payload block are placed in the major offset fragment.

2.2.4 Addressing

The source and destination of a bundle are identified by an *Endpoint Identifier* (EID). Each EID complies with the Uniform Resource Identifier (URI) format and consists of two parts: <scheme-name>:<scheme-specific part (SSP)>. The length of both fields must not exceed 1023 bytes. The representation schemes proposed for the EID are multiple, but conventionally schemes conforming to the URI (Unified Resource Identifier) scheme are used, and characterized by a <scheme-specific part> divided into two portions: the first indicating the node, the second the *demux-token*, i.e. a single application. One of the most common schemes is the one identified by the string **dtn**, which takes the form **dtn://node/demux-token**. While the presence of the **node** is mandatory, the **demux-token** may also not be present, as in the case of administrative bundles directed to the node's BPA. An EID typically represents a single node (or rather an application on a single node) and is called a Singleton, but it can also represent a group of DTN, "multicast" or "anycast" nodes, groups containing several nodes.

2.2.5 Bundle format

Each bundle consists of the concatenation of at least two blocks. The first block in the sequence, or *primary block*, contains information similar to an IP header that is needed to route the bundle to its destination. Each bundle can only have one primary block, but can be followed by a series of blocks to support protocol extensions, such as the Bundle Security Protocol (BSP). It can exist in blocks following the first, at most one payload block. Most fields have variable lengths and use a compact notation called *self-delimiting numerical values.* (SDNVs) (ref.), scalable for a variety of network protocols and payload sizes.

Primary Block

The Primary Block (Figure 2.5), in addition to version, block length, source, and destination, contains a number of typical Bundle Protocol information.

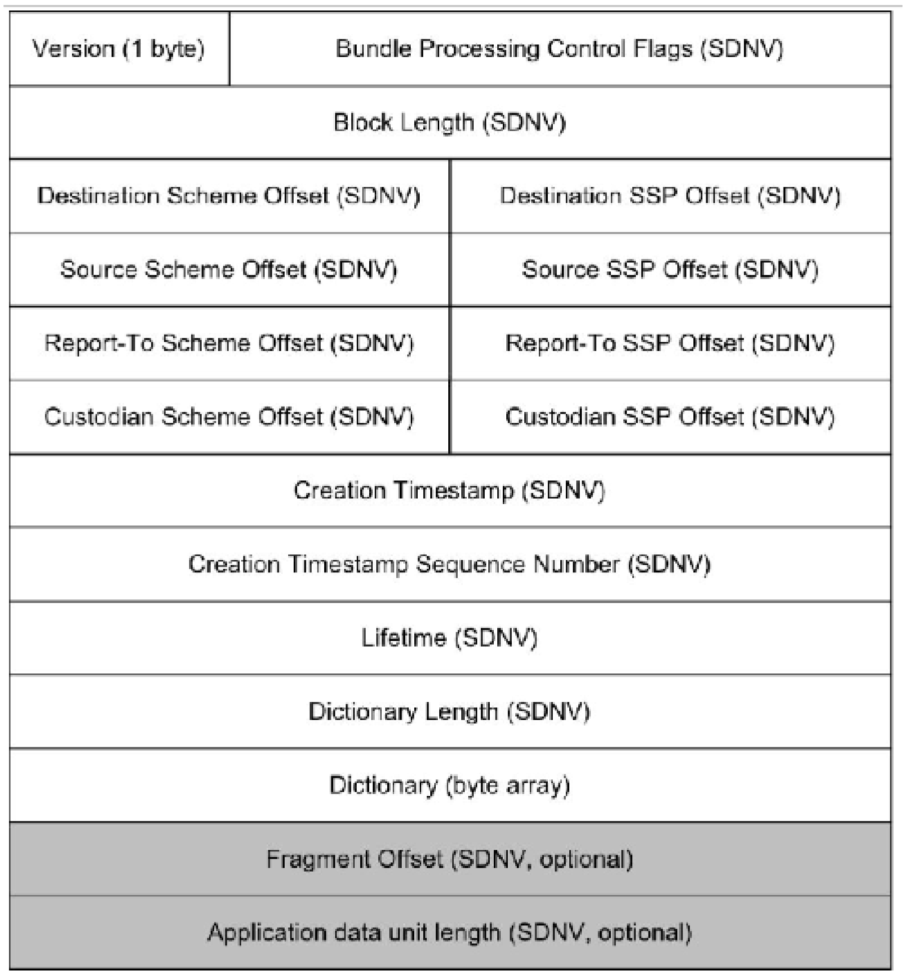

Version (1 byte)
Bundle Processing Control Flags (SDNV)
Block Length (SDNV)
Destination Scheme Offset (SDNV)
Destination SSP Offset (SDNV)
Source Scheme Offset (SDNV)
Source SSP Offset (SDNV)
Report-To Scheme Offset (SDNV)
Report-To SSP Offset (SDNV)
Custodian Scheme Offset (SDNV)
Custodian SSP Offset (SDNV)
Creation Timestamp (SDNV)
Creation Timestamp Sequence Number (SDNV)
Lifetime (SDNV)
Dictionary Length (SDNV)
Dictionary (byte array)
Fragment Offset (SDNV, optional)
Application data unit length (SDNV, optional)

Figure 2.5: Format of the primary block.

Bundle Processing Control Flag

Processing Control Flags bundles are a string of bits useful for processing the bundle. They are divided into 3 categories:

- General [0-6]: specify general information about the bundle, e.g. whether it is regular or administrative, the state of fragmentation, whether the destination is a singleton EID, whether acknowledgement or custody transfer is required.
- Class of Service [7-13]: specify the priority of the bundle, where a high value indicates a high priority, and other information useful for the routing of the bundle.
- Status Report [14-20]: specify the reports required for this bundle, e.g. if a

delivery report, forwarding report, custody acceptance report, etc. is required.

Priority

Two "Class of Service" bits are used to define the priority of the Bundle. Typically, this applies only between bundles with the same source, and may not apply to bundles with a different source.

Three values have been used so far:

- Bulk: means bundles that must be shipped with minimum effort, delivered only at the end of delivery of all bundles with the same source and destination.
- Normal: for bundles that are shipped before Bulk priority bundles.
- Expedited: for bundles with higher priority, to be shipped before Normal and Bulk priority bundles.

Endpoints

The primary block includes four EIDs of variable length, each encoded via a pair of offsets: one for the scheme, the other for the SSP. These offsets are nothing more than pointers to the strings representing the EIDs stored within the dictionary later placed in the block.

- Source: contains the endpoint from which the bundle comes,
- Destination: is the destination endpoint of the bundle,
- Report-to: indicates the node to send status reports to for events involving the bundle,
- Custodian: identifies the last node that accepted the custody of the bundle.

Since EIDs make up most of the overhead bytes due to the Bundle Protocol, the dictionary is a mechanism for reducing the amount of space required for their storage. For example, if the source and report-to EIDs coincide, two references to that EID will appear, but only one string within the dictionary.

Time

Other significant information for the elaboration of a bundle are the *creation timestamp* and the *lifetime*. The *creation timestamp* indicates the bundle creation time, expressed as the number of seconds since the beginning of the year 2000 in the UTC time zone. This value is calculated at the instant the BPA receives the transmission request. The *lifetime*, on the other hand, represents the life time of the bundle, expressed as an offset to the creation time. Using *lifetime* allows you to eliminate excess bundles within the network, as each time a node receives a bundle that has finished its life time, it discards it. Since both the *creation timestamp* and the *lifetime* use real time, the nodes participating in the DTN need to be synchronised, albeit coarsely.

Other blocks

In addition to the Primary Block within a bundle, several other blocks can be inserted. As you can see in Figure 2.6, each of these blocks is identified by the *Block Type*, an 8-bit string. The value '1' indicates a payload block and a bundle can contain a maximum of one, values between 192 and 255 are for experimental and private use, while the rest are reserved for future use. All blocks other than the primary block and the payload are called extension blocks. Then there are the block control flags, which give directions on how the block should be handled. Finally, the leotard and their length complete the block. It's also possible to insert a reference to some EIDs contained in the dictionary. A counter will trace them and two pointers, one at the beginning of the pattern and one at the beginning of the SSP in the dictionary for each entry.

Block Processing Control Flag

The *Block Processing Control Flags* are a string of bits useful for block processing. It is an SDNV field currently made up of 7 bits, indicating some special features on the block. For example, you can replicate the block in each fragment (in case of fragmentation), indicate to discard the block or the whole bundle, or send a report if you are unable to process the block, if it contains EID-References, and especially the flag indicating if it is the last block in the Bundle. The replication bit in the fragments, however, cannot be set to one on the blocks following the payload block.

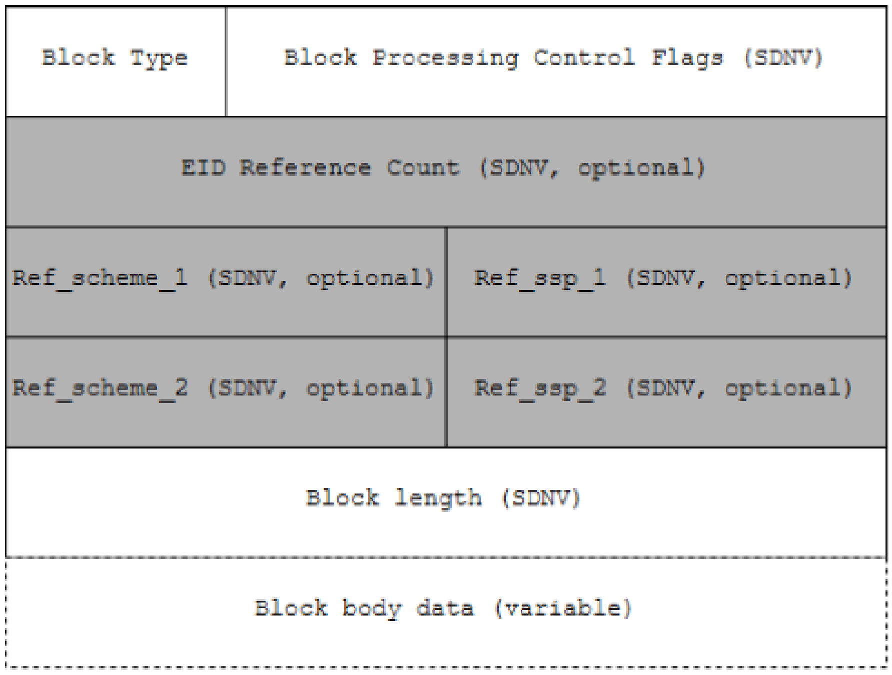

Block Type
Block Processing Control Flags (SDNV)
EID Reference Count (SDNV, optional)
Ref_scheme_1 (SDNV, optional)
Ref_ssp_1 (SDNV, optional)
Ref_scheme_2 (SDNV, optional)
Ref_ssp_2 (SDNV, optional)
Block length (SDNV)
Block body data (variable)

Figure 2.6: Generic format of a secondary block of a bundle.

2.2.6 Transmission reliability

DTNs support mechanisms for retransmission of lost and/or corrupted data at both the transport protocol and the Bundle Protocol level. However, since DTNs typically have a heterogeneity in the transport protocols used by nodes, reliability must be achieved at the Bundle Protocol level, through a node-to-node retransmission mechanism called *transfer to custody*. Basically, when the current custodian of a bundle needs to forward it, it requires custody transfer and triggers a retransmission timer. If the BPA of the receiving node decides to accept custody, it sends an acknowledgement to the sender. If no acknowledgement is received before the timer expires, the bundle is retransmitted. The value of the retransmission timer can be distributed to the nodes along with the routing information or calculated locally by the nodes themselves, depending on their past experience. The current caretaker of a bundle is then the node responsible for keeping the bundle in persistent memory until it is received by a new caretaker. A DTN node does not necessarily have to offer the custodian transfer service. A node may, for example, refuse a custody transfer request due to a lack of available resources, policy or implementation issues. However, in a context of minimising the number of losses, it would be appropriate for all nodes to use custody transfer, provided that the necessary storage resources are available and that the frequency of bundle generation does not exceed that of delivery, as well as the buffering capacity of the network. Thus, the custody transfer mechanism, combined with the use of persistent storage on the intermediate nodes, makes it possible to delegate responsibility for reliable transfers to portions of the network rather than to the sender of the bundle. Unfortunately, this is not enough to guarantee the reliability of the transmissions, but only to improve them. A further step can be taken by using the return receipt, a message confirming the delivery of a bundle to the sender of the bundle. However, too many bundles or fragments of bundles can lead to over-consumption of available storage resources and congestion of DTN. In the event of congestion, a node may adopt several strategies: remove from storage copies of bundles that have run out of life, which should be undertaken regularly anyway; transfer bundles to others; not accept bundles with transfer to custody, rather than regular bundles; delete bundles that have not expired, even if the node is the custodian. The use of the latter option is strongly discouraged as it is clearly contradictory to the core principles of the DTN.

