Extension and ns2 Implementation of Saratoga

Abu Zafar M. Shahriar,
Mohammed Atiquzzaman

TR-OU-TNRL-10-106
December 2010

Telecommunication & Network Research Lab

School of Computer Science

THE UNIVERSITY OF OKLAHOMA

110 W. Boyd, DEH, Room 260, Norman, Oklahoma 73019-6151
(405)-325-8077, atiq@ou.edu, www.cs.ou.edu/~atiq

Extension and ns2 Implementation of Saratoga

Abu Zafar M. Shahriar and Mohammed Atiquzzaman
School of Computer Science
University of Oklahoma, OK 73019, USA
Email: {shahriar, atiq}@ou.edu

CONTENTS

I. Introduction

Wood *et al.* [1] describe *Saratoga*, a UDP-based protocol that sends data at a rate independent of feedback rate, and performs loss recovery based on periodic feedback. *Saratoga* is used to download data from Internet-enabled Disaster Monitoring Constellation (DMC) satellites constructed by Surrey Satellite Technology. The six DMC satellites currently operational in low Earth orbit provide remote sensing images to support disaster relief.

At present Saratoga is being used in a controlled environment. The environment is controlled because a dedicated link with known bandwidth and delay parameters are being used, and drops only occurs due to the link error. In such environments, setting of parameters (e.g. data rate, period of feedback etc.), to get the optimal performance out of Saratoga, is simple. However, to find optimal values of the parameters for the use of Saratoga in uncontrolled environments, experimentation with Saratoga is required. Such environments are characterized by shared links with varying bandwidth and delay, and occurrence of heavy packet drops due to congestion or mobility. Moreover, use of optional features (e.g. voluntary feedback from the receiver for buffer management, congestion control) of Saratoga also requires experimentation. Simulation provides the flexibility for a wide range of experimentation in quick time and at low cost, and in theoretical environments. Therefore, our objective is to build a simulation prototype for Saratoga in ns-2 [2] which is a widely used simulation tool.

Apart from building a simulation prototype in ns-2, we also address the problem of using Saratoga in shared link environment. *Saratoga* could be used to download data simultaneously from multiple IP-enabled devices onboard satellites [3], rather than using the scheduled one-file-only-after-another model that currently avoids competition, or could be used for transfers of data from remote-sensing systems directly to end users through the public Internet [4]. Such data transfers will be over links which are shared by co-existing flows or other protocols, predominantly TCP, in the Internet. Therefore, our objective is to design a self- and TCP-friendly congestion control mechanism for Saratoga.

We have developed a simulation prototype of Saratoga in ns-2. Our prototype contains most of the basic features described in [1]. In addition, we have designed and included the mechanisms for optional features such as, buffer management and congestion control described in [1]. Our congestion control mechanism is self- and TCP-friendly, and requires modification of sender functionalities only, and no modification is required to the protocol packet format and receiver.

Our work can be used in the following way:

- Simulation prototype can be used to experiment with Saratoga
- Simulation prototype is a framework for future extensions
- Evaluation of the performance of Saratoga in shared link environment

The rest of the report is organized as follows. Overviews of *Saratoga* and the congestion control mechanism for Saratoga are presented in Secs. II and III. Sec. IV presents the ns-2 implementation of Saratoga followed by concluding remarks in Sec. V.

II. A brief overview of Saratoga

Saratoga [1], [5] is a UDP-based transfer protocol initially intended for efficient use of one-hop highly-asymmetric private links having brief periods of connectivity, when as much data as possible must be transferred, and loss is due to link errors, rather than to congestion. *Saratoga*'s loss reporting is less chatty than TCP's acknowledgement mechanism to support constrained return channels. Functionalities of *Saratoga* are presented below.

A. Basic functionality

Figs. 1 and 2 shows the signaling diagram for data transfer which can be initiated by either the sender or the receiver. In sender initiated data transfer (Fig. 1), the sender sends meta data, describing the data to be transferred, to the receiver that, in response, sends a voluntary feedback, informing acceptance or rejection of the transfer, to the sender. In receiver initiated data transfer (Fig. 2), receiver first sends a request to the sender. Remaining of the data transfer is similar to the sender initiated transfer.

The sender in Saratoga sends packets at a constant rate independent of receiving feedback from receiver. Receiver keeps track of the lost data, and sends a feedback reporting the lost data in response to periodic requests from the sender. Sender retransmits lost data, and does not transmit any new data until all lost data are retransmitted.

B. Optional functionality

Apart from the functionality described in Sec. II-A, some optional functionality is also specified in [1]. In addition, we also propose some functionalities to enhance Saratoga. Following are the functionalities that we implement:

- Sending beacons [1]:
  Saratoga peers can indicate their presence and capabilities by sending beacons. Beacons can also be used to indicate that peers are connected.
- Congestion control [1]: See Sec. III.

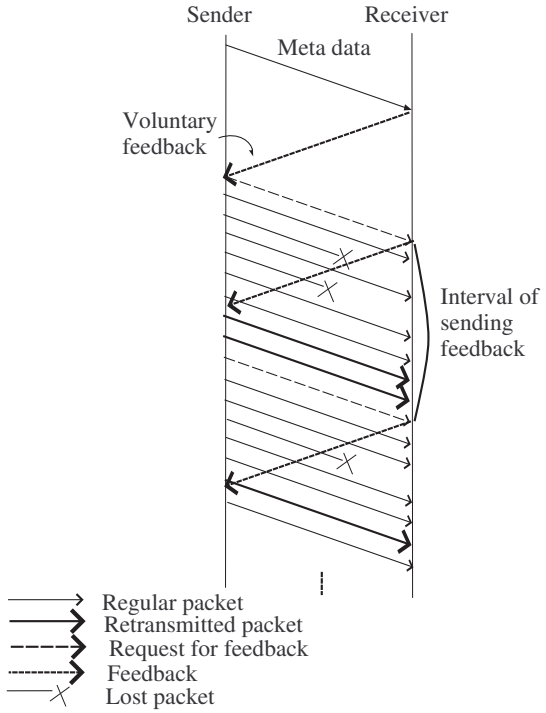

Fig. 1. Sender initiated data transfer

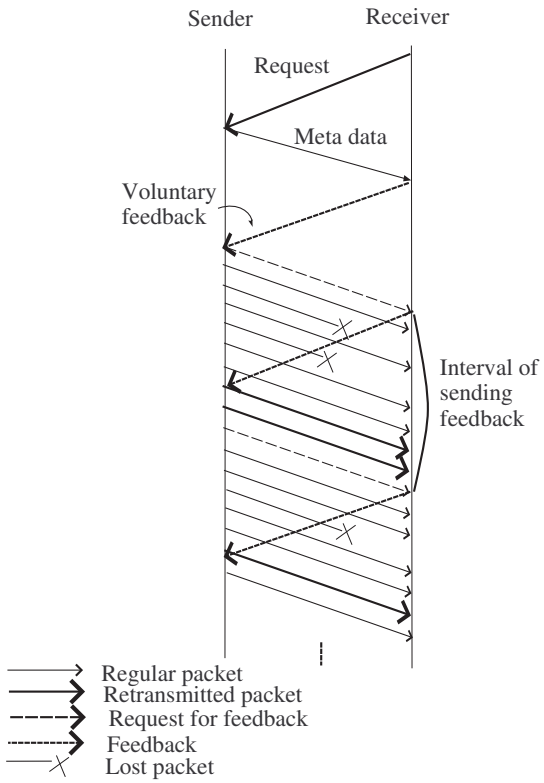

Fig. 2. Receiver initiated data transfer

- Sending feedback depending on buffer status: Too small a value of the period of sending feedback will consume more reverse link bandwidth, and processing. Too big a period will require large buffer at the receiver for partially received data, and a large buffer at the sender to reduce the probability of time-consuming re-fetching of data from disks to buffer when losses occur. Therefore, instead of sending periodic request for feedbacks, a sender can request for feedback or a receiver can send a voluntary feedback when the buffer is about to be exhausted.

