DTN Bundle Layer over TCP: Retransmission Algorithms in the Presence of Channel Disruptions

Carlo Caini, Rosario Firrincieli and Marco Livini
DEIS, University of Bologna, Bologna Italy
E-mail:{ccaini, rfirrincieli}@arces.unibo.it {marco.livini}@studio.unibo.it

Abstract*—"Challenged networks" often violate the TCP design key assumption of continuous path availability from the source to the sink node. These networks are the preferred target of Delay/Disruption Tolerant Networking (DTN), which aims at providing a more robust network architecture. Against disruptions, however, even standard TCP offers a certain level of robustness, through its many retransmission algorithms. This intrinsic TCP resilience against disruptions is retained and enhanced if TCP is inserted in a DTN architecture. Focusing on this, the aim of the paper is to present an in-depth analysis of TCP and DTN bundle protocol retransmission algorithms that are triggered by channel disruptions, thus providing the reader with a comprehensive view of the many mechanisms involved and their complex interactions. The first sections, devoted to the description of TCP and DTN algorithms, are completed by the presentation of some numerical evaluations obtained by means of a Linux testbed. They refer to a GEO satellite environment and clarify the previous algorithm descriptions. Moreover, they also offer the reader some useful insights on both TCP and DTN resilience to disruptions, considering some performance metrics, like the "maximum tolerable disruption length" and the "restart delay", introduced and explained in the paper.

***Index Terms*—DTN, Bundle layer, TCP, Disruptions, Challenged networks, Satellite communications**

I. Introduction

A key assumption of TCP design is the availability of a continuous path from the source to the sink node. This however, is not always the case in the "challenged networks", which comprise a wide variety of different environments, from sensor networks to space communications. These networks are the preferred target of Delay/Disruption Tolerant Networking (DTN) [1], [2], [3], [4], which, as the name suggests, aims at providing a more robust network architecture against long delays, channel disruptions, and limited or intermittent connectivity. Among challenged networks, Land Mobile Satellite (LMS) systems should be also included because of the long Round Trip Time (RTT), and also the possible presence of frequent disruptions and intermittent connectivity. In this case, obstructions, like tunnels or buildings, can actually prevent or severely impair satellite signal reception, causing frequent connectivity disruptions. This is precisely the environment at the basis of our study, which is, however, more general and can be applied to other disruptive channels as well.

The DTN architecture considered here relies upon the introduction of the "bundle layer" between the transport and the application layers, in order to provide end-to-end data transfer across heterogeneous networks. Aiming at data reliability, TCP is selected at transport layer and the DTN "custody transfer" option is enabled. The aim of this paper is to present an in-depth analysis of TCP and DTN bundle protocol retransmission algorithms triggered by channel disruptions. On related topics some papers have recently appeared in the literature. In particular, a survey on the issues of DTN over TCP and other transport protocols is presented in [5]; a comparison of several possible DTN reliability approaches is given in [6]; performance evaluations and architecture descriptions are presented in [7], [8], [9]. With respect to the present literature, this paper offers a novel contribution by focusing on the description and comparison of DTN and TCP retransmission algorithms and their relationships.

The analysis presented in the paper is incremental, starting from a thorough investigation of TCP retransmission algorithms, as the DTN architecture considered in the paper retains and complements most of TCP mechanisms. Although conceived essentially to cope with losses due to congestion and not to channel unavailability, TCP retransmission mechanisms provide a certain level of robustness also against channel disruptions. These procedures are well documented in many RFCs [10], [11]. However, as RFCs leave several degrees of freedom, it is also important to study, as in the paper, their actual implementation on real operating systems, like Linux. The study of TCP algorithms, as well as being significant in itself, is also propedeutic for the subsequent analysis of DTN retransmission mechanisms. In this case, there is still less documentation available about implementation details, as DTN RFCs [3] and [4] are more general on this point. Therefore, the algorithm descriptions given in the paper derive from direct inspection of the bundle protocol reference

Manuscript received April 30, 2009; revised September 25, 2009; accepted January 20, 2010.

implementation code and of real packet traces of TCP segments and DTN bundles. The DTN analysis is carried out at three different levels (end-to-end TCP, single hop DTN, and end-to-end DTN), with the aim of highlighting both TCP and DTN mechanisms and the complex interactions between them. The analysis is completed by the presentation of some numerical evaluations obtained by means of the TATPA (Testbed on Advanced Transport Protocols and Architectures) testbed [12]. These evaluations, which refer to a GEO satellite environment, support the previous algorithm descriptions and give a first assessment of disruption impact on both TCP and DTN performance, considering some metrics introduced in the paper, like the "maximum tolerable disruption length" and the "restart delay".

The structure of the paper is as follows. In the next section, a brief overview of the DTN architecture is presented; TCP and DTN retransmission algorithms are then described in Section III and IV, respectively; numerical evaluations are reported and discussed in Section V; conclusions are drawn at the end of the paper.

This paper represents an extended and largely enhanced version of [13].

II. DTN Architecture Overview

The DTN bundle layer architecture [1], [3], [4] aims to support communication in challenged environments by adding a new layer between transport and application layers, called "bundle layer". The bundle protocol can interface with different transport protocol through its "convergence layer" interface, specific for each transport protocol. In this new architecture (see Figure 1), transport protocol end-to-end features are confined to homogeneous network segments (A, B and C), while end-to-end data transfer across the heterogeneous network is provided by the bundle protocol through store-and-forward relay between DTN nodes of large data packets called "bundles". In a DTN architecture, different transport protocols, as well as different network stacks, may be used in different network segments. In this way it is possible to use optimized protocols, or optimized versions of the same protocol, to tackle the different characteristics of different network segments, like a satellite channel or a sensor network. Note that from this point of view, the DTN architecture recalls and extends the TCP connection splitting approach pursued in Performance Enhancing Proxies (PEPs) [14], a solution widely applied on satellite and wireless systems.

A. End-to-end reliability and custody transfer option

End-to-end reliability at bundle layer can be provided by selecting a reliable transport protocols, like TCP, if available at DTN nodes, and/or through two optional mechanisms: end-to-end acknowledgments and custody transfer. Applications are left free to exploit the former, which is just a signaling process, like parcel tracking of a courier, while the latter implies persistent storage (e.g. on hard disks) and bundle retransmission capabilities at some selected DTN nodes. Quoting RFC 4838 [3]: "Transmission of bundles with the custody transfer requested option specified generally involves moving the responsibility for reliable delivery of bundles among different DTN nodes in the network. For unicast delivery, this will typically involve moving bundles "closer" (in terms of some routing metric) to their ultimate destination(s), and retransmitting when necessary. The nodes receiving these bundles along the way (and agreeing to accept the reliable delivery responsibility) are called "custodians". The movement of a bundle (and its delivery responsibility) from one node to another is called a "custody transfer" ".