2.3 IBR-DTN Node

2.3.1 Introduction

IBR-DTN is the application chosen for the creation of a DTN infrastructure that, once installed on devices, allows its insertion in the network and manages communication via bundle. Modular and lightweight IBR-DTN was created by the DTN research group of the Technische Universität Braunschweig. Designed to be installed on embedded systems it provides the developer with a framework to create DTN applications [15].

2.3.2 The architecture

The version of IBR-DTN for traditional operating systems has been developed in C++. As can be seen in figure 2.7, the implementation of the IBR-DTN bundle protocol is characterised by a highly modular organisation, allowing developers to extend the software in a simple and non-invasive way. The Protocol Agent Bundle is implemented as a daemon process and exposes a socket-based API that applications can contact to interact with the Layer Bundle. By default, the API is available on TCP port 4550 in text and binary format. You can refer to the documentation[11] for more information on the features exposed.

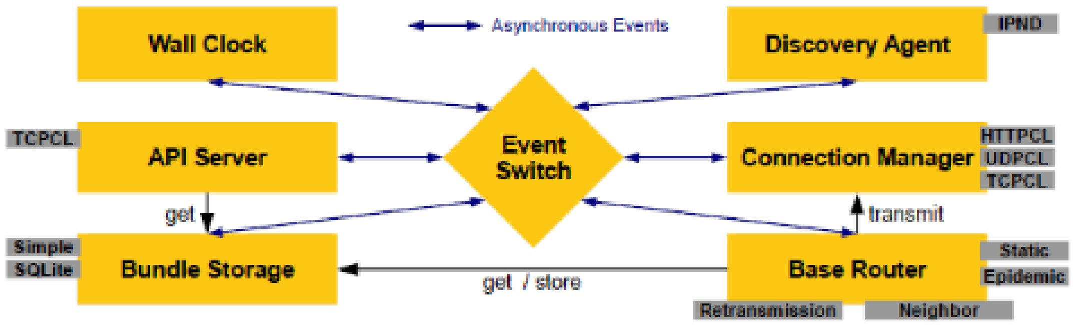


Figure 2.7: IBR-DTN Architecture

Event Switch

The modules are connected flexibly and communicate with each other via an event-based mechanism, making the *Event Switch*, responsible for entrusting the management of individual events to the corresponding sub-modules, essential. All modules can receive or trigger events to communicate with other parts of the software. A series of events are integrated in the current implementation to notify storage operations, the presence and disappearance of nodes in the neighbourhood, bundle routing operations, etc.

Discovery Agent

Another important component is the *Discovery Agent*, responsible for discovering nodes in the neighbourhood. Under the assumption that it wants IP nodes to communicate, IBR-DTN uses a module that implements the DTN IP Neighbor Discovery (IPND)[6]. This module listens to small UDP datagrams called *beacon*, used by nodes to announce their presence to neighbours, and periodically announces itself through the same datagrams. The *beacons* are sent to a known multicast IP address (and can be specified in configuration) and contain the sender's EID, to allow the receiver to bind the EID to the IP address of the neighbour.

Connection Manager and Convergence Layer

The module *Connection Manager* is responsible for managing connections with neighbouring nodes and for sending and receiving bundles. The *Connection Manager* in turn uses different *convergence layer* for the implementation of the information transfer. As described by RFC 5050 on the Delay Tolerant Network, it is the reconvergence layer that handles communication between two nodes. Each defines an interface to the underlying transport layer, allowing bundles to be transferred from the lower layer protocols. The convergence layers used are specified in the daemon configuration. Currently, IBR-DTN offers convergence layers for TCP/IP[5], UDP/IP, HTTP, Bluethooth, IEEE 802.15.4 LoWPAN, and thanks to this thesis work, for V2X. There is also an extension of TCP/IP CL for TLS support.

Bundle Storage

Since DTNs are based on the store-and-forward paradigm, each node must be able to store bundles for a certain period of time. In IBR-DTN storage interaction is managed by the *Bundle Storage*, a module that provides primitives for reading, deleting and storing bundles to/from storage. Several storage mechanisms are supported: in RAM memory, on disk (file-system) and on SQLite databases.

Base Router

The routing of bundles is carried out by the *Base Router* module, which manages the forwarding of the bundles in charge. The *Base Router* divides its work between the different routing modules. Each of them implements a specific DTN routing algorithm and is attached to the *Base Router* as a sort of plugin. All routing modules are notified by the *Discovery Agent* when neighborhood events occur and by the *Bundle Storage* when a new bundle arrives at the daemon. The routing module that will be responsible for forwarding the bundle will then contact the Connection Manager to activate the appropriate convergence layer. IBR-DTN features modules for static, epidemic and PRoPHET and MaxProp routing support.

API Server

Interaction with IBR-DTN is obtained via the server API. The server API exposes on a socket interface, configurable via config file, a textual protocol with which to make requests to the application core. The commands to be sent must follow the specific logic and syntax of the action to be executed, when a command is sent and all the information it requires is entered, you always receive a response format like <status-code> <message> [Additional data], as shown in Figure 2.8.

Service Discovery

It is a module created to propagate the services present on each node, within the DTN network. By services we mean the description of the capabilities of an application, resident on the node, that can use the DTN network to send and/or

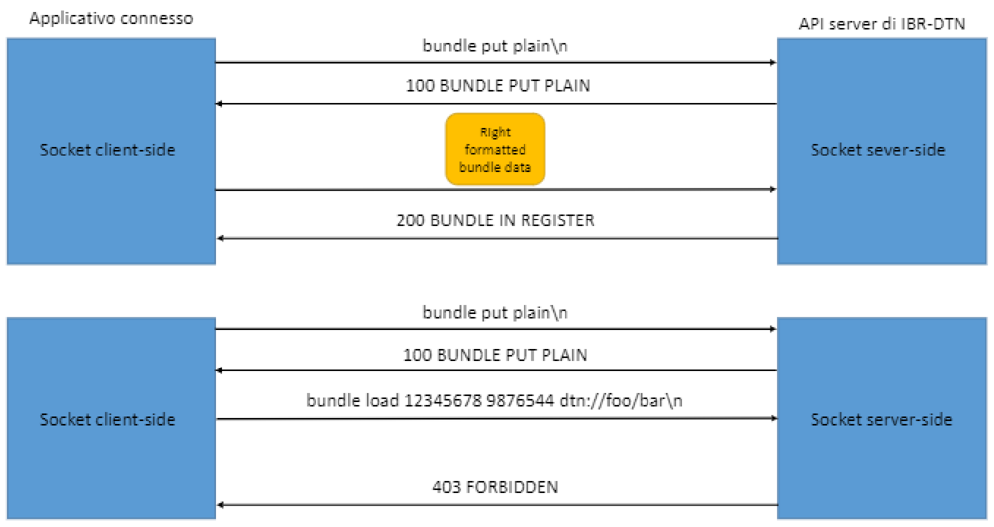


Figure 2.8: Interaction API Server.

receive data. Service Discovery allows all nodes to know about this service thanks to multicast messages sent to a group DTN address. Thanks to Service Discovery, applications can select the DTN node that comes closest to their requests and start sending packets to it.

2.3.3 Setup and startup

This paragraph explains the steps required to install and start the IBR-DTN daemon. The procedures used are valid for Linux,Debian and derivative (Raspbian) distributions.

Setup

The first step is to install the necessary libraries. Actually, the installation of some of the libraries listed below is optional, as they are useful when adding optional modules.

```
$ apt-get install build-essential libssl-dev zlib1g-dev libsqlite3-dev libcurl4-gnutls-dev
    libdaemon-dev automake autoconf pkg-config libtool libcppunit-dev libnl-3-dev
    libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev libarchive-dev git
```

The installation and activation/deactivation of the optional modules of IBR-DTN involves the use of `CMAKE`. After creating the build folder in */ibrdtn/ibrdtn*, you will launch the command `cmake ..` with the option to activate all modules your operating system can support. Then, after cloning the repository, you will configure, compile, and install the sources:

```
$ cd ibrdtn-repository/ibrdtn
$ mkdir build
$ cd build
$ cmake .. -DBUILD_ALLFEATURES=ON
$ make
$ make install
$ ldconfig
```

If you want to have a more specific control of the options, just use the cmake variables

```
$ cmake .. -DBUILD_<OPTIONS>=ON/OFF
```

Details of all possible options can be found in the *CMakeList.txt* file inside the */ibrdtn/ibrdtn* folder in the repository.

Start-up

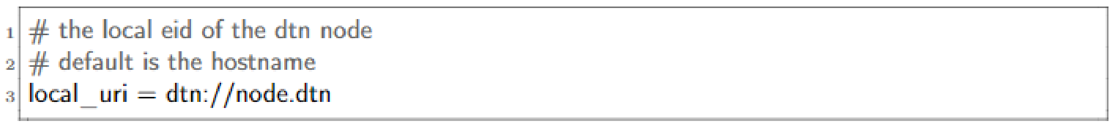
After completing the installation, you can start the IBR-DTN daemon using the **dtnd** command. This command, invoked without options, starts the daemon using the default configuration. You can use the **-i** option to specify the network interface to associate the daemon process with, or **-v** to enable printing of log messages, **-d** to choose the log level that will be printed, etc. An example of the command:

```
$ dtnd -i eth0 -v
```

With this combination of parameters we will have binding on the network interface textiteth0 and log on the console for the main information. For a complete list of available options use the **-h** flag. Once started, the IBR-DTN daemon will automatically detect the presence of daemons running on machines directly accessible via the Neighbor Discovery IP module and simultaneously announce its local EID to be discovered by others. In the default configuration, this EID uses the DTN scheme and local machine name as SSP, in the form **dtn://hostname**.

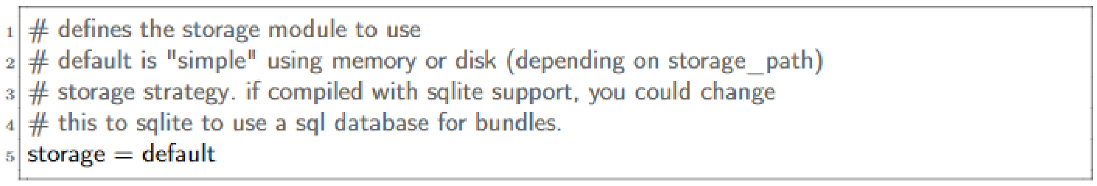
2.3.4 Configuration

To change the daemon's default behaviour you need to specify the parameters to use within a configuration file. An example configuration file can be found at the path **ibrdtn/daemon/etc/ibrdtnd.conf**, within the repository used for installation, or at the address [12]. The most significant parts are illustrated below.



```
# the local eid of the dtn node
# default is the hostname
local_uri = dtn://node.dtn
```

allows you to customize the local EID, if not specified IBR-DTN will create one for us according to the standard formatting of URI dtn **dtn://hostname**.



```
# defines the storage module to use
# default is "simple" using memory or disk (depending on storage_path)
# storage strategy. if compiled with sqlite support, you could change
# this to sqlite to use a sql database for bundles.
storage = default
```

Defines how to save bundles until the TTL expires. In volatile memory (RAM) or on disk if a save path is specified.

```
#a list(separated by spaces) of names
#for convergence layer instances.
net_interfaces = lan1 lan0

#configuration for a convergence layer named lan0

net_lan0_type = tcp # we want to use TCP as protocol
net_lan0_interface = eth0 # listen on interface wlan0
net_lan0_port = 4556 # with port 4556 (default)

#configuration for a convergence layer named lan1

net_lan1_type = tcp     # we want to use UDP as protocol
net_lan1_interface = eth1 # listen on interface eth0
net_lan1_port = 4557 # with port 4556 (default)

#configuration for a convergence layer named blue0

```

```
#net_hci0_type = bluetooth # use bluetooth as protocol
net_hci0_interface = hci0 # listen on interface
net_hci0_port = 10 # with RFCOMM channel

#configuration for a convergence layer named v2x0

net_v2x0_type = v2x # we want to use v2x as protocol
net_v2x0_interface = eth0 # listen on interface
net_v2x0_port = 0 # with no channel
```

Having different convergence layers available, it is possible to list all the interfaces available on the device in the configuration file. IBR-DTN will, therefore, try to bind the chosen protocols, thus allowing communication on bundles.

```
# routing strategy
# values: default | epidemic | flooding | prophet | none
# In the "default" the daemon only delivers bundles to neighbors and static
# available nodes. The alternative module "epidemic" spread all bundles to
# all available neighbors. Flooding works like epidemic, but do not send the
# own summary vector to neighbors. Prophet forwards based on the probability
# to encounter other nodes (see RFC 6693).
routing = epidemic
```

specifies the routing algorithm to be used by choosing from the options shown in the comments.

```
# forward bundles to other nodes (yes/no)
routing_forwarding = yes
```

enables/disables the node to forward bundles.

```
# forward singleton bundles directly if the destination is a neighbor
routing_prefer_direct = yes
```

enables/disables the direct forwarding of a bundle to its destination if it can be reached directly.

