Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 21-23, 2007, pp.281-288.

Event based Extensible Interactive Transparent Networking: Performance Study with Fast TCP Principles

Javed I. Khan, Pradeep K. Punnam and Raid Zaghal
Networking and Media Communications Research Laboratories
Department of Computer Science, Kent State University
233 MSB, Kent, OH 44242
javed|ppunnam|rzaghal@cs.kent.edu

Abstract

Interactive Transparent Networking has been proposed to support a new generation of symbiotic applications that require advance interaction with the Network. In this mode advanced applications can subscribe to state-space event based feed from network protocol local end-points by interactive version of the network protocols. This enables a whole new range of high performance extensible adaptation which requires low-time constant feedback. In this paper, we show how classical TCP can be extended to support long delay high capacity pipe. We demonstrate a system that mimics FAST TCP however, without the usual extensive reengineering required by the original. It seems matching performance can be achieved with protocol interactivity.

1 Introduction

Traditional network software stack has been designed with a layered organization. Each layer is intended to offer a specific service. Each service has a specific implementation somewhere inside these layers. These layers have provided a simple and systematic framework to meet the needs of first generation internet application- however now it seems the situation has changed. The same organization now is widely felt to be rigid and frustratingly immutable.

Various solutions have been proposed by researchers to soften the rigid layered organization to allow them to be more accessible and tunable. For example TCP is now being worked out to make it more tunable (Net100 project [4]). The paradigm of *programmable networking* took the approach one step deeper- it suggested making the API of not only the protocol end-points, but also that of router software stack to be made open. The paradigm of *active networking* [7] has moved another step further by allowing almost arbitrary custom codes to be embedded inside network software layers.

It seems the need for changing and embedding custom components inside network layer software comes from two realities- first we are seeing that the data is being required to be processed in custom and adaptive ways. No finite set of pre-agreed '*fixed*' protocols may ever foresee and satisfy the cases. Secondly, these customizations are very dynamic and communication case specific. Many of the triggers for the accompanying custom actions originate right inside network software. Thus, allowing programmable modules to be embedded inside '*fixed*' network software solves both the problems. However, it seems a less radical approach of **interactive transparent networking** [2,3], which we will explain shortly, might be able to achieve similar functional capability- with perhaps much lesser network layer complexity. The idea is to make network service implementations interactively accessible. This unique feature can allow new type of application level codes to augment, alter and influence the lower level functionalities of the protocols in the same way as they are embedded inside. This results in a surprisingly low cost path for implementing and upgrading new protocols. The approach allows a new generation high performance adaptive applications that require low feedback constant interaction with environment.

FAST TCP is one of the latest high-performance extensions of TCP proposed by Steven Low et al. [5,6] for better performance in high-bandwidth delay product networks. However, it has required a completely reengineered TCP. Can such a challenging system be implemented outside network layer? It requires very low time constant coupling- which generally is beyond the realm of end-to-end edge techniques (such as RTP). We have recently taken this challenge and designed a novel system which can mimic FAST over a Linux interactive Network kernel. This paper first explains the interactive transparent networking. Then it explains the implementation of the FAST TCP congestion control algorithm in application layer using interactive model, and finally presents the performance.

2 Interactive Transparent Networking

Instead of embedding custom codes within network layer the paradigm of *interactive transparent networking* [3] proposes creating mechanisms only to pull up the required service state information to the upper layers. And then the actual action can be formulated by the programmable components running in the upper layers- even at the application layer, and then to create handles so that the generated actions again can be pushed down below in the network layer. This relives the lower network layers from housing custom components and to re-address complex issues regarding security and resource sharing. The attraction is that the application space already has very well developed provision for housing custom codes, share resources, and handle security issues for managing multiple trust domains, etc. Much of that can be reused.

The proposed interactive and transparent networking requires three fundamental steps. The first step towards is to capture the internal operation of a target protocol in some form of event/state based language [1]. The classical protocols did not require this formalization. We call this stage protocol's *operational modeling.* The second step is to provide *protocol event subscription facility* so that a set of events and states identified in the operational model can be accessible to subscribers. The third step is to create appropriate *operating system facilities* so that custom modules can efficiently interact with these network service components.

2.1 TCP Protocol's Operational Modeling

We modeled the TCP congestion control algorithm (i.e. Reno) using finite state machine as shown in the Figure-1. All the states and state transitions occur inside the established state of the TCP state machine except the TCP initialization state. The lower part of the diagram shows the 'fast Retransmission/Fast Recovery', and the upper part shows the 'slow start/congestion avoidance' algorithm of Reno. TCP_INIT and TCP_CLOSE events are initialization and termination events occurred in TCP connection establishment and connection closing phases of a TCP connection. Table-1 shows all the states and events in TCP congestion control algorithm.