III. Congestion control for Saratoga

A. Importance

Saratoga could be used to download data simultaneously from multiple IP-enabled devices onboard satellites [3], rather than using the scheduled one-file-only-after-another model that currently avoids competition, or could be used for transfers of data from remote-sensing systems directly to end users through the public Internet [4]. Such data transfers will be over links which are shared by co-existing flows or other protocols, predominantly TCP, in the Internet.

TCP is the most widely-used reliable and rate-controlled transport protocol when multiple competing flows share common links. TCP achieves reliability and rate control based on frequent acknowledgments that can be a limiting factor for data transfer when forward/back path asymmetry exceeds 50:1 [5]. That, and TCP's assumptions about loss being caused by congestion, makes TCP unsuitable for the environment Saratoga is intended for, as a single TCP flow will not be able to fully utilize available satellite link capacity.

A TCP-friendly rate control mechanism is needed in *Saratoga* to permit fair allocation of shared paths to TCP and other traffic, and to enable *Saratoga* to be used in the public Internet, rather than only in the private networks for which *Saratoga* was designed and developed.

B. State of the art in congestion control

Widmer *et al.* [6] provide an overview of known TCP-friendly rate control mechanisms. Given *Saratoga*'s data sending mechanism, we prefer the rate-based approach to the window-based one for better integration and to keep the functionality of the *Saratoga* sender simple. More importantly, considering current use of *Saratoga* in private space links where losses can be bursty and are not due to congestion, the conservative reaction of rate-based approaches to packet loss will provide better throughput performance than the more aggressive reaction of window-based approaches. Among the rate-based protocols proposed in the literature, TCP Friendly Rate Control (TFRC) [7], [8] suits *Saratoga* in terms of low needed feedback rate.

TFRC, which is specified by the Internet Engineering Task Force as a proposed standard, has been shown to perform very well for a variety of available link capacities and number of flows [7]. Therefore, we *aim* to use a TFRC-like rate control mechanism with minimal changes to the existing *Saratoga* protocol.

In TFRC, the sender controls the data rate using a model imitating the long run behavior of TCP, and requires two parameters from the receiver to do this - a measure of loss and of the receiver's throughput. It is *receiver-based* because the parameters are computed at the receiver, and are sent to the sender through periodic feedback. A *sender*-based TFRC has been proposed in [9], where the measure of loss is computed at the sender. This sender-based version of TFRC depends on the receiver for obtaining the receiver's throughput, and the feedback type differs from that in *Saratoga*. Thus, adopting existing TFRC mechanisms would require significant modification to the *Saratoga* protocol. Moreover, sending additional data in feedback, required for receiver-based TFRC, is undesirable in *Saratoga* when asymmetry and low return channel rates are present, as acknowledgement congestion on the return channel is a concern. Therefore, we *design* a true sender-based TFRC-like mechanism that controls the rate using the existing feedback specified in *Saratoga*, which anticipates use of TFRC or a similar mechanism [1]. Our mechanism resembles receiver-based TFRC, but computes parameters at the sender. An overview of receiver-based TFRC is presented in Sec. III-C.

C. TCP friendly rate control (TFRC)

TFRC is a rate control mechanism with a smoother throughput than TCP, while sharing bandwidth fairly with TCP [8]. Advantages of TFRC over TCP are the following:

- Less variation in instantaneous throughput. Particularly, TFRC does not cut the throughput to half (fast recovery) when a loss is detected.
- Rate-based mechanism suitable for rate-based transfer protocols.
- Requires less feedback from the receiver than what is required in TCP.

TFRC has the following disadvantages:

- Slower than TCP to respond to availability of bandwidth.

The basic underlying principles of TFRC, and functionalities in TFRC senders and receivers, are now given.

1) Basic principle: TFRC uses the following simplified form of the model [10] of TCP to compute the data rate (X) as a function of packet size (s), Round Trip Time (RTT) (R), a notion of loss (p), an approximation of TCP timeout value (t_{RTO}), and the number of packets acknowledged by a TCP-acknowledgement (b):

$$X = \frac{s}{R \times \sqrt{2bp/3} + t_{RTO} \times \left(3 \times \sqrt{3bp/8}\right) \times p \times (1 + 32p^2)}$$

where p is computed and sent by the receiver to the sender. For computation of p, losses separated by a time period of an RTT or more are recognized as loss events. p is the reciprocal of the number of packets sent between the start of two successive loss events, and called the loss event rate.

2) Receiver functionality: The steps that the receiver executes are shown in Fig. 3, and described below.

1) At reception of a data packet, the receiver records the reception time. If loss of packets is detected, the supposed reception times of lost packets are interpolated using the times and sequence numbers of packets received right before and after the loss. These times are used to update a *history of lost/received packets*. The loss event rate is computed from the history. If the loss event rate has increased compared to the last computed value, the receiver's throughput is computed, and this information is sent as feedback to the sender.
2) Receiver should send a feedback at least every RTT if data packets have been received since the last feedback was sent. To send a feedback, the loss event rate and the receiver's throughput are computed using the history of packets.

3) Sender functionality: The steps executed by the sender in the receiver-based TFRC are shown in Fig. 4, and are as follows:

1) Sender starts sending data with an initial rate.
2) Sender tracks the weighted average of RTT and an approximation of the TCP timeout value. When feedback is received from the receiver, the sender updates the weighted average of RTT, and approximates the TCP timeout using the RTT. The loss event rate and the receiver's throughput contained in the feedback are used to update the sending rate. If the loss event rate is zero, the sender doubles the current rate, which is bounded by twice the receiver's throughput at the higher side, and one packet every RTT at the lower side. Otherwise, the sender computes X as the sending rate, which is bounded by twice the receiver's throughput at the higher side and one packet every 64 seconds at the lower side.
3) If no feedback is received for a certain period of time, the sender cuts down the rate to half.

D. Sender-based TFRC (STFRC) for Saratoga

In this section, we present the challenges in designing the Sender-based TFRC (STFRC) for *Saratoga*, our approaches to meet the challenges, and the algorithms used for the proposed STFRC.