The interested reader is referred to [15] for an exhaustive discussion of the many significant implications of the custody transfer option. Here let us point out the advantages that can derive from this in the presence of a discontinuous or disruptive channel. In the case of link disruption, DTN bundles continue to be safely stored at DTN custodians in local databases, waiting until the next hop becomes reachable again, and then they are transmitted as soon as the connection makes it possible. Bundles are deleted from custodian databases once the transfer of their custody is explicitly acknowledged by another custodian or by the destination node, or their lifetime expires. Of course, the implementation of the custody transfer mechanism requires the possibility to create and maintain a database in the custodian machine. The larger the database, the larger the number of incoming DTN bundles that can be temporarily stored. This requirement could be difficult to reach for nodes with limited hardware resources, like sensors. However, they usually act as source end nodes, and can greatly benefit from the custody transfer mechanism. In fact, as soon as a sensor has transferred a bundle to the next DTN node accepting custody, it can immediately cancel the bundle from its memory, by relying on custodian resources for storing and successful final delivery.

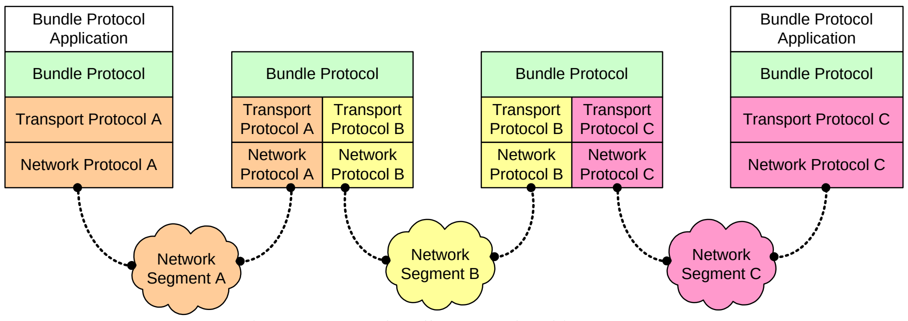

Figure 1. DTN bundle protocol architecture.

B. DTN concept development and other remarks

The concept of DTN was first conceived within the Inter Planetary Network Research Group (IPNRG) of the Internet Research Task Force (IRTF), to specifically address the problems of space communications. Later, its field of applicability was enlarged to encompass all challenged networks, either spatial or terrestrial. To this end, in 2002 the IRTF Delay Tolerant Networks Research Group (DTNRG) was established with the aim of promoting DTN. An interesting survey of DTN motivation and development, written by one of the two DTNRG leaders, can be found in [1]. DTNRG website [16] is the most important source of DTN documentation and software code. DTN bundle protocol reference implementation is freely available from [17]. Other DTN applications are also available, including DTNperf_2, a DTN performance evaluation tool, co-developed by the authors and used in the numerical evaluation section of this paper [18]. Its latest release can be found in [19]. Finally note that, although the bundle layer is the most important DTN architecture, it is not the sole possible option, as documented in the DTNRG web site.

III. TCP RETRANSMISSION ALGORITHMS

This section presents a detailed analysis of TCP behavior when a channel disruption occurs. However, before proceeding with the analysis, a preliminary discussion on TCP timers is mandatory as TCP uses timers extensively every time it does not receive feedback from the other connection peer, which is the case in the presence of channel disruptions.

A. TCP timers overview

TCP manages four different timers for each connection [20]. The *retransmission* timer is used to cope with lost data segments and/or lost ACKs. In the early versions of TCP (e.g. Tahoe) [21] its expiration was the only way to trigger segment retransmission. Because of its importance it will be discussed apart in the next subsection. Note that the same algorithm is also used to retransmit lost SYN segments at the connection set-up.

The *persist* timer is used to avoid a possible dead-lock situation. In a TCP connection, the receiver can control the data flow by specifying the amount of data it is willing to accept from the sender through the specification of the *window size* field in the returning ACKs. In some cases, the receiver can report a zero window size, which effectively stops the sender from transmitting data, until a nonzero window size update is received. However, if this update gets lost, a deadlock arises because the receiver waits to receive data since it provided the sender with a nonzero window and the sender waits to receive the window update allowing it to send. To avoid this deadlock the sender uses a persist timer through which it queries the receiver periodically, to find out if the window has been updated.

A *keepalive* timer is used to detect if the other peer on an idle connection crashes or reboots. An idle connection is a connection where nothing is exchanged between the two TCP peers. When a server wants to know if the client has either crashed and is down, or crashed and rebooted, it uses the *keepalive* timer to send probe segments.

The *2MSL* timer measures the time a connection is in the TIME_WAIT state [20]. At the end of an active closure, the connection enters the TIME_WAIT state in which it must stay for two times the Maximum Segment Lifetime (MSL), i.e. the maximum amount of time any segment can exist in the network before being discarded. The 2MSL timer enables the retransmission of the final ACK in case this ACK is lost, in which case the other peer will time-out and retransmit its final FIN.

In the next subsection we will focus on the retransmission timer, which plays a key role in our study.

B. The retransmission timer algorithm

TCP uses the retransmission timer to react to feedback absence from the remote data receiver. The duration of this timer is referred to as RTO (Retransmission Time Out) and its calculation is fully described in RFC 2988 [10]. The TCP sender sets the initial RTO value to 3 s. Then the RTO is continuously updated during connection lifetime by making use of two variables, SRTT (Smoothed RTT) and RTTVAR (RTT Variation). Following the RFC 2988 the RTO should be conservatively rounded up to 1 s to avoid spurious timeouts. Linux implementation, however, fixes this minimum value to a lower threshold (*TCP_RTO_MIN*=200 ms), because it can rely on a fine clock grain (1 ms) and on advanced spurious timeout recognition techniques, such as F-RTO [22]. The same RFC states that a maximum value may be set, provided that this threshold is greater than 60 s. Linux implements this optional feature by doubling this value (*TCP_RTO_MAX* = 120 s).

The TCP retransmission algorithm consists of many mandatory and optional features, which leave space to multiple implementations. The flow chart in Figure 2, which we are going to describe, refers to Linux OS (kernel 2.6.26).

The algorithm can be split into two phases: before and after RTO expiration.

In the former, the TCP sender applies the following actions (left-hand flow in Figure 2):

1. every time a packet containing data is sent (including a retransmission), if the timer is not running, it is started so that it will expire after RTO seconds (for the current value of RTO);
2. when all outstanding data has been acknowledged, the retransmission timer is turned off;
3. when an ACK is received that acknowledges new data, the retransmission timer is restarted so that it will expire after RTO seconds (for the current value of RTO).

