Supporting COTS Applications in Tactical Edge Networks

Alessandro Morelli[1,2], Ralph Kohler[3], Cesare Stefanelli[2], Niranjan Suri[1,3], Mauro Tortonesi[2]
[1]Florida Institute for Human & Machine Cognition (IHMC), Pensacola, FL, USA
[2]Department of Engineering, University of Ferrara, Ferrara, Italy
[3]U.S. Air Force Research Laboratory, Rome, NY, USA
[4]U.S. Army Research Laboratory, Adelphi, MD, USA

***Abstract*—Military applications are relying more and more on Commercial Off-The-Shelf (COTS) hardware and software to build their information infrastructure. However, many COTS applications have been designed for wired infrastructure networks, where bandwidth and latency are not a problem, and exhibit severe performance problems when deployed in Tactical Edge Networks. To address the issues that traditional transport protocols, such as TCP, exhibit in this environment, researchers proposed several transport protocol solutions specifically developed for Tactical Edge Networks. However, these solutions require modifications to the applications in order to be adopted. This paper presents the Agile Computing Middleware (ACM) NetProxy: a proxy-based approach to support the deployment of COTS applications on Tactical Edge Networks. NetProxy transparently intercepts the applications' TCP and UDP traffic and conveys it over Mockets-based connections, while also providing other useful features such as stream compression. We tested NetProxy in an emulated environment and the results demonstrate that, besides enabling the deployment of COTS applications, NetProxy is also capable of improving their performance.**

***Index Terms*—SOA, Tactical Edge Networks, Network Proxies, Legacy Application Support.**

I. Introduction

In recent years, military applications have increasingly relied on Commercial Off-The-Shelf (COTS) hardware and software to build their information infrastructure. COTS components enable the rapid development of new functions with relatively low costs, and therefore are being leveraged to build applications in Tactical Edge Networks.

Many COTS applications have been designed for wired infrastructure networks, where bandwidth and latency are not a problem. However, tactical missions in urban environments or on the battlefield require the support of services provided by the military applications and systems, which consequently need to fully operate in Tactical Edge Networks. These networks are highly heterogeneous and dynamic environments, generally composed of a combination of Mobile Ad-hoc NETworks (MANETs, which are prevalent with dismounted operations, convoys of ground vehicles, etc.) and Wireless Sensor Networks, and are characterized by relatively frequent disconnections, low bandwidth, and high latency. The scenarios typically involve many concurrently running applications, each with different functional requirements, which run essential tasks and compete for the scarce bandwidth and computational resources. These characteristics make COTS applications, which rely on TCP-based reliable stream communications, perform very poorly on Tactical Networks.

To address the issues that TCP exhibits in MANETs, researchers have proposed many solutions [1]. Among the proposed solutions, the Mockets communication middleware [2], whose design and implementation are specifically motivated by the needs of tactical military information networks, represents a particularly relevant solution. However, Mockets provides a richer programming model and an extended API that requires modifications to the applications, explicitly substituting TCP system calls with the equivalent function calls to take advantage of the Mockets API. Therefore, to enable application reuse, it is necessary to develop a solution to bridge the gap between applications and the communication middleware.

This paper presents the Agile Computing Middleware (ACM) NetProxy: a different, proxy-based approach to support the deployment of COTS applications on Tactical Edge Networks. Its main functionality is to transparently intercept outgoing TCP and UDP packets, extract data from them and convey it over Mockets-based connections. At the receiver side, another instance of the ACM NetProxy provides the reverse operation, so that the result is completely transparent to the communicating applications. Transparency is the key factor, since it allows access to Mockets' features and enhancements without any modification being required to deployed applications. The ACM NetProxy also provides other useful functionality, such as stream compression, buffering, traffic forwarding, support for disconnection, and logging.

The NetProxy is an essential component of the Agile Computing Middleware, a software infrastructure designed in the context of a broader perspective research project that addresses several communication problems within the tactical military environment. The goals of this middleware are the opportunistic discovery, manipulation, and exploitation of available computing and communication resources, in order to improve capability, performance, efficiency, fault-tolerance, and survivability.