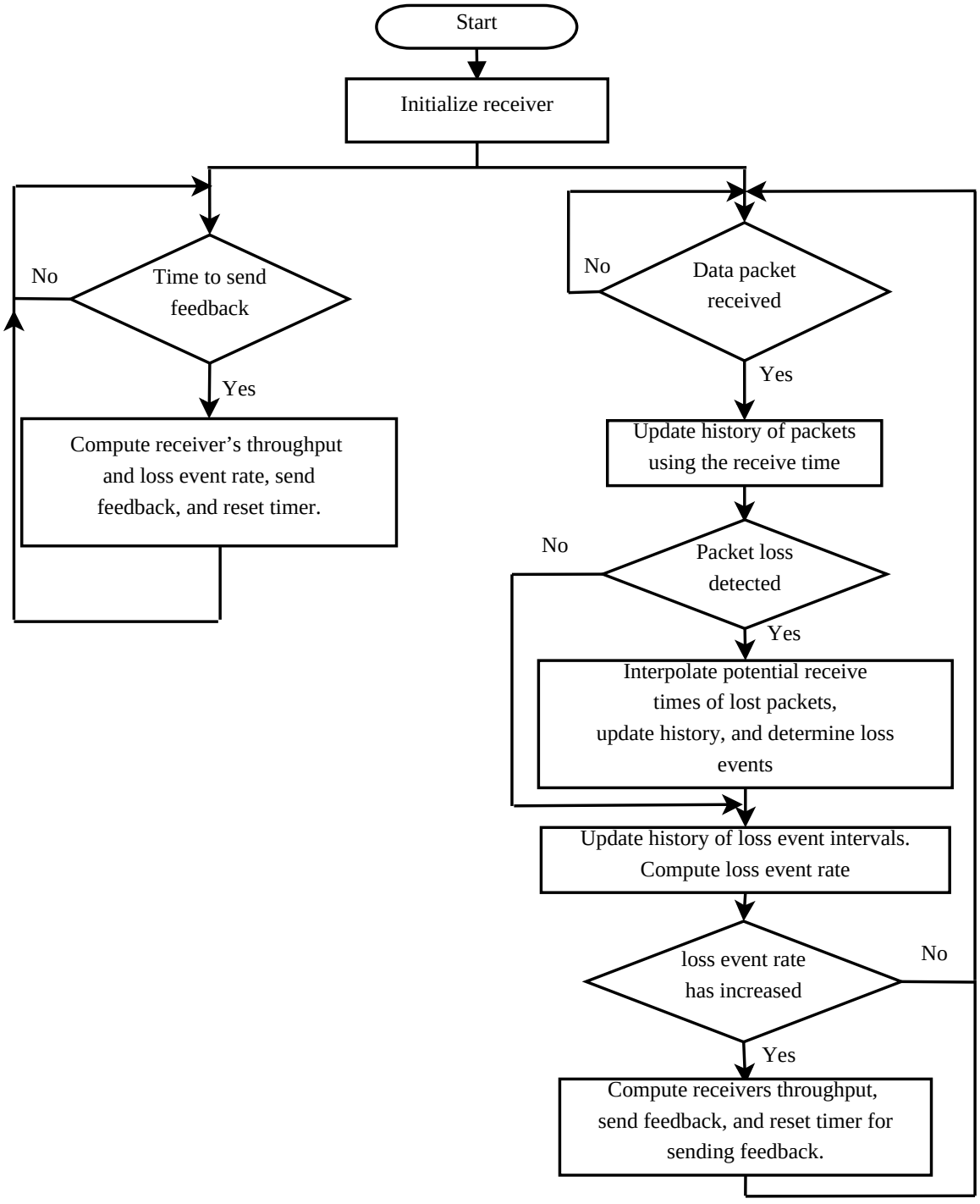

Fig. 3. Receiver's algorithm in receiver-based TFRC.

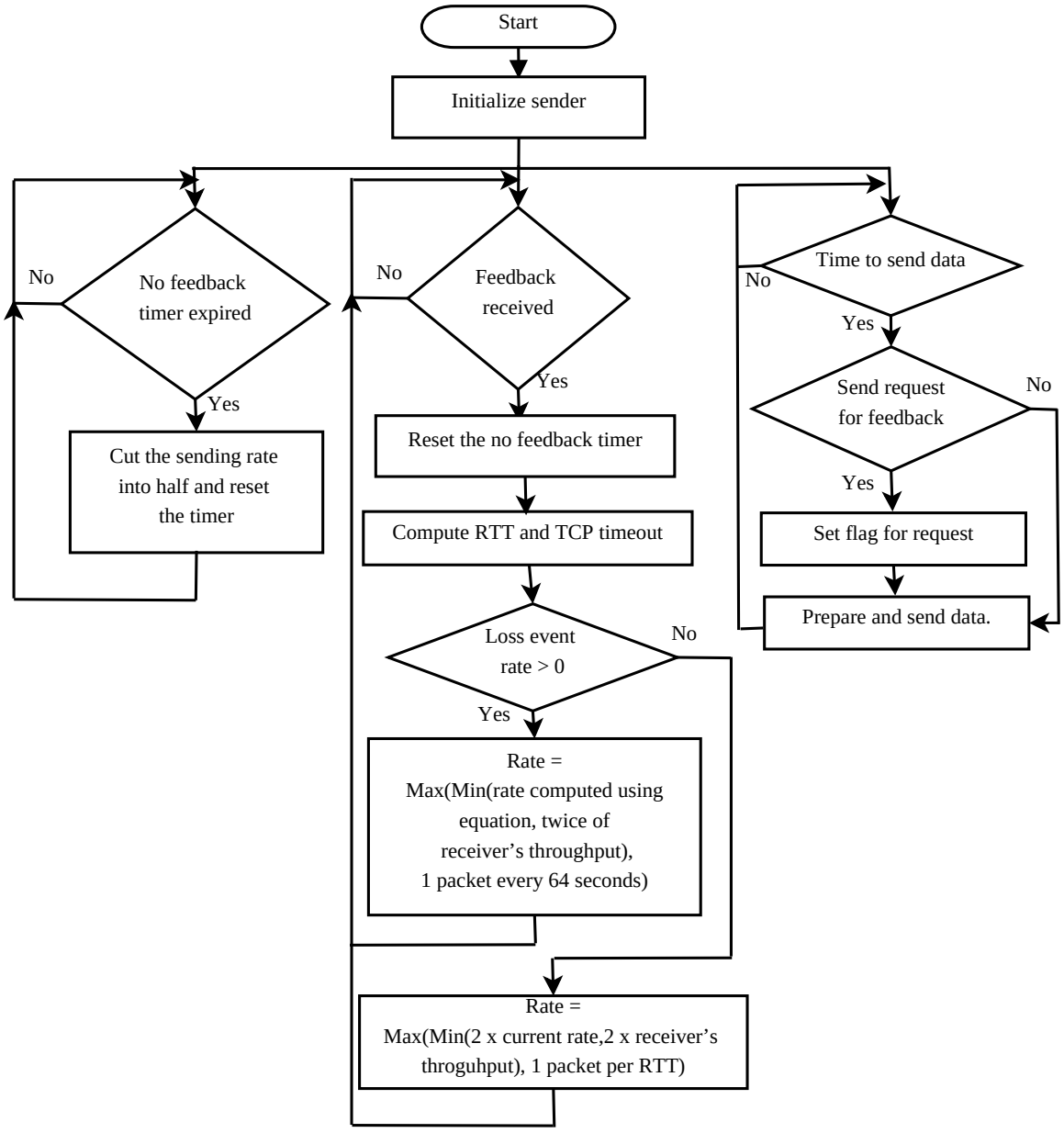

Fig. 4. Sender's algorithm in receiver-based TFRC.

1) Overview of our approach: In STFRC for *Saratoga*, the receiver is unchanged from *Saratoga*. The sender performs all rate-control related functionalities of TFRC. These functionalities include building the history of packets followed by the computation of the receiver's throughput, the loss event rate and the sending rate. Building the history of packets requires reception-times and packet-delivery-fates. Since the sender can only learn the loss status of packets when a feedback packet (so-called STATUS-feedback packet) is received, the updating of the history and computations are performed at reception of a STATUS-feedback packet. Reception-times are predicted from send-times of packets and the RTT. Therefore, the sender records the send-time of a packet in the history at the time of sending, and adds a fraction of RTT to the send-time when the STATUS-feedback packet is received. Also, the packet-delivery-fates are marked in the history using the report of loss contained in the STATUS-feedback packet. After updating the history, the computations are performed and the newly-computed sending rate is used till the next STATUS-feedback packet is received. Given Saratoga's feedback mechanism, implementation of the functionalities raises some challenges in designing the STFRC for *Saratoga*. We present these challenges, our approaches to meet the challenges, and the algorithms used for the proposed STFRC in the following subsections.