We can implement a new protocol or an improvement to an existing protocol by changing state transmission of the protocol. As shown in the figure-1 the state transition will bypass the slow start and congestion avoidance through a new state (ex. 'F' FAST congestion control), when the protocol change is enabled by the user. Instead of implementing this new state transition in the actual kernel, we implement it in the application layer which will access the state and event information from the kernel.

Interactive transparent networking service makes such *events* and *state transitions* accessible by demanding subscribers –e.g. user application. In fact, our transparency service also allows the subscriber to modify the internal state of the protocol. In general, the transparency service user (e.g. application level user program) and the target protocol (e.g. TCP) interact through a well-defined, clean interface. The interface supports four distinctive operations: (a) *subscribe,* (b) *signal,* (c) *probe*, and (d) *modify*.

An upper layer subscribes to events in the protocols below. Typically, a subscriber layer should be interested in certain events. By subscribing, the subscriber wishes to be notified when any one of the events has occurred. In real practice however, only a subset of the target layer events are subscribable. The transparency service is obliged to fulfill the subscriber wishes as far as it abides to the security restrictions and access privileges imposed by the super user. Next we describe the interactivity architecture and demonstrate with a general scenario the four operations of the transparency service.

2.2 Interactivity Architecture

In Figure-2 layer *L* wishes to subscribe for event *e* in layer *Q*. Notice that layers *L* and *Q* are separated by layer *P* to emphasize that subscriber and target layers are not necessarily direct neighbors. *L* makes a *Subscribe* call (1) that includes: Subscriber *L*, Target *Q*, subscribed event *e*, and the Transient-ware (*T-ware*) module *T*. By making this call, layer *L* is basically telling CH (central handler): Whenever event *e* happens in layer *Q*, please activate module *T*. Since *CH* maintains subscription information of all subscribers, it adds this subscription to its database. Next, *CH* forwards the subscription request to the target layer *Q* (2). Also, *Q* adds a subscription instance for (Layer *L*/Event *e*) to its internal state. When event *e* happens, a signal is sent to the *CH* (3) which in turn probes *Q* (4) to get the event information. When it receives the event information, *CH* searches its database for the appropriate *T-ware* module that matches the instance (*L, Q, e*) and invokes it (5). Once invoked, the *T-ware* module can access relevant parts of the internal state of layer *Q* and possibly make changes to it as specified by the protocol being implemented. The *T-ware* module can use the *probe* or *modify* operations (6) to access/update the internal state of *Q* in accordance with its access privileges.
This scheme obviously incurs little overhead in terms of signaling cost. Central Handler (*CH)* can be a bottleneck, if several events happen simultaneously. If we assume that the CH uses a FCFS queue, some events may experience some delay before they can be served. So we need an efficient mechanism to control the event generation and event delivery to eliminate this

State	Description
I	Initial closed state.
1	Established state communication
2	First duplicate ACK received
3	Second duplicate ACK received
4	Third duplicate ACK received – Fast Retransmission
5	Timed out (1T)
6	Fast Recovery: Send a new segment for every duplicate ACK received
7	New ACK received, shrink win to half
8	Drop win size to 1 segment
9	Slow start: Increase win exponentially
10	Congestion avoidance.

Table-1.a TCP States

event	Description
ACK	Reception of an ACK
New seg send	New segment(pkt) is sent in response to ack arrival.
Time out	Time out on an ACK
DACK(1)	First duplicate ACK
DACK(2)	Second duplicate ACK
DACK(3)	Third duplicate ACK --- packet loss
Rexmt	Retransmission of lost packet
Win=ssthresh	Congestion avoidance
TCP_init	Connection initialization – SYN Exchange
TCP_close	Connection termination – FIN exchange

