MPTCP
A Delay-Tolerant Transport Protocol for Ad hoc Networks

by

Giacomo Bernardi

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

May 2006

DECLARATION

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

_____

Giacomo Bernardi

May 31, 2006

PERMISSION TO LEND AND/OR COPY

I agree that Trinity College Library may lend or copy this dissertation upon request.

_____

Giacomo Bernardi

May 31, 2006

ACKNOWLEDGEMENTS

This thesis is dedicated to a nerd friend.

Tu, ora, diresti solo "asd".

SUMMARY

The data exchange between two nodes in infrastructure-based networks is often based on end-to-end links to implement communication models such as client-server. This approach relies on the availability of both end points at the same time and on a stable connection between the two nodes during the duration of the transfer.

In wireless - and especially ad hoc - networks these assumptions do not hold. Two mobile hosts that attempt to employ the client-server model are open to problems introduced by the mobility of either node (e.g. unavailability of one host during migration, unavailability of a route between the two, etc.)

The assumption of the "*Meeting Places*" approach is that mobile end points will always be able to connect to a "stable" (i.e. with a high connection reliability) node somewhere in the network. The key advantages of this model in situations of highly variable connectivity is that a few known and relatively stable nodes can be used to achieve point-to-point like communication requiring the two corresponding nodes to be available and able to communicate directly with each other. Moreover, the approach is scaleable through the introduction of more message points, and can accommodate a range of intermediary patterns to achieve various tradeoffs in performance.

We started our work by analysing the currently existing enhancements to the TCP protocol for ad-hoc networks, and then identified the features needed at the application level and at the underlying network layer. We then produced a “protocol draft” document that describes all the requirements and the low-level protocol specifications that can be used in MPTCP-based software development.

During the design process we kept in consideration the security outcomes of storing user’s data in the network and we modelled a specific end-to-end cryptographic system based on asymmetric algorithms and public key infrastructure. We also paid attention to the specific architectures of the devices that will be using MPTCP, considering in particular the case of embedded systems and sensor boards on which CPU processing power and available memory can be a constraint.

We supported our implementation with a sample C++ implementation of the main protocol operation and we evaluated it with an analysis of the results of lab simulations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

MPTCP is an acronym for "*Meeting Place Transport Control Protocol*".

In this first chapter we will introduce the concept of "*Meeting Place*" and the notions needed to understand the protocol operation. We will also provide some details about the specific knowledge domain and explain the reasons that leaded the main decisions in the protocol design process.

1.1 Background

The data exchange between two nodes in infrastructure-based networks is often based on point-to-point link to implement communication models such as client-server. This approach relies on the availability of both end points at the same time and on a stable connection between the two nodes during the duration of the transfer.

In wireless - and especially ad hoc - networks these assumption do not hold. Two mobile hosts that attempt to employ the client-server model are open to problems introduced by the mobility of either node (e.g. unavailability of one host during migration, unavailability of a route between the two, etc.)

Many solutions have already been proposed to address the consequences for transport protocols resulting from mobility. These solutions result in new protocols that often require support from the underlying network layer or impose the introduction of a special API for the programmers, others are enhancements of TCP protocol that try to keep a high level of interoperability with standard network nodes. While the latter approach can speed up the deployment of a new protocol over an existing network, the resulting performances are often not too far from TCP, and "workarounds" are needed in order to perform mobility operations such as hand-offs, connection migration and disconnected operation handling.

Our idea is a hybrid solution that takes advantages from both methods: MPTCP is a new transport protocol that features two different types of connection management, end-to-end security, variable header size and adaptive data compression while it adopts well-known and

extensively analyzed aspects of TCP such as three-ways handshakes, selective ACKs, flow and congestion control, etc.

1.2 Motivations

The typical motivation of a level-3 protocol is to provide a reliable or unreliable method to send data between two endpoints across a network. This concept assumes that both nodes are connected at the same time, directly via a single link or as part of a larger network. Our protocol focuses on situations in which two endpoints cannot be connected all the time, considering in particular, but not exclusively, scenarios where embedded systems are connected through "ad hoc" networks.

For simplicity, in the following paragraphs we will call the two endpoints "*Client*" and "*Server*", where the first is the "data producer" and the second the "consumer". Although this is a very common scenario, the protocol remains valid in pure peer-to-peer environments where any nodes can be defined both as Client and as Server.

The four motivations that drove the design of the MPTCP protocol were the followings:

- **Absence of contemporaneity** between the Client and the Server. This assumes that the two endpoints are rarely attached simultaneously to a network and therefore cannot establish a direct connection.

- **Resilience to disconnections.** Even if we assume that, for certain variable and unpredictable periods of time, the Client and the Server will be simultaneously connected to the network and therefore transfer data directly, this assumption can become false at any time. The two endpoints should be able to work without being affected by disconnections.

- **Minimize the processing power and transmitting time of the client.** In case the data throughput provided from the Server or the network is far below the capacity of the Client, this will be forced to keep its receiving circuits (i.e.: a radio transmitter) on for an unnecessary period of time, thus wasting power. In this situation, it is advisable to cache the data in the network and collect it later with a higher throughput.

- **Reduce the load of the client, in case of periodical repeated requests.** In case the Client needs to periodically poll a Server (e.g.: to get readings from a

sensor), it is a good idea to delegate to the network itself the task to place the requests to the Server and to store the data.

1.3 Key concepts and Philosophy

During the development of the protocol, we tried to focus on the following two key concepts:

- **"Intelligent Network, Simple Clients".** Our protocol aim to give an implicit intelligence to the network, by allowing the storage of information on the nodes themselves, without the need of level-5 proxies.

- **"Protect Against The Network".** Since we are delegating the data storage to the network itself, it is important to protect the data from malicious attacks. The only entity allowed to get the content is the Client, so a secure end-to-end encryption should be performed.

Every single design decision has been motivated by these two notions and by the common ideas included in the previous paragraph.

1.4 Dissertation Roadmap

This dissertation is composed by six chapters and provides a linear approach to the design, implementation and testing of a Transport Protocol for ad hoc networks.

Chapter 2 will discuss existing research that has influenced our work, and place our investigations in context. Comparing several protocols proposed to address the mobility problem also provides a motivation for our work and a guide to validate our initial assumptions.

Chapter 3 discusses in detail the operations of MPTCP and the packet format. We will also introduce and analyze the cryptography concepts and data compression algorithms used to develop our model.

Chapter 4 explores our attempts and results in implementing and testing a sample MPTCP stack for benchmarking purposes. This chapter includes a discussion about the

development environments that we took into consideration before finding a network software simulator suitable for our purposes.

Chapter 5 investigates the results of our work including a discussion of the seven main usage scenarios that we expect. We will conclude by summarizing the features offered by our approach.

Chapter 6 provides a final overview of the work. We will include some suggestions for possible future work in order to expand our protocol and include new features.

Finally, in the **Appendix** we enclose the document "*MPTCP Protocol Draft*" that is intended as a technical description of the protocol specifications. We refer to this text for further details about the internal structures of the protocol and the low-level functions and header format.

Chapter 2: Related Work

In this chapter, we will briefly discuss prior work in related areas of mobile ad hoc networking that can be used as an introduction to our idea. This discussion includes the definition of a taxonomy for mobile applications in order to establish the requirements of the definition of common application types. We will categorize and individually review influential transport protocols for ad hoc networks and we will conclude the chapter with a global feature comparison table.

2.1 Taxonomy for mobile applications

When we started the development of a new transport protocol, we decided to identify a number of sample user scenarios in order to better understand the application domain and to carefully drive the development process. In general, we can identify user applications according to two main "*usage patterns*" that present different requirements about transfer throughput, latency and reliability. It is important to note that this line of reasoning is not exclusive to wireless networks but can apply for every networked application. We distinguish between:

Interactive applications, in which the user or the system is continuously interacting with a remote system by sending commands and receiving answers. Typical examples for such applications are remote shells (e.g.: SSH, telnet), graphical user interfaces (e.g.: Remote Desktop, VNC) and interactive games. We include in this category also the software developed according to the Client/Server paradigm, for example an accountancy application with a remote UNIX server and a client installed on a local Windows box. The main network requirements of interactive applications are:

- **Low and constant RTT:** it is very important to keep low network latency at all times, in order to give a good responsiveness to the user.

- **Small packet size:** interactive applications tend to send small amounts of data with a high frequency rather than big packets once in a while. The size of the packets

passed from the upper layers is in general very small and data often exhibits a low "compressibility" ratio. It is also very important to support interactivity with the use of the Nagle's algorithm [1], in order to give a good feedback to the user.

- **Large number of interactions:** In many interactive user applications it is impossible to locally "cache" commands and send them at once. Instead, a single operation is composed by many transactions between the two endpoints, generating packets in both directions. Imagine for example a user entering a shell command via SSH: every single character need to be sent to the SSH server to be "echoed", while it would be possible to send the complete string at the end of the line.

Data transfer applications, such as FTP or HTTP transfers of big files or RSYNC of archives. In these cases, there is no need to ensure low latency nor for particularly responsive network links. The only factor considered by the user to evaluate performances is the data throughput, often directly measured by the end-user application, such as a Web browser. The main characteristics are:

- **Large amount of data**: we include in the "data transfer" category all the applications that send and receive big amounts of data.

- **Asymmetric transfers**: usually, a data transfer session implies sending data in only one direction over the network. Thus, one of the two endpoints can be clearly considered as a "sender" and the other as a "receiver".

- **Large packets:** data transfer applications tend to generate larger network packets and lead to reach the MTU of the network channel.

- **Few interactions:** On the contrary of interactive applications, a data transfer operation is usually composed by only two interactions: a relatively small request and a typically bigger reply.

To describe the protocol and to motivate the design decisions, it is useful to refer to these two categories of user applications.

2.2 Transport protocols for unreliable networks

A possible classification of transport layers for ad hoc networks is suggested in [2] as a tree graph:

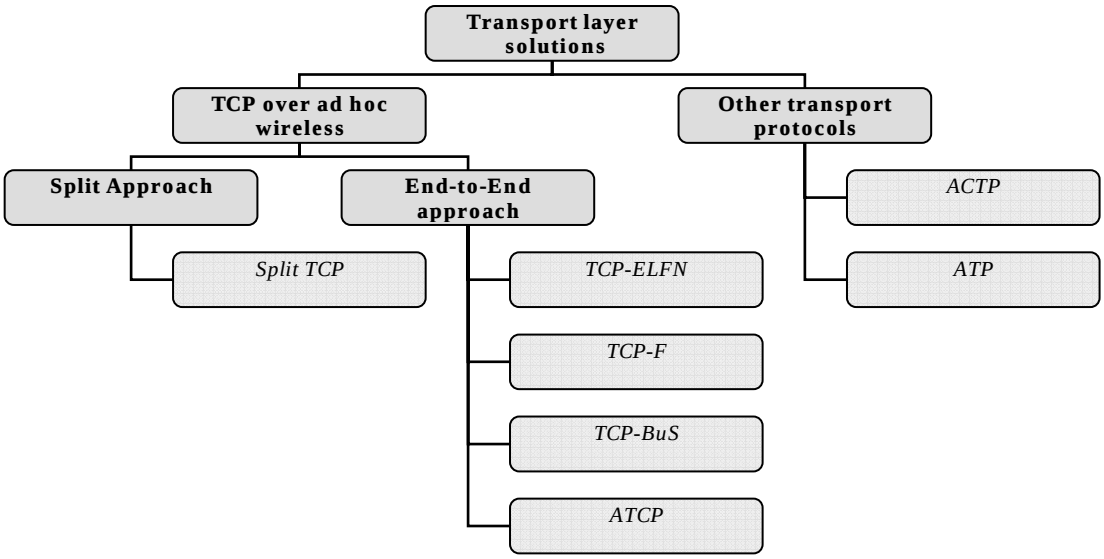


Figure 2-1: Tree classification of ad hoc transport protocols

The same publication reports also an analysis of the reasons for which TCP does not perform well in ad hoc networks, which we can resume in the following points:

- TCP over ad hoc networks leads to a misinterpretation of packet loss due to factors such as high bit error rate on the wireless channel, increased collisions because of "hidden terminals", presence of interference, location-dependent contention, unidirectional links, frequent path breaks due to mobility of nodes, etc.

- The topology of an ad hoc network changes continuously over time because of the mobility of the devices connected. This process leads to the creation of new links and the disruption of existing ones, thus requiring incessant changes in the routing tables. Often, the route reestablishment process takes a significant amount of time that can be greater than the RTO period of the TCP sender, which then will assume congestion in the network and will retransmit the lost packets. The consequences of

this process are a waste of bandwidth, power and a low throughput because of the reset of the slow start algorithm.

- A general idea is that in ad hoc wireless sensor networks “the shortest path-length, the better”. This is mainly motivated by two facts: path break probability and TCP throughput, while the first statistically increases with path length, the second degrades rapidly with an increase in the number of nodes.

- In TCP, the congestion window is a measure of the transmission rate that can be handled by the receiver. In ad hoc wireless networks, the congestion control is invoked every time the network get partitioned or when a path break occurs, this reduces the congestion window and increases the RTO period. In case of frequent path breaks, the congestion window may not reflect the maximum transmission rate acceptable to the network.

- Because of the asymmetric nature of radio links and of environmental effects on propagation, it is possible that a packet is delivered successfully to a node while the acknowledgement is not. This can lead to a wrong behaviour of the congestion control algorithm and to several unnecessary retransmissions.

- Some routing protocols use multiple paths between a source-destination pair leading, as a consequence, to a significant number of out-of-order packets, duplicate acknowledgements and additional invocations of congestion control.

- Some CSMA/CA media access protocols for wireless links show short-term unfairness: a node that has captured the channel has a higher probability of capturing the channel again. This can cause the delivery in succession of a number of ACK packets, thus creating bursts in the traffic patterns.

We will now briefly review some TCP enhancements that can lead to a better management of network resources in ad hoc networks together with other transport protocol especially suited to handle disconnected operations and poor link conditions. Some of the techniques that we found require the adoption of specific routing protocols in order to give some kind of feedback to upper layers. For example, consider:

- **Feedback-based TCP** (TCP-F) [3]. It requires the support of a reliable link layer and a routing protocol that can provide feedback to the TCP sender about path breaks. The main aim is to reduce the throughput degradation due to links breaks and, in order to achieve it, every intermediate node on the link that discovers a disruption became a “failure point” (FP) and sends back a “route failure notification”

(RFN). The protocol gets sender's address from the TCP headers and it is able to identify link problems by considering the ACK packet forwarded and received. Feedback-based tries to minimize the problems arising out of frequent path breaks in ad hoc wireless networks.

- **TCP with explicit link failure notification** (TCP-ELFN) [4]. For some aspects it is similar to TCP-F: a node that detects a link failure sends back a message (e.g.: ICMP DUR) or a RouteError message. After receiving the ELFN packet, the sender disables the retransmission timers and enters a standby state, then it tries to discover when the link is up again by periodically send a probe packet. The main advantage of TCP-ELFN over TCP-F is the independency from the routing protocol.

- **TCP with buffering capability and sequence information** (TCP-BuS) [5] requires the presence of Associativity-based Routing (ABR). After detection of path break, a Pivot Node (PN) originates an explicit route disconnection notification (ERDN) message that stops the sender from any transmission. Packets in transit are buffered and the PN tries to find a new route to the receiver. When the route reconfiguration or repair process is completed, an Explicit Route Successful Notification (ERSN) message is sent back to the source that resumes the data transmission.

Another approach to get better performances on ad hoc network is designing a new transport protocol from scratch without the problems of keeping a full compatibility with TCP. The following protocols are examples of such approach:

- **Ad Hoc TCP** (ATCP) [6] uses a network layer feedback system to inform the TCP sender about the status of the network over which the packets are sent. Based on this information, the node changes its states between "persist", "congestion control" and "retransmit". ATCP is implemented as a thin layer residing between the IP and TCP protocols and it makes use of the Explicit Congestion Notification (ECN) to update the currently active state. The main advantages of this approach are the compatibility with the traditional TCP and that ATCP maintains the traditional end-to-end semantics while introducing a new state machine to describe node's behaviour. The protocol requires some changes to the operating system networks stack in order to be deployed and also requires functions to discover route changes and partitions from the network layer protocol.

- **Split TCP** (S-TCP) [7] focuses on the problems due to the short-term unfairness of wireless MAC protocols and the 802.11 "channel capture" effect. The proposed

solution is splitting the transport layer according to two distinct goals: congestion control and end-to-end reliability. The first is solved locally to better exploit the knowledge about the local wireless link status, while to increase the probability of a successful connection S-TCP splits a long TCP connection into a chain of short concatenated TCP connections with a number of intermediate nodes. All these internal nodes act as a proxy by receiving a TCP packet, reading its contents, storing it in its local buffer and sending an ACK to the previous proxy or the source. By doing this, the protocol leave the responsibility of packet delivery to the proxy nodes and leads to improved throughput, fairness and lessened impact of mobility. The main drawbacks are the need of a modified networking stack and the performance dependency on the availability and available throughput of the proxy nodes.

- **Application controlled transport protocol** (ACTP) [8] should be considered as a light-weight transport protocol instead of an extension of TCP and offers the possibility to assign different priorities to packets to be delivered. ACTP has the advantage that the application is free to choose the required reliability level and that it has a very large scalability for large networks because of its light-weight design. The main disadvantages are the lack of a congestion control algorithm and that ACTP is not compatible with TCP.

- **Ad hoc transport protocol** (ATP) [9] uses a timer-based transmission to decide the transmission rate accordingly to the network congestion. To determine the congestion status, ATP uses a weighted average of the queuing delay and the contention delay of the packets at every intermediate node. SACKS packets are used to force the retransmission of lost packets and to ensure reliability of packet delivery. When a new node enters in the network, ATP gets information from the lower level of the network stack to estimate the transmission rate, to perform congestion control and avoidance and to detect path breaks. ATP does not offer any interoperability with TCP but offers improved performances on networks where packet prioritization is crucial.

- **Licklider Transmission Protocol** (LTP) is a transport protocol “designed to provide retransmission-based reliability over links characterized by extremely long message round-trip times and/or frequent interruptions in connectivity” [10]. The most important intended usage for LTP are interplanetary Internet links but its properties make it useful also in ad hoc networks to handle disconnected operations. It is stateful, it does not include any negotiation or handshake procedure and it uses selective-acknowledgment reception reports.

- **Indirect TCP** (I-TCP) [11] is a protocol proposed to face disconnection issues when one of the links in a TCP connection is a wireless link. I-TCP partitions a connection in two segments: a regular TCP connection between the fixed host and a Mobile Support Router (MSR) and a dynamic segment between the MSR and the mobile host. Compared with traditional TCP, this protocol introduce the "Mobility Support Routers" (MSR) that performs some transport-layer functions, which in a normal version of the protocol would be carried out by the one of the end-points. To manage the hand-off, I-TCP uses *socket migration* techniques where two new sockets with the same characteristics of the previous ones are created on the new MSR.