2) Challenges to design the STFRC for Saratoga*:* Building the history of packet delivery requires the reception-times of packets. Lost and retransmitted (or delayed and retransmitted) instances of the same packet must be uniquely identified in the history for the accurate computation of the average receiver's throughput over the last RTT, and for the computation of the

value of p.

These requirements give rise to the following challenges to design the STFRC for *Saratoga*:

- Determining the reception-times: As packet reception-times are neither known to the sender nor sent from the receiver, the sender has to predict the reception-times as correctly as possible without incurring a significant overhead.
- Unique identification of packets: In *Saratoga*, the receiver sends offsets of data in STATUS-feedback packets to report lost and received packets. These offsets are not unique or sequential, due to losses and retransmissions. The same losses may be reported in multiple STATUS-feedback packets if multiple requests are received before receiving the lost packets reported in the first of the STATUS-feedback packets. Therefore, a mechanism is needed to uniquely identify packets without incurring too much overhead.

3) Our approaches to meet the challenges: We address the above-mentioned challenges as follows:

a) Prediction of reception-times of packets: An estimation of the forward-path-delay is obtained by multiplying the RTT with a *Symmetry Ratio*, defined as the ratio of the average forward-path-delay to the average RTT. Since the size of packets on the forward path is larger than that on the reverse path, the *Symmetry Ratio* may not be equal to 0.5 because of the larger transmission delay and larger probability of developing congestion at local exit routers (considering equal delay at intermediate routers). Therefore, it would be better to obtain the factor from the long-term knowledge of the network or using a low-overhead mechanism to estimate the forward delay periodically at the cost of increased overhead. Reception-times are obtained by adding the forward delay to the send-times of packets. The RTT can be measured periodically using timestamps in the STATUS-feedback packet. (Timestamps are optional in *Saratoga*.)

b) Unique identification of packets: Unique identification is required for a history of packet-delivery-fates that can be updated by the sender when a STATUS-feedback packet is received. Although the STATUS-feedback packet contains a report up until the packet requesting the STATUS-feedback packet to be sent, the history may contain send-times of packets sent after sending the packet carrying the request. This happens because the sender will continue to record the send-times of packets that will be sent between the time of sending the packet carrying the request and the time of receiving the STATUS-feedback packet that was requested. When the STATUS-feedback packet is received, the sender needs to move back along the history from the time of receiving the STATUS-feedback packet to the time of receiving the packet that sent the request for the STATUS-feedback packet. This is done to confine the updating only to those packets whose receive-times and delivery-fates can be determined from this particular STATUS-feedback packet. Therefore, the history is updated by determining the receive-times and delivery-fates of the packets sent during the time period between sending two successive successfully-answered requests for STATUS-feedback packets.

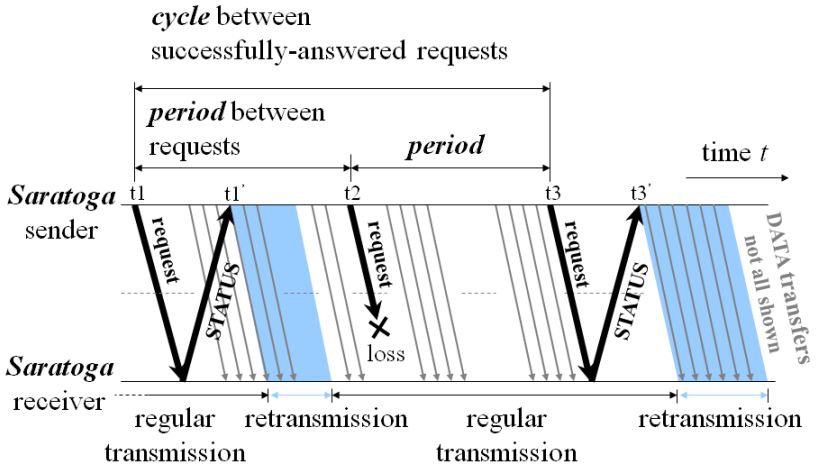

Fig. 5. Ladder diagram showing various transmission events in *Saratoga*.

To better explain which packets in the history are updated at reception of a STATUS-feedback packet, we introduce Fig. 5 demonstrating various events that may happen during an ongoing transmission in *Saratoga*. While the data transmission is going on, and a request for STATUS-feedback packet is due, the sender sends a request with the very next packet sent at time $t1$. At reception of the packet with the request, the receiver responds with a STATUS-feedback packet containing a report of losses of packets received so far. Assuming packets arrive at the receiver in the sequence they are sent, this STATUS-feedback packet received at time $t1'$ will contain report up until the packet sent at time $t1$. The sender continues to send packets and record send-times in the history after sending the packet at time $t1$. Thus, when the STATUS-feedback packet is received, the history will contain packets up until the packet sent at time $t1'$. However, the conversion of send-times to receive-times and

marking delivery-fates of packets are performed for those packets (whose send-times are in the history) that were sent up until the packet sent at time $t1$.

The next request is sent at time $t2$ and is lost on the way (alternatively, the corresponding STATUS-feedback packet might get lost). We reckon the time (e.g. from $t1$ to $t2$ or from $t2$ to $t3$) between sending two successive requests as a *period* which is significant because various information (presented later in the section) regarding some packets, sent during the *period*, is recorded. At time $t3$, the sender sends the next request which is answered successfully by the receiver. At reception of this STATUS-feedback packet at time $t3'$, the sender updates the history of packets that were sent between time $t1$ and $t3$. We call this time period, $(t3 - t1)$, a *cycle*. A *cycle* may consist of one or more *periods*.

The identification of packets sent during a *cycle* is required for updating the history. Unique identification of packets sent during a *cycle* would be possible by storing their offsets and send-times. But this is inefficient due to the requirement for large amounts of memory to store offsets and additional processing to identify packets using the send-times. Inefficiency increases when the number of packets sent during a *cycle* is large due to the high sending rate, and/or the loss of request/STATUS-feedback packets resulting in a long *cycle*. Therefore, we use an alternative method for identification of packets.

For the identification of packets using the alternative method, we introduce dummy sequence numbers used only within the sender. When a packet is sent, the send-time is recorded in the history, and the packet is assigned a unique dummy sequence number which is used to identify the packet in the history. At reception of a STATUS-feedback packet, the sender determines the range of dummy sequences of the packets sent during the *cycle* ended by this STATUS-feedback packet. A range determination is required because the history might contain packets sent in previous and subsequent *cycle*s. Dummy sequences of lost packets are also determined from the offsets of lost data reported in the STATUS-feedback packet, and from the information described below. The range of dummy sequences, and the dummy sequences of lost packets are used to identify the packets in the history to convert their send-times to receive-times by adding the predicted forward delay, and to mark them as lost/received.