The ACM NetProxy is realized as a user-space C++ application, thereby not requiring significant changes to the operating system, and runs on both the Windows and Linux operating systems.

We tested the NetProxy and the results showed that, in most of the examined scenarios, the overhead due to the use of the proxy is fully compensated by the performance improvements achieved with Mockets. Transparent stream compression further enhances the performance when transferring information encoded with verbose protocols such as XML.

II. Communication Issues in Tactical Networks

Unlike wired Internet environments, applications running in Tactical Edge Networks must deal with disconnections for multiple reasons (nodes could move and fall out of communications range, networks may become partitioned, client devices may need to be temporarily shut down or to stop network activity). They need to implement resilient behavior, allowing temporary disconnections without interrupting service sessions.

In this scenario, the communication semantics proposed by commonly used transport protocols, such as TCP, which provide reliable and sequenced delivery of a stream of data, are inefficient [3]. To minimize communication overhead, applications should take advantage of a wider range of delivery semantics, using reliable and sequenced delivery only when absolutely necessary.

In addition, TCP was designed to operate in wired networks, where it is reasonable to assume that any packet loss is due to congestion. In Tactical Edge Networks, many problems (such as a higher channel error-rate, medium contention/collision, and node mobility) can occur, increasing the likelihood of losing packets [4]. TCP reacts to packet losses by applying congestion avoidance algorithms, which decrease transmission rate, in order to reduce potential collisions. This approach leaves the wireless channel underutilized and generates traffic flows with lower throughput and higher latency than the network is actually capable of delivering. Therefore, the use of TCP over tactical edge networks is inappropriate for tactical applications that have any requirements in terms of latency or throughput, since TCP underperforms on these networks.

Finally, the limited bandwidth available for communications in the tactical environment requires applications to be as efficient as possible. Compression of message content before their transmission is one technique to reduce bandwidth consumption. But, it might increase latency and, with resource constrained clients, may be too computationally expensive.

III. Bridging the Gap: Running COTS Applications on Top of Mockets

The Mockets middleware is a solution to the issues that the standard Internet Protocols stack exhibits when used over Tactical Edge Networks. The Mockets middleware provides mechanisms for detecting connection loss, allowing applications to monitor network performance, provides flexible buffering, session mobility, and dynamic service rebinding, and supports policy-based control over application bandwidth utilization. Also, Mockets is conceived to handle most of the characteristics of Tactical Edge Networks that cause TCP to perform badly.

Using Mockets, instead of TCP, permits applications to obtain higher throughput, better bandwidth usage, and a lower latency when communicating over tactical networks. With this approach, COTS applications might still satisfy their functional requirements, even when running on tactical edge networks.

However, running COTS applications on top of Mockets instead of TCP is often impossible or impractical, as it requires modifications to the applications' source code. This approach cannot be applied to third party software and also not to legacy applications when their source code is not available anymore or if the modifications would be too expensive.

The realization of a proxy component that provides transparent remapping of the TCP protocol to Mockets before the data is sent out onto the network represents a much more feasible approach. At the receiver side, another instance of that same component is necessary to translate received data back to TCP.

This is the approach explored in this paper, which presents the ACM NetProxy: a configurable network proxy that was developed with the primary purpose of performing the translation from the TCP protocol to Mockets and vice-versa. The NetProxy is flexible enough to handle other protocol remappings and provides a number of other capabilities as described below.

IV. ACM NetProxy

The NetProxy is a component that transparently intercepts any network traffic that is generated by applications, analyzes it, and then makes decisions regarding how to handle the traffic. The actions it takes are transparent to the parts involved in the communication, thus making this approach suited for legacy and third-party applications.

The initial primary objective of the NetProxy is to convey data transported via TCP over Mockets. The proposed solution consists of using the NetProxy to capture TCP packets, extract data from them, and pass the data to the Mockets middleware, which will handle efficient delivery to the destination. In order to maintain the transparency with communicating applications, the receiver side uses another instance of the NetProxy that performs the inverse conversion. Therefore, the architecture comprises pairwise communication between proxies located at endpoints that execute applications.