2.3.5 Configuration of the time synchronization

Time synchronisation is a critical point in the configuration of IBR-DTN. When using real devices, which do not have the possibility to have a clock always synchronized with the rest of the world, it is necessary to deactivate it.

Activating time synchronization means having the possibility to discard bundles on arrival if they are too old and therefore considered useless, but this is a behaviour that can lead to the impossibility of communication between DTN devices. The DTN network as such does not foresee that the nodes inside have perpetual access to an external time synchronization service such as NTP and it is in no way guaranteed that the devices have the internal clock set within a certain delay. The possible example is that of a device with only a bluetooth connection that is used for a few hours and then switched on again much later. The system clock of the latter will never be synchronized with the rest of the network, so if time synchronization is activated the device will never create valid bundles within the network. It is therefore advisable to deactivate this behaviour.

```
# set to yes if this node is connected to a high precision time reference
# like GPS, DCF77, NTP, etc.
#
time_reference = no
```

2.3.6 Interactive applications with IBR-DTN

In order to experience the use of the DTN Bundle Protocol, in addition to the daemon process, the IBR-DTN software provides a number of command line tools. **dtnping** sends bundles to a specific destination EID and waits for responses, measuring the return time. **dtnsend** and **dtnrecv** allow files to be transferred between DTN nodes. If you want to test the text API exposed by the daemon, you can use tools like **telnet** or **netcat**, as in the following example:

```
$ telnet localhost 4550
Trying ::1...
Connected to localhost.
Escape character is '^]'.
IBR-DTN 0.11.0 (build dfb7402) API 1.0
protocol management
200 SWITCHED TO MANAGEMENT
neighbor list
200 NEIGHBOR LIST
dtn://neighbor1
dtn://neighbor2
```

In this example, after connecting to the IBR-DTN daemon running locally on port 4550, you invoke the **protocol management** command to access the Management API. You can request the list of DTN nodes adjacent to the local node using the **neighbor list** command, as shown in the example, or send commands to manage bundles.

Chapter 3

Proxies and state of art

3.1 Proxy servers

A proxy server is a machine which acts as an intermediary between the computers of a local area network (sometimes using protocols other than TCP/IP) and the Internet. The basic working principle of a proxy server is quite simple: it is a server that acts as a 'proxy' for an application by making a request on the Internet in its place. In this way, each time a user connects to the Internet using a client application configured to use a proxy server, the application will firstly connect to the proxy server and give it its query. The proxy server then connects to the server to which the client application wishes to connect and sends the request to that server. The server then provides its response to the proxy, which then sends it to the client application. Most of the time the proxy server is used for the web, and when it is, it's an HTTP proxy. However, there can be proxy servers for every application protocol (FTP, SMTP, etc..). [14]

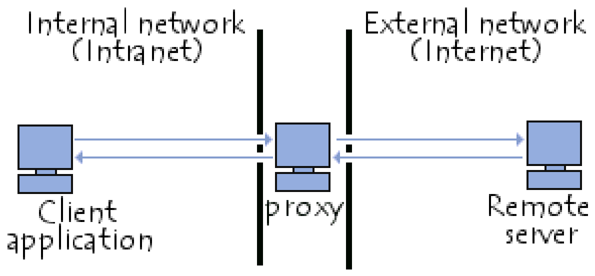


Figure 3.1: Proxy architecture

3.1.1 Types of proxy server

- **Forward Proxy:**

A forwarding proxy provides proxy services to a client or group of clients. Often these clients belong to a **common internal network**. When one of these clients makes an attempt to connect to that Internet file transfer server, its requests must first **pass through the forwarding proxy**. Depending on the settings of the forwarding proxy, it is possible to **allow or deny a request**. If allowed, the request is forwarded to the firewall and then to the file transfer server. From the file transfer server's point of view, it is the proxy server that issued the request, not the client. So, when the server responds, it directs its response to the proxy. But then, when the forwarding proxy receives the response, **it recognises** it as a response to the **request that was previously made**. And so it in turn sends that response to the client that made the request. Since proxy servers can keep track of requests, responses, their origins and destinations, different clients can send several requests to different servers via the forwarding proxy, and the proxy will act as an intermediary for all of them. Again, some requests will be allowed, while others will be refused. As you can see, the proxy can act as a **single point of access** and control, simplifying the application of security policies. A forwarding proxy is typically used in conjunction with a firewall to improve the security of an internal network by controlling traffic from clients on the internal network to hosts on the Internet. Therefore, from a security perspective, a **forwarding proxy's main purpose is to enforce security** on client computers in the internal network. [10]

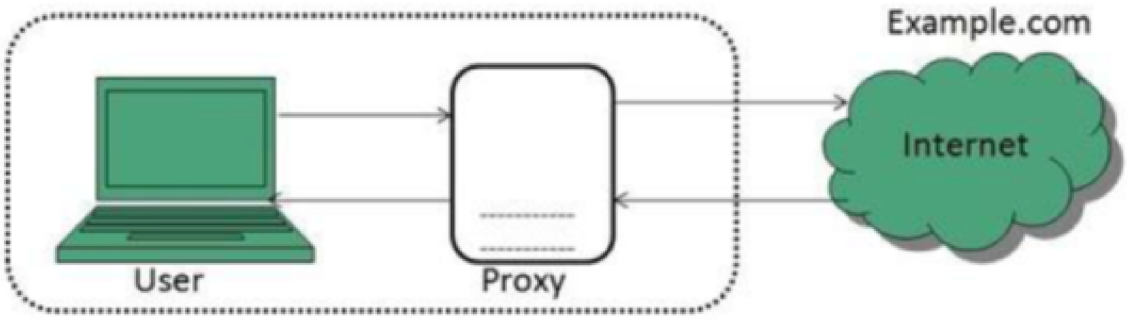


Figure 3.2: Forward proxy

- **Open Proxy:**

An open proxy is a forwarding proxy server **accessible by any Internet user**. Usually, a proxy server only enables users within a network group (i.e. a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service.

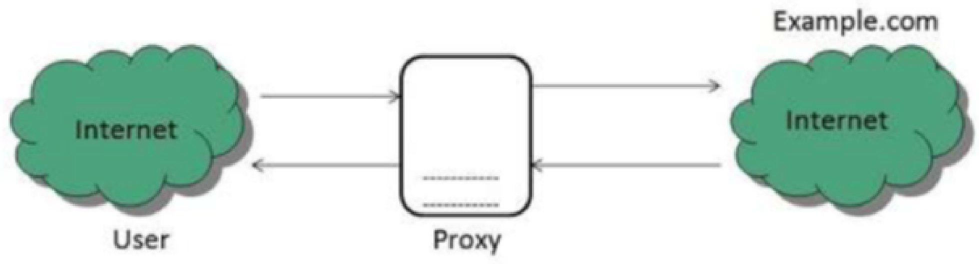


Figure 3.3: Open proxy

- **Reverse Proxy:**

A reverse proxy is a proxy server that appears to clients as a **normal server**. Requests are forwarded to one or multiple origin servers handling the request. The response is returned as if it came directly from the proxy server. A reverse proxy is a "backward" proxy cache server; is a proxy server that, instead of allowing internal users to access the Internet, allows Internet users to **indirectly access certain internal servers**. The reverse proxy server is used as an intermediary by Internet users who wish to access an internal website by indirectly sending it requests. With a reverse proxy, the web server is protected from direct external attacks, which increases the strength of the internal network. Also, the cache function of a reverse proxy can reduce the workload if the server is assigned, which is why it is sometimes called a server accelerator. Finally, with improved algorithms, the reverse proxy can distribute the workload by redirecting requests to other similar servers; this process is called load balancing. [3]

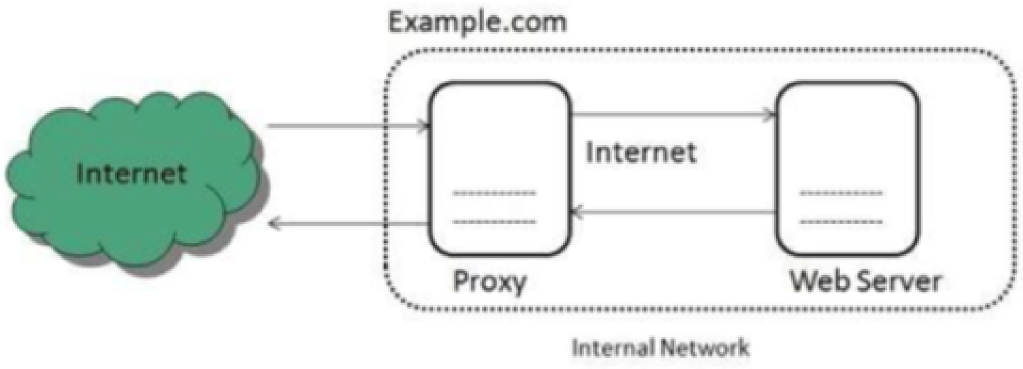


Figure 3.4: Reverse proxy

3.1.2 Proxy server as firewall

The proxy server is placed between a user's machine and the Internet. It can act as a firewall to provide protection and as a cache area to speed up Web page viewing. A firewall mechanism that replaces the IP address of a host on the internal (protected) network with its own IP address for all traffic passing through it. A software agent that acts on behalf of a user, typical proxies accept a connection from a user, decide whether or not the user or client IP address is allowed to use the proxy, perhaps perform additional authentication, and then complete a connection on behalf of the user to a remote destination. [3] An application level firewall examines the requested session to determine whether or not it should be allowed based on the source of the session requests and the purpose of the requested sessions. Such firewalls are created with the help of proxy servers. For true application-level firewalls, a **separate firewall** is needed for each different type of service. For instance, you would need separate firewalls for HTTP, FTP, SMTP and so on. Such firewalls are basically access control statements embedded in the applications themselves. As a network administrator, you place such declarations in the configuration files of the application server. A **more efficient alternative** is to use a protocol between the application layer and the transport layer - this is sometimes called the **shim layer**, to intercept application layer calls from intranet clients to connect to servers on the

Internet. Using a shim layer protocol, a proxy server can monitor all the session requests that are routed through it in an **application-independent** way to check the request sessions for their legitimacy. In this way, only the proxy server, acting as a firewall, would require direct connectivity to the Internet and the local intranet can "hide" behind the proxy server. Computers on the Internet at the large would not even know of the existence of your machine in the local intranet behind the firewall. [9]

Since shim-layer proxy servers have the characteristic of being independent of the application protocol, it was decided to go for this type of proxy. The most commonly used protocol for designing shim layer proxy servers is the **SOCKS protocol** (RFC 1928) [18].

Chapter 4

SOCKS protocol

4.1 Introduction

The use of network firewalls, systems that effectively isolate an organizations internal network structure from an exterior network, such as the INTERNET is becoming increasingly popular. These firewall systems typically act as application-layer gateways between networks, usually offering controlled TELNET, FTP, and SMTP access. With the emergence of more sophisticated application layer protocols designed to facilitate global information discovery, there exists a need to provide a general framework for these protocols to transparently and securely traverse a firewall. There is also the need to authenticate the clients so this protocol, if is required, provides authentication. The protocol described here is designed to provide a framework for client-server applications in both the TCP and UDP domains to conveniently and securely use the services of a network firewall. The protocol stands between the application layer and the transport layer.
SOCKS is referred to as a **generic proxy protocol** for TCP/IP based network applications. SOCKS, short for "**SOCKETS**", consists of two components: A**SOCKS client** and a **SOCKS server**. It is the socks client that is implemented between the application layer and the transport layer; the socks server is implemented entirely at the session layer(shim-layer).

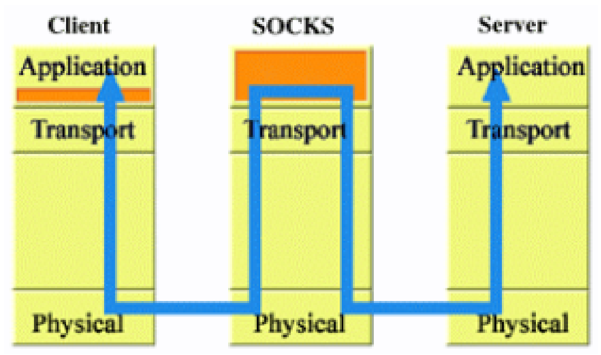


Figure 4.1: Protocol stack in the SOCKS context

The fundamental purpose of the protocol is to enable hosts on one side of a SOCKS server to access hosts on the other side of a SOCKS server, without requiring direct **"IP reachability"**. The previous version is SOCKS version 4 that provides unsecured traversal for TCP-based client-server applications. The version 5 extends the framework and provides strong authentication and extends the addressing scheme in such a way to comprise domain-name and IPv6. The implementation of the SOCKS protocol typically involves the recompilation or relinking of TCP-based client applications to use the appropriate encapsulation routines in the SOCKS library.

