Profiling Transport Performance for Big Data Transfer over Dedicated Channels

Daqing Yun,
Chase Q. Wu
Department of Computer Science
The University of Memphis
Memphis, TN 38152, USA
{dyun, chase.wu}@memphis.edu

Nageswara S.V. Rao,
Bradley W. Settlemyer, Josh Lothian
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 38152, USA
{raons, settlemyerbw, lothian}@ornl.gov

Rajkumar Kettimuthu,
Venkatram Vishwanath
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{kettimut, venkatv}@mcs.anl.gov

***Abstract*—The transfer of big data is increasingly supported by dedicated channels in high-performance networks. Transport protocols play a critical role in maximizing the link utilization of such high-speed connections. We propose a Transport Profile Generator (TPG) to characterize and enhance the end-to-end throughput performance of transport protocols. TPG automates the tuning of various transport-related parameters including socket options and protocol-specific configurations, and supports multiple data streams and multiple NIC-to-NIC connections. To instantiate the design of TPG, we use UDT as an example in the implementation and conduct extensive experiments of big data transfer over high-speed network channels to illustrate how existing transport protocols benefit from TPG in optimizing their performance.**

***Index Terms*—Transport profiling, big data transfer, high-performance networks.**

I. Introduction

Extreme-scale e-Science applications are generating colossal amounts of data, now frequently termed as "big data", which must be transferred over long distances for remote operations. Such big data transfer requires stable and high-speed connections, which are not readily available in traditional shared IP networks such as the Internet. High-performance networks (HPNs) featuring high bandwidth and advance reservation have emerged to be a promising solution to support these data- and network-intensive applications. In recent years, significant progress has been made in various aspects including the deployment of 100G networks with future 1Tbps capacity, the increase in end-host capabilities with multiple cores and buses, the improvement in large-capacity disk arrays, and the use of parallel file systems such as Lustre [4] and GPFS [2]. For example, DOE ESnet and Advanced Networking Initiatives (ANI) network infrastructures [1] have recently been upgraded to 100Gbps to meet the long-haul network demands for such data transfers.

However, even if a dedicated channel is provisioned in HPNs, the end-to-end data transfer performance still largely depends on the transport protocol being used on the end hosts. Along with the emergence and proliferation of HPNs, high-performance transport protocols are being rapidly developed and deployed, but maximizing their throughput performance over complex high-speed connections is still challenging, mainly because: i) their optimal operational zone is affected by the configurations and dynamics of the network, the end hosts, and the protocol itself, ii) their default parameter setting does not always yield the best performance, and iii) application users, who are domain experts, typically do not have the necessary knowledge to choose which transport protocol to use and which parameter value to set. Consequently, application users have not seen the corresponding increase in transport performance especially in terms of application-level throughput despite the bandwidth upgrades in the backbones of HPNs.

In many cases, for a given transport protocol, choosing an appropriate set of parameter values would result in a significant performance improvement over default settings. As a simple motivating example, we vary the data block size of UDT [5] running over a back-to-back 10Gbps link while using default values for the other parameters, and plot the corresponding instantaneous throughput measurements in Fig. 1, which shows more than four times throughput improvement on average. More improvements are expected if the other parameters such as buffer and packet sizes are properly tuned.

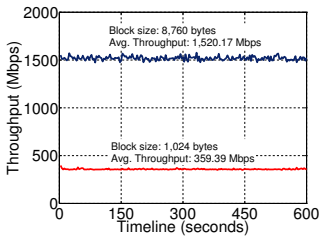


Fig. 1. UDT throughput measurements over a back-to-back 10Gbps link with different block sizes.

There are several bandwidth estimation tools such as iperf3 [3], which uses continuous data transfer to estimate the available throughput along an end-to-end network path. It provides users with various functions and options for tuning TCP, UDP and SCTP. It does not incorporate UDT, a well-adopted data transfer protocol in the HPN community, and does not provide an option to run parallel data streams over multiple NIC-to-NIC connections. A survey of bandwidth estimation tools can be found in [8].