Fig. 1 shows a simple scenario: Application A on Node A needs to communicate with Application B, which is waiting for requests on Node B. At the beginning, Application A attempts to open a TCP connection with the Application B. This connection request is intercepted by the local instance of the NetProxy, which locally responds to the connection request (note that TCP packets are addressed to the remote application and the proxy transparently intercepts them and replies as if it was the remote application). At this point, NetProxy A opens a new logical connection with NetProxy B using a Mockets connection (if the latter was not active, NetProxy A establishes it first). When NetProxy B receives the request for the logical connection it will, in turn, open a local TCP connection with Application B. Once all the connections are setup, NetProxy A can finally send data to Node B. On that side of the communication, NetProxy B will receive data in the correct order from the Mockets middleware and forward it to Application B using the local TCP connection. From this moment on, until the communication is terminated, applications can exchange data in both directions. It is relevant to note that the connection between the proxies is completely independent from the connections opened and closed by the applications which are taking advantage of the connectivity and the features provided by the NetProxy.

The NetProxy currently implements some additional functionality such as packet forwarding, packet dropping, support for temporary disconnection, stream compression, and network activity logging. By providing features at the proxy level, there is no need for every application to reimplement those features. Therefore, the NetProxy is a quick and effective approach to add functionality to COTS and other legacy applications.

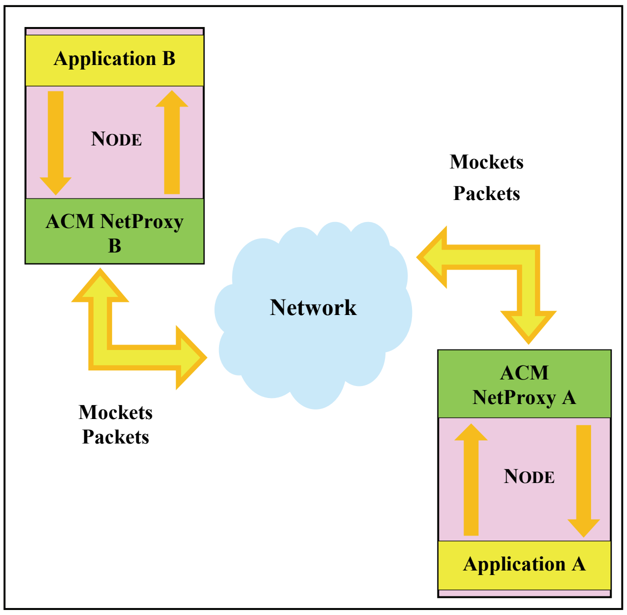

Figure 1 - Two nodes communicating via Mockets with ACM NetProxy

Users can configure the NetProxy with policies that target specific communications (depending on the type of traffic, on the protocol being utilized, or on the source/destination addresses). These policies can be dynamically updated whenever required without affecting running applications. This provides a standard and centralized configuration point for all the functionalities provided by the proxy, allowing for faster and easier management of the communication flows in the network.

All of these capabilities do introduce an overhead into the communication pathway, as a result of the time that the proxy spends analyzing and processing the packets. This overhead is intrinsic to any proxy based approach. However, the performance gains that result from using more efficient and target-appropriate protocols result in the overall performance actually improving. The additional capabilities described above further strengthen the argument for a proxy-based approach.

V. Architecture and Implementation

The ACM NetProxy is realized as an application that executes in user space and may be installed as a service or a daemon process. NetProxy supports multiple deployment configurations. The most common configuration is to install NetProxy on the same physical host as the legacy applications. In this case, NetProxy relies on a *virtual network interface* to capture and inject packets to the network stack of the system. An example is the Tun/Tap driver for UNIX (the Tap32 driver for Win32). The system has to be configured for applications to send packets over this v*irtual network interface*, on which the NetProxy continuously listens for incoming data. An application which needs to open a connection with a remote node uses standard TCP system calls to generate TCP packets addressed to the destination. The NetProxy intercepts those packets, processes them, extracts the data segments, and delivers them to the Mockets middleware, which will undertake the task of sending the data to destination.

