Chapter 8

Implementing TRUMP/RBCC in the REAL Network Simulator

In order to test the rate-based congestion control framework in a complex environment which might highlight problems or undesirable behaviour, it was decided to simulate the behaviour of the framework. I began initial work on a home-grown network simulator, but this work was discarded when version 4 of the REAL network simulator was obtained.

The REAL network simulator is designed for testing congestion and flow control mechanisms. It was written by S. Keshav and is copyright by the University of California, Berkeley. A description of an early version of REAL is given in [Keshav 88], and REAL was used to perform the simulations described in [Keshav 91]. REAL's author states that "[REAL] has been installed at over 300 sites (but I have no idea how many people actually use it)".

REAL was chosen as the network simulator to test my congestion control framework for the following reasons:

- The simulator was available in full source code form, which allowed me to implement my own congestion framework.
- The simulator was known to run on several hardware platforms.
- The simulator already had several transport flow control schemes and router congestion control schemes already built into it. I would be able to compare my congestion framework against these schemes.

The main alternative to REAL, x-sim[1], was not publically released until after substantial modifications to REAL had been made, in order to implement my congestion framework. The latest version of REAL, version 5, became available after all of my experimental results had been obtained with version 4.

8.1 The REAL Simulator

The REAL 4.0 user's manual describes REAL thus:

> REAL is a simulator for studying the dynamic behaviour of flow and congestion control schemes in packet switch data networks. It provides users with a

[1]Written by Norm Hutchinson and Larry Peterson and others at the University of Arizona and elsewhere.

way of specifying such networks and to observe their behavior. Source code is provided so that interested users can modify the simulator to their own purposes.

The simulator takes as input a scenario, which is a description of network topology, protocols, workload and control parameters. It produces as output statistics such as the number of packets sent by each source of data, the queueing delay at each queueing point, the number of dropped and retransmitted packets and other similar information.

There are 11 source types, corresponding to 11 transport protocol and workload types. The sources can be categorized into one of two types: flow-controlled and non-flow-controlled data sources. Flow-controlled sources use acknowledgments and timeouts to implement a reliable transport layer functionality over a lossy network. Non-flow-controlled sources generate data from a known distribution and do not provide a reliable transport functionality; they model cross traffic.

The router implements several scheduling disciplines including:

fcfs: First-Come-First-Served.

fq: Fair queueing, as described in [Demers *et al* 89].

fqbit: This is identical to fq, except that it does bit-setting on packets. It is described in [Demers *et al* 89].

hrr: This implements the HRR discipline described in [Kalmanek *et al* 90].

decbit: This implements DEC bit-setting described in [Ramakrishnan & Jain 90], and FCFS queueing.

REAL 4.0 also provides several packet dropping mechanisms in routers:

Drop Tail: The newly-arrived packet is dropped.

Drop Head: The oldest queued packet in the router is dropped.

Drop Random: An already-queued packet in the router is randomly chosen and dropped.

Decongest First: The traffic flow with the most bytes queued in the router is found, and the most recently-arrived packet for the flow is dropped.

Decongest Last: The traffic flow with the most bytes queued in the router is found, and the oldest queued packet for the flow is dropped.

8.2 Changes to the Simulator

REAL 4.0 as distributed did not run on any of the platforms available within my department (mainly SPARC machines running Solaris). A slow SPARC machine running SunOS 3.5 was located in another department, and initial simulator use was on this machine.

During the course of my studies, REAL 4.0 was ported to the i386/486 platform under FreeBSD 2.x. This involved some substantial changes to the simulator, mainly to the assembly code which performs thread scheduling. At the same time, I cleaned up the code somewhat in order to reduce the number of warnings generated by the Gnu C compiler.