Table-1.b TCP events

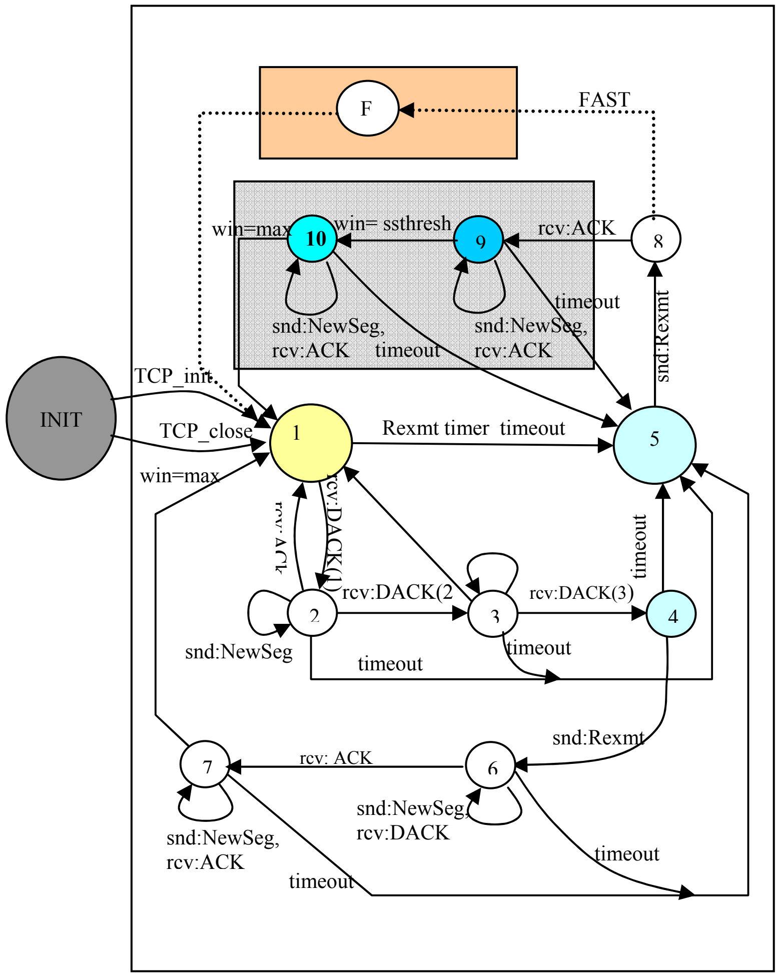


Fig-1: TCP Reno Congestion control state diagram with states and events.

bottleneck. To eliminate the event bottlenecks in Central Handler, we gave the user an ability to control the event generation process. User can subscribe for the events independently and with different priorities and frequencies. Next we will explain the Central Handler.

2.3 Central Handler

Central handler controls event generation and delivery process of the interactive model. It maintains the subscription information in the *'subscription info'* data structure, which will be used by the event filters to dynamically control the event generation. User can dynamically change the subscription information using subscription API. Central handler maintains the event information in the *'event info'* data structure, so that subscribed applications can get the event information by using probing API's.

Fig-3 shows the working diagram of the central handler. Applications register their event requirements using subscription API (1). This information will be stored in *'Subscription info'* data structures. When an event generated by a network layer is received by the central handler, the event information will be feed to the event filters (3). Event filters will process the events by matching with the '*subscription info*' data structure information of that event (4a, 4b, 4c). The filtered event's state information will be kept in the '*event info*' data structure, before it will be queued in to the signal queue (5, 6a).When the system scheduler delivers the event, the T-ware will get the event information from the *'event info'* data structures and performs the corresponding operations(6b,7).

2.4 Event registration and control

We designed an event control mechanism to give user an ability to control the event generation process. Each event generated will go through three sub-filters before it invoking corresponding Transient-ware.

1) Subscription filter: It will let only the events that are subscribed. This will make sure event generated is an outstanding and syntactically correct event.

2) Time-Filter: It will determine the time gap between successive event deliveries for a given event. If more then one instance of that event occur in this period, it will send the last occurred event.

3) Frequency-Filter: It will determine how many events to be skipped between two allowed events.

Figure-4 explains the filtering mechanism. Event E_1 is generated 11 times and the time value of the time filter is T=t1 so all the events except E4, E7, E10 are dropped. The frequency filter applies frequency of F=3 which eliminates all the events but E10. Thus, the user is given a control to eliminate excessive event occurrences of an unwanted event. User can set timer and frequency to zero, if user wants to allow all the occurrences of events.

2.5 Interactive-TCP (iTCP)

We extended the standard socket API with subscription and probing system calls to enable demanding applications to use the transparency service.