Fig. 2 shows the architecture of the NetProxy. Applications that communicate through the NetProxy are configured to send data over a virtual network interface. The *Virtual Interface Adapter* is the class responsible for reading and writing packets from and to that interface. The adapter simply reads/writes Ethernet data frames, thereby allowing it to support different protocols.

The *Receiver Thread* is the execution unit responsible for receiving packets from the virtual network interface through the Virtual Interface Adapter. The Receiver Thread continuously listens on the virtual interface for new packets and processes them. Depending on the type of packet that receives, the thread will behave differently. If a UDP unicast packet is received, the thread simply forwards it to the Mocket (or Socket) Connector. However, Mockets and Sockets are not the only available middleware to which the NetProxy can remap a communication. In fact, in case of UDP multicast packets, the NetProxy can be configured to leverage DisService to send them to multiple destinations [5]. DisService is also a component of the Agile Computing Middleware and an information dissemination service for tactical edge networks that provides a reliable multicast capability, among other features. The characteristics of such a service make it suitable for dispatching data to multiple nodes in the network.

When a TCP packet is received, the Receiver Thread processes it and performs different actions depending on the flags set in the packet. Usually, data is buffered in memory and referenced by an entry in the TCP Connections Table. Therefore, the Receiver Thread implements the TCP protocol, adapting it to the additional functionalities implemented in the NetProxy. Also, when applications open a new connection, the Receiver Thread is responsible for sending a message to the proxy running on the destination node to open a new logical connection on the top of the running Mockets instance.

The status of both local and remote connections is maintained by the *TCP Connections Table*. For every local TCP connection, this table contains an entry with a unique pair of identifiers to couple that connection with the respective one at remote side, where a symmetric pair of identifiers is also replicated. Every time the proxies exchange packets, two fields in the messages uniquely identify a specific pair, to discriminate data belonging to different logical connections. In addition to connection identifiers, the entries of the TCP Connection Table store all the necessary information to keep the status of the local TCP connection (i.e., timeouts, counters, etc.), incoming/outgoing buffered data, remote proxy address and local and remote IP addresses and port numbers.

The main task of the *Remote Transmitter Thread* is to read buffered data from any entry in the TCP Connection Table and to send it to destination, through the Socket/Mocket Connector. This latter component implements all the functions to send messages to remote proxies. Therefore, this component is the only one which interacts with the traditional Socket API or the Mockets middleware. Note that data is not passed directly to the Connector: if compression is enabled, data is compressed before being forwarded to the Connector for transmission. *Plain/Zlib/LZMA Writer* is the component responsible for data compression. In a similar manner, the *Plain/Zlib/LZMA Reader* decompresses received data. When an application closes its half of the connection with the proxy, the Remote Transmitter Thread is also responsible for notifying the remote proxy of the completed transmission.

The *Socket/Mocket Connection Thread* is also responsible for receiving proxy messages and performs different actions depending on the type of the received message. It can open new connections with local applications, close them, or leave the task to the Local Transmitter Thread. When the thread receives messages containing data, it behaves differently depending on the type of data. In case of unreliable/unsequenced data, the thread directly forwards it to the destination application through the Virtual Interface Adapter. Instead, if the data is reliable/sequenced, the thread buffers it in the respective entry of the TCP Connection Table. Finally, if data is compressed, the Plain/Zlib/LZMA Reader decompresses it in advance, so that the proxy can correctly handle the data in subsequent steps.

The main task of the *Local Transmitter Thread* is to forward sequenced data to local applications. Since data has to arrive to the destination in a sequenced and reliable manner, the thread creates and sends TCP packets over the Virtual Network Interface. Any information necessary for the TCP Protocol to behave properly is stored in the TCP Connections Table, which the thread updates as required. The thread also performs other actions required by the TCP protocol, such as packet retransmissions and handling TCP control packets such as SYN, FIN, RST, and ACK packets.