- **TCP Snoop** is a link-aware transport protocol [12] developed by UC Berkeley for wireless last-hop networks to address TCP problems due to the presence of wireless links. The protocol uses a "snoop agent" at the radio base station to detect and locally retransmit lost segment, without any intervention by the sender. Duplicate ACKs are suppressed to avoid unnecessary invocations of the congestion control procedures in the sender and, in case data is flowing from the mobile node to a fixed host in the backbone wired network, a mechanism called Explicit Loss Notification (ELN) is used to decouple the retransmission from the congestion control.

2.3 A global feature comparison

In the following table we summarize the main characteristics of the protocols listed including our MPTCP. This table has been derived from a table presented by Murthy et al. [2] and extended with the result of our research.

	1 - **TCP-F**	2 - **TCP-ELFN**
References	[3]	[4]
Packet loss due to BER or collision	Same as TCP	Same as TCP
Path breaks	RFN is sent to the TCP sender and state changes to snooze	ELFN is sent to the TCP sender and state changes to standby
Out-of-order packets	Same as TCP	Same as TCP
Congestion	Same as TCP	Same as TCP
Congestion window after path reestablishment	Same as before the path break	Same as before the path break
Explicit path break notification	Yes	Yes
Explicit path reestablishment notification	Yes	No
Dependency on routing protocol	Yes	Yes
End-to-end semantics	Yes	Yes
Packets buffered at intermediate nodes	No	No
Compatible with TCP	No, it has an additional state in the finite state machine.	No

Table 2-1: Transport protocol feature comparison

	3 - **TCP-BuS**	4 - **ATCP**
References	[5]	[6]
Packet loss due to BER or collision	Same as TCP	Retransmits the lost packets without invoking congestion control
Path breaks	ERDN is sent to the TCP sender, state changes to snooze, ICMP DUR is sent to the TCP sender, and ATCP puts TCP into persist state	Same as TCP
Out-of-order packets	Out-of-order packets reached after a path recovery are handled	ATCP reorders packets and hence TCP avoids sending duplicates
Congestion	Explicit message such as ICMP source quench are used.	ECN is used to notify TCP sender. Congestion control is same as TCP
Congestion window after path reestablishment	Same as before the path break	Recomputed for new route
Explicit path break notification	Yes	Yes
Explicit path reestablishment notification	Yes	No
Dependency on routing protocol	Yes	Yes
End-to-end semantics	Yes	Yes
Packets buffered at intermediate nodes	Yes	No
Compatible with TCP	No	Yes

	5 – **Split-TCP**	6 – **ACTP**
References	[7]	[8]
Packet loss due to BER or collision	Same as TCP	Recalculates the priority status and then retransmit
Path breaks	Same as TCP	Same as TCP
Out-of-order packets	Same as TCP	Affect the priority status of the following packets
Congestion	Since connection is split, the congestion control is handled within a zone by proxy nodes	No congestion control mechanism
Congestion window after path reestablishment	Proxy nodes maintain congestion window and handle congestion	No congestion control mechanism
Explicit path break notification	No	No
Explicit path reestablishment notification	No	No
Dependency on routing protocol	No	No
End-to-end semantics	No	Yes
Packets buffered at intermediate nodes	Yes	No
Compatible with TCP	No	No

	7 - **ATP**	8 – **LTP**
References	[9]	[10]
Packet loss due to BER or collision	Multiplicative decrease of the transmission rate triggered.	Uses selective-acknowledgment to retrieve only missing packets.
Path breaks	An ELFN packet is sent to the ATP sender.	No explicit notification.
Out-of-order packets	Same as packet loss.	Accepted, stored and reordered.
Congestion	Modelled with 3 states that depend on the current transmission rate: increase, decrease and maintain.	No congestion control mechanism
Congestion window after path reestablishment	It is recomputed from zero.	No congestion control mechanism
Explicit path break notification	Yes	No
Explicit path reestablishment notification	No	No
Dependency on routing protocol	No	No
End-to-end semantics	Yes	Yes
Packets buffered at intermediate nodes	No	Yes
Compatible with TCP	No	No

	9 – **I-TCP**	10 – **TCP-SNOOP**
References	[11]	[12]
Packet loss due to BER or collision	It influences the congestion control only for the “wireless side” of the connection.	The protocol uses SACKs over wireless links and performs local recovery.
Path breaks	Same as TCP	No difference between packets lost
Out-of-order packets	Same as TCP	Same as TCP
Congestion	Is modelled separately for wireless links and fixed networks.	Intermediate nodes monitor every packet that passes through the path and asks the sender to reduce the transmission rate when packet losses are detected only at a wired link.
Congestion window after path reestablishment	It is recomputed after every hand-off	No windows are used, instead it forces rate-based transmission
Explicit path break notification	Yes	No
Explicit path reestablishment notification	Yes	No
Dependency on routing protocol	No	No
End-to-end semantics	Yes	Yes
Packets buffered at intermediate nodes	Yes	Yes
Compatible with TCP	Yes	Yes

Chapter 3: Design

After having analyzed some of the existing TCP enhancements for wireless networks in the last chapter, this section describes the idea behind our approach and the details of the protocol proposed. As a complement to this paragraph, we included the "*MPTCP Protocol Specification*" document in the *Appendix* that contains further details about the algorithm internals and the low-level packet formats.

3.1 Connection types

In a few word, our idea consists of *splitting the end-to-end communication in two segments by inserting a "Meeting Place" node*. By dividing the connection into two parts at the transport level, we can break the common assumption that the two endpoints need to be connected at the same time and we can instead delegate to the meeting place the work of caching the data while one of the two endpoints is disconnected or out-of-range.

This approach affects the network operations in two ways:

- First, in case we are allowed to assume that the nodes will always be able to connect to a "stable" node somewhere in the network, we are implicitly stating that two arbitrary nodes can be connected via a meeting place.

- Second, the meeting places are not merely proxy or caching nodes but they include a basic "intelligence", even if the previous assumption does not hold we could program a meeting place to do some operation with the server while the client will remain disconnected.

The key advantages of this model on a highly variable network is that a few known and relatively stable network nodes can be used to achieve point-to-point communication without the otherwise necessarily persistent quality of network.

We started our design by defining four different types of connections, as described in Table 3-1.

SDR *Single Direct Request*	**SRR** *Single Reverse Request*
MDR *Multiple Direct Requests*	**MRR** *Multiple Reverse Requests*

Table 3-1: Connection types in MPTCP

The first letter in the acronyms simply specifies the number of requests that are to be carried out: a single request consists in only one operation from the Client to the Server and the relative answer. Instead, with a multiple request operation a Client could ask to a Meeting Place to collect data from a Server many times at specified intervals.

The second letter defines the type of connection ongoing between the Client (C), the Meeting Place (MP) and the Server (S), as described in the following paragraphs.

3.1.1 Direct Requests

The client delegates the MP to get the data (once or more times) from a remote server and store it locally. The dialogue between the three actors assumes the following form:

Step 1.

C to *MP*: *"Please send my payload to S"*

MP to *C*: *"Ok"*

Step 2.

MP to *S*: *"This is my payload"*

S to *MP*: *"This is my data"*

Step 3.

C to *MP*: *"Please send me the data waiting for me"*

MP to *C*: *"This is your data"*

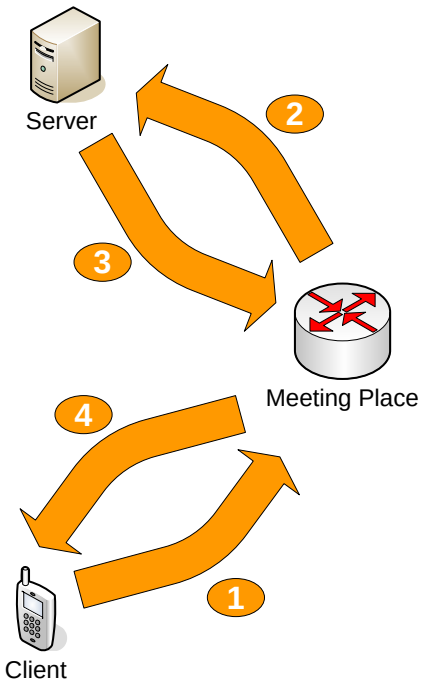


Figure 3-1: Direct Requests

3.1.2 Reverse Requests

The client directly connects to the Server and asks to send the answer to its request to a Meeting Place. Later on, the client will get the data from the MP.

Step 1.

C to *S*: *"Please send the answer to this request to MP"*

S to *C*: *"Ok"*

Step 2.

S to *MP*: *"This is data for C"*

MP to *C*: *"Ok"*

Step 3.

C to *MP*: *"Please send me the data waiting for me"*

MP to *C*: *"This is your data"*

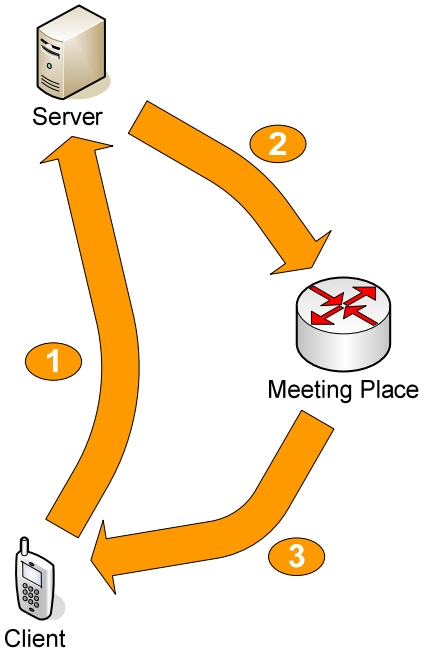


Figure 3-2: Reverse requests

3.2 MPTCP as a state machine

Considering the creation and closure of the connection and the transactions between states, MPTCP's behaviour is identical to TCP: they both share the same model for all the connection-oriented operations such as: opening and closing a connection, acknowledge the other end-point about sent data, etc.

Thus, MPTCP uses a *three-way handshake* to establish a connection, as follows:

- The server performs a passive open to be ready to accept incoming connections. This operation is usually done by creating a new socket, binding it to a network address and performing a "listen" instruction.
- The client, using a connect function, performs an active open by sending a SYN packet which includes an initial pseudo-random sequence number.
- When the server receives the client's SYN segment, it answers with an acknowledge and its own SYN.

- To complete the procedure, the client sends an ACK for the server's SYN segment notify the process that the socket is ready to send data.

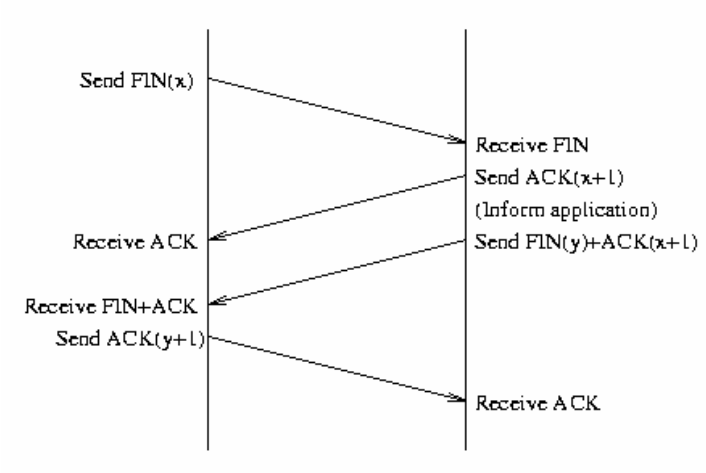


Figure 3-3: TCP handshaking for "OPEN"

While the exchange of three packets is needed to open a connection, the closing procedure requires four, as follows:

- One of the two endpoints calls the CLOSE interface (see paragraph 3.3), starting a process called active close. This involves sending a FIN segment that indicates the end of the data transmission on the current connection.

- When the other endpoint receives the FIN, it executes a passive close and acknowledges the FIN packet with an ACK. This operation is also reported to the process that opened the socket as an "end of file" after any possible packet in the sending queue has been sent.

- When the process receives this EOF message, it will call the close function to its own socket causing the transmission of another FIN segment.

- The other endpoint will receive the FIN packet and will answer with an ACK.

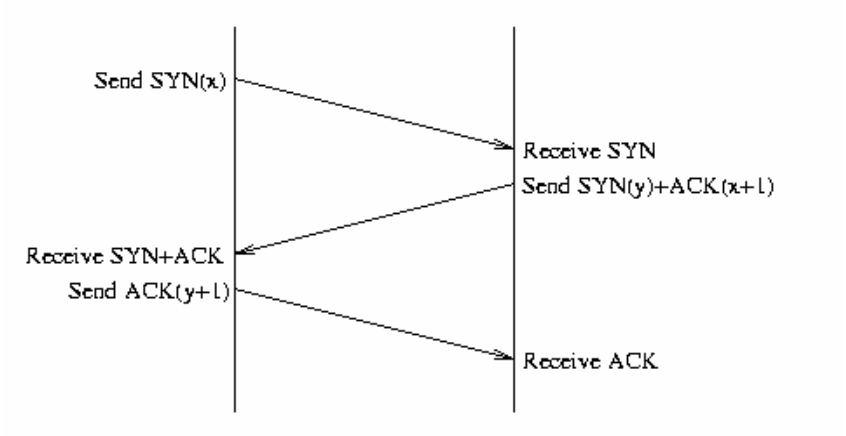


Figure 3-4: TCP handshaking for "CLOSE"

A MPTCP node has another option to abort a connection a node: it can send a packet with the RST bit set, the receiver will immediately abort the connection and warns the application program.

The MPTCP operation can be represented by using the state machine of TCP. The establishment, maintenance and closing of a TCP connection requires TCP to remember state information, timers, and variables associated with each connection. A TCP connection state changes from a state to another in response to events. The events may be user commands such as SEND, RECEIVE, STATUS, expiring timers or received packets such as RST and SYN. Ten different states are defined:

- **CLOSED:** For reference only
- **LISTEN:** represents waiting for a connection request from any remote TCP
- **SYN-SENT:** represents waiting for a matching connection request after having sent a connection request,
- **SYN-RECEIVED:** represents waiting for a confirming connection request ACK after having both received and sent a connection request
- **ESTABLISHED:** represents an open connection, data received can be delivered to the upper layer protocol (the normal state for the data transfer phase of the connection)
- **FINWAIT1:** represents waiting for a connection termination request from the remote TCP or an ACK of the connection termination request previously sent

- **FINWAIT2:** represents waiting for a connection termination request from the remote TCP

- **CLOSEWAIT:** represents waiting for a connection termination request ACK from the remote TCP

- **LASTACK:** represents waiting for an ACK of the connection termination request previously sent to the remote TCP (which includes an ACK of its connection termination request)

- **TIMEWAIT:** represents waiting for enough time to pass to be sure the remote TCP received the ACK of its connection termination request.

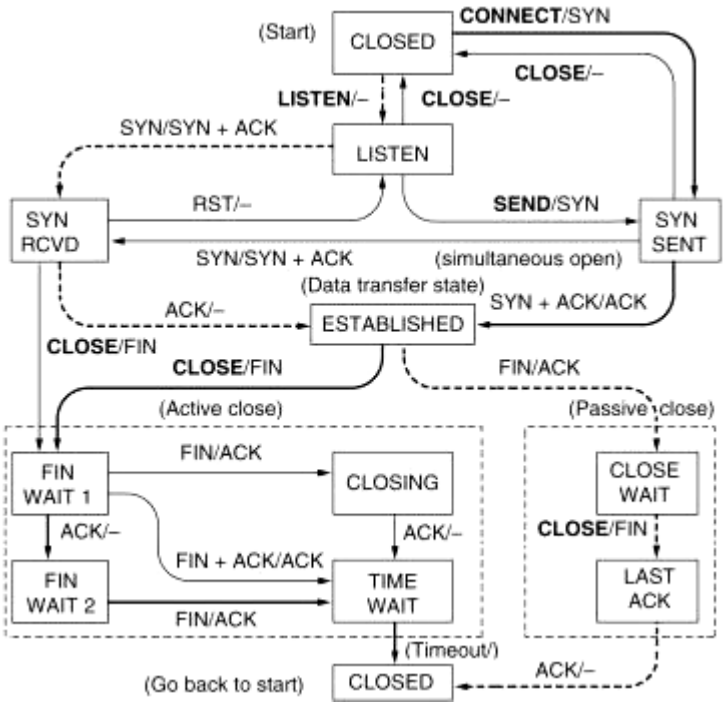


Figure 3-5: Finite state machine of TCP and MPTCP

3.3 Header structure and interfaces

As we already explained in Chapter 2, the size and number of fields in the MPTCP protocol header is not constant but can vary to adapt the protocol according to the features needed in each situation while keeping the overhead low by avoid unneeded fields. In order to achieve this result, we classified all the allowed MPTCP options and parameters in options groups that we will call "*subheaders*" in the remainder of the document.

The complete MPTCP header take the form of a "linked list" where subheader is identified by the value in the "next header" field of the previous one, using a technique similar to that implemented in the IPv6 protocol header [13]. At the moment, seven types of subheaders are defined, as explained in Table 3-2.

NextHeader Value:	Name:	Purpose:
n/a	`head_general`	Information about the content-id, the source and destination ports for layer-3 multiplexing, flow control information, session establishment and reset information and congestion control data.
0x1	`head_thirdaddress_v6`	For direct requests, this header specifies the Server IPv6 address to store the data on. For reverse requests, it specifies the Meeting Place IPv6 address. It is called "third address" because it completes the set containing the Source and Destination addresses.
0x2	`head_mpaddress_v4`	Same as `head_thirdaddress_v6`, but it uses IPv4 addresses.
0x3	`head_ttl`	Can be included by the client to specify how long the data should be stored on the Meeting Place.
0x4	`head_crypt`	Specifies the details for the data encryption, such as the encryption algorithm and the X.509 certificate that contains the node's public key.
0x5	`head_multiple`	Can be used in Multiple requests to specify the number of requests to be done and the time distance between them.
0x6	`head_answer`	Used by the MP to communicate Errors, Warnings and informative messages to the C. Eight "error levels" are defined to specify the seriousness of a problem.

Table 3-2: MPTCP Subheaders

We also impose the following rules for the use of the subheaders:

- All the subheaders are optional, except `head_general` and one between `head_thirdaddress_v4` or `head_thirdaddress_v6` that must always be present. The size of each subheader is fixed and specified in this document.

- The order of the subheaders is not relevant, except `head_general` that must always appear as first.

- All the subheaders except `head_general` can be present only in the first packet of a session.