4.2 Procedure

The SOCKS protocol performs four functions:

- Making connection requests
- Setting up proxy circuits
- Relaying application data
- Performing user authentication (optional).

When an application client needs to connect to an application server, the client machine connects to a SOCKS proxy server. [20] The proxy server connects to the application server on behalf of the client, and relays data between the client and the application server. The SOCKS service is conventionally located on TCP port 1080. If the connection request succeeds, the client enters a **negotiation for the authentication** method to be used, **authenticates** with the chosen method, then sends a **relay request**. The SOCKS server evaluates the request, and either

establishes the appropriate connection or denies it.
Let's see all the steps more in detail. The client connects to the server, and sends a **version identifier/method selection** message:

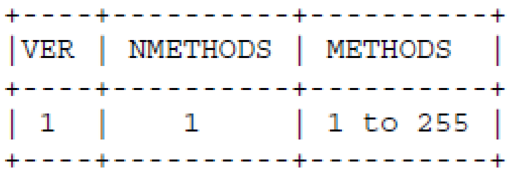

```
+----+----------+----------+
|VER | NMETHODS | METHODS  |
+----+----------+----------+
| 1  |    1     | 1 to 255 |
+----+----------+----------+
```

Figure 4.2: Client hello

with the one-byte VER devoted to the version number (SOCKS4 or SOCKS5), the one-byte NMETHOD devoted to the number of methods that will be listed subsequently for client-server authentication, and, finally, a listing of those methods by their ID numbers, with each ID number as a one-byte integer value.
For the **scope of the project** any connection is set with **no method for the authentication**. If the socks proxy server accepts the client packet, it responds back with a two-byte **"Server Negotiation"** packet:

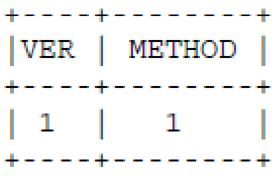
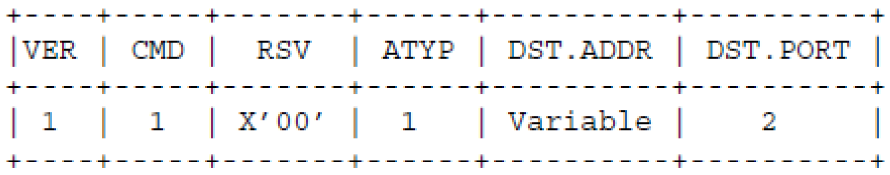

```
+----+--------+
|VER | METHOD |
+----+--------+
| 1  |   1    |
+----+--------+
```

Figure 4.3: Server negotiation

where the METHOD field is the authentication method that the server wishes to use. The socks server then proceeds to authenticate the LAN client using the specified method.

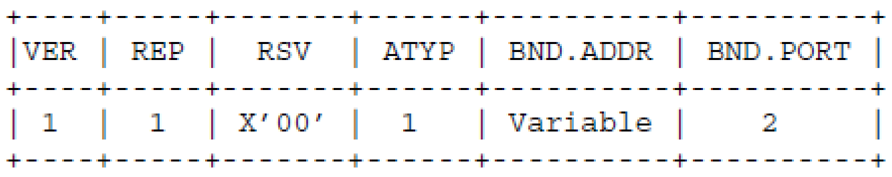
Once the method-dependent sub-negotiation has completed, the client sends the **request details** indicating which **service** it wants and which **address** on the Internet and on which port. If the negotiated method includes encapsulation for purposes of integrity checking and/or confidentiality, these requests must be encapsulated in the method dependent encapsulation. The message of the client, called the **"Client Request"** message, consists of the following fields:



```
+----+-----+-------+------+----------+----------+
|VER | CMD |  RSV  | ATYP | DST.ADDR | DST.PORT |
+----+-----+-------+------+----------+----------+
| 1  |  1  | X'00' |  1   | Variable |    2     |
+----+-----+-------+------+----------+----------+
```

Figure 4.4: Client request

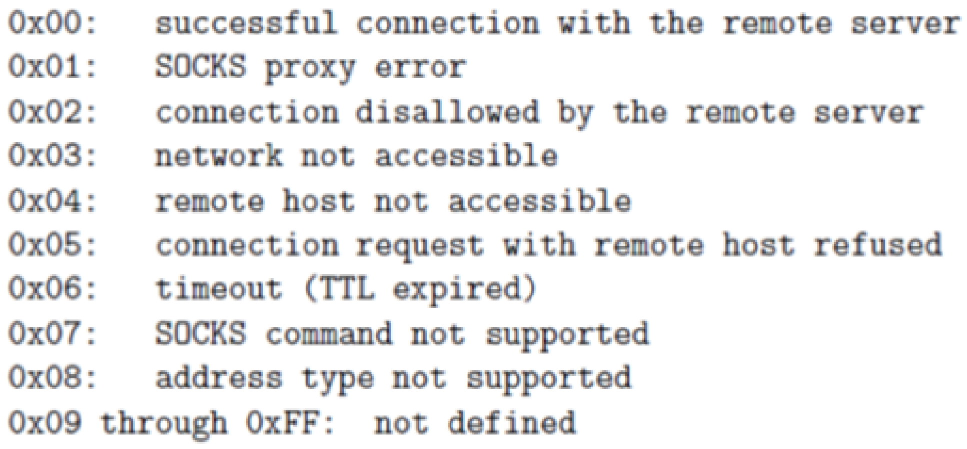
where the 1-byte CMD field contains one of three possible values: 0x01 for "CONNECT", 0x02 for "BIND", 0x03 for "UDP Associate". The ATYP field stands for the "Address Type" field. It takes one of three possible values: 0x01 for IPv4 address, 0x02 for domain name, and 0x03 for IPv6 address. As you'd expect, the length of the target address that is stored in the DST.ADDR field depends on what address type is stored in the ATYP field.
An IPv4 address is 4 bytes long; on the other hand, an IPv6 address 8 bytes long. Finally, the DST.PORT fields stores the the port number at the destination address. The RSV field means "Reserved for future use". The client **always sends a CONNECT** (value of the 1-byte CMD field) request to the socks proxy server after the client-server authentication is complete. However, for services such as **FTP**, a CONNECT request is **followed by a BIND request**. The BIND request means that the client expects the remote Internet server to establish a connection with the client. Under normal circumstances for a direct FTP service, a client first makes a so-called control connection with the remote FTP server and then expects the FTP server to establish a separate data connection with the client for the actual transfer of the file requested by the client. When the client establishes the control connection with the FTP server, it informs the server as to the address and port on which the client expects to receive the data file.
After receiving the "Client Request" packet, the proxy server evaluates the request taking into account the address of the client on the LAN side, the target of the remote host on the internet side and other access control rules typical of firewalls. If the client is **not allowed** the type of access requested, the proxy server **will terminate the connection** to the client. Otherwise, the proxy server sends one or two replies to the client socks. These replies, different in the value of the **REP field** (and possibly other fields depending on the success or failure of the connection with the remote server) are called the **"Server Reply"** are according to the following format:



```
+----+-----+-------+------+----------+----------+
|VER | REP |  RSV  | ATYP | BND.ADDR | BND.PORT |
+----+-----+-------+------+----------+----------+
| 1  |  1  | X'00' |  1   | Variable |    2     |
+----+-----+-------+------+----------+----------+
```

Figure 4.5: Server reply

where the BND.ADDR is the internet-side IP address of the socks proxy server; it is this address that the remote server will communicate with. Similarly, BND.PORT is the port on the proxy server machine that the remote server sends the information to. The REP field can take one of the following ten different values:



```
0x00:   successful connection with the remote server
0x01:   SOCKS proxy error
0x02:   connection disallowed by the remote server
0x03:   network not accessible
0x04:   remote host not accessible
0x05:   connection request with remote host refused
0x06:   timeout (TTL expired)
0x07:   SOCKS command not supported
0x08:   address type not supported
0x09 through 0xFF:  not defined
```

Figure 4.6: Response code of the SOCKS server

If the connection between the proxy server and the remote server is successful, the proxy server forwards all the data received from the remote server to the socks client and vice versa for the duration of the session.

4.3 Normal Scenario

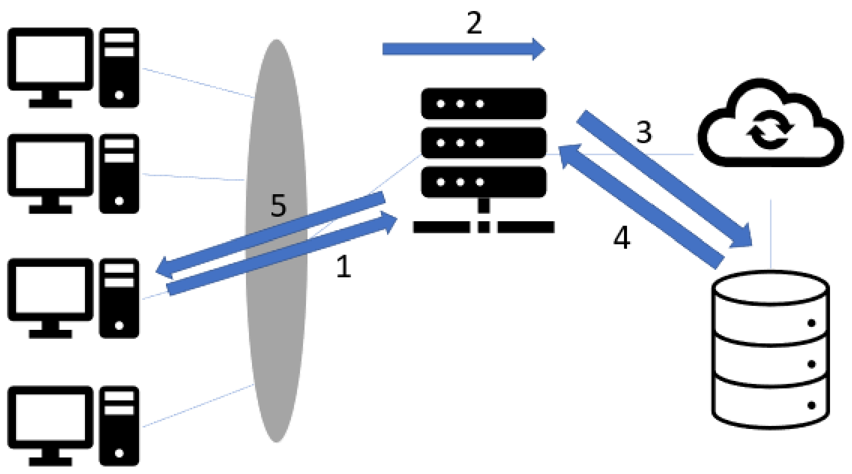


Figure 4.7: Standard scenario of a SOCKS connection

- 1) Client **contacts proxy** instead of the originally addressed server
- 2) Proxy evaluates client request and decides which to pass or which to reject
- 3) Proxy transfers request to originally addressed server on **behalf of client**
- 4) Server **answers to proxy** without knowing about client behind
- 5) Proxy **passes answer** to requesting client

Clients cannot decide whether to communicate to a server directly or via a proxy. They are forced to use the proxy by means of configuration in the corresponding application. On the other hand the application server **only knows about the proxy** and is supposed to only communicate with it. Proxies are also used as a kind of **firewall** because they protect the clients against contacts from outside. Clients to the left of the proxy are unknown to hosts on the right side (see drawing). This fact can be used to **reduce the requirement of official IP addresses**. When using a proxy server one single official IP address is theoretically sufficient to access the Internet for a whole company. The **proxy keeps a table** of all **sessions** and connections. It maps the IP addresses and port numbers from inside to a single IP address and the corresponding port number. This function is called **NAT (Network Address Translation)**. A proxy server that automates this process is called transparent proxy. Because proxy servers handle all network communication, they can log a lot. For HTTP proxies this includes requested URL's. For FTP proxies this includes every file that is transferred. They can even filter out "inappropriate" words and block sites from which the URL was retrieved or scan for viruses.

There are two **major types of proxy servers**. The first one is called dedicated or **application level proxy** and the other one **generic** or **circuit level proxy**. Dedicated proxies are specialized on one or a small amount of protocols. They know in detail about the mechanism of the proxied service and can check the contents of the packets. The name application level indicates that the proxy **can work on the application layer** of the TCP/IP stack. Generic proxies are more general proxies. In contrary to their specialized colleagues they can be called **all-rounder**. They cannot look beyond the port number. This means they **blindly trust the destination port number** of a packet. When there is something addressed to the HTTP port 80, the generic proxies treats it like a HTTP request without further checking. This may also be a disadvantage. Generic proxies can easily be faked. You can send SMTP traffic via port 80 and the proxy will pass it although smtp is not allowed. There are some more complex protocols like FTP that don't function in combination with generic proxies because they need special treatment.

4.3.1 Socks Proxy Server

The socks proxy server is a **generic proxy** and **supports nearly every application**. It only requires the application to run **socksified**.

The socks server checks the session request made by the socksified LAN client for its authenticity and then relays the request to the server on the Internet. Any replies received back from the server is forwarded to the LAN client. If a request does not violate any security policies set in the proxy server, the proxy server forwards the request to the Internet. Otherwise, the request is blocked. This property of a proxy server to receive its incoming LAN-side requests for different types of services on a single port and then forward the forward Internet requests to specific ports on specific Internet hosts is referred to as **port forwarding**. Port forwarding is also referred to as **tunnelling**. The proxy server replaces the source IP address in the connection requests coming from the LAN side with its its own IP address.(figure 4.8).