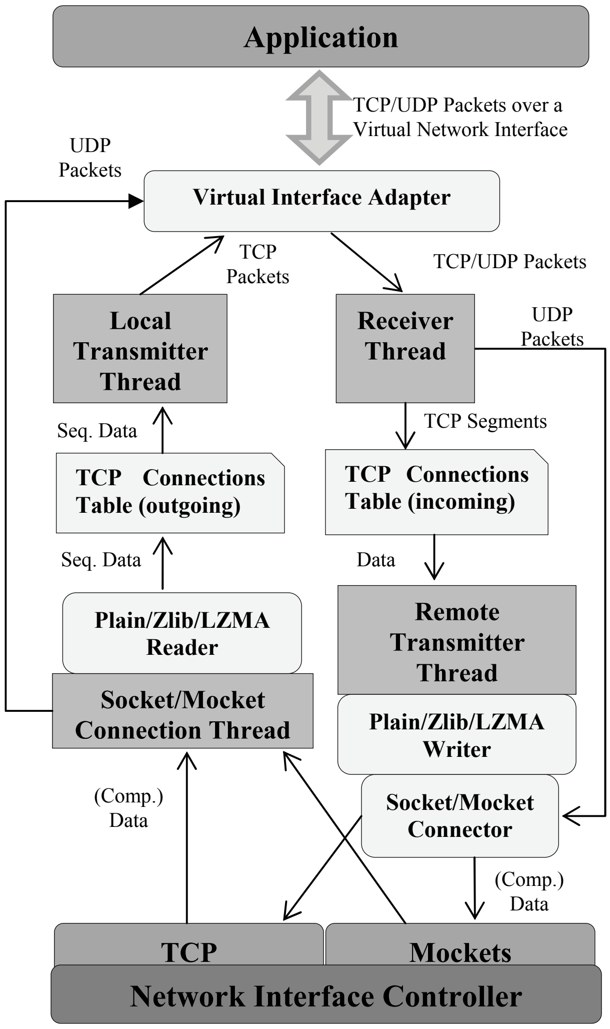

Figure 2 – Architecture of the ACM NetProxy

VI. Experimental Results

We tested the ACM NetProxy to evaluate its performance in an emulated environment that reproduced the operating conditions typically found in Tactical Networks. All the tests were run on the NOMADS Testbed, which comprises 96 servers, connected through a 100Mbps Ethernet LAN network. Hardware configuration of all the machines consists of HP DL140 Servers (Dual Xeon Dual Core CPUs at 3.06 GHz with 4 GB of RAM). The testbed uses an enhanced version of the MANE (Mobile Ad-hoc Network Emulator) software [6]. The testbed allows each link to be configured with different parameters for bandwidth, latency, and reliability. This allows the performance evaluation of different systems in a reproducible, laboratory controlled environment.

A. *Tactical Network Scenario*

The objective of the first test was to simulate the conditions of a realistic scenario of applications communicating data on a Tactical Network. Java applications were used to simulate traffic flows, which included HTTP requests as well as UDP streams working at the same time, therefore competing for the available network resources.

Two flows were HTTP requests, one for small objects of about 150KB each and the other one for larger objects, of about 3MB. Smaller objects are JPEG images and the Java client makes a request for such objects with a rate of 1 request per minute. On the contrary, larger objects are video sequences. The client repeats the request for a video every 10 minutes.

The remaining flows are two UDP streams. The first one simulates a video stream sent at the rate of 400Kbps. The second stream, instead, simulates the exchange of COT (Cursor-On-Target, a standard for exchanging location based data) information over XML messages. This stream consists of 4KB messages sent at a rate of 0.25Hz and 1KB messages sent at a rate of 6Hz. Total network bandwidth was configured to be 4Mbps. Three different reliability levels were selected – 95%, 90%, and 85% (respectively a PER - Packet Error Rate - of 5%, 10% and 15%). The operating system was Windows XP SP3. The version of JAVA and of the JVM was 1.6.0 update 27. Measurements were taken at the receiver side. At the level of the NetProxy, a fair policy to handle TCP and UDP flows was applied.

Results regarding HTTP streams show that the *throughput* measured using the NetProxy is *always higher* than the one measured using a direct TCP connection. Also, sending larger data streams allows the system to reach a higher level of performance: this is due to the time required for communication setup and to the time necessary for the protocol to reach optimal speed. In addition to throughput, another important result is the *success rate*, which is measured as the percentage of correctly completed communications. Results are very promising, especially for longer data streams, and show that the proxy-based solution also provides reliability improvements on a per-request base. Table I presents these results.