For the identification using dummy sequences, we recognize the following sets of transmissions, shown in Fig. 5, that might happen during a *period*:

- Set I of regular transmission: First set of regular transmission starts with the packet sent after the packet carrying a request (e.g. sent at time $t1$), and occurs until the reception of a STATUS-feedback packet. If the STATUS-feedback packet does not report any loss, then this set of transmission continues until sending the next request (e.g. up until the packet sent at time $t2$).
- Set of retransmissions: Retransmissions (shown in dark background in Fig. 5) start after receiving a STATUS-feedback packet containing reports of losses, and continues until all lost data are retransmitted. The set of retransmissions contains all the retransmissions that occur in the *period*. Retransmissions may not occur if no loss is reported.
- Set II of regular transmission: It starts after the end of retransmissions (if occurs), and continues until sending the next request (e.g. up until the packet sent at time $t2$).

The following information is recorded for identification purposes:

- Dummy sequences and send-times of the first and the last packet of each period: These are required to identify the packets sent during a cycle, and to determine the span of a cycle.
- Offset to dummy sequence mapping for all retransmitted packets: Since offsets of retransmitted packets are not sequential, these are required to find the dummy sequences of packets lost from retransmitted packets.
- Offsets and dummy sequences of the first packets of each set of regular transmissions: Since offsets of regular packets are sequential, dummy sequences of packets that are lost from regular transmissions can be obtained from offsets of lost packets using the first packet's dummy sequence and offset, and the packet size.

The information is recorded for each *period* because a *period* is a potential *cycle*. If two STATUS-feedback packets sent in response to two successive requests marking a *period* are received, the *period* is a *cycle*. If a packet carrying a request or the corresponding STATUS-feedback packet is lost, the *cycle* consists of more than one *period*. When a STATUS-feedback packet is received, the sender uses the timestamp in the packet and the time of the last packet sent in each of the periods to determine whether the *cycle* consists of multiple *period*s or not, and the information for those *period*s are merged for the *cycle*.

4) Sender algorithms: The steps that are executed by the sender to implement the approaches mentioned in this section are shown in Algorithm 1 and 2, and are discussed below.

The steps given in Algorithm 1 are required for unique identification of packets, and for congestion control. These steps are in addition to the steps that are executed by a *Saratoga* sender without congestion control.

Algorithm. 2 lists the steps that are executed by a sender in STFRC for *Saratoga* when a STATUS-feedback packet is received.

In Algorithm 2, Step 6 is similar to the computation of the sending rate by a TFRC sender discussed in Sec. III-C3, whereas Step 7 is for the response of a typical *Saratoga* sender. The requirement for Step 2 is explained next. As discussed earlier in this section, a *cycle* consists of multiple periods when requests or STATUS-feedback packets are lost. This requires merging of multiple *period*s into a *cycle*. The reason for finding the range of dummy sequences of packets sent in a *cycle* has been explained earlier.

Algorithm 1 Sender's algorithm when a packet is sent.

1: Detect the transmission set type, and record sequence numbers, offsets and send-times of packets as discussed in Sec. III-D3.
2: **if** (a request for a STATUS-feedback packet is due) **then**
3: Record the dummy sequence and the send-time of the packet to mark the end of a *period*.
4: **end if**
5: **if** (a request was sent with the previous packet) **then**
6: Record the dummy sequence and the send-time of the packet to mark the start of a *period*.
7: **end if**
8: Store the send-time of the packet in the history.

Algorithm 2 Sender's algorithm when a STATUS-feedback packet is received.

1: Update the RTT and the TCP-timeout.
2: Identify the *cycle* i.e. the dummy sequence number of the last packet ending the *cycle*. This might require merging of multiple *period*s into a *cycle*.
3: Using the recorded information specified in Algorithm 1, offsets of the lost packets reported by the receiver, and the information from Step 2, find dummy sequences of the packets lost in the *cycle*.
4: Update receive-times and delivery-fates (received or lost) of the packets sent during the *cycle*.
5: Estimate the receiver's throughput and compute the value of p.
6: Compute the sending rate.
7: Prepare the retransmission list from the loss reported.

Step 3 is required for two reasons. First, a STATUS-feedback packet may report losses that have been already reported by previous STATUS-feedback packets because the STATUS-feedback packet was sent before the retransmitted packets have reached the receiver. Therefore, packets lost in the current *cycle* have to be identified. Second, offsets of lost packets have to be mapped to dummy sequences to update the packet-delivery-fates in the history. Identification and mapping have been discussed earlier in this section.

In Step 4, times and delivery-fates of packets are updated in the history based on the cycle identified in Step 2, and the dummy sequences of lost packets obtained in Step 3. In Step 5, the receiver's throughput and the value of p are determined in a similar way it is done in TFRC [8].

IV. NS-2 IMPLEMENTATION

We implement Saratoga as an ns-2 *Agent* that construct or consumes packets. Following are the requirements for the implementation:

- Implementation of various types of packets.
- Implementation of states and functionalities required for the sender and the receiver.
- Implementation of the user interfaces to provide flexibility of experimentation.

Fig. 6 shows the abstraction of ns-2 implementation of Saratoga with major modules and their interactions. An arrow connecting two modules indicates the dependency of the modules. The variables and functionalities in the module, where the arrow originates, are used by the module where the arrow terminates. All modules are implemented in *saratoga.h,.cc* files that are added to *app* folder in ns-2 directory structure. In addition, *ns-agent.tcl* file was modified to initialize shadow variables in TCL. Details of the required components and functionalities of Saratoga, related modules and user interfaces are presented below:

A. Saratoga packet header

Protocol packet format, abstracted by the module *Protocl Packet Header*, is required to simulate various types of packets exchanged between Saratoga peers. This is implemented as follows:

- **File**: saratoga.h
- **Structure**: hdr_saratoga - Defines necessary variables to represent various fields, such as packet types, sequence numbers etc. of different packets header.

In addition, following definitions were added to *packet.h* to simulate various packet types used in Saratoga:

- PT_BEACON: For beacons.
- PT_SREQ: For request packets.
- PT_SMETA: For meta data packets.
- PT_STATUS: For feedback packets.
- PT_SDATA: For data packets.

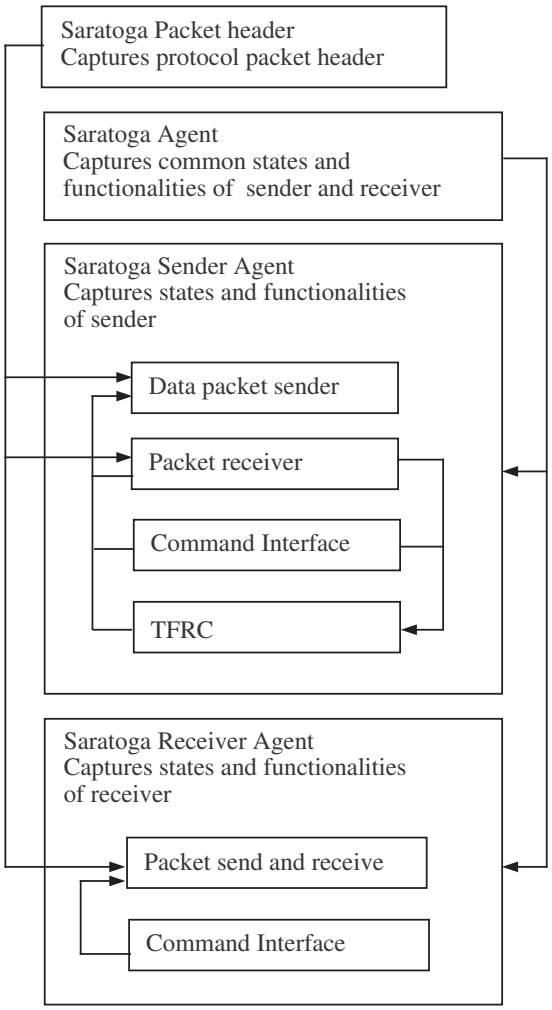

Fig. 6. Abstraction of Saratoga implementation in ns-2.