In the latter, after RTO expiration, we have the following steps (right-hand flow in Figure 2):

1. the earliest segment that has not been acknowledged by the TCP receiver is retransmitted;
2. the TCP sender backs off the RTO by doubling it (*RTO =2 ·RTO*). A maximum value may be applied to provide a ceiling to this doubling operation; as aforementioned, this optional feature is adopted in Linux (*TCP_RTO_MAX* =120 s);
3. the retransmission timer is started, such that it expires after RTO seconds (for the value of RTO after the doubling operation);
4. if the retransmission timer expires again for the same packet, the RTO is further doubled, the timer is restarted and the segment is retransmitted;
5. In [10] it is stated that the previous point must be repeated either for a given time or for a given number of retries; in Linux the second option is adopted (*TCP_RETR2* variable). After this threshold is reached, and the transmission of the segment still fails, the connection is closed. There is also a lower threshold (*TCP_RETR1* variable), which allows the TCP to pass negative advice to the IP layer, which in turn triggers dead-gateway diagnosis. Linux default values for these variables are respectively: *TCP_RETR1* = 3 and *TCP_RETR2* = 15. Note that SYN retransmissions follow the same RTO algorithm, but with a different maximum number of retransmissions (*TCP_SYN_RETRIES* = 5).

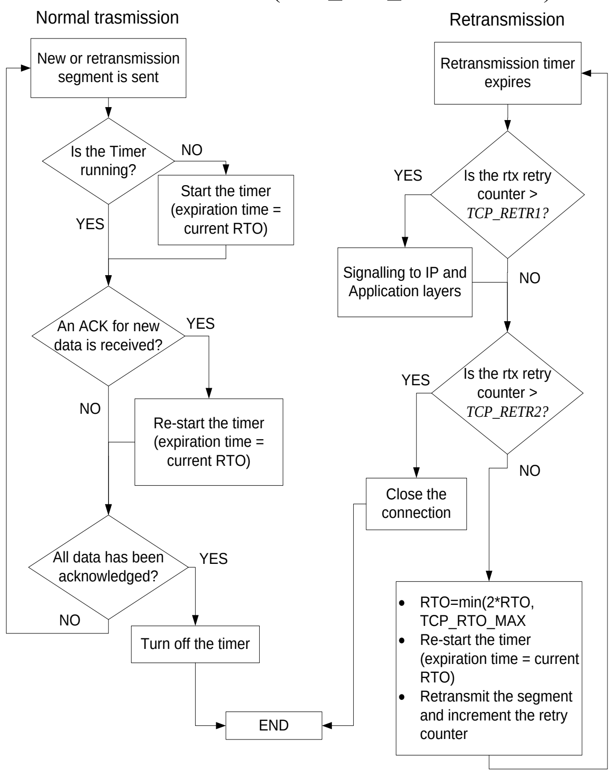
Normal trasmission
New or retransmission segment is sent
Is the Timer running?
NO
YES
Start the timer (expiration time = current RTO)
An ACK for new data is received?
YES
NO
Re-start the timer (expiration time = current RTO)
All data has been acknowledged?
YES
NO
Turn off the timer
END
Retransmission
Retransmission timer expires
Is the rtx retry counter > TCP_RETR1?
YES
NO
Signalling to IP and Application layers
Is the rtx retry counter > TCP_RETR2?
YES
NO
Close the connection
RTO=min(2*RTO, TCP_RTO_MAX
Re-start the timer (expiration time = current RTO)
Retransmit the segment and increment the retry counter

Figure 2. TCP retransmission algorithm flow chart.

C. Remarks on the maximum tolerable disruption length and TCP responsiveness to connection reestablishment

Note that the Linux choice in favor of a retry threshold implies that the Maximum Tolerable Disruption Length (MTDL), i.e. the longest disruption that does not cause the connection to close, is not directly set but depends on the following parameters: the RTO value at disruption start, the number of retries allowed and the max RTO. It is worth stressing the key role played by this last parameter in determining both MTDL and responsiveness of TCP after relatively long disruptions. Actually, in the presence of a long disruption the channel availability is probed at doubling time intervals (exponential backoff algorithm) until the maximum RTO value is reached. Then, it is probed at constant regular intervals until either an ACK is received or the maximum number of retransmission is reached (see Figure 3).

The maximum RTO value represents the maximum time interval between consecutive channel probes; therefore, it also represents the longest possible *restart delay*, i.e. the time elapsed between the end of link disruption and the following transmission restart. The longer this delay, the higher the performance penalization, as the channel is left unexploited for a longer interval.

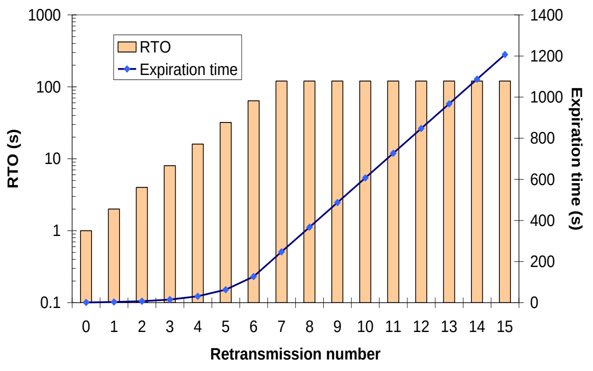

Figure 3. TCP RTO timers and their expiration time; initial RTO=1 s, max RTO = 120 s.

Finally, having fixed the maximum RTO, MTDL depends on the maximum number of retries, as shown in Figure 4 for three different initial RTO values. Note that the 3 s curve can also represent MTDL at connection set-up; in this case the maximum number of retransmissions on the x-axis must refer to the SYN retries threshold.

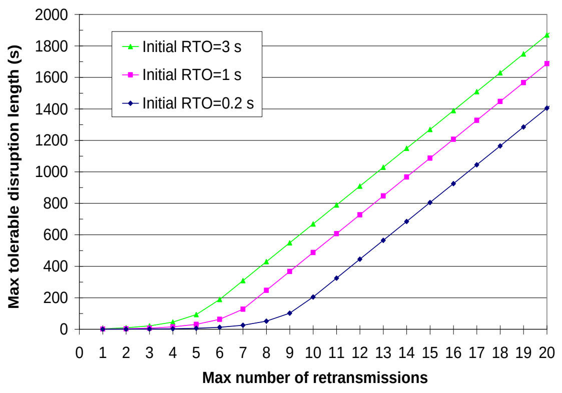

Figure 4. Maximum tolerable disruption length versus maximum number of retransmissions; variable initial RTO, max RTO = 120 s.