On the other hand, results regarding UDP streams are comparable. In fact, the NetProxy translates unreliable/unsequenced streams like UDP unicast to unreliable/unsequenced flows in Mockets, which does not entail any improvement. However, since performance is comparable, it is possible to conclude that the NetProxy overhead has minor consequences when applied to UDP traffic. Table II presents results obtained measuring the performance of UDP streams. It is important to notice the improved throughput that plain UDP seems to reach with 85% of reliability. Although the value is actually higher, we must consider that the four tests were run at the same time, thereby competing for the limited bandwidth. Using the NetProxy, HTTP requests run over Mockets, which allows a higher throughput compared to TCP and so a higher bandwidth usage. Therefore, even if unreliable unsequenced data flows might perform worse (which is reasonable to expect given that this type of flow carries less important traffic), total throughput is still much higher and available resources are better utilized.

Table I. Tactical Network Scenario Experiment - HTTP.

Rel.	Legacy (TCP/IP)				NetProxy (Mockets)			
	150 KB Images		3 MB Videos		150 KB Images		3 MB Videos	
	Data Rate KB/s	Succ. Rate %	Data Rate KB/s	Succ. Rate %	Data Rate KB/s	Succ. Rate %	Data Rate KB/s	Succ. Rate %
95%	27.48	63.64	59.37	50.00	96.59	66.67	242.07	100
90%	12.13	68.42	4.57	0.00	69.44	64.29	253.02	66.67
85%	7.94	58.82	1.29	0.00	61.91	65.00	242.09	66.67

Table II. Tactical Network Scenario Experiment - UDP.

Rel.	Legacy (UDP)			NetProxy (Mockets)		
	UDP Video	UDP COT XML Messages		UDP Video	UDP COT XML Messages	
	Data Rate KB/s	Data Rate KB/s	Succ. Rate %	Data Rate KB/s	Data Rate KB/s	Succ. Rate %
95%	30.40	4.06	58.03	30.61	4.08	58.33
90%	27.84	3.70	52.80	28.36	3.71	52.99
85%	28.81	3.73	53.35	25.88	3.24	46.31

B. *Compression*

This section presents the last set of results, which report on performance obtained with stream compression. The tests were run under both Windows XP SP3 (32-bit) and Ubuntu Linux v10.04 (32-bit). Type of data exchanged during the communication was XML over HTTP. The testbed was configured with a reliability of 95% and a bandwidth of 1Mbps. In order to have a better understanding of compression ratios and performance gains provided by different combinations of compression algorithms and levels, tests were conducted with several different XML files. Files differed in terms of the type of content and the size.

Table III shows results collected running tests under Windows XP and Table IV shows results of tests run under Ubuntu Linux. Values highlighted in bold typeface are the best and the worst results for each test. Under Windows, the gain provided by the use of Mockets, compared to TCP, is clear. Another fact that stands out pretty clearly is that, in most of the cases, Zlib performs better than LZMA. Although LZMA generally results in a higher compression ratio than Zlib, the added computational cost implied that, in most of the cases, using Zlib resulted in higher throughput. This result depends both on the characteristics of the connection and on the computing power available on the local node. At the moment, the optimization of the trade-off between compression ratio and local computation in order to reach the best performance is left to the user. Finally, it is noteworthy that, most of the time, direct TCP performs worse than the NetProxy using TCP. In fact, it is possible to configure the NetProxy to connect to the other proxies in the network via TCP; this feature is very useful to run tests to determine the introduced overhead. We chose not to report the data regarding the tests on the overhead introduced by the NetProxy in this paper since obtained results depend on many different factors (i.e., the computing resources on the nodes, the network bandwidth, the type of traffic, etc.) and further tests are necessary before we can present a clear picture of the situation. However, results reported in Tables III and IV support the claim that the introduced overhead is quite low. In fact, with the NetProxy configuration used, and for the type of traffic and network conditions considered, the buffering feature of the NetProxy compensated for the introduced overhead. For the experiment, the NetProxy was configured to turn off TCP's Nagle algorithm, hence buffering was performed only at the NetProxy level. This allowed the system to reduce the overall number of packets sent onto the network, when compared to standard Windows TCP implementation.