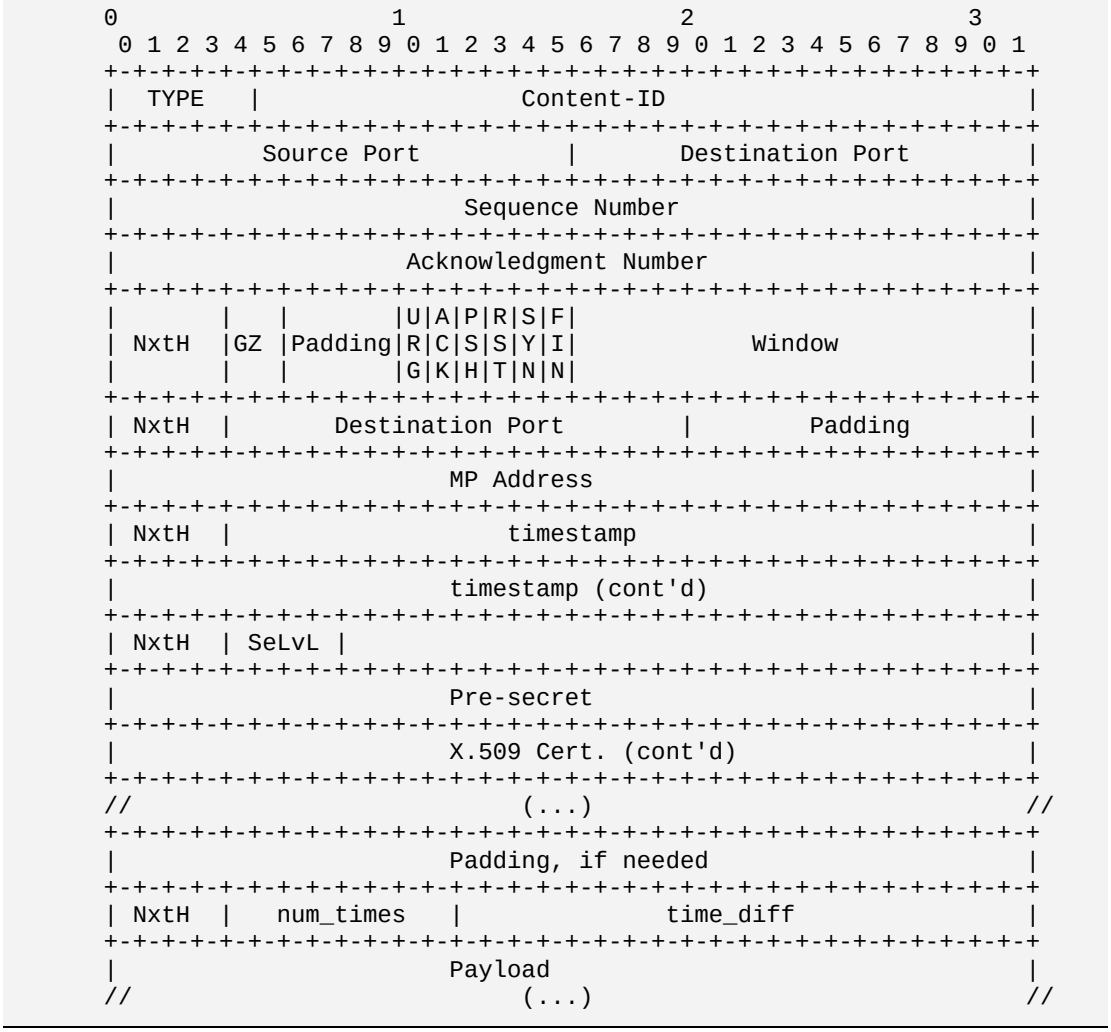
- Each of the subheader types can appear only once in the list except `head_mpaddress_v4` and `head_mpaddress_v6`, that can appear an arbitrary number of times to indicate that this session will make simultaneous use of more than one meeting place.

- The `NextHeader` value of the last subheader must be zero to indicate that the payload is following.

- The subheader `head_answer` may be used by a Meeting Place to communicate warning or errors to the Client.

All the details about the low-level format of each subheader are described in the document "*MPTCP Protocol Specification*" that we included in Appendix.

An example is represented by the following MPTCP packet, which contains five subheaders liked together by the values in the `NxtH` fields. The payload is at the end of the subheaders list and the last subheader has a zero value for `NxtH`.



3.4 Flow and congestion control

The flow control mechanisms allow the sender to determine the amount of data that it is allowed to transmit before receiving an acknowledgement from the receiver. MPTCP uses the same procedure of TCP based on a “sliding window” mechanism that contains the sequence numbers of the packet that the sender is allowed to transmit before waiting for an ACK. The window gradually slides open when wider ACKs are successfully returned. In case the receiver’s buffer is becoming full, the window size can be adapted by sending a small window size advertisement to the sender, which in turn will reduce its window to avoid receiver buffer overflow. As an extreme case, a receiver can advertise a window size of zero, causing the sender to stop transmission.

As an example we can suppose that, after the initial handshake, the receiver has a window size of 6 segments:

- The sender transmits 6 segments to the destination:

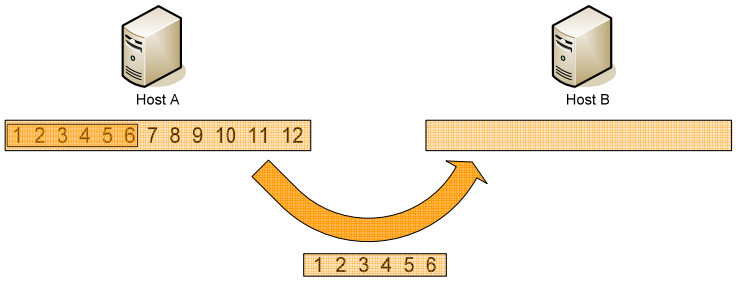


Figure 3-6: Congestion Control (step 1)

- If the receiver gets only segments 1 and 2, it sends an ACK indicating that it is waiting for segment number 3. Because of this, the window moves by two segments so that segments 7 and 8 can be sent. If no ACK is received for the previously sent segments 3 to 6, the retransmission timer gets zero. When this happens, the segments are sent again doubling the timeout time.

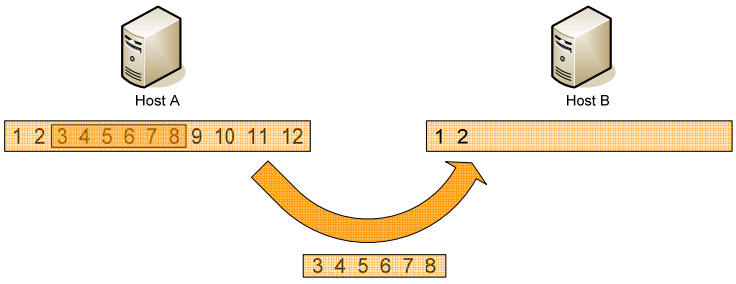


Figure 3-7: Congestion Control (step 2)

- The receiver sends an ACK indicating segment number 8, meaning that it successfully received packets till 7. The window moves after segment 7 so that it is now possible to send packets 8 to 12.

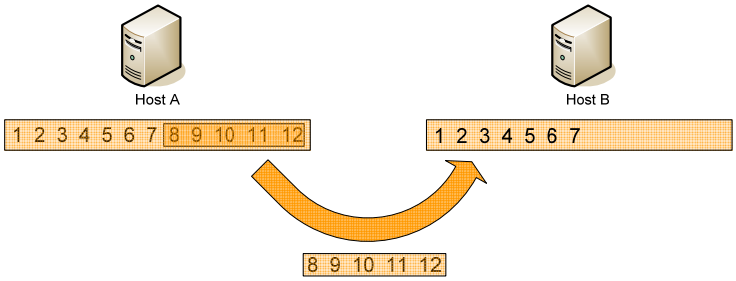


Figure 3-8: Congestion Control (step 3)

Using flow control techniques based on a "sliding window" allow us to get a performance improvement because the receiver can send a single ACK for more segments at once, thus reducing network traffic. The opposite situation, in which the window is too big, could lead to a higher packet drop and reduce the performances.

Another problem that every reliable transport protocol should address is the "congestion control": the possibility that the load applied to a network is higher than its possibilities. When congestion happens, the latency increases and the routers start to queue the packets until they cannot be routed and the only possibility is to drop them. Also, because both TCP and MPTCP include timeout mechanisms that imply retransmission of packets, the situation can get worst: if the traffic volume increases, the latency will increase as well, which will cause in turn an increase of the traffic until this "down spiral" makes the link overloaded.

The first step in congestion handling is to notice an abnormal situation: a network node can deduce the presence of a congestion by observing latency and packet loss on the link. When such a situation is detected, a useful technique to use is "Multiplicative Decrease". MPTCP maintains a second limit, beside the window size we described above, called "congestion window limit", and it uses as size for the sending window the minimum value between these two. Every time a connection is established, the sender initialize the "congestion window" to the size of the biggest segment used on the connection, then send the first segment. If it receives an ACK before the timeout, the node doubles the congestion window value and sends out a packet with this size. If this packet gets delivered, it doubles the size again and again. The congestion window keeps growing until a timeout is triggered or the receiver's window is reach. This well-known algorithm is called Slow Start and it is already deployed in all the existing TCP implementations.

Beside the congestion and receiver's windows, another threshold is used with the typical value of 64KB. In case of a timeout, this value is set to the half of the current congestion window. The Slow Start algorithm is used to determine the amount of data that can be

handled by the network and increased exponentially until that threshold is reached. Starting from that moment, the packets successfully sent makes the window increase linearly.

3.5 Cryptography and Authentication mechanisms

In the design of our transport level protocol we considered security as an important aspect since the first steps. Since we are explicitly storing data in the network, we want to be sure that no one else except the server and the client could access to it. We also addressed the problem of "node authentication" and we created a system in which the nodes can mutually authenticate themselves, leading the user to a trusted network environment.

Our system also guarantees "perfect forward security" (PFS): imagine a scenario in which an attacker has overheard all the traffic on the network and obtained both public and private of all the nodes. Even in this worst case scenario, it is impossible for everyone except the Client and the Server to decrypt the data stored on the meeting place. This is achieved by encrypting data with "session keys" that are different for every transaction and unpredictable by external nodes.

In deploying a security solution we wanted also to consider the domain in which MPTCP would be used more often: sensor networks, nodes composed by small embedded systems, etc. In these cases CPU power and memory can be very constrained so we decided for a solution in which:

- The transport protocol is not dependent from the adoption of a specific cryptographic algorithm or key length. By doing this we can enable to MPTCP hardware platforms on which the use of a specific cipersuite is not feasible and even expand the protocol by introducing new cryptographic algorithms in the future.

- It is possible for the user to define the level of security needed for any specific use of MPTCP. This is done by setting a "security level" variable called `SeLvL`, which specifies how strict the security checks should be. A `SeLvL` of zero means that cryptography is completely disabled. Higher levels provide complete mutual authentication between nodes.

We decided to adopt a mechanism based on asymmetric cryptography and certificates in the X.509 standard [14] in order to delegate to an external Certification Authority the task of declaring a host as "trusted".

To better understand the following paragraph, we remind the typical format of a X.509 certificate. Consider for example the following X.509 certificate, decoded using the OpenSSL command line utility on Linux [15]:

```
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 7829 (0x1e95)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=IE, ST=Dublin, L=Cape Town, O=Example Corporation,
               OU=Certification Services Division,
               CN=Example CA/Email=server-certs@example.com
       Validity
           Not Before: Jul  9 12:00:00 1998 GMT
           Not After : Jul  9 12:00:00 2008 GMT
       Subject: C=IE, ST=Ireland, L=Dublin, O=Giacomo Bernardi,
                OU=TCD, CN=client1.mptcp.tcd.ie/Email=bernarg@cs.tcd.ie
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (1024 bit)
               Modulus (1024 bit):
                   00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                   33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                   66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                   70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                   16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                   c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                   8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                   d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                   e8:35:1c:9e:27:52:7e:41:8f
               Exponent: 65537 (0x10001)
   Signature Algorithm: md5WithRSAEncryption
       93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
       92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
       ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
       d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
       0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
       5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
       8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
       68:9f
```

It was issued by "*Example Corporation*" as stated in its Issuer field, and it regards the object stated in the "*Subject*" field, in this case the host "*client1.mptcp.tcd.ie*". Next comes an RSA public key followed by the signature, computed by taking an MD5 hash of the first part of the certificate and encrypting it with Thawte's RSA private key.

X.509 is now a very popular standard because it is used for the SSL encryption of HTTP traffic and, because of this, many tools are already available to manage certificates and libraries for all the main programming languages are freely available (consider for example OpenSSL and GnuTLS [16]).

Please note that the MPTCP specifications do not describe how certificates are signed, distributed and revoked, so an external PKI is required. We made this design choice in order

to keep the transport protocol simple, while many certificate distribution systems are already existing and implemented.

In the following paragraph we will use the abbreviations of Table 3-3, most of which are typical of system based of X.509.

Component:	Description:	Needed by:	Secret?
$PRIV_c$	Private key of the Client.	Client only.	Yes
$PRIV_{mp}$	Private key of the Meeting Place.	Meeting Place only.	Yes
$PRIV_s$	Private key of the Server.	Server only.	Yes
$PRIV_{ca}$	Private key of the Certification Authority.	Server only.	Yes
PUB_c (from CRT_c)	Public key of the Client, included in its certificate.	Meeting Place and, for higher SeLvL, Server.	No
PUB_{mp} (from CRT_{mp})	Public key of the Meeting Place, included in its certificate.	Server and, for higher SeLvL, Client.	No
PUB_s (from CRT_s)	Public key of the Server, included in its certificate.	Client and, for higher SeLvL, Meeting Place.	No
PUB_{ca} (from CRT_{ca})	Public key of the Certification Authority, included in its certificate.	All the nodes.	No
CRL (optional)	Certificate Revocation List.	All the nodes.	No

Table 3-3: Cryptography components abbreviations

In our system, every node will be in possession of the following components:

- A pair of private and public key, generated with a chosen algorithm and key length. We will call them {$PRIV_c$, PUB_c} for the Client, {$PRIV_{mp}$, PUB_{mp}}, for the MP and {$PRIV_s$, PUB_s} for the Server.

- A X.509 certificate by which a commonly trusted Certification Authority (CA) signs the public key of the node. We will call them CRT_c for the Client, CRT_{mp} for the MP and CRT_s for the Server.

- The self-signed certificate of the CA, CRT_{ca}.

We can also introduce the use of Certificate Revocation List (CRL), a standard format to blacklist "bad" certificates (e.g. stolen, old, etc) that must not be accepted by the authenticating peers. Such a list would be distributed periodically or the nodes can use an online checking protocol (such as the Online Certificate Status Protocol, OCSP [17]) to control the certificate validity by querying a central server.

It is important to note that in the MPTCP security architecture, certificates are used for two aims at the same time: to authenticate nodes by checking the signature in the CRT against a trusted CA and to exchange cryptographic keys that can be used to generate a shared session key. In MPTCP no other authentication system is available, we thus decided to exclude common techniques such as username and password pairs sent in clear text in order to maintain a high level of security even on insecure networks links and ad hoc networks.

In the following two subparagraphs we describe the steps performed to ensure security on Direct and Reverse requests.

3.5.1 Direct Requests

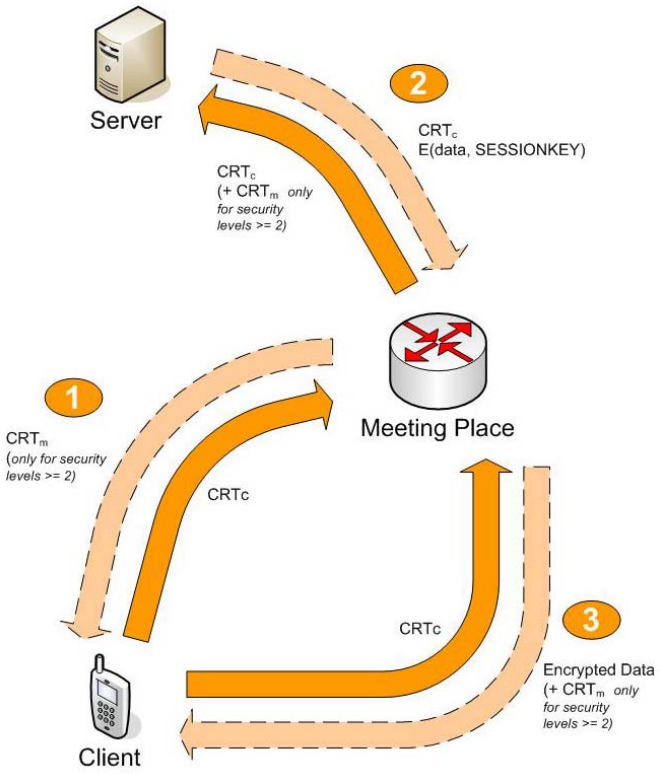


Figure 3-9: Certificate exchanges on Direct Requests

1. The Client sends a SDR or MDR packet to the MP attaching CRT_c by using an `mptcp_crypt` header as described in Chapter 3.3. Optionally for `SeLvL` higher than 2, MP sends back CRT_{mp} to C to ensure mutual authentication.

2. MP sends a SDR or MDR request to the Server attaching CRT_c. For `SeLvL` higher than 3, MP attaches also CRT_{mp} and receives CRT_s from the Server.

3. The Server sends to MP the data encrypted with the public key of the Client.

4. The Client requests data from the MP sending CRT_c and, for `SeLvL` higher than 2, requesting CRT_{mp}.

We have thus defined the following security levels:

Security Level:	Description:
0x0	No authentication at all. Cryptography disabled.
0x1	Client sends CRTc to the MP.
0x2	As above, plus MP sends CRT_{mp} to the Client.
0x3	As above, plus MP sends CRT_{mp} to the Server and receives CRTs from the Server.

Table 3-4: Security Levels in Direct Requests

3.5.2 Reverse Requests

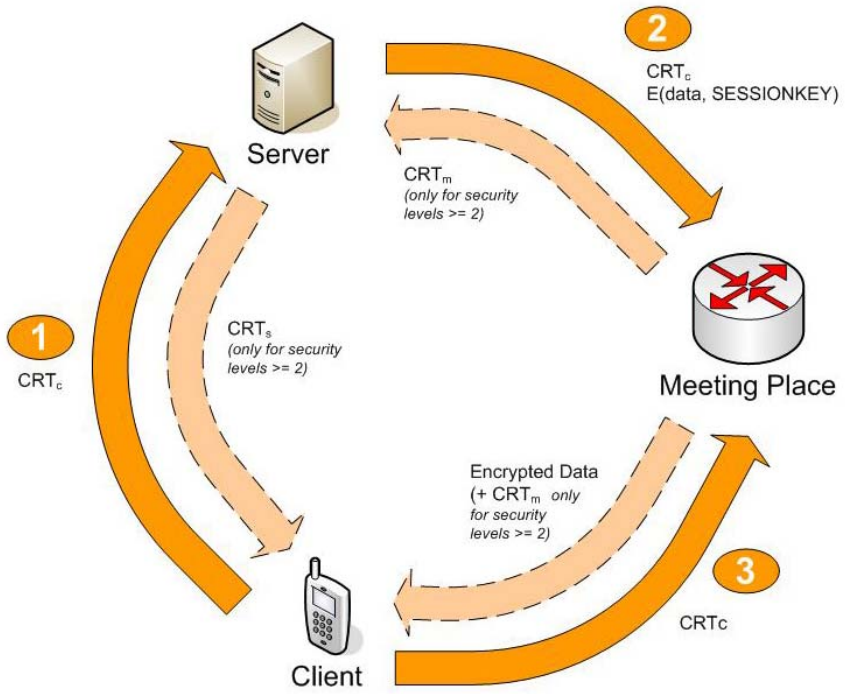


Figure 3-10: Certificate exchanges on Reverse Requests

1. The Client sends a SRR or MRR packet to the Server attaching CRT_c by using an `mptcp_crypt` header. Client and Server exchange then the pre-secrets keys in order to create a session key. At this point, the Client and the Server are mutually identified and they share the knowledge of a secret key.