MTDL curves can be obtained analytically. First, having denoted with RTO_{in} and RTO_{max} the initial and the maximum RTO, it is convenient to define $\tilde{n}$ as

$$\tilde{n} \overset{def}{=} \left\lceil \log_2\left(\frac{RTO_{max}}{RTO_{in}}\right)\right\rceil \tag{1}$$

This parameter represents the first retransmission after which RTO is set to RTO_{max} and, consequently, the start of the linear MTDL increase. Then, having denoted with n_{max} the maximum number of retransmissions, we have to distinguish two cases. In the former, RTO_{max} is attained before n_{max} is reached; in the latter, RTO_{max} is never attained, as n_{max} is too low. In formulas we have:

if $\tilde{n} < n_{max}$

$$MTDL = \sum_{k=0}^{\tilde{n}-1} 2^k \cdot RTO_{in} + \sum_{k=\tilde{n}}^{n_{max}-1} RTO_{max} =$$
$$= RTO_{in} \cdot \; 2^{\tilde{n}} - 1 \; + \; n_{max} - \tilde{n} \; RTO_{max} \tag{2}$$

otherwise

$$MTDL = \sum_{k=0}^{n_{max}-1} 2^k \cdot RTO_{in} = RTO_{in} \cdot \; 2^{n_{max}} - 1 \tag{3}$$

Note from Figure 4 that by adopting Linux defaults (RTO_{max}=120 s and n_{max}=15) for an initial RTO=1 s the corresponding MTDL is a little shorter than 1200 s. This means that disruptions longer than 20 minutes result into a forced closure of the TCP connection. This disruption resilience can be improved by changing TCP defaults. However, from the previous discussion it should be clear that this sort of TCP tuning, although possible, is far from being straightforward. Finally, note that the TCP resilience is basically the same for all TCP variants, included those experimental. In fact, they generally differ in the congestion control policies, but retain standard retransmission mechanisms.

IV. DTN Bundle Protocol Algorithms

In this section a detailed analysis of DTN bundle protocol behavior in case of disruptions is presented. Focusing on reliable transmissions, TCP is assumed as transport protocol in all DTN nodes and custody transfer option is enabled. For treatment convenience, the analysis is divided into two phases: first, we consider a single DTN hop between two consecutive DTN nodes; then, a typical end-to-end bundle flow from a DTN sender to a DTN receiver.

A. DTN "link class" retransmission timers and back-off policy (single hop DTN analysis)

We start our analysis by considering a disruptive link between two consecutive DTN nodes. Having considered TCP at transport layer, all the TCP timers and retransmission mechanisms are retained and bundle protocol algorithms are applied on top of them. In particular, we focus on the DTN retransmission timers specified in the DTN "link class" (upper part of convergence layer), with emphasis on their interaction with TCP timers. The algorithm, derived from direct inspection of the DTN2 bundle protocol reference implementation code (rel. 2.6.0) [17], is summarized in Figure 5. Note that other bundle protocol implementations may differ. Bundle protocol interactions with TCP are emphasized in bold.

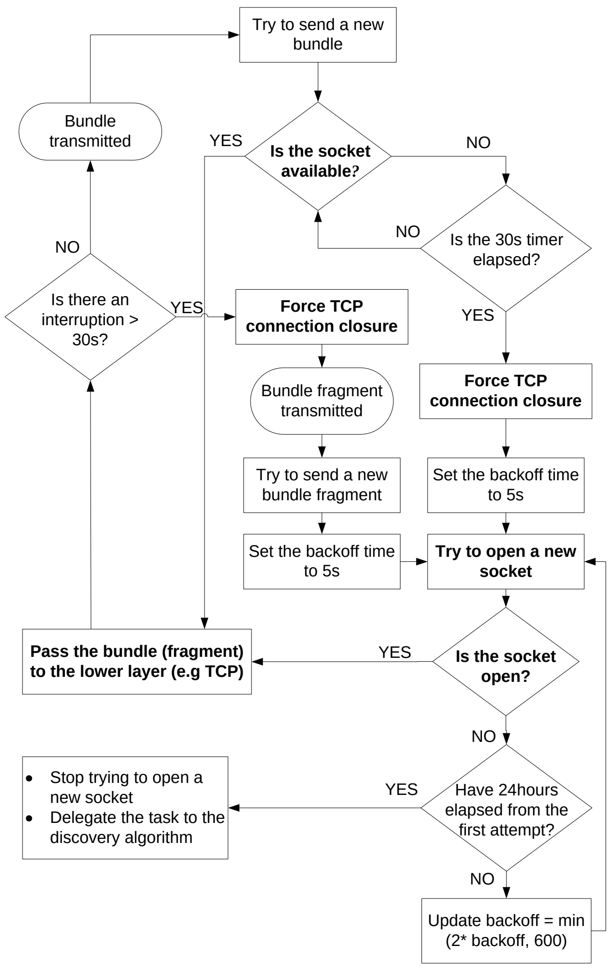
Try to send a new bundle
Bundle transmitted
YES
Is the socket available?
NO
NO
Is the 30s timer elapsed?
NO
Is there an interruption > 30s?
YES
Force TCP connection closure
YES
Bundle fragment transmitted
Force TCP connection closure
Try to send a new bundle fragment
Set the backoff time to 5s
Set the backoff time to 5s
Try to open a new socket
Pass the bundle (fragment) to the lower layer (e.g TCP)
YES
Is the socket open?
NO
Stop trying to open a new socket
Delegate the task to the discovery algorithm
YES
Have 24hours elapsed from the first attempt?
NO
Update backoff = min (2* backoff, 600)

Figure 5. DTN "link class" retransmission timers and back-off policy (single hop DTN analysis).

When the bundle protocol wants to send a bundle to the next DTN node, it generally checks the status of the link towards this destination by asking the lower layer (TCP in our case) for socket availability. If the answer is positive, the link is considered open and active and the bundle is passed to the lower layer for transmission. However, let us consider first the opposite case (right-hand flow in Figure 5).

If the socket is unavailable, DTN application sets a timer, whose expiration time is given by the DTN variable *data_timeout_* (set to 30 s as default), and waits for the socket to be available again. If this does not happen before timer expiration, the socket itself is closed (the underlying TCP agent sends a FIN segment to the other peer), and after the retry interval given by the *min_retry_interval_* variable (file Link.cc [17], initial value of 5 s) tries to open a new socket. As DTN relies on TCP to open a new socket, the standard TCP procedure is adopted. This means that TCP will try to open the new socket by sending a SYN segment to the other peer. In case of failure, the SYN segment transmission is repeated and so on, until either the SYN is acknowledged or the maximum number of SYN retransmissions is reached. As mentioned before, the interval between consecutive SYN retransmission is progressively increased following the RTO exponential back-off algorithm, starting from an initial value of 3 s. If the other peer is still unreachable after the maximum number of SYN retries, the DTN application doubles the retry interval and again asks TCP to open a new connection. This back-off algorithm fixes the maximum retry interval to 600 s; moreover, the maximum amount of time during which the DTN application will try to open a new connection is set to 24 h. If, after 24 hours, the other peer is still unreachable, the DTN application stops trying to use this link and delegates to the discovery algorithm the task of checking the link availability. This extends the MTDL of the DTN architecture beyond 24 hours, a value definitively larger than the 20 minutes of TCP with Linux default. As a final remark, note that DTN resilience is independent from the bundle size (either variable or fix) and from the TCP version adopted.