Table III. Compression Experiment – Win32.

	Windows XP SP3 32bit (values are end-to-end throughput in KB/s)								
	TCP	**NetProxy (TCP)**			**NetProxy (Mockets)**				
SIZE (Bytes)	plain	plain	zlib:1	lzma:1	plain	zlib:1	zlib:3	lzma:1	lzma:3
1133	**1.22**	1.64	2.12	1.71	37.08	**43.99**	29.38	21.44	13.76
2268	**1.98**	4.83	4.64	3.56	49.22	**66.22**	51.48	32.66	26.44
2615	**3.08**	5.71	7.33	3.67	52.14	**85.27**	46.91	44.18	30.74
3111	**3.44**	6.58	5.18	5.14	51.75	**79.21**	44.35	44.66	35.58
5287	**5.47**	15.16	14.08	9.02	47.09	**78.05**	48.46	70.31	49.20
8765	**7.33**	13.35	23.20	12.28	41.08	**127.28**	85.01	86.90	78.91
9570	**5.75**	16.19	25.18	12.64	46.66	**126.55**	93.17	107.27	81.59
116520	**18.57**	24.79	173.40	145.42	60.34	171.21	178.29	245.93	**265.96**
141349	25.05	**24.18**	75.66	103.19	74.72	113.27	135.61	**168.64**	159.96

Table IV. Compression Experiment – Linux32.

	Ubuntu Linux v10.04 32bit (values are end-to-end throughput in KB/s)								
	TCP	**NetProxy (TCP)**			**NetProxy (Mockets)**				
SIZE (Bytes)	plain	plain	zlib:1	lzma:1	plain	zlib:1	zlib:3	lzma:1	lzma:3
1133	**2.06**	7.13	6.80	7.05	34.33	**42.12**	40.94	17.91	7.51
2268	**5.77**	17.89	12.92	14.15	35.18	42.96	**46.07**	32.81	19.36
2615	**4.14**	15.74	15.20	15.04	48.23	69.35	**73.54**	34.49	14.34
3111	**7.01**	18.47	17.46	14.78	54.64	**71.03**	67.66	39.27	23.60
5287	**9.43**	36.95	31.82	23.53	58.98	137.19	**167.63**	61.33	26.47
8765	**19.48**	47.80	51.06	56.17	55.47	**208.39**	148.73	99.41	68.55
9570	**16.88**	47.76	90.09	46.59	49.85	**161.29**	130.47	110.06	75.98
116520	118.19	90.98	408.14	314.86	**65.52**	466.92	**730.82**	549.44	410.64
141349	94.03	94.15	212.27	**268.50**	**79.61**	199.50	222.03	235.30	174.02

With regards to the tests run under Linux, it is more difficult to derive general conclusions. For example, TCP performs better with larger files, or rather with longer data streams. The clearest example of this behavior is the result of the test run with the largest file (bottom line in Table IV), where compressing the stream with LZMA algorithm and using TCP to connect the two proxies produced the best results. This is due to the slow start of TCP, which does not allow good throughput in case of short data streams, although the Linux implementation of TCP seems able to reach a good performance level if streams are long enough. For this same reason, using the NetProxy connected via plain TCP to the remote proxy, and thus keeping the TCP connection open between subsequent HTTP requests, allows TCP to reach higher throughputs. This effect is more evident when small amounts of data are exchanged, because these are the cases when the TCP slow start algorithm impacts performances the most. Also, it is noteworthy that, while using higher compression levels under Windows was generally not good, under Linux many of the tests run indicate higher compression levels to be a good choice. In fact, in 5 of the 9 experiments, level 3 compression with the Zlib algorithm achieved higher throughput than any other algorithm-level combination. Note that this situation does not occur with LZMA algorithm, probably due to the fact that the latter requires more processing resources, thus making it disadvantageous to raise the compression level.

VII. Related Work

Many research projects attempted to develop transport layer protocols specifically suited for wireless environments [1] [2].