We propose a Transport Profile Generator (TPG) to characterize and enhance the end-to-end throughput performance of transport protocols in support of big data transfer over dedicated channels. TPG provides end users with a light-weight and easy-to-use toolkit for transport performance profiling. TPG automates the tuning of various transport-related system configurations and protocol-specific parameters for optimal performance, and supports multiple data streams and multiple NIC-to-NIC connections. To instantiate the design of TPG, we use UDT as an example in the implementation and conduct extensive experiments of big data transfer to illustrate how existing transport protocols benefit from TPG in optimizing their performance.

978-1-4799-6959-3/15/$31.00 ©2015 IEEE

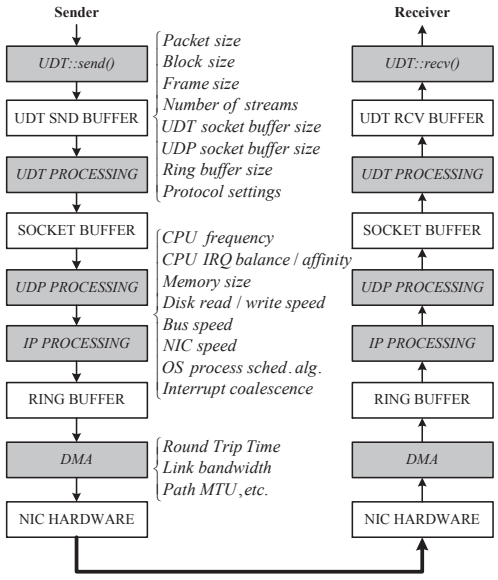


Fig. 2. Transport performance-related components.

The rest of the paper is organized as follows. Section II analyzes various performance-related factors and defines the transport profile. Sections III and IV present the TPG design and experimental results, respectively. Section V concludes our work.

II. Transport Profiling

A. Performance-related Components

The end-to-end data transfer is a complex process that involves both network and end-host components. Fig. 2 shows various software/hardware entities together with their parameter settings that may affect the end-to-end transport performance in a typical data send/receive process using UDT as an example. Any of these entities could become the bottleneck and hence limit the throughput performance of data transfer. Among these entities, some can be accessed and controlled by the application, such as packet size, block size[1], socket buffer size, jumbo frame size, number of data transfer streams; while the others are mainly determined by the hardware configurations and network infrastructures, such as CPU frequency, memory size, memory bandwidth, bus speed, disk I/O speed, path MTU size, round trip time (RTT), and link bandwidth. In this work, we focus on those entities directly accessible and controllable by the user or application.

B. Transport Profile

A transport profile $TP_t(\langle h_s,\ h_r\rangle,\ l,\ v)$ is a control-response plot illustrating how a set v of control parameters affect the transport performance (mainly throughput) of a transport protocol t over a network connection or link l between a sender h_s and a receiver h_r. Such profiles indicate the qualitative behavior of each component involved in the data transfer process and provide useful information for maximizing the transport performance.

The transport profile of a given protocol t is obtained by varying $\langle h_s,\ h_r\rangle$, l, and v to exhaust the combination of parameter settings over different network connections and collecting the corresponding throughput measurements. During a specific profiling process, $\langle h_s,\ h_r\rangle$ and l are given, and v is set by default or specified with values within an appropriate range.

[1]In our design, TPG calls `tpg_send()`/`tpg_recv()` to send/receive a data block, which may in turn call the underlying transport protocol APIs several times to completely deliver an entire data block. Here, we use the term "block size" to denote the size specified in `tpg_send()`/`tpg_recv()`, and "packet size" to denote the size of a transfer unit in the transport protocol.