Now, let us describe the opposite case, i.e. what happens when the socket is found available (left-hand flow in Figure 5). The bundle, waiting in a DTN queue, starts to be transferred to the transport layer. If no disruptions occur, the bundle is successfully transmitted to the next DTN node. Otherwise, the socket becomes unavailable and the *data_timeout_* timer is set to 30 s as before. At its expiree DTN forces a link closure (a FIN segment is sent in the TCP case) and starts a new socket opening, which follows the algorithm just described. However, in order to avoid retransmissions of data already successfully received, the current bundle is split into two bundle "fragments" as a result of connection closing ("reactive fragmentation" [3]). The first segment consists of bundle data already sent, the second of its complement. The algorithm continues with the attempt to send this second fragment, following the same rules as before. Note that in the most unfavorable conditions consecutive fragmentations are possible, i.e. a fragment can be further fragmented. The algorithm terminates when all the fragments of the same bundle are sent.

B. Bundle transmission process from DTN sender to DTN receiver (end-to-end DTN analysis)

Having examined in the previous subsection single hop transmission, the next step is to investigate the transmission of one bundle from DTN sender to DTN receiver. The focus is on the bundle transmission process, i.e. on custody transfer mechanism, bundle fragmentation and reassembling, possible deletion and final delivery. As these features are specified in RFC they should be common to all bundle protocol implementations.

A simplified description of the end-to-end process, derived from RFC documentation and from validation tests carried out by the authors, is given in Figure 6.

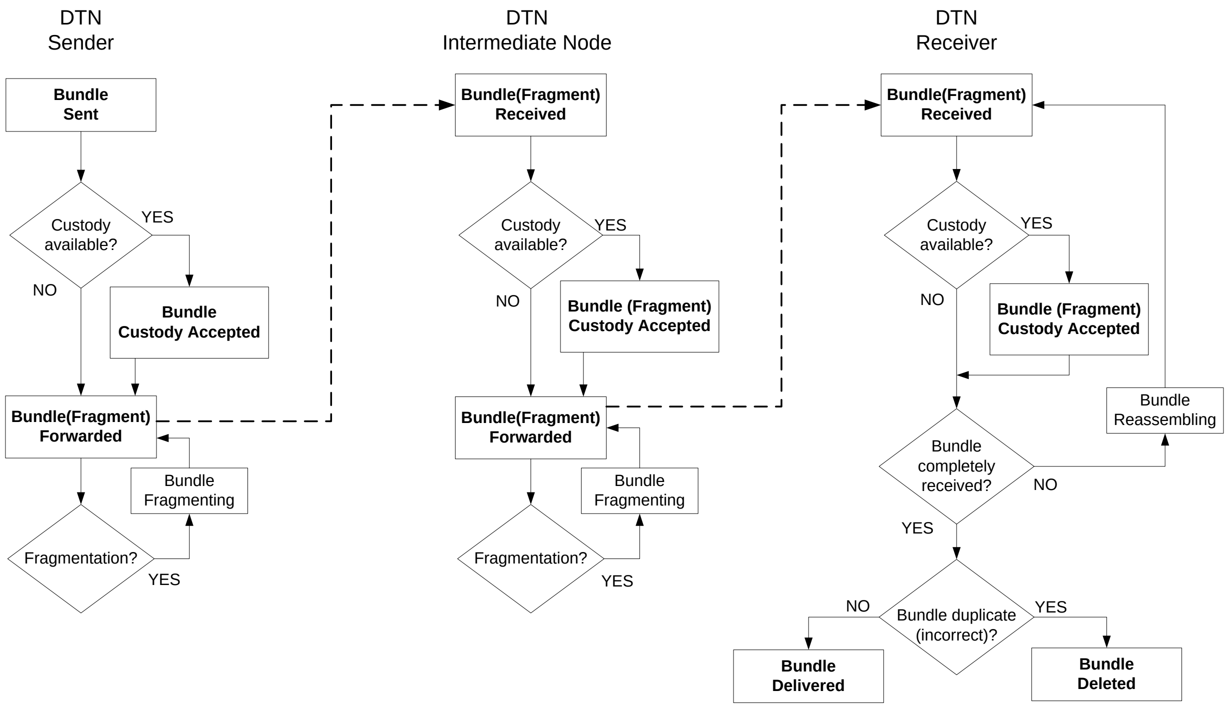

Figure 6. Bundle transmission process from DTN sender to DTN receiver (end-to-end DTN analysis).

Here, the state of a bundle is described by the name of the corresponding "status report" administrative bundle (in bold), which can optionally be sent to the "report-to" node (e.g. the DTN sender). First, a status "sent" indicates that the application has passed the bundle content to the bundle protocol. We assume that custody transfer is required. If the node can accept this request ("custody accepted"), the bundle is written on a local database, waiting for transmission. As described in the previous subsection, the bundle can be fragmented (reactive fragmentation) if the transport socket becomes unavailable because of link disruption. Fragments are then treated as ordinary bundles by subsequent nodes until reassembling. The transmission of each bundle, or bundle fragment, is acknowledged by a status "forwarded"; its reception by a status "received". The intermediate node may accept or not custody. The process continues as before possibly through other DTN intermediate nodes (not shown in the figure) until the receiver is reached. Here bundle fragments are reassembled and the bundle is eventually either delivered to the application or deleted (status "delivered" or deleted", respectively). The deletion of a bundle may result from a duplicate reception. This figure is simplified as it does not consider the possibility of bundle reassembling at intermediate nodes and "proactive" fragmentation decided by a DTN node before bundle transmission and not after disruption. Proactive fragmentation allows transmission of large bundles over channels with intermittent limited time availability, as in LEO (Low Earth Orbit) satellite systems. Finally, let us recall that custody transfer is carried out between DTN nodes through the exchange of "custody signals" independent of the (optional) corresponding status reports.

V. NUMERICAL EVALUATIONS

Numerical evaluations in this section are presented in support of the description of retransmission algorithms presented in the previous sections through the direct inspection of real segment or bundle time traces. TCP and DTN retransmission procedures are tested by considering, as an application example, a GEO satellite connection (RTT=600 ms) with channel disruptions of variable length. Tests are carried out through the TATPA testbed [12], whose logical topology, with a typical "butterfly" layout, is sketched in Figure 7.

Satellite peers are represented by the satellite sender and receiver, while an intermediate satellite emulator adds desired delays and disruptions. In our analysis we used NistNet [23] to emulate delays, and an in-house tool based on Netfilter/iptables [24] utility to emulate disruptions. Wired peers, at the bottom of the figure, may insert wired cross traffic to study congestion effects on the 10 Mb/s bottleneck between the routers R1 and R2.

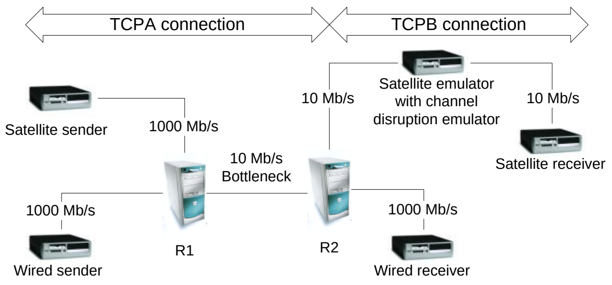

Figure 7. TATPA layout.