B. Saratoga Agent

Module *Saratoga Agent* implements the states and functionalities that are common to both sender and receiver. This includes packet size being used, file size specifier, congestion control related state variables and debugging related functionalities. It is implemented as follows:

- **Files**: saratoga.h,saratoga.cc
- **Class**: *SaratogaAgent* - This contains the following *methods*:
  - **print_rx_list** - Used for printing list of packets to be retransmitted for debugging purpose.
  - **hole_data_size** - Determines the amount of space required in the feedback packets to report lost packets. It is used by receiver module to decide whether fragmentation of feedback packet is required.

C. Saratoga sender agent

Module *Saratoga Sender Agent* captures the sender states and functionalities, Following are the implementation details of this module:

- **Files**: saratoga.h,saratoga.cc
- **Class**: SaratogaSndAgent - The class contains the state variables and methods to implement the sender. The methods can be categorized depending on their purpose of use, and are presented below:
  - *Data Packet Sender*: The methods in this category implements the functions of initiating communications, and preparing and sending packets. Following are the methods in this module:
    - **start_data** - Initialize the interval of sending data packets, computes the packet size based on specified payload and header options, and triggers the event to send the first packet after successful completion of handshaking.

    - **sendseg** - It is called periodically to prepare and send a data packet. Sending a request for feedback packet, retransmission from the list of packets to be retransmitted are also performed by this method. It also performs mapping of dummy sequence numbers to sequence numbers of regular or retransmitted packets for congestion control.
    - **status_required** - Determines whether it is time to send a request for feedback depending on the condition of the buffer at the sender.
  - *Packet Receiver*: Methods under this category implements the functions of receiving packets, and initiate responses accordingly.
    - **recv** - Receives packets, and initiate responses depending on the type of packets received.
    - **recv_req** - Responds to the reception of a request packet, sent from the sender, by sending a meta packet. This method is also used by **start** in method in *Command Interface* category to start data transfer from the sender side.
    - **prepare_meta_pkt and send_meta** - It prepares and sends meta packets, respectively.
    - **recv_status_pkt** - It responds to the reception of a feedback by initiating preparation of a retransmission list, and updating the buffer.
    - **prepare_rx_list** - Prepares the retransmission list using the feedback.
  - *TFRC*: The methods in this category perform necessary actions to control rate. Some functionalities are performed by the **sendseg** method in *Data Packet Sender* category, and is mentioned here.
    - **sendseg** - Detect phase of transmissions and record information for cycle detection while sending packets. Also, records the dummy sequences of packets.
    - **tfrc_on_feedback** - When a feedback packet is received, this method executes the major steps of the sender-based TFRC to compute the sending rate. It directly or indirectly uses the methods described below except the last one.
    - **find_cycle** - Uses the recorded information regarding phases of transmissions to find a valid cycle identifying the time period between two successive successful requests for feedback. A successful request is the one for which a feedback has been received.
    - **prepare_dholeslist** - Identifies the those packets that are lost from the packets sent during the identified cycle, and prepares the list of lost packets in terms dummy sequences.
    - **update_rcv_list** - Updates the history of packets using the predicted receive times of packets sent during the cycle. Also, marks the lost packets in the history using the list of lost packets.
    - **estimate_recv_rate** - Using the history of packets, estimates the receiver's throughput over the last RTT period.
    - **estimate_p** - It computes the loss event rate as the reciprocal of the average loss event interval.
    - **estimate_loss_events** - Detects loss events from the history of packets, updates the history of loss event intervals.
    - **find_mean_interval** - Find the weighted average of some previous loss event intervals.
    - **estimate_tcp_rate** - Computes the average rate of the TCP sender using Eqn. 1.
    - **determine_rate** - Determine the sending rate as described in Step 2-b in Sec .III-C3.
    - **no_feedback_action** - This method updates the sending rate as described in Step 3 in Sec . III-C3, when a feedback is not received within a time period determined using the RTT and sending rate.
  - *Command interface*: Methods in this category are used to implement direct user commands given from TCL.
    - **start** - Starts handshaking by sending meta packets that are used in Saratoga at the start of a communication. It is used to start sender initiated data transfer.
    - **stop** - Stop sender's activity i.e. sending packets.
    - **print_stats** - Used to print statistical data for debugging purposes.
  - *Buffer management*: Methods in this category are used to implement buffer management at the sender to determine if a feedback is required to avoid buffer overflow. **update_buffer** is the method that updates the left and right limits of the buffer when data is sent or a feedback is received.

D. Saratoga receiver agent

Module *Saratoga Receiver Agent* abstracts the states and functionalities of a Saratoga receiver, and is implemented in class *SaratogaRcvAgent*. Following are the implementation details:

- **Files**: saratoga.h,saratoga.cc
- **Class**: SaratogaSndAgent - The class contains the state variables and methods to implement the sender. The methods can be categorized depending on their purpose of use, and are presented below:
  - *Packet send and receive*: Methods in this category are used for receiver's sending and receiving packets, and for handshaking purposes. The methods are as follows:
    - **update_buffer** - Updates the buffer left and right when a packet is received.
    - **report_holes** - Determines when a feedback has to be sent due to low buffer space.
    - **recv** - Receives packets and initiate responses depending on the type of a packet.

    - **send_status** - Sends a feedback packet when a data packet with the request for feedback is received or the buffer space if low.
    - **prepare_req_pkt** - Prepares a request packet by inserting necessary information in the header.
    - **send_req** - Sends a request packet when sender wants to initiate data transfer.
    - **recv_data_pkt** - When a data packet is received, it detects whether a packet is a new, retransmitted or duplicate packet, and updates the record of lost packets.
    - **recv_meta_pkt** - Respond to the reception of a meta packet from the sender by sending a feedback packet.
    - **recv_beacon** - Respond to the reception of a beacon.
    - **find_allowed_size** - Determines the space required to report lost packets, and whether the report needs to be fragmented into multiple packets.
    - **retry_handshaking** - Retry handshaking in the event of loss of a packet used for handshaking.
  - Command interface: These methods are related to the provision of user interfaces from TCL, and are as follows:
    - **start** - To initiate communication from the reciever, this method initiates the sending of a request packet.
    - **on** - Turns on sending voluntary periodic feedback from the receiver when buffer management is not used.

E. A summary of user interfaces

This section summarizes the *TCL* user interfaces available to the user of this implementation. Interfaces are used to initialize and set various state variables, and start or stop the activity of the sender/receiver.

1) User interfaces common for the sender and the receiver: These user interfaces are used to set various states that are used in both sender and receiver. The shadow variables corresponding to these interfaces belong to the TCL class **Agent/Saratoga**, and are as follows:

- **descriptor_length_** - The value specified in bytes is used to define the descriptor length to be used by Saratoga peers.
- **payload_size_** - Set the size of payload of data packets. The value is specified in bytes.
- **binterval_** - Set the interval of sending beacons in seconds.
- **retry_period_** - Interval in seconds to retry handshaking in case a packet is lost.
- **cgs_cntrl_** - Used to specify whether the rate control will be used or not. If it is set zero, the sender will send data in a uniform rate. Otherwise, TFRC will be used for rate control.
- **shadow_tfrc_** - If it is set non-zero, rate control uses receiver-based TFRC except that the parameters computed at the receivers are directly used at the sender rather than being sent through feedback packets. The objective of this is to compare the efficiency of the sender-based TFRC with that of the receiver-based TFRC.
- **dbg_** - To enable printing of messages for debugging.
- **dbg_tfrc_** - To enable printing of messages for debugging of TFRC.