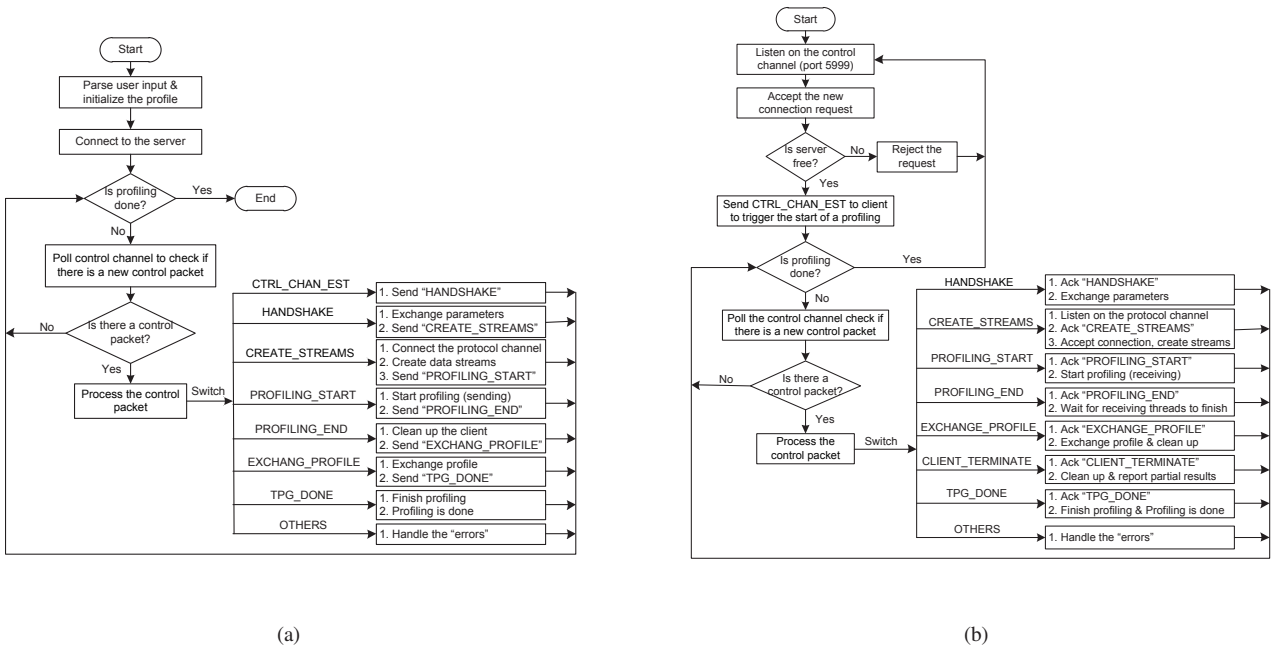
III. Design of Transport Profile Generator

A. Overview

Transport Profile Generator (TPG) consists of a pair of sender and receiver. The sender (client or source node) generates and delivers a certain amount of test data to the receiver (server or destination node) via a specific data transfer protocol being profiled. The sender also informs the receiver of the initialization and termination of a data transfer process (one-time profiling) through an independent TCP-based control channel. The client drives the entire profiling process and terminates after a one-time profiling, while the server is always reset for the next cycle of profiling. This way, user-specific profiling strategies can be automatically applied by repeatedly running the client with different parameter settings.

The control flowcharts of TPG client and server are shown in Fig. 3(a) and Fig. 3(b), respectively. A typical TPG profiling process carries out the following steps:

1) The server starts listening on the control channel;
2) The client parses the user input (if any), initializes a profile, and then connects to the server through the control channel;
3) The server accepts the connection request if it is free, and then sends back the state "`CTRL_CHAN_EST`" within the control packet to inform the client that the control channel has been established;
4) Upon the receival of "`CTRL_CHAN_EST`", the client sends "`HANDSHAKE`" to the server requesting control parameter exchange;
5) Upon the receival of "`HANDSHAKE`", the server sends "`HANDSHAKE`" back to the client as an acknowledgement;
6) Upon the receival of "`HANDSHAKE`", the client and server exchange control parameters;
7) The client sends "`CREATE_STREAMS`" to the server requesting data stream creation;
8) Upon the receival of "`CREATE_STREAMS`", the server listens on each protocol-specific data channel and then sends the same state back as an acknowledgement;
9) Upon the receival of the acknowledgement, the client connects to the server on each data channel by calling the protocol-specific "`connect()`" function and the server accepts the connection by calling the corresponding "`accept()`" function;



(a) (b)

Fig. 3. TPG control flow chart: (a) client, and (b) server.