Either Drop Tail or RED queuing policies with variable parameters can be set on R1, where all congestion effects are confined. In numerical evaluations a RED policy (qlen=50 seg., maxth=15 seg., and minth=5 seg.) is adopted.

Note that wired peers, although present in TATPA, are actually not used here, where the focus is on the satellite peers. Iperf [25] and DTNperf_2 (latest release 2.3) [18], [19] are used to transfer data between the two satellite nodes in the end-to-end TCP and DTN case, respectively. As regards TCP, Linux retransmission timer parameters are set to their default values.

In the DTN case a further description is required. Both the satellite sender and receiver are DTN nodes (DTN source and sink), as the router R2 (DTN intermediate node) is. Unlike the end-to-end TCP case, where there is just a TCP connection between the two satellite peers, here we have to distinguish between two TCP connections: a first connection, TCP A, from the satellite sender to R2, and a second connection, TCP B, from R2 to the satellite receiver. For clarity, these connections have been highlighted with arrows in Figure 7. The corresponding DTN protocol stack is given in Figure 8.

DTN link and routing configurations are given in Table I for the three DTN nodes considered. Routing names, IP addresses and TCP ports allow each DTN node to communicate with other DTN nodes through TCP sockets. These sockets can be opened following different approaches. The adopted ONDEMAND approach implies that a TCP socket is open only when there are data to transfer. Other approaches are possible, as ALWAYSON and OPPORTUNISTIC. The former forces the creation of a new TCP socket as soon as the DTN node starts up. The latter implies that the TCP socket can either be opened manually, or when another DTN node tries to connect and finds an unopened OPPORTUNISTIC link. Comparing these different approaches lies out with the scope of this paper; hence, we limit our attention to the ONDEMAND approach. Given the bidirectional nature of DTN communications, this approach implies the opening of two (bidirectional) TCP connections for each TCP node pair. This results in a total of four TCP sockets, two for TCP A and two for TCP B (see Figure 8). Once opened, data and signaling between two consecutive nodes can be transferred on either of the two TCP sockets available, depending on the bundle protocol choice.

TABLE I.
DTN LINK AND ROUTING CONFIGURATION ADOPTED.

Satellite Sender (SS)	R2 (Gateway)	Satellite Receiver (SR)
link add Gateway IP_R2:4556 ONDEMAND tcp	link add Sender IP_SS:4556 ONDEMAND tcp	link add Gateway IP_R2:4556 ONDEMAND tcp
	link add Receiver IP SR:4556 ONDEMAND tcp	
route add SR.dtn Gateway	route add SS.dtn Sender	route add SS.dtn Gateway
route add G.dtn Gateway	route add SR.dtn Receiver	route add G.dtn Gateway

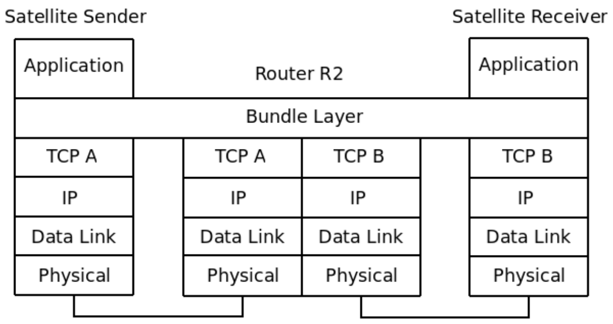

Figure 8. DTN stack considered in the numerical evaluations.

Here we noticed a prevailing use of only one of the two sockets; however, in [26], we observed how, in some circumstances, both sockets can be profitably used to transfer data in the same direction.

In the following subsections real time traces are presented and discussed. These traces were obtained through tcpdump [27] logs for TCP, and DTNperf_2 logs for bundles. Tcpdump monitors were placed on both left and right links of the satellite emulator to capture every data segments. Tcpdump traces were then analyzed through the Wireshark network analyzer [28]. The obtained results are presented into the following three subsections. The first is related to the end-to-end TCP case and should be considered as a reference. The second to the single hop DTN case; the third regards the end-to-end DTN bundle transfer process.

A. End-to-end TCP

In order to test the algorithms previously analyzed, two numerical examples are given, considering disruptions of different lengths, occurring after the TCP connection has reached the steady state. In the first case, a disruption of 600 s (10 min) causes 12 retransmissions before restarting. This behavior is coherent with MTDL, which is of 1039 s considering the initial RTO value of 0.78 s at disruption start, and default TCP_RTO_MAX and TCP_RETR2 values. Total actual transmission interruption time is of 682 s. By subtracting the disruption length, we obtain a restart delay of 82 s. Note that at steady state, the restart delay value is roughly independent from the exact moment when the disruption takes place, although limited variations are possible due to the possible variations of the initial RTO timer.

In the second case, a 1200 s (20 min) disruption is considered. As in this launch at disruption start the initial RTO is 1.39 s, the corresponding MTDL with default TCP_RTO_MAX and TCP_RETR2 values is 1137 s, which is actually lower than the disruption length. In accordance with expected behavior, TCP is finally forced to close the connection after 1257 s from last received ACK. Retransmissions and corresponding RTO timers derived from tcpdump traces are given in Figure 9. Each point represents a retransmission. More precisely, on the y-axis are plot the new RTO timer values set at retransmission times given in the x-axis.

It is worth noting that the long disruption lengths considered in the examples, as well as being significant for an exhaustive description of TCP RTO timer, may actually be encountered in land mobile satellite communications (e.g. long railway tunnels).

B. DTN "link class" over TCP (single hop DTN analysis)

With reference to Figure 8, here TCP A connection is assumed to be ideal, while disruptions are present only on TCP B. The DTN behavior is evaluated by examining the TCP B segment flows. Channel disruptions are the same as before, 600 s and 1200 s, respectively.

As aforementioned, while both R2 and the satellite receiver tries to open (or re-open after a disruption) a TCP socket, data are transferred only on sockets opened by R2 (sockets opened by the satellite receiver remain basically in idle state). Therefore, here we limit our analysis only to these TCP sockets.

Starting with the 600 s disruption test, it caused 4 new socket opening retries with a total Tx interruption time of about 672 s (i.e. with a restart delay of 72 s). After this interval, the normal data flow restarted. The gain in restart delay with respect to the analogous TCP case is of about 10 s.

The 1200 s disruption case shows DTN superiority over TCP. In fact, while TCP with Linux default settings was unable to tackle such a long disruption, DTN can cope with it. Analogously to Figure 9, referring to TCP case, Figure 10 reports retransmissions of the last unacknowledged segment (Retransmissions in the legend). However, being the DTN case much more complex, other series of transmitted segments have to be reported too. In particular, we have a FIN segment at 30 s. This is triggered by the DTN bundle protocol, which forces the TCP connection closure after DTN *data_timeout_* seconds from the last ACK (see Section IV). Then we have 5 consecutive unsuccessful socket opening attempts (spaced by an increasing retry interval), each consisting of 6 SYN segment transmissions (the first plus TCP_SYN_RETRIES retransmissions). As in the retransmission case, on the y-axis are plot the new SYN timer values set at transmission times given in the x-axis. After the first five unsuccessful attempts, the first SYN segment of the sixth attempt finds the channel available again. The transmission therefore restarts with a new TCP connection after the corresponding SYN_ACK segment reception (1290 s from the disruption start). The restart delay is of 90 s.