Now it is possible to **pass a name** to the socks server that **resolves it on behalf** of the client. The socks server then asks his name server. Prior to Version 5 UDP packets were not supported, this means they were dropped. Now applications based on UDP like i.e. DNS and NFS can communicate through socks proxy server. As the following drawing illustrates a socks server listens by default on port 1080. Applications that want to use the socks server have to be socksified. The socks client wraps the original request into a socks request. This means all outbound traffic is redirected to the socks server at the listening port. The source address will be replaced by the socks servers own and the source port will be randomly chosen by an available high port of the socks server.

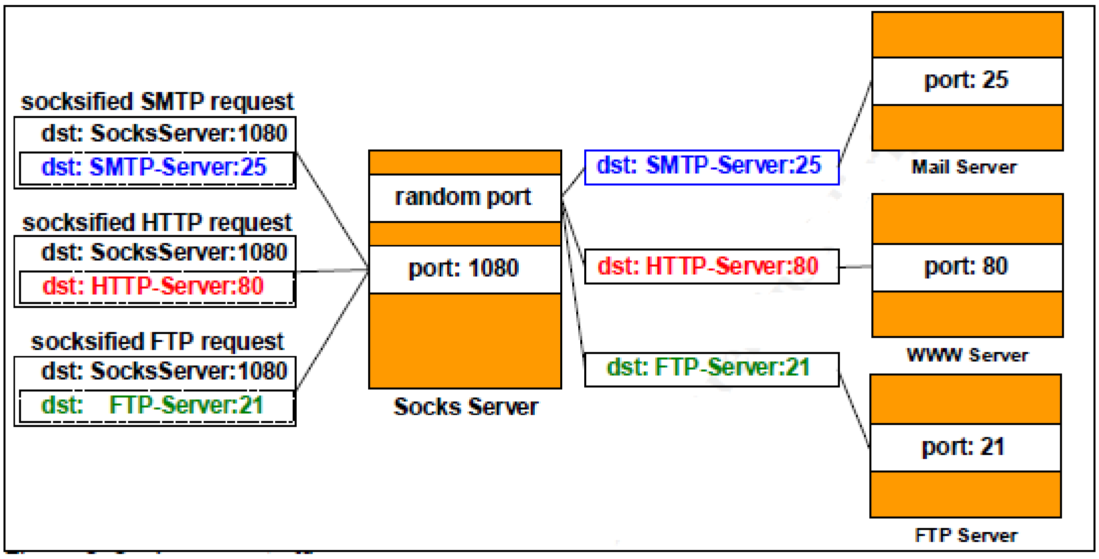
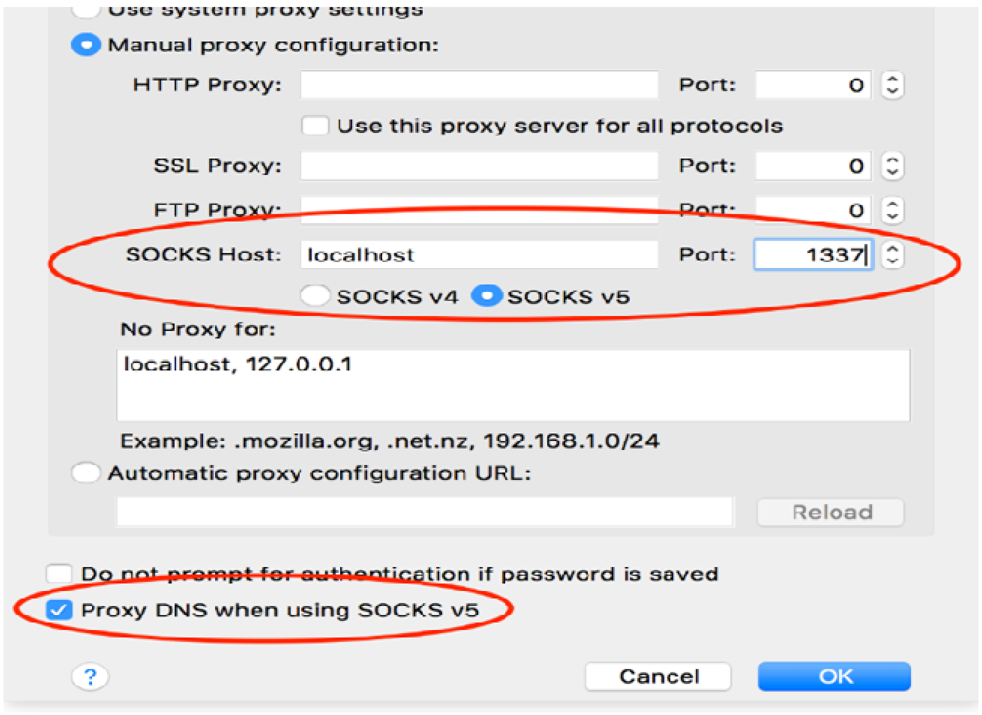


Figure 4.8: Socksification process

On the security of data communication between the socks server and the remote service provider, note that since then socks works independently of application-level protocols, it can easily host applications that use encryption to protect their traffic. To give a good example, regarding the socks server is concerned, there is **no difference between an HTTP session and an HTTPS session**. Since then, after establishing a connection, a Socks proxy server **does not care about the nature of the data** that moves back and forth between a client and the remote host on the Internet, such a proxy server is also referred to as a circuit-level proxy.

4.3.2 Socks Client

The socks customer wraps all network-related system calls made by a host with their own calls to the socket so that the host network calls are sent to the socks server on a designated port (as said before, the port 1080). This step is usually referred to as socksifying the customer call. An application doesn't connect automatically to a socks server. The operating system or the application itself has to be modified to use socks. This can be achieved in **two general ways**. The application has to be compiled against the **socks libraries** or a socks library has to be made **available at runtime**. To run an application socksified means to **redirect standard network calls** to the corresponding programs provided by the socks library. There are a lot of possibilities in doing so. Socks-capable applications such as web browsers, FTP, Citrix or IRC clients have an **optional socks functionality**. When selecting this option the socks server address and the port number socks is listening to, has to be provided. The socks libraries are included in these applications. Following a sketch of how to socksify firefox.



Use system proxy settings
[✓ Manual proxy configuration:]

HTTP Proxy:		Port:	0
	Use this proxy server for all protocols		
SSL Proxy:		Port:	0
FTP Proxy:		Port:	0
SOCKS Host:	localhost	Port:	1337

SOCKS v4 [✓ SOCKS v5]

No Proxy for:
localhost, 127.0.0.1

Example: .mozilla.org, .net.nz, 192.168.1.0/24

Automatic proxy configuration URL:

Do not prompt for authentication if password is saved
[✓ Proxy DNS when using SOCKS v5]

Figure 4.9: Socks configuration in firefox

Single windows applications can be socksified by means of programs like SocksCap from NEC. It intercepts the networking calls from winsock applications and redirects them through the socks server. In the SocksCap setup the socks server has to be defined. To run a single program socksified it has to be inserted in the SocksCap program list and started from there. A whole windows client can be socksified by manipulating the TCP stack. This is usually done by a socksify program or more common called a socks client. It replaces parts of the TCP stack or introduces an additional layer to the stack. When an application requests a network connection the destination IP address is checked against the socks client's configuration. If an entry is found the request will be redirected to the corresponding socks gateway. In the Linux/Unix environment socks libraries are available for free. To make an application socks capable it has to be **run with the according socks libs**. You can either recompile the application against the shared libraries or load these libraries at runtime by starting the application with programs like socksify or runsocks.

To accomplish our project was selected a already existing generic proxy server, and after a careful review of some projects, 3Proxy was chosen.

Chapter 5

Understanding 3Proxy

5.1 Introduction

3Proxy is a combination of many specific application proxies, then in general it can be considered a **general server proxy.** You can use every proxy as a **standalone program** (socks, proxy, tcppm, udppm, pop3p) or use **combined program** (3proxy). Combined proxy additionally supports features like access control, bandwidth limiting, limiting daily/weekly/monthly traffic amount, proxy chaining, log rotation, syslog and ODBC logging, etc.

5.2 Architecture

The architecture reflects what was described before. In fact, 3Proxy is a **shim-layer proxy** that accepts connections on different port. In our case of study, 3Proxy will be mostly **used as a SOCKS server** and will wait on port 1080 for incoming connections and will use only this port to communicate with the client applications. 3proxy is implemented as multithread application. Server model is implemented as **"one connection – one thread"**. It means, for every client connection new thread is created. Below an architecture's sketch:

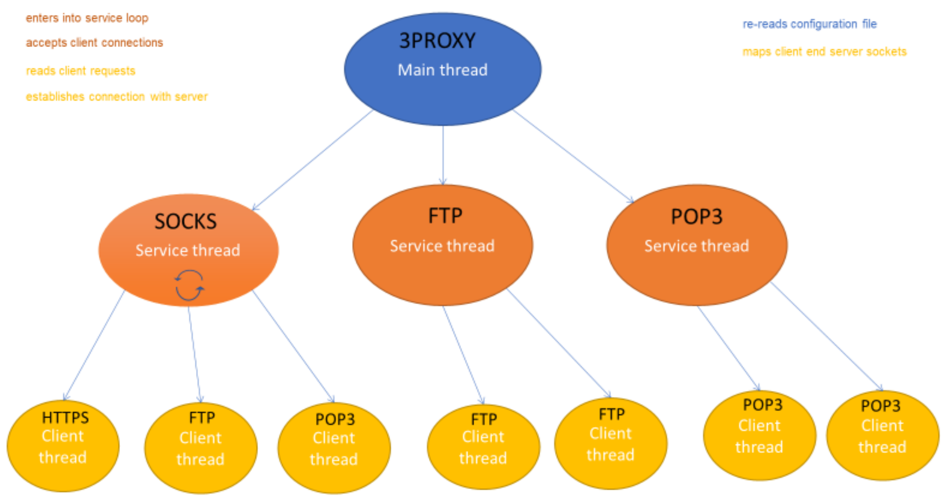


Figure 5.1: 3Proxy architecture

The program starts with the **main thread**, that has the aim to read and parse the configuration file. Each command of the configuration file, starts a new thread, called **service thread**. The service thread loops waiting for requests from the client, and spawn a new thread, the **client thread**. The last one, has the to perform the **SOCKS handshake** and to **relay the connection**. One focus on the fact that SOCKS can **handle whatever application**, while the other services are specialized and oriented only to one application.

5.2.1 Main thread

3proxy begins with main thread. This thread parses configuration file and starts main loop. During configuration file parsing struct extaparam conf; structure is filled and service threads are started. [1]

Main loop cycle takes approximately 1 second and does these tasks:

- re-reads configuration file, if necessary
- performs scheduled tasks
- monitors files ('monitor' command), approximately once in a minute
- rotates main log file
- dumps counters to file, approximately once in a minute
- performs termination, if required

It's guaranteed every configuration and schedule command is executed from the same thread. Main thread is implemented in *3proxy.c*.

5.2.2 Service thread

Service threads are started immediately, than **service command** (e.g. 'proxy' or 'socks') are found during configuration file parsing. Each command creates new thread. Thread does these tasks:

- parses service command arguments and fills the structures with default client configuration
- initializes filters
- creates and initializes listening service socket
- enters into **service loop**
- terminates filters

service loop:

- checks for configuration reload (approximately every second), thread exits if configuration reloaded or 3proxy is in terminating state.
- **accepts client connection** and creates the new client structure for the connection with client configuration
- creates/checks client filters
- creates client thread with newly created data structure

service threads are implemented in *proxymain.c*

5.2.3 Client thread

Client threads are **started from service thread**. Client thread:

- **reads client request** with authentication information and request headers (if any).
- filters request (if any)
- filters headers (if any)

- performs **authentication and authorization**
- **establishes connection with server**
- **sends request to server**
- filters server headers (if any)
- **maps client end server sockets** to transmit data between client and server
- logs request. Global counters are also updated on this operation
- clears client filters
- frees all the client data structures

In some point client thread may loop to process few client requests from the same connection (e.g. HTTP 'established' connection in 'proxy').

Socket mapping does:

- caches data in internal client and server buffers
- delays data transmit to limit bandwidth
- performs data filtering

client threads are implemented in *proxy.c, socks.c, pop3p.c* etc.

5.3 Hack into 3Proxy with plugins

3Proxy plugin is any **dynamic/shared library**. There is no specific requirement for plugin, actually you can load any dynamic library with 'plugin' command. **No linking** with any libraries are required. However, to interoperate with 3proxy dynamic library must have an **export function 3proxy may call** to pass the structure with required information.