2) User interfaces for the sender: The shadow variables corresponding to these interfaces belong to the TCL class **Agent/Saratoga/SaratogaSnd**, and are as follows:

- **srate_** - The sending rate in bits per second. It must be specified if *cgs_cntrl_* is set zero.
- **hinterval_** - The interval sending periodic request for feedback in seconds.
- **iinterval_** - If no feedback is received for this time period specified in seconds, the sender stop sending data packets.
- **optimistic_** - If the value is set non-zero, the sender starts sending data packets after sending the meta packet without waiting for the first feedback packet to be received from the receiver.
- **buffer_size_** - Specify the buffer size at the sender in mega bytes.
- **safe_factor_** - The minimum tolerable value for the ratio of the amount of unsent data in the buffer to the amount of data that can be sent during the next RTT. If the ratio goes below this value, a request to send a feedback is sent with the next data packet when buffer management is being used. The higher the value, the lower the risk of overrunning the buffer.

Apart from the above shadow variables, following are the TCL methods defined in the same class:

- **start** - This method starts the sender initiated data transfer. The flow of execution starting from this method is shown in Fig. 7. While normal line indicates explicit flow, dashed line indicates indirect flow through the use of timers.
- **stop** - It stops sender's activity.
- **print-stats** - Used for debugging purposes.
- **get-receiver** - Used to obtain a reference to the receiver to simulate receiver-based TFRC. Required to compare the sender-based TFRC with the receiver-based one.

3) User interfaces for the TFRC: The shadow variables corresponding to these interfaces also belong to the TCL class **Agent/Saratoga/SaratogaSnd**, and are as follows:

- **start_rate_** - The initial rate, specified in bits per second, for the sender.
- **numsamples_** - The number of samples to be used to compute the mean loss event interval.

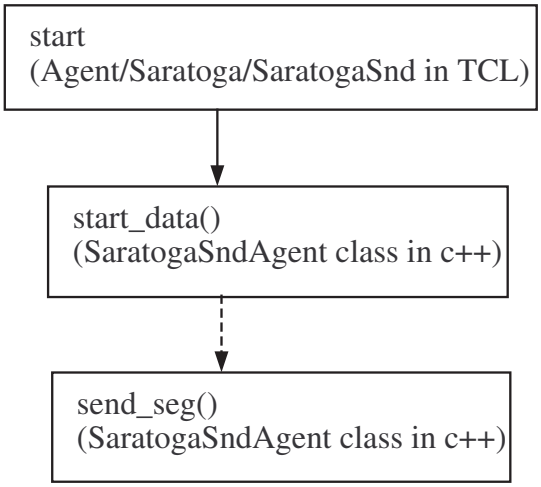

Fig. 7. Execution flow starting from the TCL interface **start** at the sender.

- **no_feedback_interval_** - If no feedback is received for the specified time period in seconds, the sender responds by executing **no_feedback_action** method.
- **symmetry_** - The fraction of RTT considered as the forward delay.
- **rtt_factor_** - A fraction of the expected value of RTT used to combine with recent sample of RTT to compute the new expected value of the RTT.
- **rtt_factor2_** - A fraction used to compute long term expected value of the RTT to make the change of sending rate smooth. Used in a similar way **rtt_factor_** is used.
- **smoothing_** - If a non-zero value is set, smoothing technique as specified in [8] is used to make the rate of change of the sending rate smooth.
- **hsz_** - Amount of packets for which history can be accommodated at any time instance.
- **gdf_** - See history discounting mechanism in [8]. History discounting mechanism is used to gradually reduce the weight of the older samples of loss event intervals.
- **hd_** - If set non-zero, history discounting mechanism is enabled.
- **threshold_** - A minimum value for **gdf_**.
- **overhead_** - A value of either zero or one used to randomize the packet sending interval.

4) User interfaces for the receiver: The shadow variables corresponding to these interfaces belong to the TCL class **Agent/Saratoga/SaratogaRcv**, and are as follows:

- **hsinterval_** - Interval of sending a voluntary feedback in seconds.
- **iinterval_** - If no data packet is received for this time period specified in seconds, the receiver stops its functionality.
- **buffer_size_** - Size of the buffer in the receiver for data that have not been received in sequence.

Apart from the above shadow variables, following are the TCL methods defined in the same class:

- **start** - This method starts the receiver initiated data transfer. The flow of execution starting from this method is shown in Fig. 8.
- **on** - Turns on the voluntary sending of feedback by the receiver.
- **print-stats** - Used for debugging purposes.

V. Conclusion

This report provide a documentation of the newly designed congestion control mechanism for Saratoga and its ns-2 implementation. The congestion control mechanism, requiring minimal changes to the protocol, enables Saratoga to be used in shared link environments. ns-2 implementation of Saratoga will facilitate experimenting with Saratoga, and thus, will lead to the development of the protocol for widespread use and optimal performance.

References

[1] L. Wood, W. M. Eddy, C. Smith, W. Ivancic, C. Jackson, and J.Mckim, "Saratoga: A scalable file transfer protocol," work in progress as an Internet-draft, Sep. 2010.
[2] K. Fall and K. V. (eds.), "ns notes and documentation," http://www.isi.edu/nsnam/ns/.

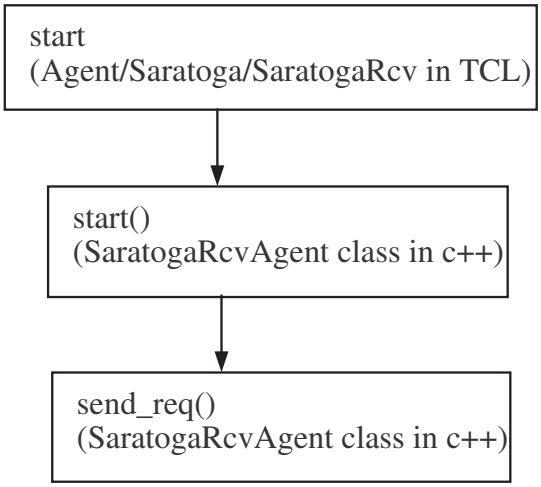

Fig. 8. Execution flow starting from the TCL interface **start** at the receiver.

[3] A. Z. M. Shahriar, M. Atiquzzaman, and W. Ivancic, "Performance evaluation of NEMO in satellite networks," in *IEEE Military Communications*, San Diego, CA, Nov. 17-19, 2008.
[4] K. Delin, Y. Chao, and L. Lemmerman, "Earth science system of the future: observing, processing, and delivering data products directly to users," in *IEEE International Geoscience and Remote Sensing Symposium*, University of New South Wales, Sydney, Australia, Jul. 9-13, 2001.
[5] L. Wood, "Saratoga a bundle convergence layer," Presentation, Delay-Tolerant Networking session IETF 69, Chicago, Jul. 2007.
[6] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-Friendly congestion control," *IEEE Network*, vol. 15, pp. 28–37, 2001.
[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *ACM SIGCOMM*, Stockholm, Sweden, Aug. 28-Sep. 1, 2000.
[8] ——, "TCP Friendly Rate Control (TFRC): protocol specification," RFC 5348, Sep. 2008.
[9] G. Jourjon, E. Lochin, and P. Sénac, "Towards sender-based TFRC," in *IEEE International Conference on Communications*, Glasgow, Scotland, Jun. 24-28, 2007.
[10] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation," in *ACM SIGCOMM*, Vancouver, B.C., Canada, Sep. 2-4, 1998.