Figure-5 depicts the basic architecture of iTCP. Upon opening the socket, an adaptive application may bind a T-ware module to a designated TCP event by subscribing with the kernel. The binding is optional; if the application chooses not to subscribe, the system defaults to the silent mode identical to TCP classic. The logical sequence of operations follows the general transparency framework described in the previous section and proceeds as follows: (1, 2) subscribe, (3a, 3b) send a signal, (4a, 4b) probe kernel for event type, (5) invoke appropriate T-ware module to serve the event, (6a, 6b) probe/modify internal state, and (7) means that a T-ware module can also access or modify certain state variables in the user application if

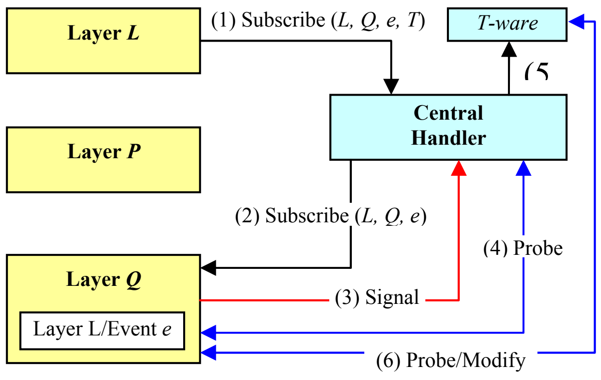
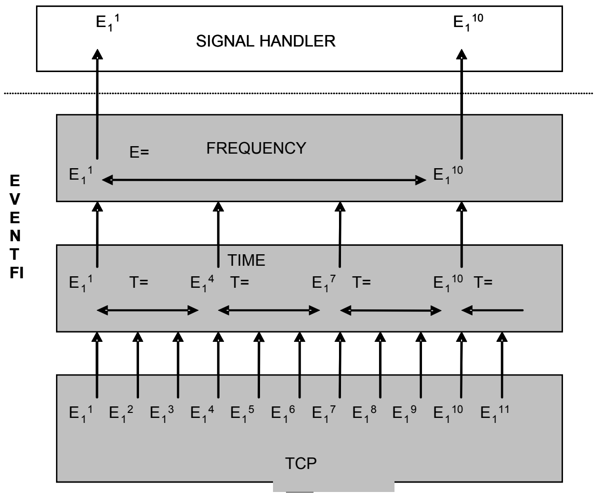


Figure-2: The sequence of operations in a general interactive service model scenario.



Figur -4. event filtering

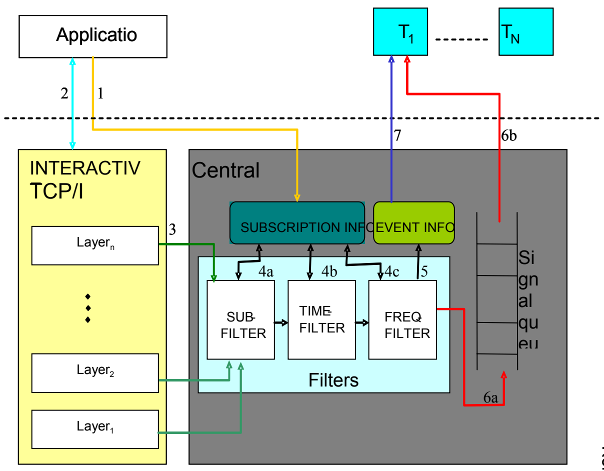


Figure3. Central Handler working

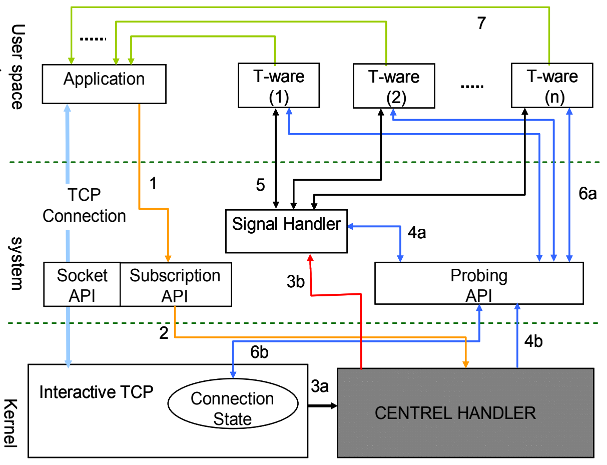


Figure-5. The TCP-interactive extension and API.