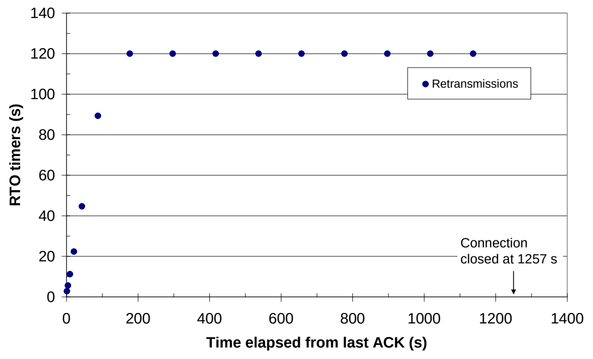

Figure 9. End-to-end TCP experimental data: retransmissions and RTO timers; disruption length = 20 min; initial RTO = 1.39 s.

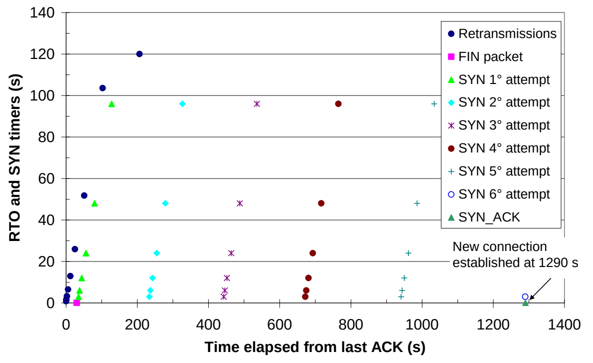

Figure 10. Single hop DTN experimental data: flow of TCP retransmissions, FIN and SYN segments; disruption length = 20 min; initial RTO = 0.82 s.

C. DTN bundle process (end-to-end DTN analysis)

In this section, we present numerical results supporting the description of end-to-end bundle transmission given in Section IV.B. Unlike the two previous cases, here we consider a much shorter disruption (150 s) in order to preserve result readability. It is worth noting that the considered disruption length is long enough to trigger a TCP socket closure through a FIN segment (see Figure 10). In the test examined here (Figure 11) we consider a total transfer length of 300 s with the disruption starting after 30 s from the beginning. Each bundle "history" is given in the figure through its most significant status reports, namely "sent", "forwarded" (only by the intermediate DTN node R2) and "delivered", while other possible reports are not shown for the sake of readability.

Note that, in contrast to the TCP traces shown in all previous figures, here the x-axis represents the absolute time from transfer start, and not from the interruption start. Looking at the graph, the first four bundles (bundle#0-bundle#3) are regularly sent by the satellite sender and delivered after a few seconds to the satellite receiver. By contrast, the transmission of bundle#4 is interrupted by disruption start. After 30 s the DTN bundle protocol forces the TCP connection closure and a first status report "forwarded" is sent from R2 to the "report-to" DTN node (here the satellite sender).

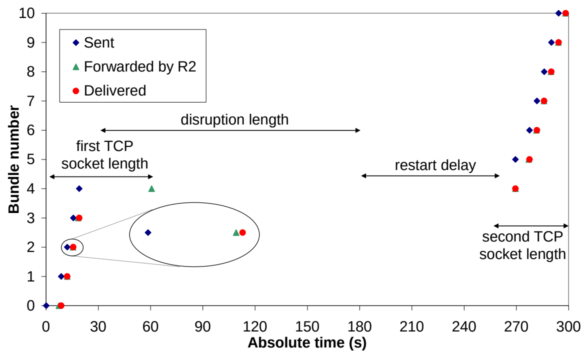

Figure 11. End-to-end DTN experimental data: flow of main DTN status reports.

By inspection of the tcpdump traces, it is possible to discover that the bundle was only partially transmitted to the satellite receiver (meaning that a reactive bundle fragmentation happened) and that transmission was re-established at the first SYN of the second re-opening attempt (at 263 s). After connection reopening the second fragment of bundle#4 is transmitted (second "forwarded" report for bundle#4 from R2) followed by a "delivered". Then, bundle transmission continues regularly till the end of the data transfer. The figure also shows the duration of the first TCP socket terminated by the disruption, the disruption length, the restart delay (about 83 s) and the duration of the second TCP socket opened after disruption end.

Finally, for the sake of completeness, in Table 2 the complete history (all available status reports) for bundle#3 (typical of all bundles transmitted without fragmentation) and bundle#4 (typical of reactive fragmentation induced by disruption) is given. The first column shows the time at which the satellite sender received the corresponding status report (including those sent by itself). The second, third and fourth columns, contain the report type, the report sender, and the bundle ID, respectively. Finally, the last column contains the fragment byte offset; an offset greater than zero indicates a bundle fragment whose first payload byte is equal to the offset.

VI. Conclusions

In this paper, an in-depth analysis of TCP and DTN bundle protocol retransmission algorithms in case of channel disruptions is presented. The study, carried out at different levels (end-to-end TCP, single hop DTN and end-to-end DTN), aims at pointing out the many mechanisms involved and their complex interactions. The analysis is completed by some practical examples referring to satellite communications, a promising DTN application field. The inspection and discussion of real packet traces provides the reader with a first quantitative evaluation of TCP and DTN resilience to disruptions. The analysis and the numerical results presented highlight the efficacy of the DTN architecture in coping with disruptive channels. It is authors' commitment to up-date the analysis presented here whenever retransmission features are added or modified in future DTN bundle protocol implementations.

TABLE II.
BUNDLES PROCESS: STATUS REPORTS FOR BUNDLE 3 AND 4.

Absolute Time (s)	Status report	Report Sender	Bundle number	Fragment offset
15.63019	SENT	SS	3	0
15.63836	CUSTODY ACCEPTED	SS	3	0
16.66910	RECEIVED	R2	3	0
16.68216	FORWARDED	SS	3	0
16.76918	CUSTODY ACCEPTED	R2	3	0
18.45637	RECEIVED	SR	3	0
18.48267	FORWARDED	R2	3	0
19.03292	DELIVERED	SR	3	0
19.04216	SENT	SS	4	0
19.05054	CUSTODY ACCEPTED	SS	4	0
20.10472	RECEIVED	R2	4	0
20.11976	FORWARDED	SS	4	0
20.12058	CUSTODY ACCEPTED	R2	4	0
60.62024	FORWARDED	R2	4	0
265.3536	RECEIVED	SR	4	0
265.9308	CUSTODY ACCEPTED	SR	4	0
269.472	FORWARDED	R2	4	4002
269.4756	RECEIVED	SR	4	4002
269.4992	CUSTODY ACCEPTED	SR	4	4002
269.5002	DELIVERED	SR	4	0

ACKNOWLEDGMENTS

The authors would like to thank P. Cornice, D. Lacamera and S. Tamagnini for their valuable support in the design and development of the software tools used in the paper.