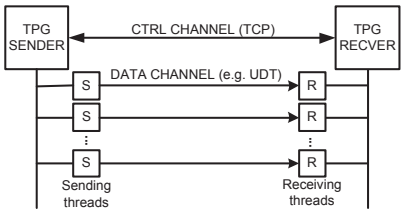


Fig. 4. TPG control and data channels.

10) The client sends "PROFILING_START" to the server, and then the server acknowledges;
11) Upon the receival of the acknowledgement, the client/server starts sending/receiving data blocks;
12) Once a one-time profiling is completed, the client sends "PROFILING_END", and the server responds with an acknowledgement;
13) The client and server exchange results by sending and acknowledging "EXCHANGE_PROFILE", and then the client exits while the server cleans up and waits for next profiling by sending and acknowledging the state "TPG_DONE".

During the above process, whenever an error or failure occurs, the client/server sends an error message through the control channel before it exits or aborts.

B. Support of Multiple Data Streams and Multiple NIC-to-NIC Connections

TPG uses one TCP-based control channel and supports multiple protocol-specific data channels, as shown in Fig. 4. The control channel is created at the starting stage of a profiling to exchange the control information between TPG client and server. TPG creates a separate working thread to perform independent profiling over each data channel. The data streams over different data channels can be bound to the default IP address or different ones (if multiple NICs are equipped). TPG maps each data stream to a pair of source-destination IP addresses to support multiple NIC-to-NIC connection-based profiling.

TPG features a flexible structure for an easy extension to new protocols, where a transport protocol is defined by its callback functions with a set of tunable control parameters.

IV. Implementation and Experimental Results

A. TPG Implementation

TPG is implemented in C/C++ on Linux platform. To instantiate the design of TPG, we include UDT as an example in the implementation. TPG is able to adjust the following parameters that may affect the UDT throughput performance:

- `-s`: run as a TPG server;
- `-c`: run as a TPG client;
- `-t`: select UDT for profiling (default is TCP);
- `-l`: set the data block size;
- `-P`: set the number of data streams;
- `-w`: set TCP socket buffer size;
- `-f`: set UDT send buffer size;
- `-F`: set UDP send buffer size (UDT socket option);
- `-r`: set UDT receive buffer size;
- `-R`: set UDP receive buffer size (UDT socket option);
- `-M`: set UDT socket option `UDT_MSS` [5];
- `-m`: set multiple NIC-to-NIC connections;
- `-a`: set CPU affinity;
- `-B`: set the bandwidth used by one UDT connection;

B. Experimental Results on a Local Back-to-back Connection

1) Configuration: We set up a local network testbed by back-to-back connecting two Dell workstations. The average round-trip time (RTT) between these two hosts through a direct 10Gbps link is around 0.04 milliseconds, resulting in a Bandwidth-Delay Product (BDP) of 50KB. The Internet connection between them has a RTT of around 0.25 milliseconds, and a bandwidth of around 95Mbps, resulting in a BDP of around 3KB. Both of the client (`dragon.cs.memphis.edu`) and the server (`rabbit.cs.memphis.edu`) are equipped with a 2.93 GHz Intel Core(TM) 2 Duo E7500 CPU, 2.9 GB RAM, and Fedora 17 Linux Operating System updated with `3.9.10-100.fc17.i686` kernel. The system's default (i.e. `net.core.rmem_default` and `net.core.wmem_default`) and maximum (i.e. `net.core.rmem_max` and `net.core.wmem_max`) memory space allowed for the UDP socket buffer size is configured to be 32MB and 64MB, respectively.