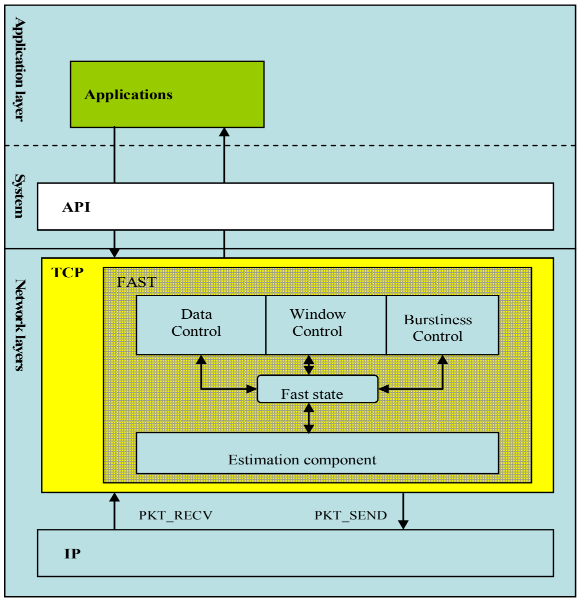


Fig-6 FAST TCP in the Kernel

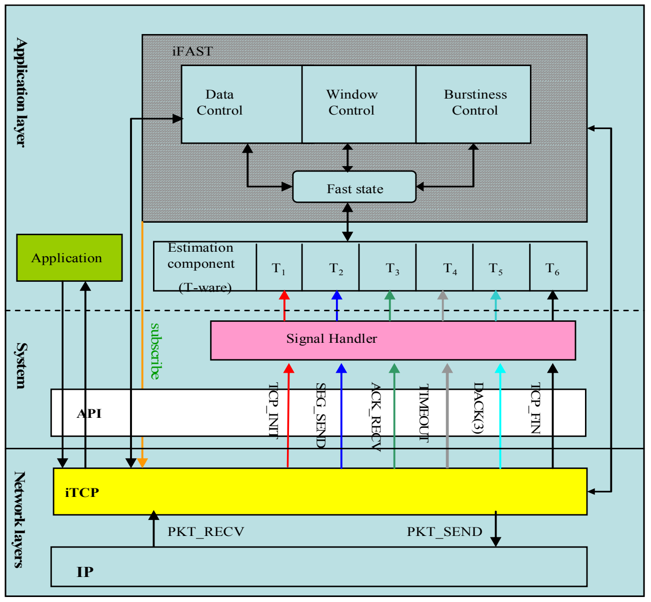


Fig-7 FAST TCP i in user space using iTCP

necessary.

Table-1.a shows all the events that can occur in the operation of congestion control algorithm, and corresponding event variables. In the FAST implementation we selected a set of 6 events that will effect the operation of the FAST TCP. TABLE-2 shows the set of these events and the corresponding *transientwares* (handlers).

Transient-wares T1 invoked by the TCP_INIT event initializes the data structures in *'fast state'*. The transient-ware T6 removes all the state information of the connection from the *'fast state'*, when a TCP_FIN event occurs.When an ACK received, iTCP sends ACK_RCV event notification to trigger the transient-ware T3, which calculates the average RTT (avgRTT), average queuing delay (qdealy) and change the state variables of the *'fast state'* . When a packet is sent, a SEND_SEG event is generated which will invoke Transient-ware T2, which will note the timestamp in the *'fast state'*. This timestamp is used by burstiness control to calculate the gap between successive packet transmissions.

3 FAST TCP

FAST TCP is a congestion control algorithm proposed by Steven Low et al. [5] for better performance in high bandwidth delay product networks. Unlike the TCP RENO which is a loss based algorithm, FAST TCP is a delay based congestion algorithm. FAST TCP calculates the congestion window of a connection based on the queuing delay sensed in the network. For every positive ACK received it will calculate AvgRTT and queuing delay using the following equations.

$$\overline{T}_i(k+1) = (1-\eta(t_k))\overline{T}_i(k) + \eta(t_k)T_i(k) \qquad (1)$$

$$q_i(k) = \overline{T}_i(k) - d_i(k) \qquad (2)$$

Equation 1 is used to calculate the average RTT of the network, Where $\overline{T}_i(k)$ is moving average RTT and $\eta(t)$ = min {3/w(t) , 1/4} is the weight used to calculate the moving average which depends on the window size W(t) at time t. Equation 2 calculates the queuing delay using the difference between the average RTT and the base RTT. FAST TCP is equation based congestion control algorithm where congestion window is calculated periodically using the following formula.

$$w \leftarrow \min\left\{2w, (1-\gamma)w + \gamma(\frac{baseRTT}{RTT}w + \alpha(w, qdelay)\right\} \qquad (3)$$

$\gamma \in (0,1]$ Controls the convergence speed of the protocol and the baseRTT is the minimum rtt observed so far. $\alpha(w, qdelay)$ is taken as constant as suggest by the authors of FAST TCP.
FAST TCP has been implemented by adding custom code to the kernel, but we show that it can be implemented as user module using the interactive model without loosing any performance.