2. The Server sends to the MP: its own certificate CRT_s and data encrypted using the shared key (that MP ignores). Optionally, for `SeLvL` higher than 3, MP sends back CRT_{mp} to S to ensure to be trustworthy.

3. The Client gets the data from the MP by using its own CRT_c as identifier. Optionally, for the `SeLvL` 4, MP sends its own CRT_{mp} to C.

We define the following security levels:

Security Level:	Description:
0x0	No authentication at all. Cryptography disabled.
0x1	Client sends CRT_c to the Server.
0x2	As above, plus Server sends CRT_s to the Client.
0x3	As above, plus MP sends CRT_{mp} to the Server.
0x4	As above, plus MP sends CRT_{mp} to the Client.

Table 3-5: Security Levels in Reverse Requests

3.6 Adaptive Compression

When we started to design our transport protocol we realized that it would have had to face several constraints on two different sides: first, because it is a protocol especially developed for embedded systems and sensor networks, we must take into account the limited CPU power and memory resources that any node of the network can be limited to. Also, since MPTCP is designed to work on ad hoc networks, we have to consider the limited throughput that an end-to-end link can offer to upper layers.

In order to increase network availability while addressing both the problems exposed, we evaluated the possibility of implementing an adaptive compression mechanism that can optimize the compression ratio according to the type of data transmitted and, for more accurate results, to the current network characteristics.

We started the design process by analyzing the existing technologies for self adaptively compression over remote links (e.g. OpenVPN [18]) and with a search for academic

publications that leaded to the following results, mainly taken from [19] and [20]. To begin, we can consider the three main factors that influence the choice of an optimal compression ratio:

- The ratio by which data can be compressed,
- The CPU processing power and available memory on the sender and on the receiver,
- The network end-to-end latency.

By combining these three values we are able to determine a compression level that could reduce the total data transmission time. Consider for example the case in which data is predominantly uncompressible or pre-compressed (e.g.: FTP or HTTP transfer of a large compressed archive): the compression can only introduce a computational overhead without giving any significant advantage in size or transmission time, so the ratio should be set to zero or to a low value.

It is possible to better define the compression trade-off by doing the following definitions, using the notation of [19]:

- T is the type of data we want to send. If we know T, we could choose a compression algorithm designed for this type of data.
- P is the parameter set controlling the compression algorithm.
- $C = f(CPU, T, P)$ is time to compress one byte, given the currently available CPU, type of data T and parameter set P.
- τ is the time to send one bit, i.e. $1/bandwidth$.
- $\rho = g(T, P)$ is the compression ratio we achieve for the type of data T and parameter set P.
- $C' = f'(CPU', \rho)$ is time to decompress one byte, given the currently available CPU and the compression ratio ρ of the input data.
- b is the amount of data we want to send.

Thus, the time needed to send the uncompressed data will be $(\tau \times b)$ and, for compressed data, $(\tau' \times b')$. The time we have available to process data is the reduction in sending time and if:

$$Cb + \left(\tau \times \frac{b}{\rho} \right) + C'b < \tau \times b$$

then compressing data before sending it would lower the total transfer time. To control the process we can modify the parameter P depending on the current network conditions.

We did not discuss an implementation of any specific compression algorithm because we consider this beyond the scope of the dissertation and we leave this extension as a future work, but for the moment we suggest two possible logics to determine the compression parameters:

- **Continuous compression benchmarking:** MPTCP will periodically sample the compression process to determine its efficiency, measured as $\frac{b'}{b}$. If the data being sent over the tunnel is already compressed, the resulting efficiency value will be low and the algorithm will disable the compression for a period of time until the next re-sample test. The sampling rate and the size "gain" threshold are not specified by the protocol but are implementation-dependant. It is anyway advisable to let the user "tune" these values if required.

- **Compression based on network throughput:** without any modification to the protocol, it is possible to derive an estimation of the current network throughput (i.e.: by analyzing the trend of the sequence numbers ACKed by the receiver or by directly counting the number of octets in the payload). Adjusting the compression ratio on this value may be a good idea if the network condition varies: the value can be increased if the network is slow or reduced if there is no real advantage against sending uncompressed data.

This behaviour of the compression system can be controlled using the parameter `GZ` in `head_general`, as follows:

Hex value:	Mnemonic:	Description:
0x0	OFF	Compression is always off
0x1	ON	Compression is always on.
0x2	AUTO	Compression is adaptive.

Table 3-6: Values for the GZ field in head_general

Chapter 4: Implementation

After the protocol design step was completed and the description of the low-level specifications was ready, we wanted to implement a simple MPTCP stack in order to test the main programmable interfaces and evaluate the protocol performances in sample user scenarios. Our experiments were first focused on the development of a complete transport layer on a Linux box but were slowed by the complexity of the network stack and the need of handling the protocol internals such as buffers, sliding windows, etc. We then opted to modify a user-space TCP stack and finally to use a network simulator that would allow us to concentrate on the protocol evaluation without spending excessive effort on the interfaces with the Operating System and the internal memory management. This chapter analyzes in detail the evaluation of the various experiments we ran during the implementation step.

4.1 Guidelines for an implementation

In order to simplify the deployment of MPTCP in existing network stacks, we reserved a paragraph in the "*MPTCP Protocol Specification*" document to give a guideline for the implementation of the protocol. Assuming a Unix-style Operating System, in that section we suggest three approaches that would represent an efficient protocol implementation:

- Using **"IP raw" sockets** in a user-space software to accept incoming packets from the network. To generate the new segments to be sent, the program would have to explicitly generate the headers, the checksums and manage ACKs and buffers,

- Writing an **external shared library** that could be included by user-space software to handle MPTCP operation. Such a library would provide to the programmers a set of APIs to create and manage MPTCP sockets,

- Developing a **kernel module** that can be loaded into memory to handle the MPTCP protocol and provide functions to create network sockets that use our protocol.

Each of these approaches offers advantages and generates specific issues. The first method requires the development of a complete architecture able to manage all the required networking functions, such as: parsing the packets coming from the underlying network layer, fragmenting data, generate headers, set retransmission timers, send and receive ACKs, handle flow and congestion control, etc. While it is a complex technique that requires a substantial programming effort, it is suitable for deploying MPTCP on embedded platform running an Operating System that offers limited functionalities.

Instead, writing an external shared library would allow the reuse of the produced code and it would simplify the support of the transport protocol by third-party applications. By distributing the binary version of the "MPTCP library" and some API documentation, every programmer would be able to link the support routines from his own code without having to implement any of the low-level functions needed.

The third option can be considered the most "elegant" in Unix-based systems, since the MPTCP functions would be managed directly by the Operating System kernel in the same fashion of the other well known protocol such as TCP, UDP, etc. For non-monolithic systems, there are two common ways to add the support for a new protocol that we will call "patching" and "loading". The former is the most "direct", since it requires hacking the kernel code and producing a "patch" file that can be later applied to other systems. The latter consists in writing an external kernel module that is loaded by the user at runtime. While both these approaches are architecture-dependant, a direct hack of the kernel source is also strictly version-dependant and could require code modifications to support newer OS versions.

4.2 Experiments with MPTCP

In the next four sections we describe the different attempts we made in order to develop a test implementation of MPTCP. We show the details of each of them, explaining the advantages of each solution and illustrating the reason for which we thought them as not optimal.

We ran all our experiments, especially those which required any software development, on systems with Linux kernels 2.6 for x86 architectures.

4.2.1 Hacking the kernel-space network stack in Linux

The first solution we decided to evaluate was to modify a "vanilla" Linux kernel in order to add the support for MPTCP. Since in the Linux architecture the TCP/IP stack can be compiled only as "built-in" feature and not as a loadable module, adding a new protocol from scratch would require a modification of the kernel code. It is not possible to write an external kernel module that can be loaded in memory by the end-user and thus the deployment of MPTCP on existing systems would require one of the following actions:

- Distributing a specific "MPTCP-enabled" version of the kernel, as source code or as binary executable.
- Providing a patch for a given version of the "vanilla" kernel.

We must also consider that in case the characteristics of the architecture on which MPTCP is to be deployed are not known in advance, the kernel cannot be distributed as a binary because this would make impossible for the user to add or modify the device drivers he needs for his hardware or user-space applications.

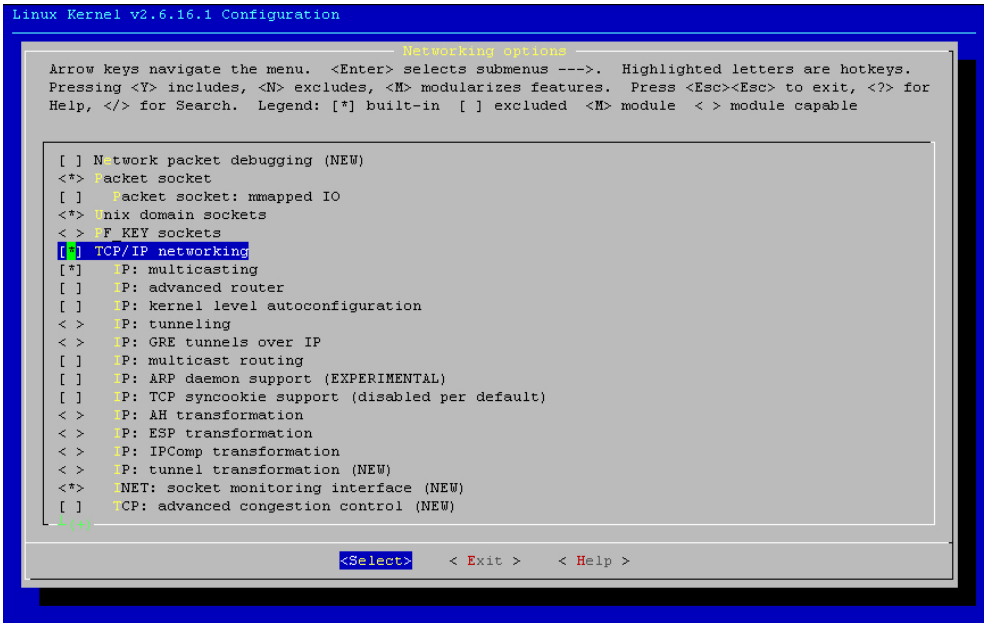


Figure 4-1: The TCP/IP networking option in the Linux kernel

At the moment of writing, the latest kernel version is 2.6.16 and, on this release, the networking code is spread over around 2,017 files reaching a total of 612,288 lines of code, while the files in `net/ipv4/tcp_*` are only 16. From these figures it easy to understand the

amount of code that a developer would have to look into in order to implement a new transport protocol. Most of the networking code is included in the directories shown in bold in Figure 4-2, a graph taken from Rio et al. [21].

Since the MPTCP protocol would be included in the built-in network functions, testing even a single line of code would require a complete recompilation of the kernel and a system reboot in order to load the new binary. Such development model requires spending a significant amount of time in compiling and testing the code, thus making more difficult a rapid application deployment.

Finally, it is important to note the set of instructions includes dependencies to the current system architecture for low level instructions (e.g.: for Little/Big Endian bits alignment), so a portable MPTCP implementation must include the support for each of them using techniques such as multiple `#define` to compile the proper code for the current platform.

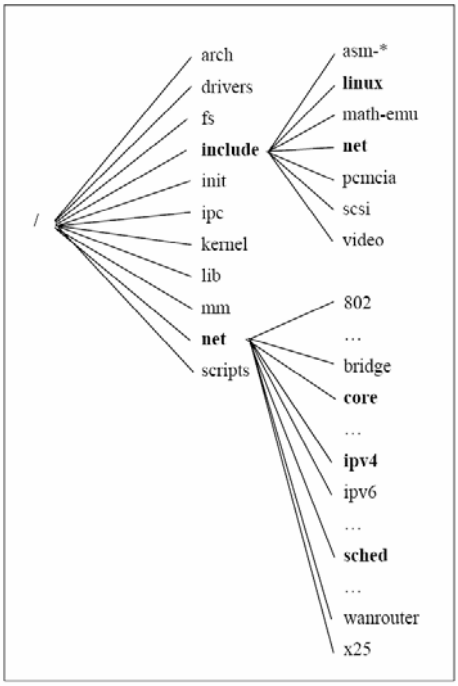


Figure 4-2: Networking code in the Linux kernel

4.2.2 Using a user-space TCP stack

Because of the problems explained in the previous paragraph, the development of a new protocol in the Linux kernel cannot be considered an optimal solution for testing purposes, while it offers performance advantages and direct access to the kernel internals.

To overcome the limitations of a kernel-space approach we then evaluated a solution based on the development of a network stack completely in user-space. This method allows a more rapid development cycle because there is no need to compile a complete kernel and reboot the system in order to test the program, and also grants a higher portability to different architectures.

By doing a search for academic papers, we found a similar project named “Daytona” [22], initially developed by the IBM Watson Research Center, the University of Illinois and the Princeton University. The development was then abandoned and is currently maintained by Prashant Pradhan and Srikanth Kandula, which we contacted for an advice. The authors define Daytona as a complete “stack TCP stack for Linux” and describe it to be “an invaluable tool for TCP performance research, network performance diagnosis, rapid prototyping and testing of new optimizations and enhancements to the TCP protocol, and as a tool for creating adaptive application-level overlays”.

Daytona is basically the same TCP networking code of the kernel extracted and assembled in a way that can be compiled as stand-alone and run in user-space. The last version is developed to run on kernels of the series 2.4 and includes a “fake” kernel module that integrates many functions of the operating system. When we started our experiment we realized that the available code could only be compiled with old versions of GCC 2.3.x and that it included many deprecated constructions and paradigms, so we spent a couple of days to modify the code to make it compatible with the latest GCC 4.x on a Linux 2.6 series kernel.

From the resulting source it would have been possible to develop an MPTCP implementation by using the TCP code as a starting point and writing from scratch all the new features. While it is difficult to imagine a real large-scale deployment of a new transport level protocol based on this solution, Daytona represents a good technique to shorten the development and testing times. One last open question we have is related to the lack of maturity of the Daytona code that, in many occasions, makes assumptions about the system on which the protocol is to be ran and that all the incoming packets are in a correct format.

4.2.3 Using raw-sockets in Linux

According to our purposes of developing MPTCP applications that are able to run on embedded systems, remote sensors and portable clients, we targeted one of our experiments to a lightweight implementation of the transport protocol features. The resulting solution should contain all the main characteristics of the algorithm and should be completely platform and architecture independent, while maintaining a full compatibility with the protocol specifications. In order to reach these targets, we aimed for an implementation fully contained in the user-space and written in a high-level programming language which compilers are available for various platforms.

The use of raw-sockets is an efficient and low-level approach to network programming without using the transport level functionalities offered by the Operating System. The developer is requested to program the entire code needed to handle the incoming packets, parse their headers, allocate buffers in order to rejoin and store fragmented data and explicitly manage congestion windows and retransmission timers. With raw-socket, the programmer passes a "protocol identifier" to the OS declaring that his software is willing to receive the incoming network layer segments with the given identifier in the "protocol" field of the IP header.

The main tasks that the algorithm would have to perform are:

- Accepting data from the local application, fragmenting it into segments, generating the headers list and assembling outgoing packets before sending them to the destination using the Operating System network layer.

- Handling incoming MPTCP packets from the network, parsing their headers and passing them to the application level

- Managing data encryption and the authentication procedures, including the generation of session keys, X.509 certificates validity checks, etc.

- If the local node is a Meeting Place, it is also necessary to implement some sort of storage of user's data. In real scenarios, this procedure has high efficiency requirements because data should be stored safely and retrieved quickly not to represent a communication bottleneck.

There are already a good amount of TCP implementations in high-level languages such as C and Java, and most of them are distributed as "open source" or are part of the public domain. A valuable starting point we evaluated is the C implementation of the complete

TCP/IP stack designed by Comer and Stevens [23]. While it is programmed for the Xinu operating system, it is reasonably easy to adapt to any other platform. The documentation of their algorithms is very detailed and extensively covers all the internal functions to simplify modifications from programmers.

Comer's code is available as a tar.gz archive that contains 656 C and headers files, with the tendency of including a single function for each source file. For this reason and because the code has been initially developed for educative purposes, many of the functions are self explicatory.

Many other open source implementation of the TCP/IP stack is available in books and on the Internet, so it is very unlucky that a developer would have to write a full MPTCP code from the beginning. Another starting point is represented by the networking code of the Linux kernel, which is freely available. Because of its complexity, it has a steep learning curve but many books (such as Stevens [24]), websites and guides include a complete explication of the algorithm.

4.2.4 Using OPNET Modeler

Our experiments focused also on the use of network simulators to reproduce the behaviour of MPTCP on a controlled environment and thus get statistics about the global overhead imposed by the protocol and other performance values. Simulate a network protocol is also useful for demonstrative purposes, to explain the functioning of the algorithms and to better understand the finite state machine model and the interaction between hosts.

Among all the network simulators that would be suitable for MPTCP, we chose to use the OPNET Modeler [25], a commercial software produced by OPNET Technologies, Inc.

The application describes each simulation as a series of discrete events over time and includes ready to use libraries with network objects ranging from wireless antennas to wired hosts including many well-known protocols for each of the ISO/OSI levels. It is possible to add new modules to the simulation by defining a state machine and writing C code to manage the transaction between states and send and receive packets. The software has a very modular approach and several modules, which mutually interact, compose a single object. As an example, in Figure 4-3 we can observe the internal structure of a standard LAN host.