*typedef int (*PLUGINFUNC) (struct pluginlink *pluginlink, int argc, char** argv)*

struct pluginlink is a structure with export information, argc and argv are argument counter and array of arguments of "plugin" command. Plugin should report it's status with integer return value. 0 is success, positive value indicates non-recoverable error, 3proxy do not parse rest of configuration and enters into termination state, negative value indicates recoverable value, 3proxy logs warning (if possible). In case of C++, all 3proxy functions/structures must be extern “C”. All 3proxy structures/functions descriptions are located in *structures.h* Because there is no linking between 3proxy and plugin, all 3proxy functions and structures are passed with pluginlink structure. Pluginlink is actually a **collection of pointers to 3proxy internal structures and functions**. One insight was to put all the pointers to functions in that structures to allow the plugin to interact with the original workflow of the project. With a simple call of this functions inside 3Proxy was possible **to switch in the bundle context** and execute commands and actions that interact with Aether in such a way that **3Proxy was not involved and was not aware of this context switch**. Besides not only pointers were stored inside that collection but also data structures and variables useful for the bundle plugin.

Chapter 6

3Proxy2Bundle

This chapter considers the design and implementation of the work. In a first step will be discussed the architecture of the work and the workflow of a single connection. After will be showed the crucial point of the implementation.

6.1 Architecture

There are three actors involved: the client node making the request, the intermediate nodes and the gateway node that has the router role. As we have seen, there is the DTN layer that allow the connection between the two sides.

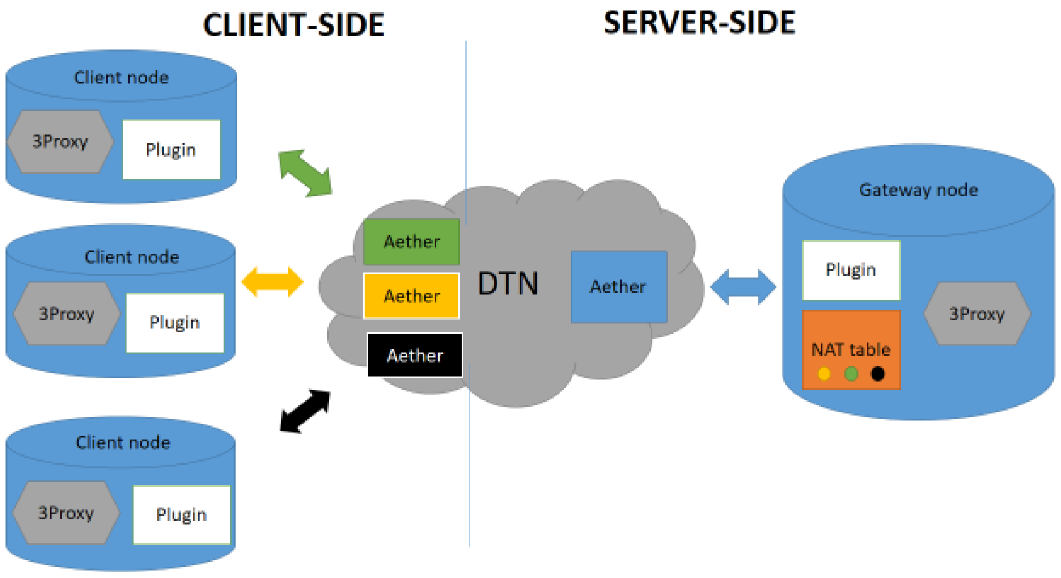


Figure 6.1: 3Proxy2Bundle architecture

It is worth to know that the Aether boxes are the application that runs over each device and that there are more than one connection instantiated between the nodes and Aether. The task of Aether is to **route packets** to the correct destination, send and receive data in bundle format and provide the application data to the proxy. From now on we call 3Proxy2Bundle the original 3Proxy project plus all the modifications and plugins, that are essential to the proxy to **"talk" the bundle protocol**. Basically, the original architecture of 3Proxy is modified in such a way that the socks server is **splitted in two sides**, client and server's. The client side provides the function to accept connections of the clients and to send the requests to the gateway. Therefore, it **relays the connection** between the client and the gateway. As we can see in the figure below, the client application establishes a connection with 3Proxy2Bundle listening on port 1080.

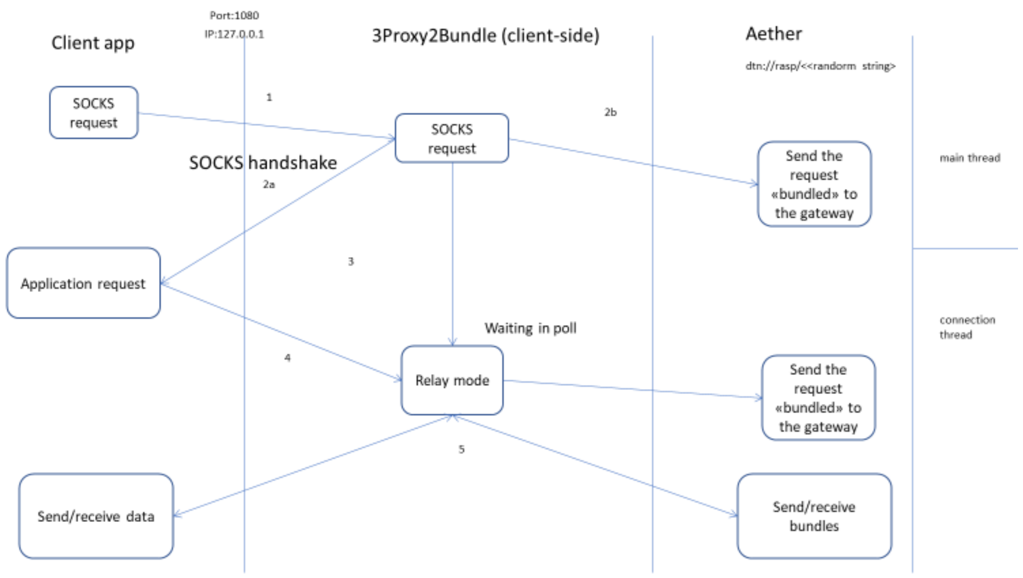


Figure 6.2: Client-side workflow

The client sends the **socksified request** to the proxy (1). The SOCKS handshake is performed (2a), the proxy sends everything it receives from the client in bundle format to the gateway (2b). The proxy then waits to receive the actual request from the client (3). The client in turn waits for the outcome of the handshake. If the latter is successful, the client sends the application request to the proxy (4). The proxy, which has entered the **relay phase**, is therefore waiting to both **receive and send bundles** (6). Once it receives the request, it forwards it via DTN and from here on the connection occurs automatically, in relay mode.

The server side provides the **connectivity to the internet** and **send back the answers** to the client side. The server side workflow is shown below.

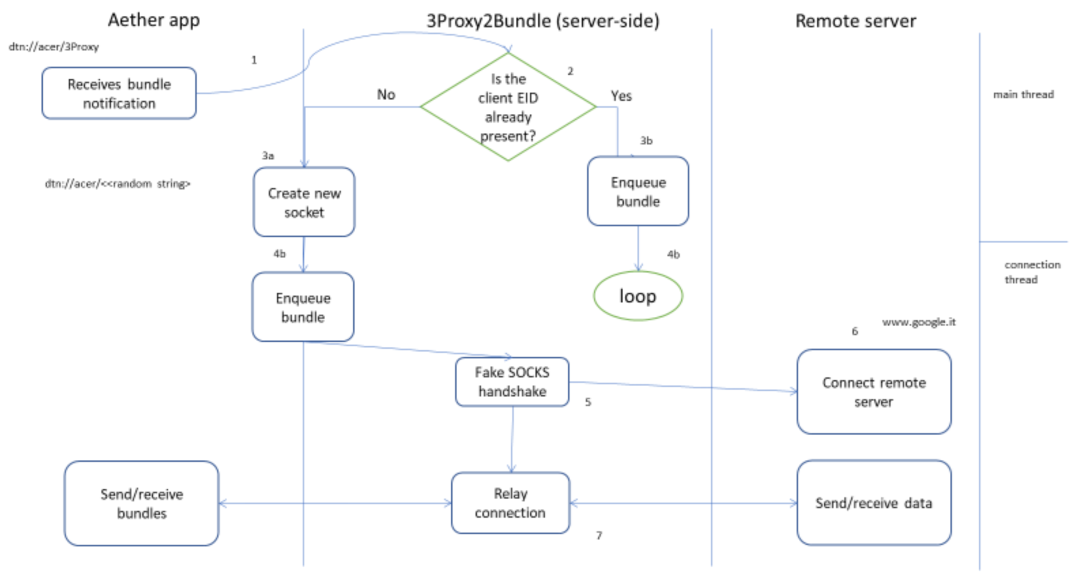


Figure 6.3: Server-side workflow

On the server side once that the application is entered, it creates a thread that listens on the DTN node (i.e. dtn://acer/3Proxy). As soon as the node receives a bundle notification (1), the proxy parses the message and retrieves the **client Endpoint ID** (EID). The proxy has a **map** that contains all the combinations between **client EID and the socket** that has been created to communicate with the remote server. For example an entry could be "dtn://rasp/«random string» -> 4", where the "4" is the open socket connected to the remote server. Then, after obtained the EID, the proxy checks if is already present in the table (2). In case that is the first message that the client node sends to the gateway, is obvious that there isn't an entry in the table. So the app will create a **new socket connected to Aether**, that has the aim to receive new messages from the client node and to send messages from the remote server to the client node (3a). After the socket creation, the proxy will **enqueue the bundle** in the queue of bundles that have to be nextly consumed. A new thread is created and as the client side the SOCKS handshake is performed. This time, anyway, it is a **fake handshake** performed to do not modify the original workflow (5). Once that the proxy has got the address or DNS name, it can establish a connection with the remote server. After that, the thread enters in the relay mode, in which it listens both sockets for new messages and sends/receives data from the remote socket and the DTN node. In the case that the entry was already present in the table, it means that a connection has already been established, then the proxy will only enqueue the bundle in the queue (3b) and will loop to receive another notification (4b).

6.2 FTP example

An example of how to use the application can be using the FTP protocol thanks the FileZilla client. FileZilla can connect to the Socks proxy on the port 1080. 3Proxy2Bundle is running a thread waiting on that port for connections. At this point a new thread is spawned and the connection is taken in account. A initial handshake is instantiated between the app and the client. After that a socket to the Aether is generated and all the requests are redirected to the 3Proxy2Bundle router node waiting in the DTN network. At this point the 3Proxy2Bundle server side has already created a thread waiting on the DTN socket for some requests. As soon as a notification arrived, the request is taken in account and a dummy handshake is performed between the DTN node and app. At that point the server connects to the remote server and send the original request. After that the server receives the answers by the remote one and generating a temporary socket for the connection sends through DTN the packets. After that point on, all the connection is relayed on this 2 sockets on the edge of the DTN and Filezilla can demand for any file in the directory listing.

6.3 Configuration

6.3.1 Configuration file

3Proxy has originally a own configuration file to set up some options and behaviours. In this work has been chosen to use that file with some insertions and modifications.

```
#3Proxy original options
nscache 65536
nserver 8.8.8.8
nserver 8.8.4.4

#Timeouts for connections
timeouts 200000 200000 200000 200000 200000 200000 200000
   200000

#Server-side plugin
```

```
#plugin ./Server.ld.so server

#Client-side plugin
plugin ./Client.ld.so client

#3Proxy original options
config 3proxy.cfg
rotate 60

#Authentication is not required
auth none

#Allow any client to access the service
allow *

#Start the socks program
socks -p1080

#clean the option file
flush
```

Listing 6.1: Configuration file

The options above specify the functionalities of the program and the behaviour. The first three lines derive from 3Proxy and are necessary to specify to which DNS server the application will connect and define the dimension of the cache. The next option defines the value of the timeouts. In the program there are different timeout, so you define them separating each one by a space. As we have seen, the main thread loops each second to fetch some command. In the configuration above, there are 2 plugin commands, actually only one should be defined because one excludes the other. The ***plugin ./Server.ld.so server*** means that a plugin command is executed. It implies that a dynamic library is loaded, in this case *Server.ld.so*, and the *server* function is called as an **entry function**. After the server function is called the main thread exits by the plugin function and the thread returns looping. Since that the application is a proxy, it has to be defined who has the **rights to access the service**. In fact, the next two options define how to do that. The former, establishes if authentication is required or not; for simplicity is set to not authenticate. The latter, states if every client can connect and use the service or there is some restriction. The last command shown above **starts the socks program with a new thread**. It also defines the port on which the thread will listen for incoming connections.