4 Fast TCP implemented with iTCP

4.1 General Scheme

Figure-6 shows the schematics of the original FAST TCP implementations. When an acknowledgement received, estimation component computes the average queuing delay and registers into the fast state. When a packet is transmitted, estimation component will

Table-2 iFAST T-wares

EVENT	SUB	EVENT-VARIABLES	T-Ware	EVENT & T-WATE PURPOSE
TCP_INIT	S	TIMESTAMP, SOCK-ID.	T1	T-ware will create the iFAST data structure for the connection.
SEG SEND	S	TIMESTAMP. NO_PKTS_SENT	T2	T-ware will note the time stamp for burstiness control of the connection, and notes the number of packets sent.
ACK RECV	S	TIMESTAMP,RTT, PIF,NO_PKTS_ACKED, SSTHRESH, SND_CWND.	T3	T-ware will calculate the AvgRTT, qdelay by using the new RTT value, and calculate the new congestion window value and pushes the value back to the connection state.
TIMEOUT	S	TIMESTAMP, NO_SKB_RTX.	T4	This event will cause T-ware to drop congestion window to 1.
DACK(3)	S	TIMESTAMP,PIF, NO_PKTS_ACKED.	T5	T-ware will decrease the ssthresh and congestion window.
TCP_FIN	S	SOCK_ID.	T6	T-ware will delete corresponding connection data structures.

register the time stamp to *'fast state'* data structures. This time stamp will be used in calculating the time gap between two consecutive transmitted packets. Window control component will compute the new congestion window value based on the queuing delay and current window size for every RTT. Burstiness control component will distribute the outstanding packets into a RTT to simulate the fluid-like behavior in data transmission. Data control component will select the packet to transmit from the pool of outstanding packets. All these changes are implemented inside kernel by modifying the TCP code.

Figure 7 shows the FAST TCP implementation in the application layer with estimation component as transient ware and the other components as user modules. When an event occurs in the TCP, the iTCP (interactive TCP) will invoke the corresponding transient ware which will change the *'fast state'* accordingly. The data control, window control and burstiness control components will use the *'fast state'* information in controlling the data transfer. We describe the T-wares next.

4.2 iFAST T-ware

Table-1.a shows all the events that can occur in the operation of congestion control algorithm, and corresponding event variables.

We need to select events that are required for the implementation of FAST TCP. In the FAST implementation we selected a set of 6 events that will effect the operation of the FAST TCP. TABLE-2 shows the set of these events and the corresponding transientwares (handlers). Transient-wares T1 invoked