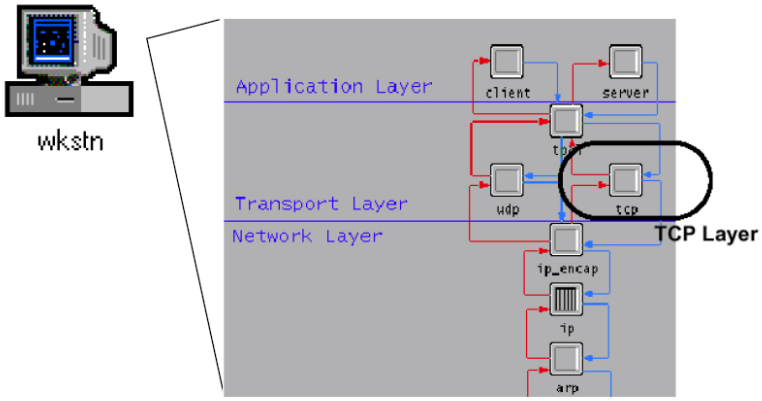


Figure 4-3: The TCP layer in the OPNET networking stack

As we can see, the TCP module is directly connected to the network layer and the application layer as in the normal ISO/OSI stack and furthermore the TCP state machine is very similar to the one defined in RFC 793 (see paragraph 3.2), as reported by Figure 4-4.

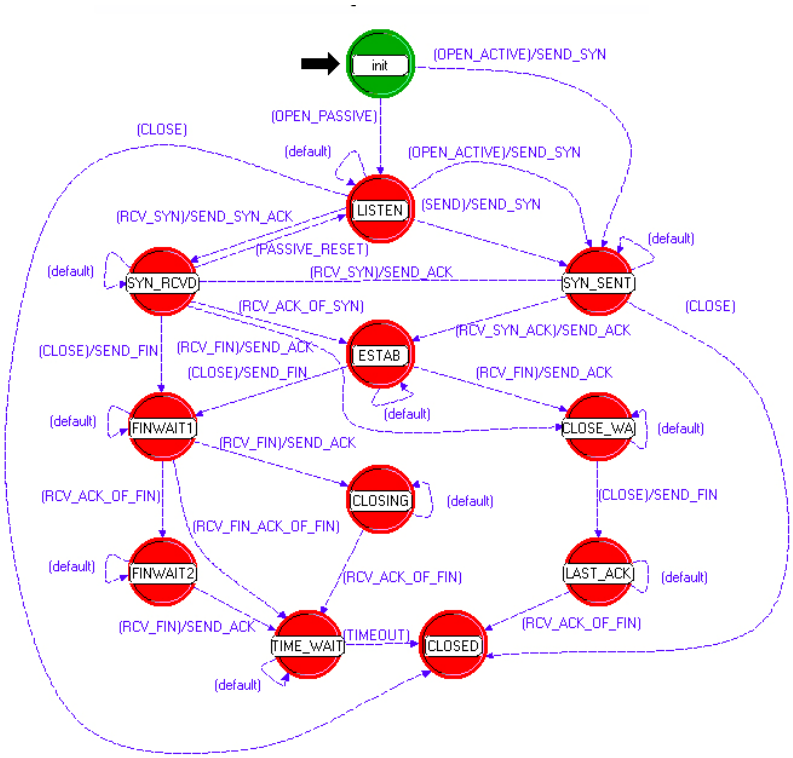


Figure 4-4: The OPNET representation of the TCP state machine

We spent several days on implementing a sample version of MPTCP that included only the most basic components and while the OPNET modelling approach looks simple at a first glance, its programming language derived from ANSI C sometimes shows to be problematic. The most difficult operations to implement in the simulator were the creation of a list of variable subheaders and some complex functions like the authentication procedures. Because of the constraint of the C-like language, it was also difficult to implement “storage” functions to memorize data on the meeting place nodes.

4.3 Simulations in OMNeT++

Because of these difficulties, we made a survey to find a network simulator with a better programmability. We then tried OMNeT++ [26], a software developed by András Varga and distributed as open-source under the GNU license.

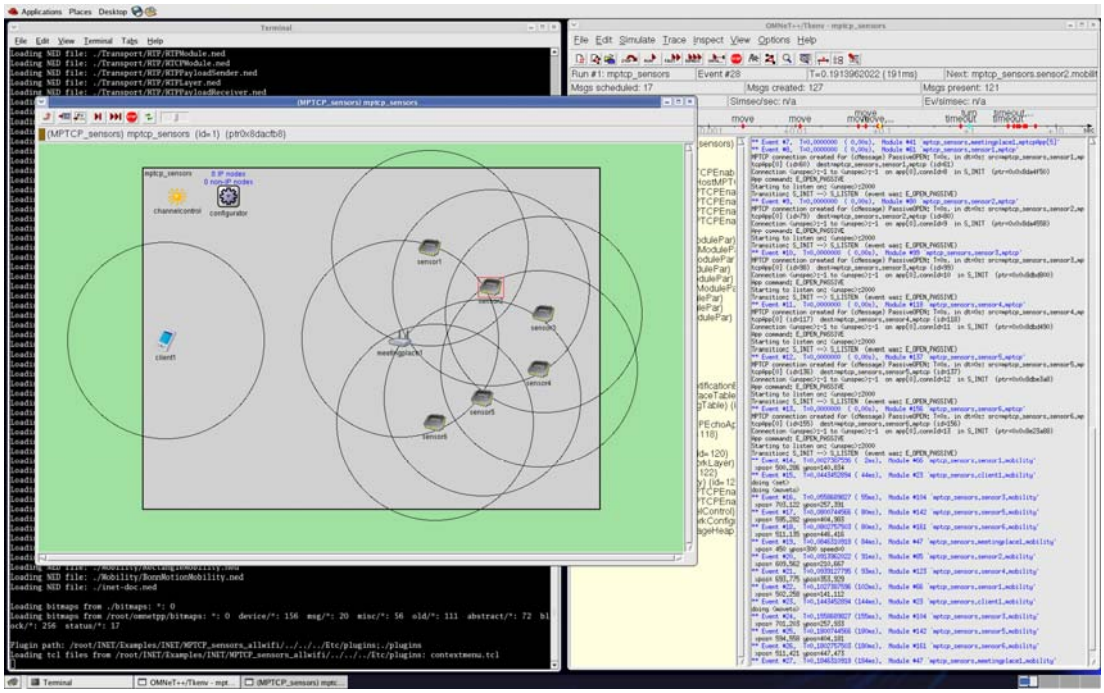


Figure 4-5: The OPNeT++ Workspace

OMNeT++ is available as source or as binary for Linux, Windows and many other platforms, and also includes libraries to support the most famous network protocols, wired connections and host mobility.

A single OMNeT++ simulation is composed by various components with different syntaxes and stored on physically different files. The two essential parts that are needed even for a basic simulation are:

- one or more NED file, that describe the objects used in the network and define the connections between them,
- `omnet.ini`, a simple text-file that includes all the parameters for the objects used in the network.

If the user needs to develop new modules, for example to support a new transport protocol as in our case, it is possible to do so using standard C++ code in the application framework provided by the simulator. In figure Figure 4-5 we can see the components of the working space: on the left the main simulation window with the grey playground area and the controls to start the action and to control the simulation speed, on the right the main program window with a tree graph of the used modules and an area for the debug output. On the top of the main window there is the timeline on which events such as packet transmissions, timers, and host mobility are represented as red dots.

In the playground we can notice the presence of a mobile Client, a Meeting Place and six sensor nodes. Each of the hosts has an 802.11 wireless interface which range of coverage is shown as a black circle. The experiment consists in the following steps:

1. A MPTCP-enabled mobile client, shown with the icon of a PDA, is initially out of the scene.
2. The client gets in the area of coverage of a Meeting Place and issues a Single Direct Request asking for the reading from the six sensors. It then moves out of range.
3. The Meeting Place queries each sensor and stores the data locally.
4. The client returns on the scene and gets the sensor readings that the Meeting Places had stored.

To shorten the development time we made several simplifications and assumptions that not affect directly the aim of the simulation, for example the sensors readings always return zeros, no authentication is used and no cryptography or data compression are implemented.

Another scenario we prepared is `MPTCP_basic`, in this case the playground includes only three nodes connected with wired PPP links. The purpose of this experiment is simply to test

and demonstrate the correctness of the protocol and to measure some quantities as packet sizes.

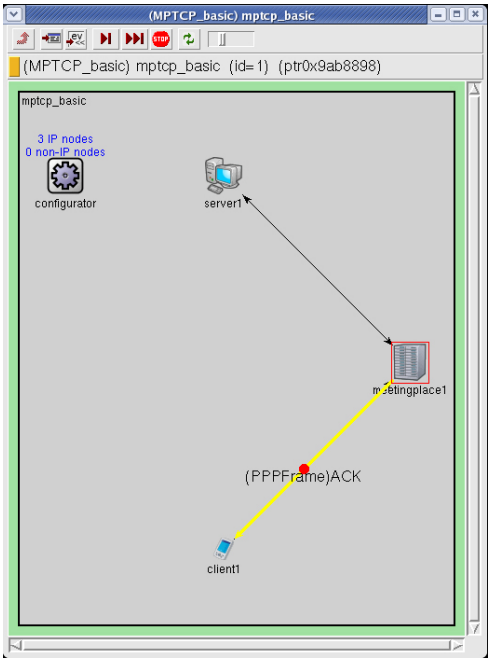


Figure 4-6: The mptcp_basic scenario

Using wireless communication in a simulated environment can lead to a difficult analysis of the performances of a protocol, since it is hard to identify the packet exchanges destined to the correct nodes. For example, imagine a situation with three nodes (A, B and C) is each in the coverage area of the other two. When host A sends a wireless packet to node B, the simulator will show two distinct messages: one travelling from A to B and the other from A to C. While this behaviour is correct because in reality both B and C antennas will receive the wireless message but only host B will recognize it as valid and will pass it to the upper network layers, it generates confusion in the observer because the number of packets showed in the playground window is potentially bigger than expected.

To overcome the problem, in our `MPTCP_sensor_wired` scenario (see Figure 4-7) we replaced the wireless connections between the Meeting Place and the sensors with standard wired links, which use is straightforward.

We regard this simplification as reasonable because the purpose of the simulation is not benchmarking transmission times but a verification of the protocol correctness.

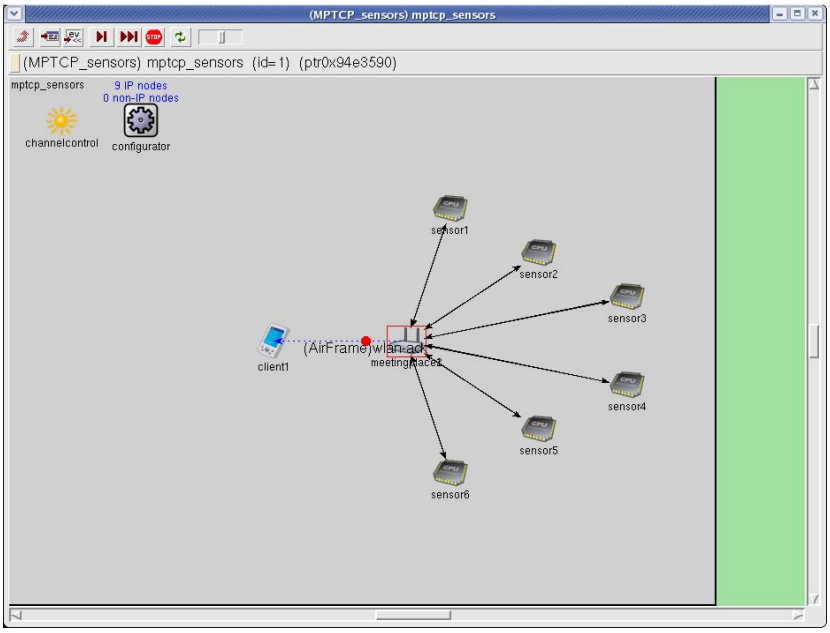


Figure 4-7: The mptcp_sensors scenario

The modular organization of OMNeT++ can be easily understood by double clicking on a network host in the playground in order to open the detail widows reported in figure X. On the right we see the internal composition of the network stack, with a PPP interface at the bottom, the IP network layer in the middle just below the three supported transport protocols: TCP, MPTCP and UDP. In this case, a MPTCP application is sitting on top of the stack and the connections are showed as black arrows. Further details about the memory occupation and internal organization of the components can be obtained by double-clicking their icons, as an example on the left side we notice the “Info” panel of the MPTCP application showing the Stack size in Bytes, the current state and other information while on the bottom the “Gates” panel reports the communication links between the transport layer and the network and applications.

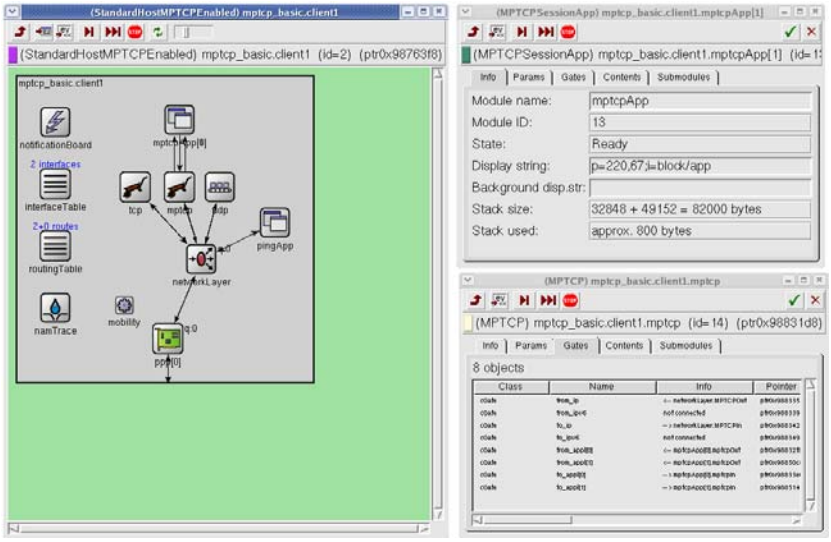


Figure 4-8: MPTCP component in the OMNeT++ stack

To better understand the functioning of the simulator, consider the C++ instructions needed to open a MPTCP socket listening on a specific port:

1. As in normal Unix sockets, in case the node has more than one IP address or network interface ("multihoming"), we can bind the socket to a specific address by doing:

```
socket.bind(Address, port)
```

2. The process of listening on a port is called "passive open", and is done simply by:

```
socket.listen()
```

3. To wait until a valid incoming connection is created, the application can use the following `while()` loop which literally means "process all incoming messages until a connection is made. If we get a `SOCKERROR` message, give up".

```
while (socket.state() != MPTCPSocket::CONNECTED) {
    socket.processMessage(receive());
    if (socket.state() == MPTCPSocket::SOCKERROR)
        return;
}
```

4. To send data on the socket, we use the "`send`" command, where "`msg`" is a reference to an object of type "`cMessage`".

```
socket.send(msg);
```

5. To close the connection:

```
socket.close();
```

The client will instead perform an "active connection" to contact the Server. The procedure is similar to the one explained above except about step 2, where we need to use:

```
socket.connect(Address, port)
```

in order to establish a connection with the node of given address on the specified port. All the socket objects are of type `MPTCPSocket` and exports their active states as public variables such as:

```
MPTCPSocket::NOT_BOUND
MPTCPSocket::BOUND
MPTCPSocket::LISTENING
MPTCPSocket::CONNECTING
MPTCPSocket::CONNECTED
MPTCPSocket::PEER_CLOSED
MPTCPSocket::LOCALLY_CLOSED
MPTCPSocket::CLOSED
MPTCPSocket::SOCKERROR
```

Our implementation work in OMNeT++ began by duplicating the TCP model to have a starting point for the MPTCP development. We then developed a script to modify all the references to the TCP structures by the use of regular expression that substitutes all the “TCP” strings with “MPTCP”, respecting the context in which the replacement is done. After that we changed the packet header format to include the new fields and we developed two basic applications that are useful for testing the protocol. We also created two host objects that include the support for our protocol, one with an only a PPP wired interface and another including also one or more 802.11 wireless cards and the support for host mobility.

To summarize, the following are the component that have been developed:

- `Transport/MPTCP/` and `Transport/Contract/MPTCP*`, that contains the actual C++ implementation of the MPTCP transport layer.
- `Applications/MPTCPApp/` that contains the C++ code of the MPTCP sample applications.

- `Nodes/INET/MobileHostMPTCPEnabled.ned`, which is a mobile node offering the support for mobility and a wireless interface complying with the 802.11 standard.

- `Nodes/INET/StandardHostMPTCPEnabled.ned`, which is a modified version of the OMNeT++ standard node including the support for MPTCP, zero or more PPP interfaces, zero or more Ethernet connections and any number of 802.11 wireless receivers.

- The three showed scenario are stored in the directories `Examples/INET/MPTCP_basic/`, `Examples/INET/MPTCP_allwifi/` and `Examples/INET/MPTCP_wiredsensors/`.

While it is not a full implementation of the protocol because we did not include authentication procedures nor the support for cryptography and data compression, the resulting MPTCP model is valid for demonstration purposes and as framework for further developments.

Chapter 5: Evaluation

After the experiments discussed in Chapter 4 and after the analysis of a basic MPTCP implementation in a network simulator, we faced the questions about how to properly evaluate a transport protocol which main aims are not performance results such as throughput or latency but instead offering new characteristics like disconnection handling and local data storage. Since benchmarking would not be possible to demonstrate the validity of those operations, our evaluation approach will be an analysis of the feature list of MPTCP.

We will conclude the chapter including a review of the security issues addressed, an estimation of the overhead induced by protocol headers and a comparison table between our protocol and others, using the format we discussed in Chapter 2.

5.1 Sample scenarios

MPTCP uses are potentially unlimited, ranging from unreliable wireless links to high-availability wired networks. In the following seven paragraphs we will illustrate various different scenarios in which we believe MPTCP can give a valuable contribution.

5.1.1 Sensor Networks

In the last years, a number of “smart dust” solutions have appeared on the market. These small devices, often only few centimetres long, are equipped with a radio and one or more sensors to get readings of the current temperature, air pressure, humidity, acceleration, etc. In a typical environment, the boards would be connected via wireless links to form an ad-hoc network in which the routing is done by all the nodes by re-broadcasting the incoming packets.

Programming a software application for a sensor network is difficult for several reasons: the embedded operating system running on the sensor boards usually is very constrained and offer scarce network functionalities, available memory and CPU power are limited, the

network does not offer any speed or reliability guarantee and it is often mandatory to bound the number of calculations and packets sent to the network in order to save energy. It is also important to develop a stable application before arranging the sensors on the field of operations because in many cases it would be extremely difficult or impossible to modify the boards once deployed.

All these issues lead to a difficult development lifecycle during which the need for a reliable data transport protocol face with a very constrained programming environment. MPTCP provides two efficient methods to save sensor readings. Assume that the user is carrying a mobile device such as a PDA to collect the data periodically, for example to get the temperature values of ten sensors in an area with an interval between two values of an hour.

Normally, the boards would have to store the readings in the local memory and, every hour, the user would need to physically reach the area of monitoring to download the data on his PDA. Following the MPTCP paradigm we can imagine to program and deploy a Meeting Place, such as a small board equipped with a wireless radio and a storage chip, that will periodically query the sensors and store the received data. The user will then be able to download on his PDA all the readings of the last few days or weeks at once.