2) UDT Profiling on Packet Size: The throughput performance can be improved by using a larger packet size to reduce the per-packet overhead. For example, many Gigabit Ethernet NICs support "jumbo" frames with a packet size up to 9000 bytes or more. In the protocol stack of modern OS, the largest MTU supported along the network connection is automatically discovered and used [6], [7]. UDT provides a socket option `UDT_MSS` to configure the packet size. The profiling results across different packet sizes are plotted in Fig. 5, which shows that the UDT throughput performance is improved by using larger packet sizes and setting the UDT option `UDT_MSS` to be the maximal allowable MTU size along the path. In this experiment, both UDT and UDP are configured with sufficient socket buffer space to maintain the link speed.

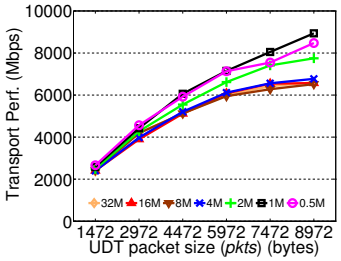


Fig. 5. Average throughput vs. packet size ($pkts$): 1 data stream, 120 seconds of transfer. Different curves correspond to different UDT/UDP send/receive buffer sizes.

3) UDT Profiling on Block Size: We plot the profiling results on the block size in Fig. 6, where the x-axis uses a multiplicity (n) of the payload size ($pkts - 16$) to represent the block size. We observe that when the buffer size is limited, increasing the block size does not bring too much throughput gain, especially when the block size is comparable with the buffer size. However, when there is sufficient buffer space, increasing the data block size significantly improves the UDT throughput performance, but the improvement becomes less obvious as the data block size increases.

Particularly, in Fig. 6(a), when the buffer size is set to be 128KB, which is larger than the $Bandwidth \times RTT$, the peak throughput we observe is less 7Gbps over the 10Gbps link; when the block size is further increased from 107471 bytes to 179119 bytes (i.e. n from 12 to 20, since $\lceil 107471/8956 \rceil = 12$ and $\lceil 179119/8956 \rceil = 20$), the throughput drastically decreases. If we increase the buffer size to 0.5MB or 1.0MB, UDT achieves the peak throughput around 8.5Gbps. If we further increase the buffer size to 2MB or 4MB, the throughput performance decreases slightly, as shown in Fig. 6(b). When we continue to increase the buffer size from 8MB to 64MB, the peak throughput further decreases, as shown in Fig. 6(c). Our profiling results on the data block size show that a larger block size generally leads to a better performance, however, an appropriate buffer size is also necessary to ensure a satisfactory throughput performance. In this test case, a buffer size of 0.5MB or 1.0MB seems to be appropriate.

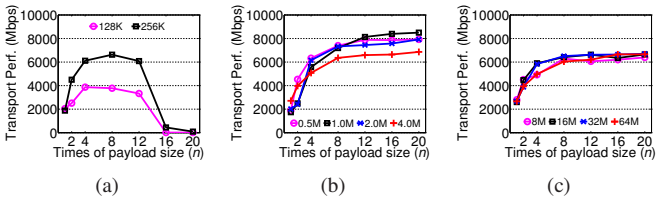


Fig. 6. Average throughput vs. block size (bs): 8972 bytes packet size ($pkts$), $(pkts - 16) = 8956$ payload size, 1 data stream, 120 seconds of transfer. Different curves correspond to different UDT/UDP send/receive buffer sizes.