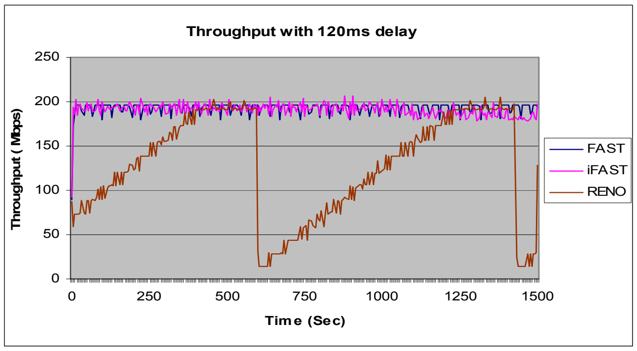


Figure-9.a: Throughput with 120ms delay

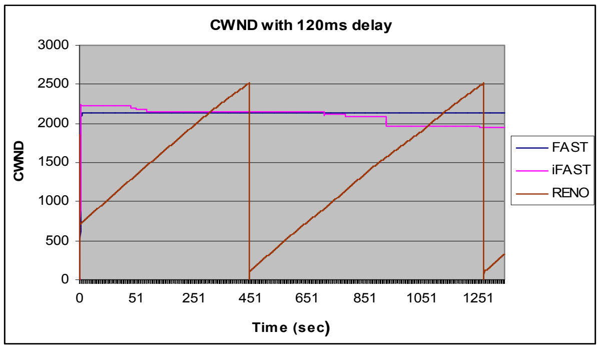


Figure-9.b: Congestion window with 400ms delay

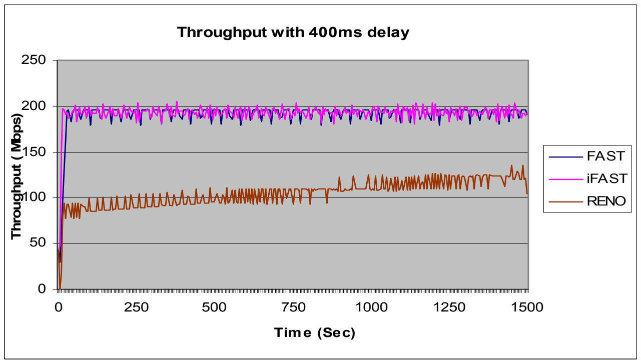


Figure-9.c: Throughput with 400ms delay

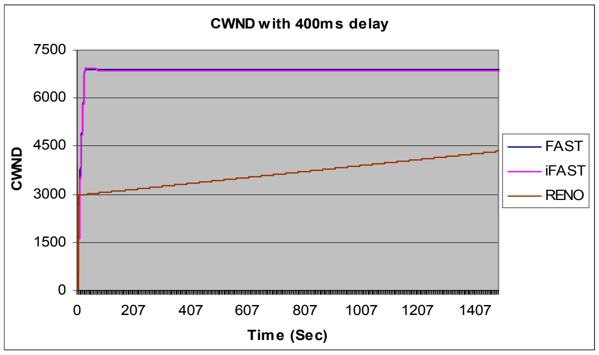


Figure-9.d: congestion window with 400ms delay

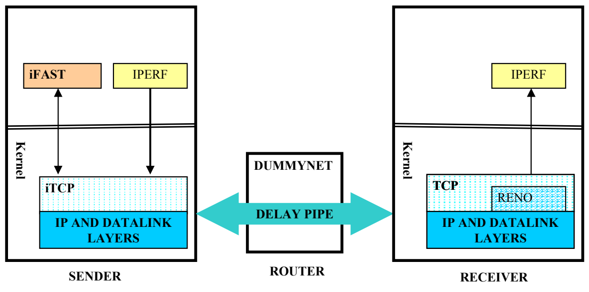


Figure-8: Experimental setup

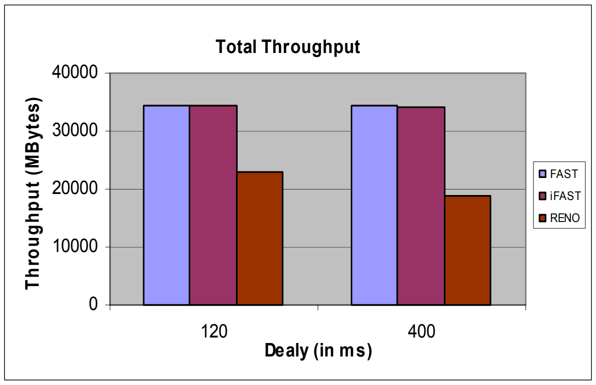


Figure-10: Total throughput achieved

by the TCP_INIT event initializes the data structures in 'fast state'. The transient-ware T6 removes all the state information of the connection from the 'fast state', when a TCP_FIN event occurs. When an ACK received, iTCP sends ACK_RCV event notification to trigger the transient-ware T3, which calculates the average RTT (avgRTT), average queuing delay (qdealy) and change the state variables of the 'fast state' . When a packet is sent, a SEND_SEG event is generated which will invoke Transient-ware T2, which will note the timestamp in the 'fast state'. This timestamp is used by burstiness control to calculate the gap between successive packet transmissions. Packet-loss is notified by TIMEOUT and DACK(3) events, which occurs when a retransmission timer expires and three duplicate ACK are received.

Transient-ware T4 is invoked when a TIMEOUT event occurs; it will reduce its congestion window to initial value and starts the congestion algorithm again. Transient-ware T5 will reduce the window size by Half. Experiments and Performance analysis

5.1 Experimental Setup

We constructed an experimental testbed with a sender, receiver running Linux 2.6.7 kernel, and router running FreeBSD. Each machine has AMD Athlon 1.6 GHz processor with 512MB of memory and D-Link 500T gigabit Ethernet cards. We have used the FreeBSD dummynet service in the in the router to simulate the queuing delay, bandwidth and queue size limits. We can configure the dummynet to simulate different queuing delays and the queue sizes. Figure-8 shows the test-bed setup. We used IPERF traffic generator to generate the TCP traffic between the sender and receiver. The maximum throughput achieved by the above hardware setting is 250 Mbps, so we selected a bandwidth limit of 200 Mbps to simulate the bottleneck behavior. To simulate the high bandwidth and delay product network we choose to use a relatively high delay of 120ms and 400ms.

5.2 Performance Comparison

We captured the throughput and congestion window behavior of all the three implementations to understand the behavior of the new protocol implementation (iFAST). Figure 9-(a,b) shows the throughput achievedby each of the protocols in 1500 seconds of experiment.

As we can see from the graphs, both iFAST and FAST are able to achieve throughputs around 190 constantly, while Reno perforce poorly because of packet losses caused by the queue overflow at the router. Figure-9(c,d) shows the behavior of the protocols in the form of congestion window. Both FAST and iFAST behaves the same, while Reno constantly going to slow start state because of the packet losses. Figure-10 shows the performance of the three kinds of congestion control algorithms (iFAST, FAST, Reno). In the form of how much data transfer they are able to achieve, as we can see from the graph both FAST and iFAST able to achieve a data transfer of 34.5 GB but Reno performs very badly in both the cases.