A different approach uses Multiple Reverse Requests: the sensor boards can be programmed to actively transmit their readings to a Meeting Place, which will store them on the disk. Later, the user will be able to collect all the gathered data directly from the Meeting Place.

5.1.2 Requests for time-consuming services.

Many mobile applications are developed following a Client/Server approach in which a small software client is installed on a portable device (i.e. a PDA, a Java-enabled mobile phone, a gaming console etc.) in order to interact with a remote server.

The Server applications can offer several services at once and will typically interact with other remote resources such as databases, archives, web services, etc. We can thus divide a single request from a mobile device in three steps: first, the user sends a command to the remote server, then the data is processed and the results are computed and finally the data is sent back to the portable application. In many situations the second step we just defined will require a variable amount of time for its completion and the user must remain connected to the network in order not to loose the answer.

An example can be represented by a mobile application for financial trading running on a mobile phone. The remote server handles the “buy”, “sell”, “get price of” commands by querying a complex SQL database with the details of each stock option, investment fund, and analyst ratings before sending back to the user a confirmation of the command or a numeric requested value.

A valid enhancement to this Client/Server application is represented by “SQL over MPTCP”, according to which the database operations on the Server can be performed while the client is disconnected. At first, the user can ask to store the data on a meeting place then disconnect from the network. Later, the Client will connect again to the network and download the data from the Meeting Place.

5.1.3 Large Requests to Servers with low bandwidth.

In mobile networks, no assumption can be made about the bandwidth available to a remote server for providing a service. Moreover, if data routing is based on packet forwarding, it is even possible for a single mobile client to have get bandwidth than a central server.

As an example we can consider that the Operating System of a PDA is connecting to a remote file storage using an 802.11 wireless connection in order to automatically download software updates. If the file being downloaded is large and the server is overloaded, the data transfer will require more time than necessary thus reducing the general network efficiency. As a matter of fact, even if the client would be able to download at a faster rate, the server (or the network, if many PDAs are updating the software at the same time) cannot dedicate more bandwidth forcing the user to wait for a longer time and wasting battery power.

By using “HTTP over MPTCP”, a client that wants to download a large file from a slow server can delegate a Meeting Place to get the file, then disconnect and collect it later at a higher speed.

5.1.4 Many requests, one collection.

Many network applications are composed by a pattern of requests that repeats constantly over time. Common examples are administrative protocols that periodically gather statistics about the status of links and hosts, such as SNMP.

For instance consider a user application which queries every 5 minutes each of the routers in the network to obtain a value of the bandwidth utilization of each Ethernet link.

In this case, with the adoption of “SNMP over MPTCP” the client can instruct a Meeting Place to send an SNMP payload to a list of IP addresses every 5 minutes and to store the data on the local disk. The user software will then download all the gathered readings at once. Thus, instead of sending a request and obtaining an answer for each data sampling, the client will only send out a request to the Meeting Place and then get only one cumulative reply.

5.1.5 Implicit redundancy, keeping a local copy of the data.

The transport protocol that we developed can also be used for operations typically carried out by higher levels of the network stack. In case the network administrator wants to improve network reliability by increasing the redundancy of user data, MPTCP provides an automatic technique to mirror information and store it on the Meeting Places of the local network.

A good example is Database replication: a mobile device accessing a remote database can request to the Meeting Place to store a copy of the tables on its disk. By doing this, the user will be able to work even in case of a failure of the remote network link.

5.1.6 Reduce bandwidth requirements for external networks.

Sometimes, mobile and ad hoc networks do not have a direct upload to the Internet nor to any Wide Area Network but only include hosts that are geographically near to each other. An “uplink” to other public networks is often included for management purposes and it is typically constituted by a GSM or GPRS connection.

The adoption of links that have a direct correlation between costs and produced traffic must be planned in advance to avoid unnecessary high network costs. Imagine the case in which dozens of PDA are programmed to download firmware updates from the Corporate FTP server: if each client downloads a copy of the file, the accounted traffic on the dial-up link will be elevated.

The use of MPTCP for file repositories leads to an optimization of the external network resources, since a keeping a copy of the data on the local Meeting Place reduces the demand for external connectivity.

5.1.7 Transmitting data between mobile users.

In the last sample scenario we point out an end-user application developed with MPTCP support.

Instant Messaging software are very popular for both mobile clients (e.g.: SMS, Bluetooth paging, etc.) and public networks (e.g.: software like ICQ, Microsoft MSN, Jabber, etc.) and they represent an effective communication tool.

Since most of the IM applications are based on a Client/Server paradigm, their deployment on generic mobile networks is more difficult because of the poor links reliability and because the user can connect and disconnect from the network at any time. This problem is even more evident for pure peer-to-peer messaging protocols in which no central server is used but the clients directly connect to each other.

By developing an IM software that uses MPTCP, the programmer can exploit the presence of the Meeting Places in the network and use them as a temporary "base point" to transfer data between two mobile user that can be disconnected at any time because of insufficient network coverage, hand-off errors, etc.

5.2 Overhead considerations

As previously explained in Chapter 3, MPTCP includes a communication model based on variable size headers and on the use of a linked list of optional "subheaders". This approach was adopted in order to include in the transmission only the options that are actually needed in order to reduce the packet size and thus increase the throughput, reduce the transmission times and save power. The following table gives a rapid overview of the admitted subheaders, the number of times each of them can appear in a packet and the size of each in octets.

Subheader name	Number of occurrences	Size (octets)
head_general	{1}	24
head_thirdaddress_v6	{0, ∞}	20
head_thirdaddress_v4	{0, ∞}	8
head_ttl	{0, 1}	8
head_crypt	{0, 1}	Variable
head_multiple	{0, 1}	4
head_answer	{0, 1}	4

Table 5-1: MPTCP subheaders sizes

It is important to note that in MPTCP all the headers have a fixed size except `head_crypt` that contains the X.509 certificates and the session keys used for cryptography. In order to be considered valid, a packet need only to contain `head_general` and one between `head_thirdaddress_v4` or `head_thirdaddress_v6`, so the size of the smallest accepted packet in MPTCP, assuming an empty payload, is of 32 or 44 bytes respectively.

Another technique that our transport protocol incorporates in order to reduce the packet size consists in sending all the subheaders only once in the first packet sent to the destination, thus the first with the "SYN" bit set. After the sender receives an ACK for this segment, all the subsequent packets include only `head_general`, having thus an overhead of only 32 bytes.

As a comparison, we recall that TCP has a fixed header size of 20 octets, IPv4 is from 20 to 60 bytes long and IPv6 has a minimum length of 40 bytes. By considering all these values, the minimum size of a MPTCP packet once encapsulated by the network layer is 20+32=52 bytes for MPTCP over IPv4 and 40+44=84 bytes for MPTCP over IPv6.

5.3 Security aspects of MPTCP

Paragraph 3.5 analyzes in detail the security features of MPTCP, explaining the operation needed for authentication and data cryptography. In this section we will examine some scenarios in which an external malicious "actor" tries to overhear, change, corrupt or destroy data running over a MPTCP-enabled network by performing attacks aimed to the transport layer.

In the first scenario we imagine an attacker node that tries to impersonate a legitimate network node in a reverse request, as shown in Figure 5-1.

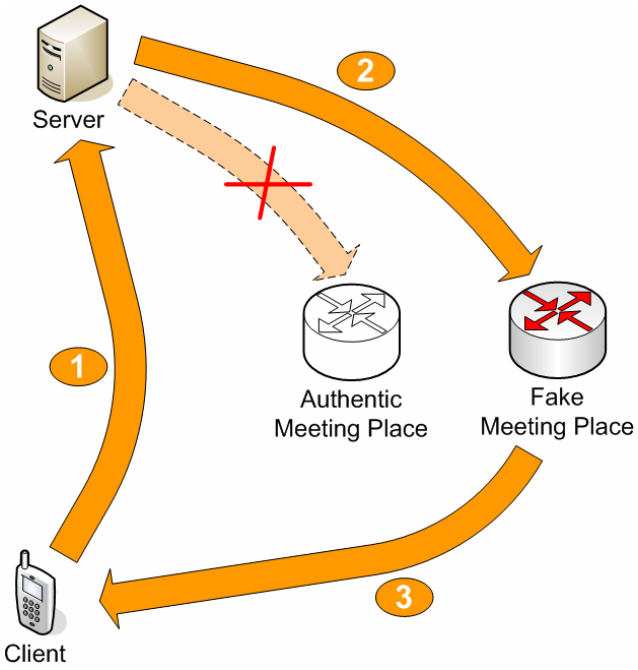


Figure 5-1: MPTCP security attack based on IP spoofing

The attacker will try to "spoof" the IP Address of the authentic Meeting Place in order to take its place and to direct towards itself the traffic coming from the Server. As a consequence, two different situations are possible:

- If the security level (variable `SeLvL`) is bigger or equal to 3, mutual authentication between the Meeting Place and the Server is in place, therefore the Meeting Place must send its certificate CRT_{mp} to the Server, which will check its validity. By requiring this extra check, the Server will identify the Meeting Place as fake and it will refuse the connection or trigger an external security countermeasure.

- If the security level is less than 3, the Server does not have any method to control the Meeting Place validity and thus the attacker could impersonate it successfully.

In both cases, it is important to remember that the malicious node will not be able to read in clear the data sent by the server since it is encrypted at the source using a session key known only to the Server and the Client. As a result, with `SeLvL >= 3` no attack is possible using the "spoofing" technique while with `SeLvL < 3` the attacker can only receive and destroy data but not decrypt it.

In the second scenario, we imagine an attack that exploits the properties of insecure wireless links. Typically, sensor networks do not use any encryption at the MAC layer to

prevent data stealing from unauthorized hosts, on the contrary to infrastructure based wireless network where often standard protection systems like WEP or WPA are enabled. It is often difficult to implement any data encryption scheme at the physical or media access layer when the hardware characteristics of an embedded system are very constrained, therefore sensor and on-the-field applications often transmit data on the air without any encryption at all. In this case, it is easy for an attacker to "sniff" all the packets it is able to hear and potentially it is even possible to capture every single octet sent in the transaction between the Client, the Meeting Place and the Server. This attack is easily feasible with limited computing power also on 802.11 network protected with the WEP standard, because of the well documented flaws of this standard [27].

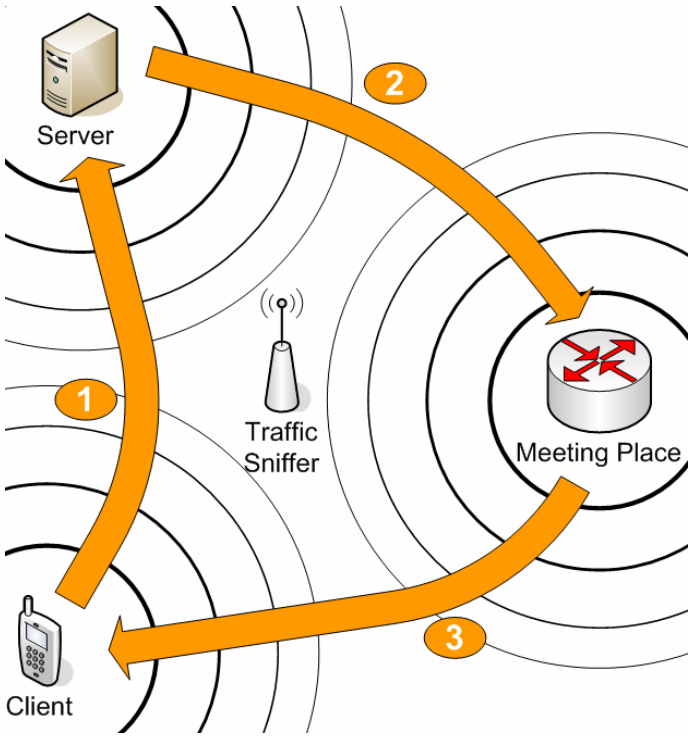


Figure 5-2: MPTCP attack based on traffic sniffing

With the MPTCP protection scheme based on asymmetric cryptography and certificate exchange, even if an eavesdropper can capture all the data send from the beginning of the transaction, it would be still impossible for the attacker to determine the session key that is used to encrypt the data. Consequently, it is completely useless for an unauthorized node to perform an attack based on the systematic and extensive capture of the network traffic and the only data it could gather are statistics about the number of connections, network addresses and other header fields that are sent as clear text.

The third security scenario we propose is an extension of the second, since we suppose that a malicious node has captured all the network traffic relative to a full MPTCP transaction. After the transaction is finished and the Client has got its data from the Meeting Place, the attacker manage to get a copy of the both the X.509 certificates and the private keys of each of the three nodes.

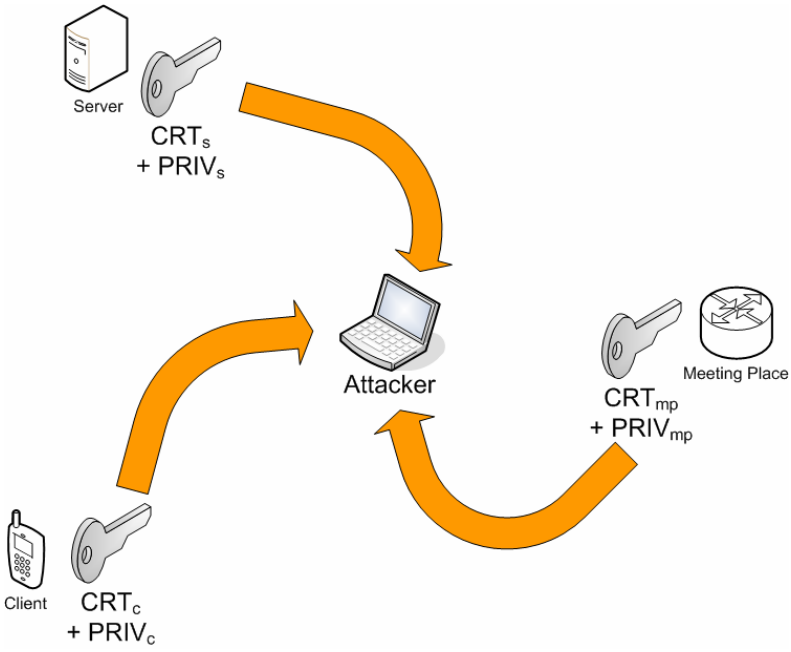


Figure 5-3: MPTCP attack based on certificate stealing

At this point, the node would be able to "forge" any type of packet and impersonate any other node by using their private key. Despite this, since data destined to the Client is encrypted directly on the Server using a temporary session key that is known only to the Client and the Server and never sent in clear over the network, MPTCP ensures "perfect forward secrecy" of the data stored on the Meeting Place disk and sent between the hosts. Because of this approach the data can be deciphered only by the Client and the Server by using the session key and any other approach based on network "sniffing" would fail.

The only scenario in which knowing both the public and private keys of all the nodes represents an advantage is one where an attacker gets a copy of the keys before the MPTCP transaction begins. Nevertheless, if the hosts use a "Certificate Revocation List" or an online certificate validity protocol and their discover that the Private keys have been stolen, it would be possible to revoke their certificates on the fly and consequently deny any further use.

5.4 Feature list and protocol comparison

While MPTCP includes performance enhancements and techniques to adapt the transmission to the network capacity, it is a transport protocol that does not aim directly to deliver a better throughput, latency or reliability but it is intended to offer new qualities to existing networks.

In this paragraph we analyze the main features offered by the protocol, comparing them with the other protocols we discussed in Chapter 2.

The central functionalities introduced by MPTCP are:

- The support for handling multiple, periodical requests by issuing one command in a single transaction. This result can be achieved by using Multiple Direct Requests (MDR) or Multiple Reverse Requests (MRR) or by including more than one `head_thirdaddress` header in the packets.

- MPTCP maintains the end-to-end paradigm of the traditional transport protocols while offering two different operation modes: direct and indirect connections between the Client, the Meeting Place and the Server.

- To reduce protocol overhead and subsequently require less power and transmission time, the protocol uses headers with variable length to include only the options needed by the current transaction or supported by the actual implementation.

- A transparent and adaptive payload compression algorithm is supported, but can be disabled by using a specific flag in the headers.

- MPTCP works over unmodified IPv4/IPv6 networks and does not require any modification to the network layer of the hosts or the use of a specific routing protocol on the network.

- Security features are included in the protocol specification. A scheme using X.509 certificates and asymmetric cryptography is used to avoid many common attack patterns and to ensure Perfect Forward Security (PFS) of the data stored in the network. The implementation is independent from the adoption of a specific “cipersuite” and no encryption algorithm nor key lengths are enforced.

The table reported in the following page presents a basic comparison of the features of MPTCP with other two transport protocols, following the scheme adopted in paragraph 2.3.

	MPTCP	TCP-BuS	Split-TCP
References	n/a	[5]	[7]
Packet loss due to BER or collision	On a single link: same as TCP. Note that between C and S no contemporaneity is needed.	Same as TCP	Same as TCP
Path breaks	Thanks to the MP paradigm, the user does not experience any single link disconnection.	ERDN is sent to the TCP sender, state changes to snooze, ICMP DUR is sent to the TCP sender, and ATCP puts TCP into persist state	Same as TCP
Out-of-order packets	Same as TCP	Out-of-order packets reached after a path recovery are handled	Same as TCP
Congestion	Same as TCP	Explicit message such as ICMP source quench are used.	Since connection is split, the congestion control is handled within a zone by proxy nodes
Congestion window after path reestablishment	Recomputed for every transaction	Same as before the path break	Proxy nodes maintain congestion window and handle congestion
Explicit path break notification	No	Yes	No
Explicit path reestablishment notification	Yes	Yes	No
Dependency on routing protocol	No	Yes	No
End-to-end semantics	No	Yes	No
Packets buffered at intermediate nodes	Yes	Yes	Yes
Compatible with TCP	No	No	No

Table 5-2: Comparison table including MPTCP

Chapter 6: Conclusions

This chapter discusses the purpose of the thesis and how the objectives were achieved. The first paragraph lists the contributions to the state of the art in the area of transport protocols for delay-tolerant links and Ad Hoc networks, we then analyze the completed work, the results of the experiments discussed in Chapter 4 and the evaluation presented in Chapter 5. The final section on future work suggests any further development which could be carried out on the protocol.

All the protocol specification documents and the developed C++ code can be downloaded from the project website, at the URL: **http://mptcp.sourceforge.net**

6.1 Contribution

At the beginning of the thesis we briefly introduced ten existing transport protocols that were designed to improve the performances of the traditional TCP stack in scenarios such as: ad hoc routing systems, wireless degraded links, "interplanetary" networks and generic architectures with delay-tolerant properties. All these standards are based on the assumption that two endpoints connects between them, directly or by packet forwarding with the help of other nodes, thus creating a single link over the network. In some cases (such as S-TCP or LTP), intermediate nodes that lie on the path between the sender and the receiver can act as transparent "proxies" by locally caching data before passing it to the next node, increasing thus the overall network reliability. The user is not aware of the mechanism and what he sees is only a direct connection to a remote node.