However, the most interesting of these proposals break API compatibility with TCP, thus forcing to change the applications' source code in order to be adopted.

Other proposals have attempted to retain API compatibility with TCP by developing communications middleware on top of existing network stacks. In particular, I-TCP [7], Mobile-TCP [8], and the Remote Sockets Architecture [9] address both the performance and the mobility issues in TCP by proposing proxy-based architectures. However, these proposals only focus on (partially) mitigating the problems of TCP, as they simply reroute connections through proxies deployed at the edge of the wireless and the wired portion of the network.

NetProxy goes beyond these proposals by realizing an extensible system that is capable of improving the performance of COTS applications in a transparent way.

VIII. Conclusions and Future Work

The initial results are very encouraging and show that the NetProxy works correctly in many different scenarios and with heavy workloads. In addition, in the tested configurations and scenarios, the performance gains as a result of using Mockets completely compensate for the overhead introduced by the NetProxy. In the future, we will perform more tests in order to be able to analyze the impact of the use of the NetProxy separately from the benefits provided by Mockets.

Several capabilities will be added to the current version of the ACM NetProxy. One, for example, is the possibility for the users to associate a priority to the traffic of a certain type or with some specific source-destination pairs. This way, it would be possible to specify the services that should receive available bandwidth first when there is insufficient bandwidth to serve all the applications in the network. A similar feature is the possibility to configure a filter in order to stop specified traffic flows when adverse conditions occur.

Furthermore, we are working on porting the ACM NetProxy to the Android operating system, thus enabling it to operate on mobile devices such as smartphones and tablet computers.

Finally, it would be interesting to compare the performance of the presented solution with other efforts which have been made in the same direction, such as the Bundle Protocol developed for DTNs under DARPA sponsorship, as well as some of the proposed modifications to the TCP protocol.

Acknowledgment

This effort has also been funded in part by the U.S. Air Force Research Laboratory under cooperative agreement FA8750-11-2-0095 and by the Office of Naval Research / U.S. Navy Space and Naval Warfare Systems Command (SPAWAR) under contract N66001-11-C-0045.

References

[1] A. Boukerche, "Algorithms and Protocols for Wireless and Mobile Ad Hoc Networks", Wiley-IEEE Press, 2009.

[2] N. Suri, E. Benvegnù, M. Tortonesi, C.Stefanelli, J. Kovach, J. Hanna, "Communications Middleware for Tactical Environments: Observations, Experiences, and Lessons Learned", *IEEE Communications Magazine*, Vol. 47, No. 10, pp. 56-63, October 2009.

[3] S. Schütz, L.Eggert, S. Schmid, M. Brunner, "Protocol Enhancements for Intermittently Connected Hosts", *ACM SIGCOMM Computer Communication Review*, Vol. 35, No 3, July 2005

[4] X. Chen, H. Zhai, J. Wang, Y. Fang, "TCP performance over mobile ad hoc networks", *Canadian Journal of Electrical and Computer Engineering*, Vol. 29, No. 1, pp. 129-134, Jan-April 2004.

[5] N. Suri, G. Benincasa, M. Tortonesi, C.Stefanelli, J. Kovach, R. Winkler, R. Kohler, J. Hanna, L. Pochet, S. Watson, "Peer-to-Peer Communications for Tactical Environments: Observations, Requirements, and Experiences", *IEEE Communications Magazine*, Vol. 48, No. 10, October 2010.

[6] Mobile ad-hoc network emulator (MANE), available at: http://cs.itd.nrl.navy.mil/work/mane/index.php

[7] A. Bakre, B. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts", in: *Proc. of 15th IEEE International Conference on Distributed Computing Systems (ICDCS '95).*

[8] Z. Haas, "Mobile-TCP: An Asymmetric Transport Protocol Design for Mobile Systems", in: *Proc. of 3rd International Workshop on Mobile Multimedia Communications (IWMM'95).*

[9] M. Schlager, B. Rathke, S. Bodenstein, A. Wolisz, "Advocating a Remote Socket Architecture for Internet Access Using Wireless LANs", *Mobile Networks and Applications*, Vol. 6, N. 1, pp. 23-42, Jan./Feb. 2001.