5.3 Overhead Cost Analysis

Interactivity adds some extra cost to the original scheme. It adds signaling and system calls overhead. To get a real measurement of interactivity service overhead we performed a simple experiment on iTCP. We ran iFAST on a similar test-bed as shown in figure-8 with three different modes: 1) Invoke only- In this mode iFAST will subscribe the events and receives the event notifications, 2) Probe mode- this mode includes invoke only operations and also iFAST probes the kernel for event state information, 3) Full iFAST- this mode will perform the complete iFAST operations. We performed iFAST experiments over a period of 100 sec in all the three modes.Table-3 shows the experimental results over 10 runs for each experiment. We used the Linux getrusage() utility to collect the following resource usage information from the:

1) User time (utime): The total amount of time spent executing in user mode.

	User CPU time (in sec)		System CPU time (in sec)		Total CPU time (in sec)	
	Avg	SD	Avg	SD	Avg	SD
Signal only	1.29	0.14	5.88	0.6	7.17	0.7
Probe mode	1.89	0.71	7.69	0.32	8.58	0.36
Full iFAST	2.17	0.79	7.57	0.54	9.74	0.55

Table-3. CPU overhead of iFAST, averaged over 10 samples.

	Voluntary CSW		Forced CSW		Total CSW	
	Avg	SD	Avg	SD	Avg	SD
Signal only	575466	61441.7	3758	539.8	579224	61807.29
Probe mode	496225	17122.2	4852	539.6	501077	17594.66
Full iFAST	486582	10369.9	4795	517.7	490777	10742.38

Table-4. context switching of iFast, averaged over 10 samples.

2) System time (stime): The total amount of time spent in the system executing on behalf of the process.
3) Voluntary context switches (vcsw): The number of times a context switch resulted due to a process voluntarily giving up the CPU beforeits time slice was completed (usually to await availability of a resource).
4) Forced context switches (fcsw): The number of times a context switch was forced by the OS due to a higher priority process gaining the CPU or because the current process exceeded its time slice.
From the Table-4 we can observe that iFAST implementation takes more of system time (stime) than the user time (utime) where the actual protocol is implemented this also proves that the size of the t-ware has little effect on the overhead. The total amount of CPU overhead is a small percentage of the total running time of the experiment (9.5 %). The standard deviation of the iFAST increases from signal only to full implementation. Table-4 shows the context switching overhead experienced by the iFAST implementation. It can be seen from the table that a small increase in the total number of forced context switched and small decrease in the Voluntary context switches is observed.

6. Concluding Remarks

Interactive Transparent Networking has been proposed to support a new generation of symbiotic applications that require advance interaction with the Network. This enables a whole new range of high performance extensible adaptation which requires low-time constant feedback. In this paper, we show how classical TCP can be extended to support long delay high capacity pipe. We demonstrate a system that mimics FAST TCP like principles however, without the usual extensive reengineering required by FAST TCP. We show how performance similar to FAST TCP can be achieved with protocol interactivity. It seems, there is much little to worry about performance sacrifice than to gain from smart networking. This model will incur small overhead of signaling and data exchange between user and kernel, but drastically simplifies protocol extension and modification.

7. References

[1] J. Khan and R. Zaghal, "Event Model and Application Programming Interface of TCP Interactive," Technical Report (TR2003-02-02), February 2003.

[2] Javed I. Khan and Raid Y. Zaghal, Symbiotic Rate Adaptation for Time Sensitive Elastic Traffic with Interactive Transport, Journal of Computer Networks*, Elsevier Science Volume 51, Issue 1, 17 January 2007, Pages 239-257.

[3] Javed I. Khan and Raid Y. Zaghal, Interactive Transparent Networking: Protocol Meta-modeling based on EFSM, Journal of Computer Communications, Elsevier Science, Volume 29, Issue 17, 8 November 2006.

[4] Net100, "http://www.net100.org" The Net100 Project-Development of Network-Aware Operating Systems, 2001.

[5] David X. Wei, Cheng Jin, Steven H. Low and Sanjay Hegde, FAST TCP: motivation, architecture, algorithms, performance. IEEE/ACM Trans. on Networking.

[6] W. Stevens, "TCP/IP Illustrated, Volume 1: The Protocols," Addison-Wesley, 1994.

[7] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Network Research," IEEE Communications Magazine, Vol. 35, No. 1, pp80-86. Jan. 1997.