The transport protocol that we developed follows a different paradigm in which is the user (or the application he is running) that directly controls data caching on the network hosts. It is also important to note that, while in traditional TCP a connection can occurs only between two endpoints, in MPTCP a "transaction" can include a minimum of three end-to-end packet exchanges without any upper maximum.

To conclude, we believe that the introduction of MPTCP in an existing network can represent a useful aid for scenarios like the seven described in Paragraph 5.1. Whilst all these situations can be solved using only traditional transport protocols and possibly delegating the packet delivery to upper layers, a technique based on Meeting Places provides a more accurate control of the data transfer.

6.2 Completed Work

Alongside this dissertation we prepared a separate document, "MPTCP *Protocol Functional Specification*", that is reported in Appendix I and includes a formal description of the algorithm and structure used. A central part of the text is dedicated to describe the finite state machine used to model the protocol. The format of all the low-level structures used by MPTCP is specified in detail, including also the headers layout and the allowed options of each.

Starting from that document it is possible to develop an actual implementation of the transport protocol in virtually any programming language that offer networking support. As described in Chapter 4, we evaluated several solutions for testing purposes, including a Linux kernel modification, a user-space application which can be compiled without modifying or rebooting the system and a shared library in C that uses IP RAW sockets. We completed by producing two simulation environments using OPNET and OMNeT++ and by running various scenarios including wired and 802.11 wireless networks.

To conclude, a new delay-tolerant transport protocol has been designed, formalized in the "Protocol Specs" document and tested by the use of network simulators. We believe that MPTCP can represent a useful technique in the deployment "on the field" of many applications.

6.3 Future Work: How to expand MPTCP

During the course of our research, many opportunities for future investigations were made clear. First of all, a full implementation of the MPTCP functionalities would allow the deployment of the transport protocol in a testing environment and the measurement of performance values. The suggested approaches have been described in Chapter 4 for Linux platforms but of particular interest is also the implementation of MPTCP on embedded hardware such as sensor motes running TinyOS [28], µCLinux [29] or similar.

As explained in the last chapter of the "*Protocol Specification*" document, a useful extension to MPTCP would be the development of a new sub-layer that the Meeting Places could use to communicate among them. The exchanged information is mainly constituted by routing and signalling data for the following purposes:

- **Synchronization**, to copy the stored data on more than one Meeting Place in order to achieve higher redundancy,

- **Failover**. In case a node suddenly becomes unavailable, neighbouring hosts can signal the problem to the Server, the network administrator or the other Meeting Places.

- **Administrative routing**. If a Meeting Place has to be shut down, the administrator could notify the other hosts to avoid sending data to the offline node. Example scenarios are scheduled maintenance works, software upgrades, etc.

- **Traffic redirection** between Meeting Places. When a node get too busy to manage the incoming data, store it on the local disk and notify the user, the signalling protocol would allow the routing of requests to other nodes. This function creates a "load-balancing" solution that also solves storage related problems such as full disks, RAID problems, etc.

Because of the security issues that such a routing problem can generate, it is advisable to exchange all the information using asymmetric cryptography and verifying the X.509 certificates that are already used in the protocol.

The development of a signalling sub-protocol would allow for a better management of the requests coming from the Clients and of the data being sent by the Servers, with both performance and reliability improvements.

Chapter 7: References

[1] Nagle, J. (1984). *Congestion Control in IP/TCP Internetworks.* IETF RFC #896.

[2] Murthy C. Siva Ram, Manoj B.S. (2004). *Ad Hoc Wireless Networks.* Prentice Hall Communications Engineering and Emerging Technologies Series.

[3] Chandran K., Raghunathan S., Venkatesan S and Prakash R. (2001, February). A Feedback-Based Scheme for Improving TCP Performance in Ad Hoc Wireless Networks. *IEEE Personal Communications Magazine*, vol.8, no. 1, pp. 34-39.

[4] Holland G. and Vaidya N. (1999, August). Analysis of TCP Performance over Mobile Ad Hoc Networks. *Proceedings of ACM MOBICOM 1999*, pp. 219-230.

[5] Kim D., Toh C. K. and Choi Y. (2001, June). TCP-BuS: Improving TCP Performance in Wireless Ad Hoc Networks. *Journal of Communications and Networks*, vol. 3, no. 2, pp. 1-12, June 2001.

[6] Liu J. and Singh S. (2001, July). ATCP: TCP for Mobile Ad Hoc Networks, *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 7, pp. 1300-1315.

[7] Kopparty S., Krishnamurthy S. V., Faloutsos M. and Tripathi S. K. (2002, November). Split TCP for Mobile Ad Hoc Networks. *Proceedings of IEEE GLOBECOM 2002*, vol. 1, pp. 138-142.

[8] Liu J. and Singh S. (1999, September). ACTP: Application Controlled Transport Protocol for Mobile Ad Hoc Networks. *Proceedings of IEEE WCMC 1999*, vol. 3, pp. 1318-1322.

[9] Sundaresan K., Anantharaman V., Hsieh H. Y. and Sivakumar R. (2003, June). ATP: A Reliable Transport Protocol for Ad Hoc Networks. *Proceedings of ACM MOBIHOC 2003*, pp. 64-75.

[10] *Licklider Transmission Protocol website* (n.d.). Retrieved May 2006. http://irg.cs.ohiou.edu/ltp/

[11] Bakre A., Badrinath B. R. (1995). I-TCP: indirect TCP for mobile hosts. *Proceedings of the 15th International Conference on Distributed Computing Systems*, page 136.

[12] Song Y., Suh Y. (2003, March). Rate-control snoop: a reliable transport protocol for heterogeneous networks with wired and wireless links. *IEEE Wireless Communications and Networking 2003*, vol. 2, pp. 1334-1338.

[13] Deering S., Cisco, Hinden R., Nokia. (1998, December). Internet Protocol, Version 6 (IPv6) Specification. *IETF RFC 2460*.

[14] ITU-T Working Group. (n.d.). *Public-Key Infrastructure (X.509) charter*. Retrieved May 2006: http://www.ietf.org/html.charters/pkix-charter.html

[15] *The OpenSSL project website* (n.d.). Retrieved May 2006: http://www.openssl.org

[16] *The GNU Transport Layer Security Library website.* (n.d.). Retrieved May 2006: http://www.gnu.org/software/gnutls/

[17] Myers M., Ankney R., Malpani A., Galperin S., Adams C. (1999, June). X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP. *IEEE RFC 2560*.

[18] *The OpenVPN website.* (n.d.). Retrieved May 2006: http://www.openvpn.net

[19] Jeannot E., Knutsson B. (2002). Adaptive Online Data Compression. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing 2002*. page 372.

[20] Krintz C., Sucu S. (2006, January). *Adaptive On-The-Fly Compression IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 17, number 1.

[21] Rio M., Kelly T., Goutelle M., Hughes J. R., Martin-Flatin J. P. and Li Y. (2004, March). *A map of the networking code in Linux kernel 2.4.20*. Internal Technical Report. Retrieved May 2006: http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf

[22] Pradhan P., Kandula S., Xu W., Shaikh A. and Nahum E. (2002). *Daytona: A user-level TCP stack*. Retrieved May 2006: http://nms.lcs.mit.edu/kandula/data/daytona.pdf

[23] Comer D., Stevens D. (1999). *Internetworking With TCP/IP Volume II: Design, Implementation, and Internals*. Prentice Hall.

[24] Stevens W. R. (1996). *TCP/IP illustrated*. Addison-Wesley.

[25] *The OPNET website.* (n.d.). Retrieved May 2006: http://www.opnet.com

[26] *The OMNeT++ website.* (n.d.). Retrieved May 2006: http://www.omnetpp.org

[27] Borisov N., Goldberg I., Wagner D. (2001). Intercepting mobile communications: the insecurity of 802.11. *Proceedings of the 7th annual international conference on Mobile computing and networking*, pp. 180-189.

[28] *The TinyOS opensource project website.* (n.d.) Retrieved May 2006: http://www.tinyos.net/

[29] µCLinux, *Embedded Linux/Microcontroller project website*. (n.d.) Retrieved May 2006: http://uclinux.org/

Appendix A: MPTCP Protocol Draft

In the following pages we include "*MPTCP Protocol Specification*" document, which contains the formal description of the protocol operations and of the low level structures used such as headers, interfaces, states, etc.

This text represents a useful documentation for developing a real MPTCP implementation in a high-level programming language.

MPTCP
MEETING PLACES TRANSMISSION CONTROL PROTOCOL
PROTOCOL SPECIFICATION

1. INTRODUCTION

1.1 Background

The communication between two nodes in infrastructure-based networks is often based on point-to-point communication to implement communication models such as client-server. This approach relies on the availability of both nodes at time of the communication and on a stable connection between the two nodes during the duration of the communication.

In mobile networks – and especially mobile ad hoc networks – these assumption do not hold. Two mobile nodes that attempt to employ the client-server model are open to problems introduced by the mobility of either node e.g. unavailability of one node during migration, unavailability of a route between the two nodes, etc.

The assumption of the “Meeting Places” approach is that mobile nodes will always be able to connect to a “stable” node somewhere in the network. The key advantages of this model on a highly variable network is that a few known and relatively stable network nodes can be used to achieve point to point like communication without the otherwise necessarily persistent quality of network. Moreover, the approach is scaleable through the introduction of more message points, and can accommodate a range of intermediary patterns to achieve various tradeoffs in performance.

We started our work by analyzing the currently existing enhancements to the TCP protocol for ad-hoc networks and then we identified the features needed at the application level and at the underlying network layer.

This document represents a formal description of the operation of the algorithm that can be an aid to the understanding, implementation and use of the protocol. We are supporting our work publishing a sample implementation as a standard C library for Linux operating systems.

1.2 Motivations

The typical motivation of a level-3 protocol is to provide a reliable or unreliable way to send data between two endpoints across a network. This concept assumes that both nodes are connected at the same time, directly via a single link or as part of a bigger network. Our protocol focuses on the situations in which the two endpoints cannot be connected at all times considering in particular, but not exclusively, scenarios where embedded systems are connected trough “Ad-hoc” networks.

For simplicity, in the following paragraphs we will call the two endpoints “Server” and “Client”, where the first is the “data producer” and the second the “consumer”. Although this is a very common scenario, the protocol remains valid in pure “peer-to-peer” environment where any nodes can be defined both as Server and as Client.

The four motivations that drove the design of the MPTCP protocol were the followings:

a. Absence of contemporaneity between the Client and the Server.
   This assumes that the two endpoints are never connected simultaneously to a network and therefore cannot establish a connection.

b. Resilience to disconnections.
   Even if we assume that, for certain variable and unpredictable periods of time, the Client and the Server will be simultaneously connected to the network and therefore transfer data directly, this assumption can become false at any time. The two endpoints should be able to work without being affected by disconnections.

c. Minimize the processing power and transmitting time of the client.
   In case the data throughput provided from the Server or the network is far below the capacity of the Client, this will be forced to keep its receiving circuits (i.e.: a radio link) on for a unnecessary period of time, thus wasting power.
   In this situation, it is advisable to cache the data in the network and collect it later with a higher throughput.

d. Reduce the load of the client, in case of periodical repeated requests. In case the Client needs to periodically poll a Server (e.g.: to get readings from a sensor), it is a good idea to Delegate to the network itself the task to place the requests to the Server and to store the data.

2. PHILOSOPHY

This chapter gives some general concepts and guidelines about the MPTCP specification and implementation to better understand its operations.

2.1 Key concepts of MPTCP

During the design of the protocol, we tried to focus on the following key concepts.

a. “Intelligent Network, Simple Clients”.
   Our protocol aim to give an implicit intelligence to the network, by allowing the storage of information in the network nodes themselves without the need of level-5 proxies and enforcing end-to-end cryptography.

b. “Protect Against The Network”.
   Since we are delegating the data storage to the network itself, it is important to protect the data from malicious attacks. The only entity allowed to get the content is the Client, so a secure end-to-end encryption should be performed.

2.2 Implementation guidelines

This document describes the operations of the protocol and the programmable interfaces that an implementation must provide to the upper layer.

It is possible to implement the protocol in three ways: using “raw” sockets directly inside the user software (if the Operating System allows this), writing an external library or developing a kernel module that will handle the MPTCP protocol when opening a new socket. Each of these approaches offers advantages and issues.

In the paragraph 4.8 we describe the high layer interfaces that a MPTCP implementation must support, anyway the actual MPTCP API functions could differ because of the differences between operating systems.

2.3 Application examples

To further clarify the purposes of the protocol, we point out some typical usage of MPTCP.

a. Sensor Networks.
   Imagine a scenario with “sensor motes” deployed on the field and a mobile device to collect the data periodically. MPTCP provides two solutions for this scenarios: the sensors could be programmed to send their reading to a Meeting Place so that the user will be able to collect them later. Alternatively, the user application can delegate the MP to collect the data periodically from each sensor and store it locally.

b. Requests for time-consuming services.
   For example: SQL over MPTCP. A Client that requests a complex database operation to the Server could ask to store the data on a meeting place then disconnect from the network. Later, the Client will connect again to the network and download the data from the meeting place.

c. Large Requests to Servers with low Bandwidth.
   For example: HTTP over MPTCP. A client that wants to download a

large file from a slow server could delegate a meeting place to download the file then disconnect and collect it later.

d. Many requests, one collection.
   For example: SNMP over MPTCP. A client that wants to get a SNMP reading from a remote host every 5 minutes for many days could delegate a meeting place to do so, and then collect all the data together later.

e. Implicit redundancy, keeping a local copy of the data.
   For example: database replication. A mobile device accessing a remote database could request a replication on the local meeting place, to be able to work even in case of a failure of the remote network link.

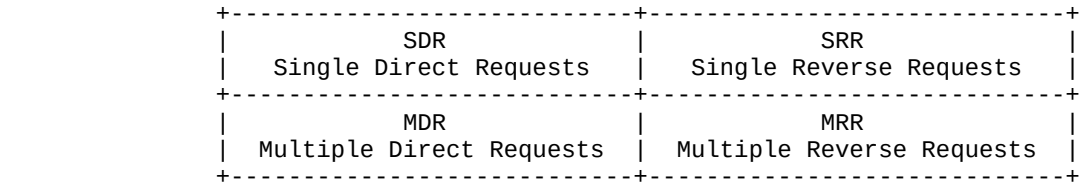
f. Reduce bandwidth requirements for external networks.
   For example: file repositories. By keeping a copy of the data on the local meeting place, it is possible to reduce the demand for external connectivity.

g. Transmitting data between mobile users.
   For example: Instant Messaging in Ad Hoc networks. A meeting place could be used as a temporary base point to transfer data between two mobile user that can be disconnected at any time.

3. OPERATION OF THE ALGORITHM

3.1 Types of Connections

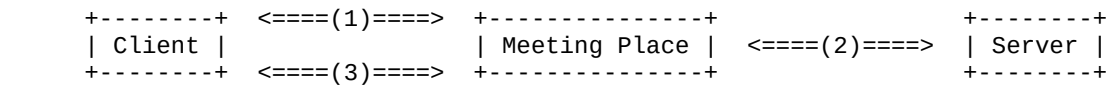
In MPTCP four different types of requests are defined, they can be classified as follows:



The first letter in the acronyms specifies the number of requests to be carried out. A single request consists in only one request from the Client to the Server and its relative answer. Instead, with a multiple request a Client could ask to a Meeting Place to collect data from a Server many times at specified intervals.

The second letter defines the type of connection ongoing between the Client (C), the Meeting Place (MP) and the Server (S), as follows.

- Direct Requests: The client delegates the MP to get the data (one time or more) from a remote server and store it. The dialogue between the 3 actors assumes the following form:

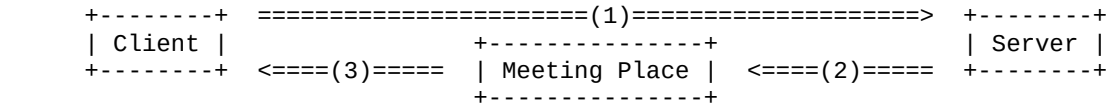


1) C to MP: “Please send my payload to S”
   MP to C: “Ok”
2) MP to S: “This is my payload”
   S to MP: “This is my data”

Later...

3) C to MP: “Please send me the data waiting for me”
   MP to C: “This is your data”

- Reverse Requests: The client directly connects to the Server and ask to send the answer to its request to a Meeting Place. Later on, the client will get the data from the MP.



1) C to S: “Please send the answer to this request to MP”
   S to C: “Ok”
2) S to MP: “This is data for C”
   MP to C: “Ok”

Later...

3) C to MP: “Please send me the data waiting for me”
   MP to C: “This is your data”

3.2 Authentication procedures

In this paragraph we describe the security mechanisms integrated in MPTCP in order to provide the identification of the “actors” involved and the secrecy of data.

MPTCP uses a mechanism based on asymmetric cryptography and certificates in the X.509 standard. MPTCP specification does not describe how certificates are signed, distributed and revoked, so an external PKI becomes required.

For the purposes of this document, we assume that every node in the system has the following elements:

- a pair of private and public key, generated with a chosen algorithm and key length. We will call them {PRIVc, PUBc} for the Client, {PRIVmp, PUBmp} for the MP and {PRIVs, PUBs} for the Server.
- a X.509 certificate by which a commonly trusted Certification Authority (CA) signs the public key of the node. We will call them CRTc for the Client, CRTmp for the MP and CRTs for the Server.
- the self-signed certificate of the CA.

MPTCP uses the CRT for both authentication and authorization, since no account names nor password are defined.

Every single MPTCP operation has a “security level” specified by the variable SeLvL contained in the head_crypt subheader. The user can decide how strict the security checks should be by modifying its value. A SeLvL of zero means that cryptography is completely disabled. Higher levels provide complete mutual authentication between nodes.

In the following paragraphs the steps performed for Direct and Reverse requests are described.

3.2.1 Direct Requests

1. The Client sends a SDR/MDR packet to the MP attaching CRTc by using a mptcp_auth header. Optionally for SeLvL >= 2, MP sends back CRTmp to C to ensure mutual authentication.

2. MP sends a SDR/MDR request to the Server attaching CRTc. For SeLvL >= 3, MP attaches also CRTmp and receives CRTs from the Server.

3. The Server sends to MP the data encrypted with the public key of the Client.

4. The Client requests data from the MP sending CRTc and, for SeLvL >= 2, requesting CRTmp.

We define the following security levels:

SeLvL	Description
0x0	No authentication at all. Cryptography disabled.
0x1	Client sends CRTc to the MP.
0x2	As above, plus MP sends CRTmp to the Client.
0x3	As above, plus MP sends CRTmp to the Server and receives CRTs from the Server.