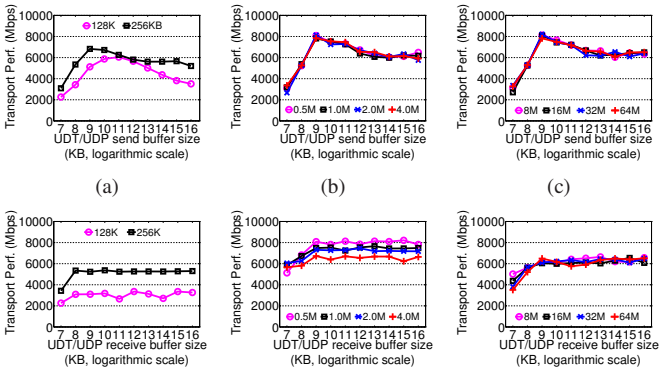


Fig. 7. Average throughput vs. UDT/UDP send buffer size in subfigures $(a)-(c)$, where different curves correspond to different UDT/UDP receive buffer sizes. Average throughput vs. UDT/UDP receive buffer size in subfigures $(d)-(f)$, where different curves correspond to different UDT/UDP send buffer sizes. Other parameters: 8972 bytes packet size ($pkts$), 89559 bytes block size (bs), 1 data stream, 120 seconds of transfer.

4) UDT Profiling on Buffer Size: We plot the performance measurements in response to various send/receive buffer sizes in Fig. 7, where the x-axis takes the logarithm of the actual send/receive buffer size (e.g. $7 = log_2 128$ represents 128KB). A rule of thumb for obtaining good transport performance is that both the send buffer and the receive buffer should be no less than $Bandwidth \times RTT$ (BDP), which is also true in our experiments with UDT. In Fig. 7(a), Fig. 7(b), and Fig. 7(c), as the send buffer size increases from 128KB (2^7KB) to 1024KB (2^{10}KB), we observe a significant throughput improvement. Afterwards, in the case of a small receive buffer size, e.g. 128K or 256K in Fig. 7(a), increasing the send buffer from 1.0MB (2^{10}KB) to 64MB (2^{16}KB) drastically decreases the throughput. It is probably due to the fact that a larger send

buffer results in a longer RTT and in turn a larger BDP [5], which requires a larger receive buffer to maintain the transfer speed. However, in the case of a large buffer size, e.g. from 1.0M to 64M, we observe that the throughput first decreases, and then stabilizes around 6Gbps. As shown in Fig. 7(d), Fig. 7(e), and Fig. 7(f), a larger receiver buffer also generally leads to a better performance, but the improvement becomes less obvious as the receive buffer increases. In the case of a small send buffer (128KB and 256KB), increasing the receive buffer does not have an obvious positive effect as shown in Fig. 7(d). In the case of a large send buffer size, increasing the receive buffer greatly improves the performance, as shown in Fig. 7(e) and Fig. 7(f). The profiling results on the buffer size show that to maintain a high UDT data transfer rate, a large receive buffer is needed, and an appropriate send buffer is also necessary. Since a larger send buffer may incur a longer RTT and therefore may not yield the best performance, in this test case, a send buffer of 0.5MB or 1.0MB turns out to perform well, which is consistent with the results in Section IV-B3.

5) UDT Profiling on Parallel Streams: We vary the number of parallel streams and plot the corresponding aggregate throughput performance in Fig. 8. We observe that with two parallel data streams, we achieve a throughput of 8Gbps. However, a larger number of parallel streams may not necessarily lead to a better performance as shown in Fig. 8, which is mainly due to the significant overhead incurred by memory copying, context switching, and multi-threaded implementation. However, on the hosts with sufficient computing resources, running multiple parallel streams would generally improve the throughput performance, although the transport protocol itself may not be able to fully utilize the link bandwidth. Determining an optimal number of streams is not straightforward as it highly depends on the specific configurations of the hosts.

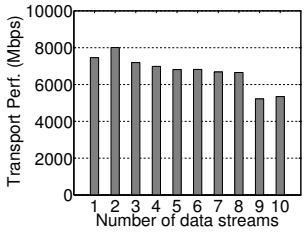


Fig. 8. Average throughput vs. number of streams: 1MB UDT/UDP send/receive buffer size, 8972 bytes packet size ($pkts$), 89559 bytes block size (bs), 120 seconds of transfer.

6) Comparison of Default UDT and TPG-tuned UDT: To illustrate how TPG improves the performance of UDT, we run 10 sets of data transfer experiments using default UDT and TPG-tuned UDT. The performance results are tabulated in Table I and further plotted in Fig. 9 for a visual comparison. Note that the *italic* numbers in Table I indicate that they are UDT default values. We observe that the TPG-tuned UDT in experiment 8 achieves a significant performance improvement over any other parameter settings.