6.3.2 Plugins

The aim of the plugin, in our case, is to **initialize correctly the pointers to functions** that will called inside the 3Proxy original project. Following is reported the *Client.cpp* file. *Server.cpp* is pretty similar to *Client.cpp*, for this reason is not reported. The basic difference is that instead of the client flag is raised, in the case of the server side, the server flag is set to 1.

```

#ifdef  __cplusplus
extern "C" {
#endif

struct pluginlink * pl;


PLUGINAPI int PLUGINCALL client (struct pluginlink *
    pluginlink, int argc, char** argv){

        pl = pluginlink;
        std::cout << "Hi,␣I␣am␣3ProxyClient" << std::endl;

        if (argc>1) {
                if (!strcmp((char *)argv[1], "debug")) {
                        fprintf(stdout, "Traffic␣correct␣
                            plugin:␣debug␣mode␣enabled.\n");
                }
        }

        pl->connect_to_aether = utils::connect_DTN;
        pl->client = 1;
        pl->DTN = 1;
        pl->send_DTN = utils::send_DTN;
        pl->read_DTN = utils::read_DTN;
        pl->close_fd = utils::closefd;
        pl->queue_is_empty = utils::queue_empty;
        pl->check_and_react = utils::checkandreact;
        return 0;
 }

#ifdef  __cplusplus
}
#endif
```

Listing 6.2: client()

According to this approach, you can call some functions that are external from the original project, for instance, simply by calling the function as follows:

pluginlink->readDTN(parameters).

6.4 Common part

To understand more what are the functions involved in the workflow of a standard connection, let's see the sketch below:

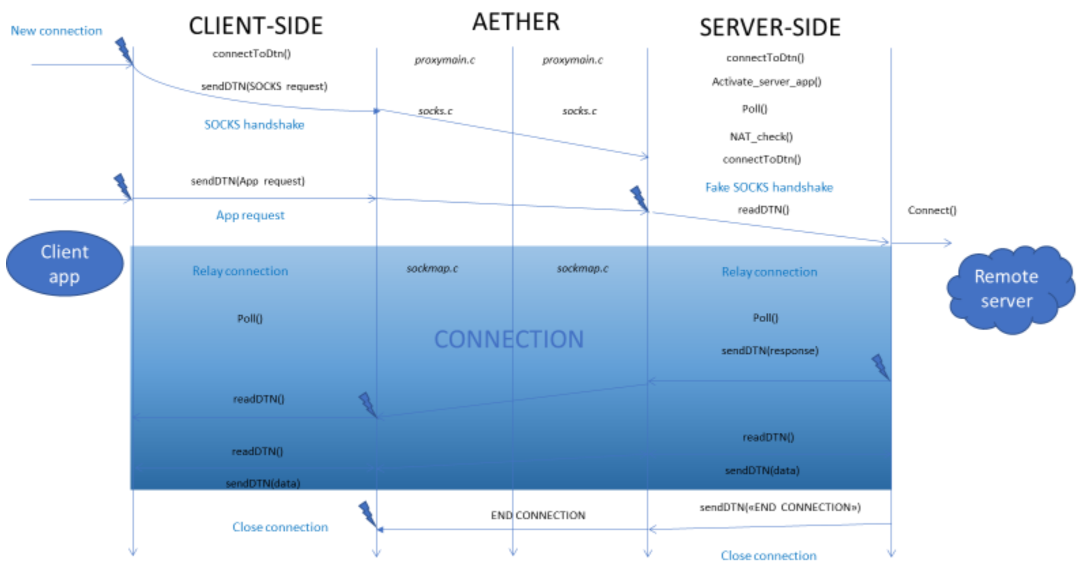


Figure 6.4: Workflow of establishing a new connection

As shown in the figure above, there are some functions that are common in both sides. For the sake of simplifying the presentation they are grouped in this paragraph. The functions that are responsible to intermediate between the proxy and Aether are grouped in the *Utils.cpp* file. *Utils.cpp* is a wrapper that embeds the function calls to function in *BundleServer.c* and *BundleClient.c*. The most important function are the send and receive ones, that are responsible to deliver and receive every packet or data to/from the other edge. The bundle management part has been taken from *Tinyproxy* [4] and some changes have been added. *Bundle.c* and *network.c* remained unchanged. Basically they were used as an interface to the proxy with the bundle protocol. In the relaying phase, where the connection has been already established, both the sides applications enter in the *sockmap.c* file where the connection is relayed. With respect to the original file, some modification have been added such as an extra timeout in the case a bundle is already available and the functions read/send have been substitute by the *readDTN* and *sendDTN* functions.

6.4.1 Read and write functions to DTN

The readDTN implementation is the following:

```c
int readDTN(int s, unsigned char * buf, size_t len){
        Dtnd_bundle_id *dtnd;

        if(bundle_is_bundle_id_queue_empty(s)){
                dtnd = bundle_react_to_notify(s);
                if (dtnd != NULL)
                        bundle_enqueue_bundle_id(s, dtnd);
                else
                        return -1;
        }
        if(bundle_read_header_incoming_bundle(s)){
                fprintf(stdout,"Errore␣read␣incoming␣
                   bundle!\n");
        }
        int ret = read_buffer(s, buf);

        bundle_set_delivered(s);

        return ret;
}
```

Listing 6.3: readDTN()

The basic task of this function is to check if the bundle's queue is empty. If yes, it will react and try to retrieve the bundle asking it to Aether. If not, it

means that there is yet a bundle to consume. So it goes directly to the bundle*_read_header_incoming_bundle(s)*. At that point, the function have put the bundle in the queue and it is possible to read it. The result is stored in the buffer *buf* that was passed by reference. The last action is to set the bundle as delivered. In such a way, it will be erased from the queue and from Aether.
The sendDTN implementation is the following:

```c
int sendDTN(unsigned char* buf, int sock, size_t bufsize){
        int res;
        int connect_method =
           connections[sock].connect_method;
        char* client_local_eid =
           connections[sock].client_local_eid;

        if(bundle_get_remote_eid(sock) == NULL){
                if(bundle_start_send(sock, client_local_eid,
                   bundle_get_remote_eid(sock)) != 0)
                        return -1;
        }
        else{
                if(bundle_start_send(sock, client_local_eid,
                   bundle_get_remote_eid(sock)) != 0)
                        return -1;
        }
        res = safe_write(sock, buf, bufsize);
        bundle_finalize_send(sock, client_local_eid);

        return res;
}
```

Listing 6.4: sendDTN()

Basically, this function retrieves the receiver and the sender thanks the data structure previously filled. Once it has both them, it calls *bundle_start_send* that initializes the structures for sending the bundle. Then, the socket will expect some data and the *safe_write* function will write on it. At the end, to close correctly the bundle and turn back to the correct state is called the *bundle_finalize_send* function. From now on, it will be presented the different implementations that characterise the client and server side. The main differences between the 2 sides are shown into the *BundleClient.c* and *BundleServer.c*.

6.5 Client implementation

The interaction with the client application is left the same of the original project, in fact the listening thread on port 1080 is running and waiting for connections from the client. The only modification in the *proxymain.c* is the connection to the remote server. For our scope, *param->remsock* will point to a socket connected to Aether. Below is shown how to get it:

```c
int connect_to_aether(){
        char* remote_eid;
        char* server_local_eid;
        int fd = bundle_establish_dtnd_connection();
        server_local_eid = bundle_get_local_endpoint_eid(fd);
        char host[16] = "3proxy.acer.dtn";
        remote_eid = bundle_format_server_eid(host);
        if (remote_eid == NULL) {
                fprintf(stdout, "Server␣EID␣is␣not␣valid,␣
                   critical␣error");
                bundle_close_fd(fd);
                close(fd);
                return -1;
        }
        bundle_set_remote_eid(fd, remote_eid);
        connections[fd].host = host;
        connections[fd].remote_eid = remote_eid;
        connections[fd].server_local_eid = server_local_eid;

        return fd;
}
```

Listing 6.5: connect_to_aether()

The value returned is the socket connected to Aether so it can be assigned to *param->remsock*. After that the connection thread enters in the *socks.c* file, does the handshake, sends the SOCKS request and App request to the server through DTN and enters in the **relay mode**. Once the server will have answered, the connection will proceed automatically.

6.6 Server implementation

In the server part, the *proxymain.c* is a bit different from the original one. A first difference is that the creation of the client socket (param->clisock) is done by the following code:

```
/* create the socket */
    sock = pluginlink.activate_server_app();
        sprintf((char *)buf, "Connected␣to␣DTN␣[%d]\n",
           sock);
        (*srv.logfunc)(&defparam, buf);
```

It is worth nothing that, the socket, while previously was connected to the client application, in that case is **connected to Aether**. The *active_server_app()* does the following stuff:

```
int activate_server_app(char* name_service, char* address,
   int port){
        dtnd_server_ipaddress = (char*)
           malloc(10*sizeof(char));
        strcpy(dtnd_server_ipaddress, address);
        dtnd_server_port = port;
        Dtn_service* ds;
        ds = (Dtn_service*) malloc(sizeof(Dtn_service));
        ds->ip_address = dtnd_server_ipaddress;
        ds->port = dtnd_server_port;
        ds->app_name = name_service;
        int ret = bundle_establish_dtnd_connection();
        ds->app_eid = bundle_build_nodename(ret,
           ds->app_name);
        bundle_set_service(ret, ds);
        bundle_set_endpoint(ret, ds->app_name);
        return ret;
}
```

Listing 6.6: activate_server_app()

Basically, in the above code the server app will connect to Aether on address and port provided in the configuration file. Then is defined the service, that is a name written in the configuration file. So every node in the DTN that wants to **send bundles to the server** has to set as receiver *dtn://name-device/name-service.* Then the connection with Aether is established and is set the name of the service and the name of the endpoint.

After the structures of the bundle plugin are set, has to be shared with the node the active service. It is done and shown in the next code snippet:

```
int register_service(int fd,char*name, char* type, float
   min, float max, int osi_level, char* ext_conn, char*
   unit_min, char* unit_max){
        Dtn_service dtn_service;
        dtn_service.app_name = name;
        dtn_service.dtnd_description = "Test";
        dtn_service.type = strdup(type);
        dtn_service.min_bandwidth = min;
        dtn_service.max_bandwidth = max;
        dtn_service.unit_max = unit_max;
        dtn_service.unit_min = unit_min;
        dtn_service.max_bandwidth = max;
        dtn_service.osi_level = osi_level;
        dtn_service.external_connection = ext_conn;

        bundle_register_to_SD(fd, &dtn_service);
}
```

Listing 6.7: register_service()

The next step is that the **service thread enters in the loop** and **wait for some notifications** from the client app. When a new notification arrives to the listening socket, the poll wakes up and the following code is executed:

```
 new_sock = pluginlink.accept_connection(sock);
         if (new_sock == -1)
                continue;
        so._setsockopt(new_sock, SOL_SOCKET, SO_LINGER,
           (char *)&lg, sizeof(lg));
        so._setsockopt(new_sock, SOL_SOCKET, SO_OOBINLINE,
           (char *)&opt, sizeof(int));
```

The *accept_connection* references to *acceptconnection* that is a wrapper for the *accept_connection* function. The wrapper has the function to know if a **connection has already taken place** or not. If yes, the function returns -1 and the bundle will be enqueued and the function continues the loop. Otherwise a new socket is created calling the *accept_connection* function inside the wrapper.

```
int acceptconnection(int listenfd){
    char buf[512];
    int len;
    int connfd;
    Dtnd_bundle_id* dtnd = react_to_notify(listenfd);
    if (dtnd == NULL) {
        printf("Errore␣50␣utils.cpp\n");
        return -1;
    }
    std::string client_eid = get_eid(listenfd);

    std::map<std::string, int>::iterator connection;
    connection = get_connection(client_eid);

    if (connection != connections.end()){
        //already present -> enqueue bundle
        enqueue_bundle(connection->second, dtnd);
        return -1;
    }else{
        connfd = accept_connection(listenfd, dtnd);
        connections.insert(std::make_pair(client_eid,
           connfd));
    }
    return connfd;
}
```

Listing 6.8: acceptconnection()

As said before, only if a connection hasn't already been created the *accept_connection* will be called. To do that, a **map** with all the connections is stored and **updated dynamically** in the course of the program. The *accept_connection* function is reported as follows:

```
int accept_connection(int listenfd, Dtnd_bundle_id*
   dtnd){
        int connfd;
        if (dtnd != NULL) {
                connfd = bundle_establish_dtnd_connection();
                bundle_copy_service(listenfd, connfd);
                bundle_enqueue_bundle_id(connfd, dtnd);
                bundle_set_remote_eid(connfd,
                   strdup(bundle_get_remote_eid(listenfd)));
                bundle_clear_remote_eid(listenfd);
                int flags = fcntl (connfd, F_GETFL, 0);
                fcntl (connfd, F_SETFL, flags & ~O_NONBLOCK);
        }
        if (dtnd == NULL) {
                usleep(10);
        }
        char* local_eid =
           bundle_get_local_endpoint_eid(connfd);
        connections[connfd].client_local_eid = local_eid;
        return connfd;
```

Listing 6.9: accept_connection()

Once a connection is accepted, all the further messages will be addressed to the **connection-specific node** (*dtn://acer/«random-number*) and not anymore to the **listening-service node** (*dtn://acer/3proxy*). It is possible since that every time a node receives a message, it sets up the **receiver for further messages** as the **sender of the last message**. Then, when the node on the client side will receive the message from the temporary socket on the server side, it will set up the temporary node as the receiver and not anymore the "3proxy" node. So the connection enters in the relay mode and the connection thread will enter in the *sockmap.c* file and will relay the connection.