APPENDIX

A sample script showing the usage of ns-2 prototype for Saratoga.

```
ns-random 0

global ns set ns [new Simulator]

#############################################################################
# Global configuration parameters #
#############################################################################
Agent/Saratoga set dbg_ 0       ;#for debug output Agent/Saratoga
set dbg_tfrc_ 0      ;#for debug output Agent/Saratoga set
payload_size_ 960 Agent/Saratoga set descriptor_length_ 4

Agent/Saratoga/SaratogaRcv set retry_period_ 2.0

Agent/Saratoga/SaratogaSnd set iinterval_ 66.0
Agent/Saratoga/SaratogaSnd set hinterval_ 0.04
Agent/Saratoga/SaratogaSnd set optimistic_ 0


Agent/Saratoga/SaratogaRcv set iinterval_ 66.0

#TFRC related parameters Agent/Saratoga set cgs_cntrl_ 1
Agent/Saratoga set rtt_factor_ 0.9

Agent/Saratoga set rtt_factor2_ 0.9
```

```
Agent/Saratoga/SaratogaSnd set start_rate_ 1;#pkt/s
Agent/Saratoga/SaratogaSnd set symmetry_ 0.75
Agent/Saratoga/SaratogaSnd set numsamples_ 8
Agent/Saratoga/SaratogaSnd set smoothing_ 1
Agent/Saratoga/SaratogaSnd set hd_ 1

Agent/Saratoga/SaratogaSnd set overhead_ 0

Agent/Saratoga set shadow_tfrc_ 0       ;#for debug

#pass these parameters from command line set tfrcflow [lindex $argv
0] set tcpflow [lindex $argv 1] set linkbw [lindex $argv 2] set
qqtype [lindex $argv 3] set runno [lindex $argv 4]

set flows [expr $tfrcflow + $tcpflow] set randomflows 0 if
{$randomflows  > $flows } {
    set randomflows $flows
}


global opt set opt(ifq) Queue/DropTail set opt(qtype) $qqtype set
opt(qlim)       50

set opt(start)      5 set opt(stop)       130


#############################################################################
# Tracing #
#############################################################################
set outfname "test.tr"

set outfile [open $outfname w] $ns trace-all $outfile

#############################################################################
# Create nodes                                  #
#############################################################################

$ns node-config -ifqType $opt(ifq) \
        -ifqLen $opt(qlim)

# Create saratoga source for {set i 0} {$i<$tfrcflow} {incr i} {
    set ss_($i) [$ns node]
} #sartoga destinations for {set i 0} {$i<$tfrcflow} {incr i} {
    set sd_($i) [$ns node];
} #tcp sources for {set i 0} {$i<$tcpflow} {incr i} {
    set tcps_($i) [$ns node];
} #tcp dest for {set i 0} {$i<$tcpflow} {incr i} {
    set tcpd_($i) [$ns node];
} set r1 [$ns node]; set r2 [$ns node];

#random tcp sources for {set i 0} {$i<$randomflows} {incr i} {
    set rtcps_($i) [$ns node];
} #random tcp dest for {set i 0} {$i<$randomflows} {incr i} {
    set rtcpd_($i) [$ns node];
}
```

```
#############################################################################
# Set up links #
#############################################################################
#TFRC for {set i 0} {$i<$tfrcflow} {incr i} {
    $ns duplex-link $ss_($i) $r1 100Mb 2ms DropTail
    $ns duplex-link $sd_($i) $r2 100Mb [expr $i/3]ms DropTail
} #TCP for {set i 0} {$i<$tcpflow} {incr i} {
    $ns duplex-link $tcps_($i) $r1 100Mb 2ms DropTail
    $ns duplex-link $tcpd_($i) $r2 100Mb [expr $i/3]ms DropTail
} #Random flows for {set i 0} {$i<$randomflows} {incr i} {
    $ns duplex-link $rtcps_($i) $r1 100Mb 2ms DropTail
    $ns duplex-link $rtcpd_($i) $r2 100Mb [expr $i/3]ms DropTail
}

$ns duplex-link $r1 $r2 [expr $linkbw]Mb 20ms $opt(qtype)
$opt(qtype)

#scaling q set t [expr $linkbw*25] $ns queue-limit $r1 $r2 $t $ns
queue-limit $r2 $r1 $t

if {$opt(qtype)=="RED"} {
    set redq [[$ns link $r1 $r2] queue]
    $redq set thresh_ [expr 3 + $t/15]
    $redq set maxthresh_ [expr 13 + $t/2]
}

#############################################################################
# Set up connection between wired nodes #
#############################################################################
proc lsrc_dst {} {

    global ns opt start stop ss_ sd_ tcps_ tcpd_ tfrcflow tcpflow src_ dst_
    defaultRNG mrng_ randomflows rtcps_ rtcpd_

    set rate 8Mb

for {set i 0} {$i<$tfrcflow} {incr i} { #saratoga peers
   set src_($i) [new Agent/Saratoga/SaratogaSnd]
   $src_($i) set descriptor_length_ 4
   $src_($i) set srate_ $rate
   $ns attach-agent $ss_($i) $src_($i)

   set dst_($i) [new Agent/Saratoga/SaratogaRcv]
   $dst_($i) set hsinterval_ 20000.0
   $ns attach-agent  $sd_($i) $dst_($i)

   $ns connect $src_($i) $dst_($i)


    #TFRC debug
    $src_($i) get_receiver $dst_($i)

   $ns at [expr $opt(start) - [$defaultRNG uniform 0 20]/20] "$src_($i) start"

} for {set i 0} {$i < $tcpflow} {incr i} {
    #TCP peers
    set tsrc_($i) [new Agent/TCP/Sack1]
```

```
    $tsrc_($i) set window_ 10000
    $ns attach-agent $tcps_($i) $tsrc_($i)
    set tdst_($i) [new Agent/TCPSink/Sack1]
    $ns attach-agent $tcpd_($i) $tdst_($i)

    $ns connect $tsrc_($i) $tdst_($i)

    set ftp_($i) [new Application/FTP]
    $ftp_($i) attach-agent $tsrc_($i)

    $ns at [expr $opt(start) - [$defaultRNG uniform 0 20]/20] "$ftp_($i) start"

} for {set i 0} {$i < $randomflows} {incr i} {
    #TCP peers
    set rtsrc_($i) [new Agent/TCP/Sack1]
    $rtsrc_($i) set window_ 20
    $ns attach-agent $rtcps_($i) $rtsrc_($i)
    set rtdst_($i) [new Agent/TCPSink/Sack1]
    $ns attach-agent $rtcpd_($i) $rtdst_($i)

    $ns connect $rtsrc_($i) $rtdst_($i)

    set rftp_($i) [new Application/FTP]
    $rftp_($i) attach-agent $rtsrc_($i)

    $ns at [expr $opt(start) - [$defaultRNG uniform 0 20]/20] "$rftp_($i) start"
    #$ns at $opt(start) "$ftp_($i) start"
} }

lsrc_dst

$ns at $opt(stop) "finish"

proc finish {} {
    global ns outfile src dst
    $ns flush-trace
    close $outfile
    printstats
    exit 0
} $ns run
```