REFERENCES

[1] K. Fall, S. Farrell, "DTN: an architectural retrospective", IEEE Journal on Selected Areas in Commun., vol.26, no.5, pp.828-836, June 2008.
[2] S. Farrell and V. Cahill "Delay- and Disruption-Tolerant Networking", Artech House, 2006.
[3] V. Cerf , A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss "Delay-Tolerant Networking Architecture", IETF RFC 4838, Apr. 2007.
[4] K. Scott, S. Burleigh, "Bundle Protocol Specification", IETF RFC 5050, Nov. 2007.
[5] S. Farrell, V. Cahill, D. Geraghty, I. Humphreys, and P. McDonald, "When TCP Breaks: Delay- and Disruption-Tolerant Networking", IEEE Internet Computing, vol. 10, no. 4, pp. 72-78, 2006.
[6] K. Harras and K. Almeroth, "Transport layer issues in delay tolerant mobile networks", IFIP Networking, vol. 3976, pp.463-475, May 2006.
[7] C.Rigano, K. Scott, J. Bush, R. Edell, S. Parikh, R. Wade, "Mitigating naval network instabilities with disruption tolerant networking", in Proc. IEEE MILCOM 2008, San Diego, Nov.2008, pp.1-7.
[8] W. Ivancic, L.Wood, P. Holliday, W.M. Eddy, D. Stewart, C. Jackson, J. Northam, "Experience with Delay-Tolerant Networking from Orbit", in Proc. ASMS 2008, Bologna, Aug. 2008, pp.173- 178.
[9] J. L. Torgerson, L. Clare, S. Y. (Cindy) Wang, J.Schoolcraft, "The Deep Impact Network Experiment Operations Center", in Proc. IEEE Aerospace Conference 2009, March 2009, pp.1-12.
[10] V. Paxson, M. Allman, "Computing TCP's Retransmission Timer", IETF RFC 2988, Nov. 2000.
[11] R. Braden, "Requirements for Internet Hosts - Communication Layers", IETF RFC 1122, Oct. 1989.
[12] TATPA website: http://tatpa.deis.unibo.it.
[13] C. Caini, P. Cornice, R. Firrincieli, D. Lacamera, and M. Livini, "Analysis of TCP and DTN retransmission algorithms in presence of channel disruptions", accepted for publication in Proc. IEEE SPACOMM'09, Colmar, France, Jul. 2009.
[14] J. Border, M. Kojo, J. Griner, G. Montenegro, Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations", IETF RFC 3135, June 2001.
[15] K. Fall, W. Hong, S. Madden, "Custody Transfer for Reliable Delivery in Delay Tolerant Networks", Technical Report IRB-TR-03-030, Intel Research, Berkeley, July 2003, pp1-6. Available at DTNRG web site.
[16] DTNRG web site: http://www.dtnrg.org/wiki
[17] DTN reference code (2.6.0): http://sourceforge.net/projects/dtn/.
[18] C. Caini, P. Cornice, R. Firrincieli, M. Livini, "DTNperf_2: a Performance Evaluation tool for Delay/Disruption Tolerant Networking" in Proc. of E-DTN 2009, St.-Petersburg, Russia, October 2009, pp. 1-5.
[19] DTNperf_2 source code : DTN2 Mercurial repository , http://dtn.sourceforge.net/hg/
[20] W. R. Stevens, "TCP/IP Illustrated, Volume 1", Addison-Wesley, Reading, MA, Nov. 1994.
[21] M. Allman, V. Paxon, W. Stevens, "TCP Congestion Control", IETF RFC 2581, Apr. 1999.
[22] P. Sarolahti, M. Kojo, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)", IETF RFC 4138, Aug. 2005.
[23] NistNet website: http://snad.ncsl.nist.gov/nistnet/.
[24] Netfilter/iptables wiki page: http://it.wikipedia.org/wiki/Netfilter.
[25] Iperf wiki page: http://en.wikipedia.org/wiki/Iperf.
[26] C. Caini, P. Cornice, R. Firrincieli, and D. Lacamera, "A DTN Approach to Satellite Communications", IEEE Journal on Selected Areas in Commun., vol. 26, no. 5, pp. 820-827, June 2008.
[27] Tcpdump website: http://www.tcpdump.org/.
[28] Wireshark website: http://www.wireshark.org/

Carlo Caini received the Dr. Ing. Degree (summa cum laude) in Electronic Engineering from the University of Bologna, Italy, in 1986. In 2001, he joined the Department of Electronics, Computer Science, and Systems (DEIS), University of Bologna, where he is presently an Associate Professor in Telecommunications.

His main scientific interests are in the field of satellite systems and wireless networks, with a special emphasis in the last years on the study, design and implementation of transport protocols and architectures for satellite and "challenged networks", including enhanced versions of TCP, Performance

Enhancing Proxies (PEPs) and Delay/Disruption Tolerant Networking (DTN). He is author of many international publications on these and other topics.

Prof. Caini is member of IEEE Communications Society and participates to several international research projects. He was corecipient of the Best Paper Award at SPACOMM'09 and IEEE IWSSC'09 conferences with works on DTN.

Rosario Firrincieli received his Master's and Ph.D. degrees in Telecommunications Engineering from the University of Bologna, Italy, in 2001 and 2006 respectively. He is a senior researcher at ARCES, University of Bologna, Italy. His present interests are focused on the study and the evaluation of enhanced transport protocols, PEP solutions, and DTN architectures over wireless network. Moreover, he is interested in congestion control algorithms for unicast and multicast protocols, traffic shaping, retransmission techniques and UL-FEC coding techniques.

During 2005-2008 he spent 10 months as Visiting Researcher at Department of Computer Science, the Henry Samueli School of Engineering and Applied Sciences, UCLA, US. He is co-author of many scientific publications on International Journals and Conferences.

Dr. Firrincieli was corecipient of the Best Paper Award at SPACOMM'09 and IEEE IWSSC'09 conferences with works on DTN.

Marco Livini received his Bachelor Degree in Computer Science Engineering from the University of Bologna with a thesis on Delay Tolerant Networks in 2007. Currently he is going to conclude the Master of Science degree course in Computer Science Engineering at the University of Bologna.

His scientific interests are focused on the study of infrastructures for information exchange in smart environments, Delay/Disruption Tolerant Networks and IP Multimedia Subsystem architectures. Moreover, he is interested in software solutions for mobile and embedded systems. During his Bachelor thesis work, he developed the DTNPerf_2 software for DTN performance evaluation, which currently he contributes to maintain and upgrade.

Marco Livini was corecipient of the Best Paper Award at SPACOMM'09 and IEEE IWSSC'09 conferences with works on DTN.