6.7 Relay mode

After a connection is established and the first notifications arrived to the gateway, the whole system enters in the **relay mode**. This part of the workflow is the heart of the connections. For instance, if the application involved is HTTPS, there will be a lot of requests and a lot of connections. The average duration is very small with regards to a FTP connection, since that in the latter, more data will pass through the sockets in the relay mode, then on the same connection. The core of the relay mode is in *sockmap.c*. *sockmap()* function has a poll with the 2 sockets, *clisock* and *remsock* and whenever there are data for one side, after triggered a wake up of the poll, they are sended to the destination. Since that on client side and server side the socket are different from the native ones, respectively, *remsock* and *clisock*, some changes were necessary to make everything working.

6.7.1 Read/Send function calls

Due to the interoperability with the bundle protocol a simple call to send became:

```
if(pluginlink.client && pluginlink.DTN){
                res = pluginlink.send_DTN(parameters);
}else{
                res = so._sendto(parameters);
}
```

Listing 6.10: send_DTN in sockmap()

The same is done for the reading function. In such a way the *sendDTN* and the *readDTN* are called instead of calling *read* and *send* as done by the normal flow of the original project.

6.7.2 Handling silent bundles

Therefore, since that sometimes new bundles are silently added to the queue without being noticed by the poll it was necessary to add a check before the poll to be sure that no bundle is waiting in the queue to be consumed:

```
timeo = 300000;
        if(pluginlink.client && pluginlink.DTN &&
           !pluginlink.queue_is_empty(param->remsock)){
                timeo = 2;
        }
        res = so._poll(fds, 2, timeo);
        if(res < 0){
                if(errno != EAGAIN && errno != EINTR) return
                   91;
                if(errno == EINTR) usleep(SLEEPTIME);
                continue;
        }
        if(res < 1){
                if (pluginlink.client && pluginlink.DTN &&
                   !pluginlink.queue_is_empty(param->remsock))
                        fds[1].revents |= POLLIN;
                else{
                        return 92;
                }
        }
```

Listing 6.11: Modification to handle silent bundles

Originally the timeout is set up on the configuration file. In this case, since every time it is modified, *timeo* is **always redefined.** If there is no bundle that has to be consumed, the timeout is left equal to a long timeout, in the code above is set to 300000. In the case that there is **already a bundle**, the timeout is **set approximately to 0** to let trigger the timeout and go out immediately from the poll. This change is really important for the correct workflow of the connection, if that part would not be present, the poll will wait until the timeout expires and the bundle will never acknowledged. After the exit from the poll, the fds.revent specific is set in such a way that on the socket will be read data.

6.7.3 Connection closure

Having the sockets connected to the client and the remote server on a single device ensures that when one is closed, the other is notified about the closure of the other side and it will also close the connection. Since in this case, the sockets are on two different machines and are **intermediated by sockets connected to Aether**, the connection must be closed explicitly with text plain messages. One solution is to send a text message to the receiver so that it detects and knows that the connection on the other side is closed and acts accordingly, that is in turn closes the connection. The solution chosen is shown below:

```
if (res==0 || end_connection(res, param->srvbuf +
    param->srvinbuf)) {
                        if(pluginlink.client &&
                            pluginlink.DTN)

                        so._shutdown(param->remsock,
                            SHUT_RDWR);
                        so._closesocket(param->remsock);
                        fds[1].fd = param->remsock =
                            INVALID_SOCKET;
                        stop = 2;
                }
```

Listing 6.12: Handle the closure of the connection

In a nutshell, a check is performed after receiving data from Aether; if the bytes read are 0 or the message reports the closure of the other part, the socket immediately exits from the loop and close the connection. Specifically, if *end_connection* returns 1, means that the connection on the other side is closed and it sended

"END CONNECTION" in plain text. Below is reported the *end_connection* function:

```
int end_connection(int count, unsigned char* buf){
        char subbuff[15];

        memcpy( subbuff, buf, 14);
        subbuff[14] = '\0';

        if(strcmp("END CONNECTION", subbuff) == 0)
                return 1;

        return 0;
}
```

Listing 6.13: end_connection

Chapter 7

Prototype validation

In this chapter, we will carry out some tests and evaluate the results. To this end, a scenario has been configured in which two nodes are connected via a dedicated Wi-Fi network. The metrics measured are **throughput** and the **overhead introduced by the implemented software layer.**

7.1 Setup

The tests were carried out using 2 computers connected to a dedicated Wi-Fi network with a 100 MBps access point. The two devices used are:

- Acer, 8 GB of RAM, i5-2410M processor with 2.3 GHz frequency and four cores on Ubuntu 16.04 operating system.
- Dell 3550, 8 GB of RAM, i5 processor with frequency 2.7 GHz and four cores on XUbuntu operating system.

Two of the tools offered by Aether were used to generate and receive traffic, namely *dtnsend* and *dtnrecv*, which are launched from the terminal with the following commands:

```
dtnrecv --name abc
```

which puts the receiver node on listen; the parameter passed is the name to be given to the application;

```
dtnsend dtn://nodename/abc F.txt
```

with which the sender sends the F.txt file to the node with the corresponding dtn name via the application of name *abc*. The configuration of the proxy is set to

craft packets of 4096 bytes. This decision was taken in account because on one side it implies that an **high number of packets are created** each time, but the great advantage is that it guarantees that **it is not required to resend large packets in the case one of them is corrupted.**

7.2 Throughput and network load

In this section we will analyse the throughput and latency of transmission as the sending characteristics change. The times measured below are given in seconds. The protocol used to test the application were **FTP and HTTP**. The former, using FileZilla gives a more detailed look at the real performance of the proxy service. It is possible to obtain the **average speed of the downloads** and uploads and so on. In the latter case it is possible to see how many connections are really established and closed correctly. Following a particular study is carried on the former which a focus was taken on the throughput and performance.

7.2.1 Troughput

As can be seen from the following list, throughput was measured in different scenarios:

- receiving a single 1 MB file,
- receiving a single 10 MB file,
- receiving a single 100 MB file,
- receiving a single 1 GB file

When receiving a single 1MB file, the average transfer time was 1.1 s. This implies an average throughput of this implies an average throughput given by:

$$Throughput = \frac{2^{20}B}{1s} \simeq 1MB/s \tag{7.1}$$

The same experiment repeated with a 10MB file shows an average transfer time of 12.3 s, with a throughput given by:

$$Throughput = \frac{10 \times 2^{20}B}{28s} \simeq 0.35MB/s \tag{7.2}$$

As can be seen, the performance in this case is lower than the previous case by about 65%; this is due to the excessive network load and, therefore, a waiting time for the data in transit in the network buffers.

As can be seen, the performance in this case decrease not linearly with the dimension of the file. However, it should be noted that the larger size of the data sent compared to the single 1MB file, the throughput in this case is worse. So the overhead introduced for the passage from one bundle to the next allows for faster sending, in the sense that in the 10MB case the receiver has to wait for all the fragments to be received, reorder them and recompose the file, while by sending smaller files the data to be ordered and rearranged is smaller, resulting in a slight improvement in performance.

Nextly the prototype was challenged with files of 100 MB and 1 GB. In the former case, the average transfer time was is about 5 minutes; therefore the throughput in this case is:

$$Throughput = \frac{100 \times 2^{20}B}{286s} \simeq 349{,}65KB/s \tag{7.3}$$

The latter was not successful, in fact after have downloaded 1 quarter of the total file, the throughput fell down and the connection was closed by the server side. This is due to the high number of packets that are generated when files of such dimensions are demanded and it causes a dropdown of the connection speed and the **fulfilment of the 4 buffers** in the software(2 original by 3Proxy and other 2 in Aether).

7.2.2 Aether load

Due to the load of the requests, it is possible that Aether cannot keep up with all of them. This brings the system, on both side, in a state that is not guaranteed stable downloads. As soon as the load becames less, the downloads rocket up to the maximum speed possible. This behaviour is justified from the fact that, on the overload situation, there is a congestion in delivering the packets; this results in slowing down the speed of the download and in some cases also the interruption of it.

7.2.3 Performance

Generally the performance have suffered a drop due to the computational expanse of Aether intermediation. It was possible to notice a decrease of 5 times with respect to the normal speed of the proxy running alone. It is acceptable considering that the plugin has to listen on 2 sockets more, generate packets and send them. Moreover, in the case that there is a lot of packets to deliver, there is a congestion phase that tends to drastically reduce the overall throughput.

Chapter 8

Conclusion and future developments

The aim of this work was to provide a proxy service to each node in the DTN network that requests its use. For this reason, a design that already acted as a proxy was studied and from there adapted in the context of a DTN. The plugin mechanism was used in order to be able to transparently call functions that had access to the data structures of the bundles in order to communicate with the Aether. Specifically, the part of bundle management, sending, receiving and processing used some code and files from Tinyproxy project. In general, 3Proxy2Bundle works correctly and efficently. In some cases, there might be problems with the reordering of packages or security issues.
Future developments may focus on improving failures and delay situation, e.g. when the connection fails and the downloads of files via FTP don't terminate properly. Performance could also be improved analysing the congestion problem in Aether.

Bibliography

[1] *3Proxy project.* URL: https://github.com/z3APA3A/3proxy.

[2] *An explanation of the SOCKS protocol and application proxy gateway systems.* URL: http://www.infosecwriters.com/text_resources/pdf/what_is_socks.pdf.

[3] Daya Ram Budhathoki. *Computer Network : Lecture Notes.* URL: https://dayaramb.files.wordpress.com/2011/03/computer-network-notes-pu.pdf.

[4] Calogero Carrabbotta. *Comunicazione self-optimized tra dispositivi eterogeneamente connessi ad una rete con tolleranza ai ritardi.* Master thesis. Politecnico di Torino, 2019.

[5] Michael Demmer, Jörg Ott, and Simon Perreault. *Delay-tolerant networking tcp convergence-layer protocol.* RFC 7242. IETF, 2014.

[6] D. Ellard and D. Brown. *Dtn ip neighbor discovery (ipnd).* Internet-Draft draft-irtf-dtnrg-ipnd-01. IETF, 2010. URL: https://tools.ietf.org/html/draft-irtf-dtnrg-ipnd-01.

[7] Mignini Fabio. "User-oriented Network Service on a Multi-domain Infrastructure". IBM Global Services, Network ServicesVersion 2.0, Dec. 2014.

[8] Martin Arndt Gérard Ségarra. *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions.* ETSI document TR 102 638. ETSI, 2009.

[9] *Lecture Notes on "Computer and Network Security".* 2020. URL: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture19.pdf.

[10] *Managed File Transfer and Network Solutions.* 2012. URL: https://www.jscape.com/blog/bid/87783/forward-proxy-vs-reverse-proxy.

[11] Johannes Morgenroth. *Ibr-dtn api.* Tech. rep.

[12] Johannes Morgenroth. *Ibr-dtn configuration file example.* Tech. rep.

[13] *OpenDaylight project website.* URL: http://www.opendaylight.org/software.

[14] *Proxy and reverse proxy servers.* URL: http://www.idc-online.com/technical_references/pdfs/data_communications/Proxy_and_reverse_proxy_servers.pdf.

[15] Sebastian Schildt et al. “Ibr-dtn: A light-weight, modular and highly portable bundle protocol implementation.” In: *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011* (2011). DOI: http://dx.doi.org/10.14279/tuj.eceasst.37.512.544.

[16] K. Scott. *Disruption tolerant networking proxies for on-the-move tactical networks.* Tech. rep. 2005.

[17] K. Scott and S. Burleigh. *Bundle Protocol Specification.* RFC 5050. IETF, 2007. URL: https://tools.ietf.org/html/rfc5050.

[18] K. Scott and S. Burleigh. *SOCKS protocol.* RFC 1928. IETF, 2007. URL: https://tools.ietf.org/html/rfc1928.

[19] SDN and OpenFlow World Congress. “Network Functions Virtualization white paper”. In: Oct. 2012. URL: http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[20] *Understanding and implementing socks server.* URL: https://www.giac.org/paper/gsec/2326/understanding-implementing-socks-server-guide-set-socks-environment/104018.