3.2.2 Reverse Requests

1. The Client sends a SRR/MRR packet to the Server attaching CRTc by using a mptcp_auth header. Client and Server exchange then the pre-secrets keys in order to create a session key.

At this point, the Client and the Server are mutually identified and they share the knowledge of a secret key.

2. The Server sends to the MP: its own certificate CRTs and data encrypted using the shared key (that MP ignores). Optionally, for SeLvL >= 3, MP sends back CRTmp to S to ensure to be trustworthy.

3. The Client gets the data from the MP by using its own CRTc as identifier. Optionally, for the SeLvL 4, MP sends its own CRT to C.

We define the following security levels:

SeLvL	Description
0x0	No authentication at all. Cryptography disabled.
0x1	Client sends CRTc to the Server.
0x2	As above, plus Server sends CRTs to the Client.
0x3	As above, plus MP sends CRTmp to the Server.
0x4	As above, plus MP sends CRTmp to the Client.

By using these security measures we can avoid the following attack patterns:

a. A malicious node could “spoof” the IP Address and take the place of the actual MP to direct towards itself the traffic coming from the Server. This attack is not feasible because:
   - the malicious node cannot read the data since it was encrypted with a session key known only to the Server and the Client.
   - with SeLvL >= 3, MP must send CRTmp to the Server.
b. Even if an eavesdropper would capture all the data send in the three steps above, it would be impossible for it to find the session key.
c. In case the Server or the MP or both would become compromised, MPTCP ensures “perfect forward secrecy” of the data stored on the MP’s disk, since they are encrypted with different and unknown session keys.

4. FUNCTIONAL SPECIFICATION

This chapter formally describes the format of the packets, in order to make different MPTCP implementation fully compatibles. Please note that the ordering of the bytes “on the wire” follows the standard used in the lower layer and that in this document we keep a data alignment of 32 bits words (4 octets) for presentation purposes.

4.1 Header format

The size and number of fields in the header of the MPTCP protocol is not constant but can vary to adapt the protocol to each situation while keeping the overhead low.

The fields allowed in the MPTCP header are classified in “options groups” that we will call “subheaders” in the remainder of this document.

Please note the following constraints:
1. All the subheaders are optional except head_general and one between head_thirdaddress_v4 or head_thirdaddress_v6, that must always be present..

2. The size of each subheader is fixed and specified in this document
3. The order of the subheaders is not relevant, except head_general that must always appear as first.
4. All the subheaders except head_general can be present only in the first packet of a session.
5. Each of the subheader types can appear only once in the list except head_thirdaddress_v4 and head_thirdaddress_v6, that can appear an arbitrary number of times to indicate that this session will make simultaneous use of more than one server or meeting place by sending a copy of the payload to each of them.
6. The NextHeader value of the last subheader must be zero to indicate that the payload is following.
7. The subheader head_answer can be used by a MP to communicate warning or errors to the C.

The subheaders present in a packet take the form of a “linked list”, where each one is identified by the value in the “next header” field of the previous subheader. Note that this technique is similar to the IPv6 protocol, described in RFC2460.

In the current protocol description, seven groups are present.

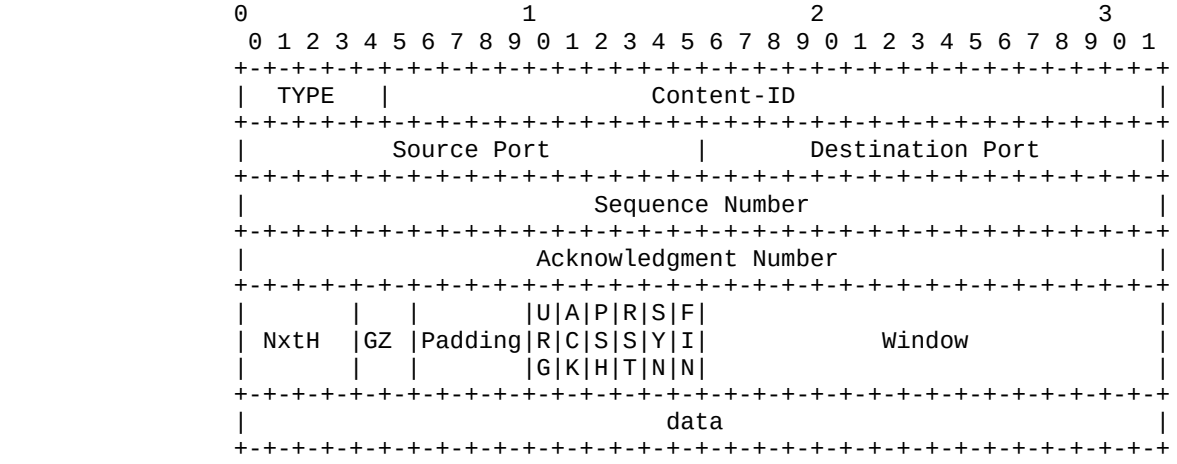
NextHeader Value	Name
n/a	head_general
0x1	head_thirdaddress_v6
0x2	head_thirdaddress_v4
0x3	head_ttl
0x4	head_crypt
0x5	head_multiple
0x6	head_answer

We will describe them in detail in the following paragraphs.

4.1.1 The MPTCP head_general Subheader

This header is mandatory and must appear in each packet of the session, always in the first position of the subheader list. The first bit of the “TYPE” field must be the first bit of the underlying network layer.

The head_general subheader contains information about the content-id, the source and destination ports for layer-3 multiplexing, flow control information, session establishment and reset information and congestion control data.



Fields specification:

- TYPE: 5 bits. Defines the type of request issued by the client, as described in section 3.1 of this document. The allowed values are:

Hex Value	Mnemonic	Description
0x1	SDR	Single Direct Request
0x2	MDR	Multiple Direct Request
0x3	SRR	Single Reverse Request
0x4	MRR	Multiple Reverse Request
0x5	GET	Get Content
0x6	NFY	Notify Content Availability/Errors
0x7	SIG	Signalling Between MPs

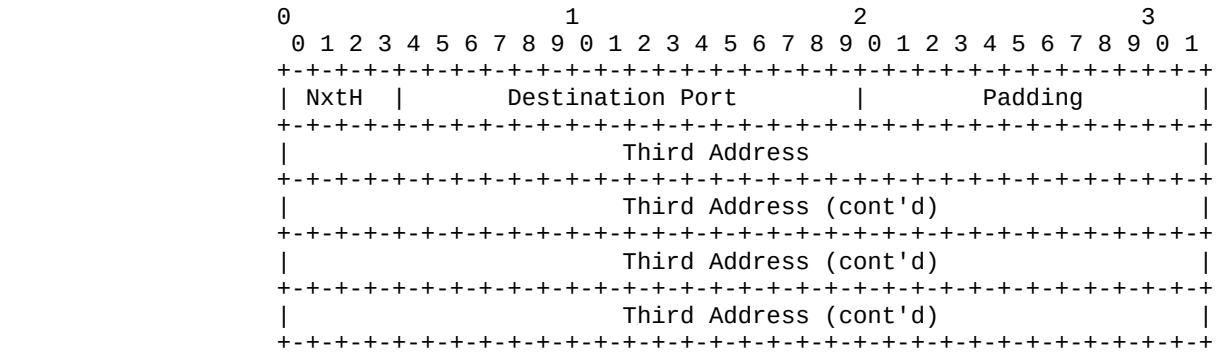
- CONTENT-ID: 27 bits. It is a unique identifier for the information that the client is requesting.

- GZ: 2 bits. For requests packets, defines if the payload data should be compressed in the answer. In answer packets, it defines if the payload is compressed. The compression algorithm is described the standard GZIP algorithm.

Hex Value	Mnemonic	Description
0x0	OFF	Compression is always off
0x1	ON	Compression is always on
0x2	AUTO	Compression is adaptive

4.1.2 The MPTCP head_thirdaddress_v6 Subheader

For direct requests, this header specifies the Server address to store the data on. For reverse requests, it specifies the Meeting Place address. It is called “third address” because it completes the set containing the Source and Destination addresses.



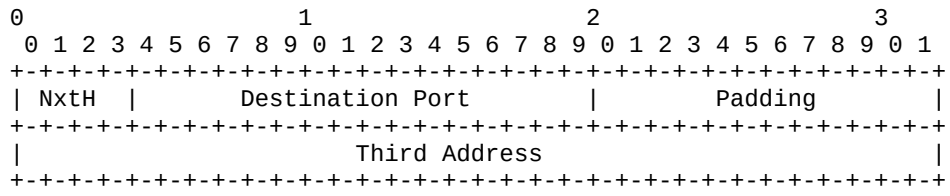
Fields specification:

- Destination Port: 16 bits. It specifies the layer-3 endpoint on the Server or the Meeting Place.

- Third Address: 128 bits. The IPv6 address of the Server or Meeting Place.

4.1.3 The MPTCP head_thirdaddress_v4 Subheader

For direct requests, this header specifies the Server address to store the data on. For reverse requests, it specifies the Meeting Place address. It is called “third address” because it completes the set containing the Source and Destination addresses.



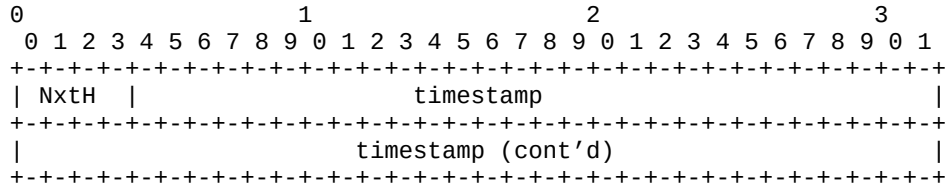
Fields specification:

- Destination Port: 16 bits. It specifies the layer-3 endpoint on the Server or the Meeting Place.

- Third Address: 32 bits. The IPv4 address of the Server or Meeting Place.

4.1.4 The MPTCP head_ttl Subheader

This subheader can be included by the client to specify how long the data should be stored on the Meeting Place. This Time-To-Live value is expressed as the number of integer seconds since the Unix Epoch (01/01/1970 at 0:00GMT). Using this standard and allocating 60 bits for the timestamp field, we are able to describe time without compatibility problems in the future. Also, by using this approach, the 64bit implementations could just copy the system timestamp and removing the 4 most significant bits.

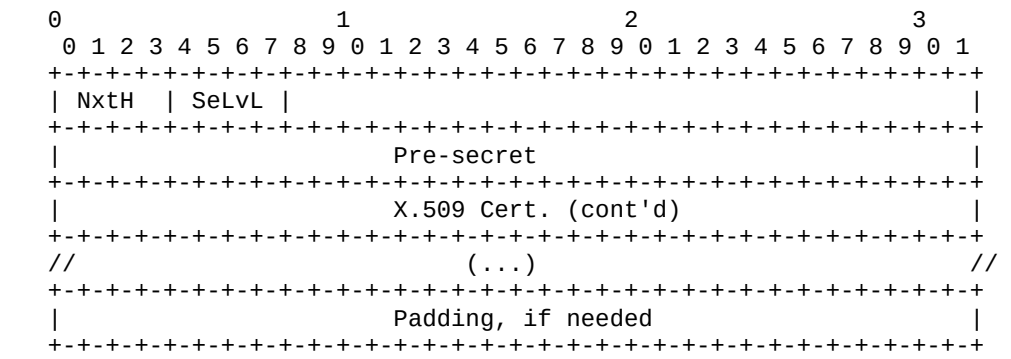


Fields specification:

- timestamp: 60 bits. Number of seconds since Unix Epoch.

4.1.5 The MPTCP head_crypt Subheader

This header specifies the details for the data encryption, such as the encryption algorithm and the X.509 certificate, that contains the node's public key.



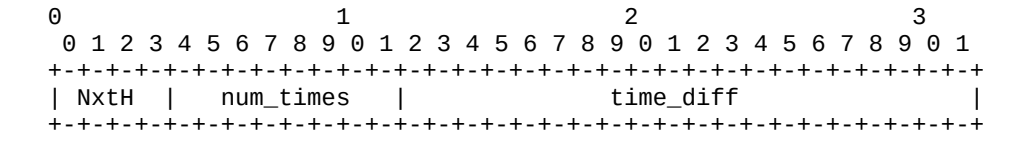
Fields specification:

- SeLvL: 4 bits. Define the security level of the operation, with the values described in paragraphs 3.2.1 and 3.2.2.

- Public Key: contains the X.509 Certificate with the public key of the node, its technical details (cipersuite used, key length, hashing algorithm, etc) and a "common name".

4.1.7 The MPTCP head_multiple Subheader

This subheader can be used in Multiple requests (see paragraph 3.1) to specify the number of requests to be done and the time distance between them.

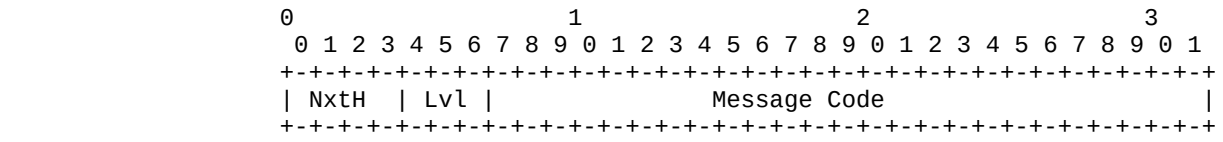


Fields specification:

- num_times: 8 bits. It express the total number of requests that must be performed.
- time_diff: 20 bits. It is the number of seconds between each request. Thus, the maximum delay between two consecutive requests is 2^20 seconds (more than 12 days).

4.1.8 The MPTCP head_answer Subheader

This subheader is used by the MP to communicate Errors, Warnings and informative messages to the C. Eight "error levels" are defined to specify the seriousness of a problem. Higher levels are simply informative and do not affect the operation of the protocol.
A Warning is considered a non-blocking error given to the C to signal a unusual condition (i.e. space quota almost full). An Error requires the closing of the connection, with the use of the FIN and RST fields.



Fields specification:

- Lvl: 8 bits. Specifies the level of the table, according to:

Level	Name
0x0	Blocking errors
0x1	“Strong” warnings
0x2	“Soft” warnings
0x3	Notice
0x4	Debug

- Message Code: 25 bits. A code associated with the specific situation.

Code	Description
0x0	General Error, no other details given
0x1	User quota full
0x2	High user quota usage
0x3	Storage-related general problem

4.2 Establishing a connection

The MPTCP behaviour in point-to-point connections (e.g. from the Client to the Meeting Place) follows the TCP state machine as described in RFC 793.

To establish a new connection, a "three-way handshake" procedure is initiated by one end-point. This consists in sending a packet with the SYN flag set in the head_general header.

Please note that MPTCP specifications allow the node that actively opens the connection to send optional subheaders in the same SYN packet. Another option is waiting for the connection to be in the ESTABLISHED state before sending any optional data.

4.3 Closing a connection

As in TCP, CLOSE is an operation meaning "I have no more data to send".

Since MPTCP imply a full-duplex connection, the user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also.

The three cases of:
1) User telling the TCP to CLOSE the connection,
2) Users sending a FIN control signal and
3) Both users sending a CLOSE command simultaneously
are discussed in section 3.5 of RFC 793, and MPTCP strictly comply to the TCP specifications.

4.4 Flow and Congestion control

MPTCP provides a means for the receiver to govern the amount of data that the sender writes on the communication channel. This is achieved by the use

of a "window" indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

MPTCP strictly comply with the TCP specification for the sliding window internals, as described in RFC 793.

4.5 Payload compression

MPTCP features an adaptive compression algorithm that tries to optimize the compression ratio according with the type of data transmitted.
In the case the data is predominantly uncompressible or pre-compressed (eg: FTP or HTTP transfer of a large compressed archive) the compression can only introduce a computational overhead without giving any significant advantage in size or transmission time.

With adaptive compression, MPTCP will periodically sample the compression process to measure its efficiency. If the data being sent over the tunnel is already compressed, the compression efficiency will be very low and the algorithm will disable the compression for a period of time until the next re-sample test.

This behaviour can be controlled using the parameter GZ in head_general, as follows:

Hex Value	Mnemonic	Description
0x0	OFF	Compression is always off
0x1	ON	Compression is always on
0x2	AUTO	Compression is adaptive

The sampling rate and the size “gain” threshold are not specified by the protocol but are implementation-dependant. It is advisable to let the user “tune” these values if required.

4.6 Interfaces

As in every transport protocol, there are two types of interfaces to describe: the user/MPTCP and the MPTCP/lower-level interface. We specify only the first since the latter is dependant on the lower level protocol and the system in which MPTCP is deployed.

Following the example of the standard TCP, we identify the following abstract user procedures:

- OPEN
  Parameters: local port, foreign socket, active/passive, timeout
  Description: If “active”, this command starts the creation of a new connection, in “passive” mode the system listens for incoming connection on the specified port. It returns a reference to the newly created connection.

- SEND
  Parameters: connection identifier, buffer address, byte count.
  Description: Copy the specified amount of bytes from the buffer starting address to the socket specified.

- RECEIVE
  Parameters: connection identifier, buffer address, byte count
  Description: Read the specified number of bytes from the socket and copy them to the buffer.

- CLOSE
  Parameters: connection identifier
  Description: This command causes the connection specified to be
               closed, if open.

- STATUS
  Parameters: connection identifier
  Description: This command returns the status of the specified socket
               to the user.

- ABORT
  Parameters: connection identifier
  Description: This causes all pending SEND and RECEIVE to be aborted.
               Also, a RESET message is sent to the other side of the
               connection.

We also introduce the following abstract interfaces:

- SET_CONNECTION_OPTION
  Parameters: connection identifier, parameter name, parameter value
  Description: This command set a value for the specified parameter,
               inserting a new subheader in the following packet if
               needed.

- GET_CONNECTION_OPTION
  Parameters: connection identifier, parameter name, parameter value
  Description: With this command the user gets the current value of a
               given option sent in the packets of the specified
               connection.

5. BETWEEN-MEETING-PLACES OPERATIONS

MPTCP includes also a sub-layer that the MPs can use to communicate between them in order to manage the requests coming from the Clients and the data being sent by the Servers.

These “between-meeting-places” operations include:

1. Synchronization, to support redundancy,
2. Failover, in case a MP suddenly become unreachable,
3. Routing of request between MPs.

We plan to describe this communication sub-layer specifically in a further document.

GLOSSARY

In this section we report the meaning of the main abbreviations used in the document.

C	Client
S	Server
MP	Meeting Place
NxTH	Next Header
SDR	Single Direct Request
SRR	Single Reverse Request
MDR	Multiple Direct Requests
MRR	Multiple Reverse Requests
PRIVc	Private key of the Client
PRIVmp	Private key of the Meeting Plac
PRIVs	Private key of the Server
PUBc	Public key of the Client
PUBmp	Public key of the Meeting Place
PUBs	Public key of the Server
CRTc	X.509 Certificate for the Client
CRTmp	X.509 Certificate for the Meeting Place
CRTs	X.509 Certificate for the Server