V. Conclusion and Future Work

We proposed TPG, a transport profiling toolkit to characterize and improve the performance of existing transport protocols to support big data movement. We used UDT as an example and conducted extensive experiments in real network environments. TPG is useful to explore the optimal protocol parameters and system configurations prior to the actual data movement. It is of our interest to extend TPG with more transport protocols and methods, and integrate it into existing large data transfer tools such as XDD [9] .

TABLE I
Throughput performance comparison between default UDT and TPG-tuned UDT.

Idx no.	packet size (B)	block size (B)	UDT sndbuf (M)	UDP sndbuf (M)	UDT recvbuf (M)	UDP recvbuf (M)	Perf. (Mbps)
1	1472	1455	*10*	*1*	*10*	*1*	450
2	1472	8955	*10*	*1*	*10*	*1*	1762
3	1472	65536	*10*	*1*	*10*	*1*	2487
4	1472	89559	*10*	*1*	*10*	*1*	2584
5	1472	89559	1	1	1	1	2810
6	1472	179119	1	1	1	1	2847
7	8972	89559	1	1	1	1	7216
8	8972	179119	1	1	1	1	8713
9	8972	179119	16	16	16	16	6300
10	8972	179119	32	32	32	32	6536

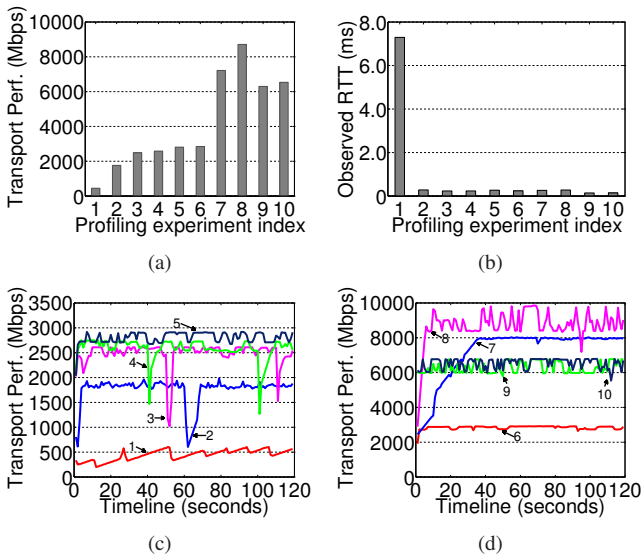


Fig. 9. Throughput performance comparison between default UDT and TPG-tuned UDT: 120 seconds of transfer, 1 second sampling interval. (a) Average throughput in each experiment; (b) Average RTT in each experiment; (c) Instantaneous throughput measurements in experiments 1 to 5; (d) Instantaneous throughput measurements in experiments 6 to 10.

Acknowledgment

This research is sponsored by U.S. Department of Energy's Office of Science under Grant No. DE-SC0010641with University of Memphis.

References

[1] Energy Sciences Network. http://www.es.net.
[2] IBM General Parallel File System. http://www-03.ibm.com/systems/software/gpfs/.
[3] iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf.
[4] Lustre. http://wiki.lustre.org.
[5] UDP-based Data Transfer. http://udt.sourceforge.net/.
[6] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End System Optimizations for High-speed TCP. *Comm. Mag.*, 39(4):68–74, Apr 2001.
[7] J. Mogul and S. Deering. Path MTU Discovery, Nov. 1990. IETF RFC 1191.
[8] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35, 2003.
[9] B.W. Settlemyer, J.D. Dobson, S.W. Hodson, J.A. Kuehn, S.W. Poole, and T.M. Ruwart. A Technique for Moving Large Data Sets over High-performance Long Distance Networks. In *Proc. of the 27th MSST*, pages 1–6, May 2011.