As well as porting REAL 4.0 to the i386/486 platform, I made further changes and additions to it in order to implement the TRUMP transport protocol and the RBCC rate-measurement algorithm. The changes required were:

- Support for the TRUMP node type and for RBCC_QUENCH and RBCC_TIMER packet types. Files affected: kernel/config.h.
- Extra code to allow an arbitrary number of arguments to be passed to each node. Files affected: kernel/parameters.h, kernel/sim.c, lang/lang.lex and lang/lang.yacc.
- Extra fields in each packet as required by RBCC, and data structures to hold the Allocated Bandwidth Tables. Files affected: kernel/types.h
- New results output code to bypass the limit on the number of open file descriptors per process. Files affected: kernel/plotting.c
- Extra code to deal with the Allocated Bandwidth Tables and detection of traffic flows. Files affected: router/queues.c
- Extra code to perform manual route changes and to plot the number of bytes dropped per router. Files affected: router/router.c and router/router.h
- Extra code to plot the sequence numbers received at the destination. Files affected: sources/sink.c

Three new C files were added to the simulator in order to implement the RBCC algorithm, the TRUMP source and the TRUMP destination: router/rbcc.c, sources/trump.c and sources/trsink.c, respectively.

The simulator was also modified to include an implementation of TCP Vegas, described later in Section 8.5. REAL 4.0 with these changes has been tested on both a SunOS 3.5 machine and a FreeBSD 2.x machine. Instructions on how to obtain the modified REAL simulator are given in Appendix G.

All modifications and extensions to REAL were managed with the RCS source code control system [Tichy 87]. The results in the following chapters were obtained with the following files:

File	Version
sources/jk_reno.c	1.4
sources/ftp_vegas.c	1.11
sources/sink.c	1.3
sources/trump.c	1.23
sources/trsink.c	1.15
router/router.c	1.19
router/rbcc.c	1.53

8.3 Implementation of TRUMP

REAL does not simulate all aspects of the network. For example, although packets have a size depending on their contents (data, acknowledgments etc.), no actual contents are exchanged between the source and the destination. Following this, it makes sense only to implement those parts of a protocol which can be simulated in REAL. The implementation of TRUMP in REAL concentrates mainly on the flow-control aspects of the protocol.

TRUMP is implemented as two source files in the simulator; `sources/trump.c` containing the code for the source, and `sources/trsink.c` containing the code for the destination. The two files contain 400 lines and 140 lines of C code[2], respectively.

[2]Comment lines are excluded, and the source has been formatted with the *indent(1)* program.

8.3.1 TRUMP Destination Code

The TRUMP destination code is relatively trivial, but is complicated by the fact that selective acknowledgments (and their size) may or may not be chosen at run-time for each scenario simulated. The TRUMP destination code has three main modes of operation: setting up a connection, acknowledging data packets, and closing the connection down. See Section B.5 for a proper state diagram of the TRUMP receiver.

A received SETUP packet indicates the start of a connection; a special acknowledgment (SACK) packet is returned in response to this. The destination responds to any SETUP packet received with a SACK packet.

If a DATA packet is received, it must be acknowledged. If selective acknowledgments are *not* being used, an acknowledgment (ACK) packet is returned to the source immediately. If selective acknowledgments are being used, a selective ACK packet is built up, and transmitted if the sequence number of the newly-arrived DATA packet cannot be acknowledged in the ACK's bitmap.

A received connection TEARDOWN packet indicates that the connection is being closed, and is acknowledged with a returned TEARDOWN packet. If there is any outstanding selective ACK packet, this is also returned to the source.

With all returned packets, the source and destination address fields are swapped, the most recent **Rate** field from the source is copied into the **Return_Rate** field. The **Bottle_Node** field is set to identify the destination. The **Desired_Rate** field is set as follows:

$$Desired\_Rate = \frac{Rate * acknowledgment\ packet\ size}{Size\ of\ data\ packets\ acknowledged} \tag{8.1}$$

where **Rate** is the most recent **Rate** value received by the destination. Finally, the **Rate** field in the outgoing packets is initialised to infinity.

8.3.2 TRUMP Source Code

The code for the TRUMP source is more complicated, as it must deal with handshaking and connection termination, rate-based flow control, packet retransmission, Rate Quench packets and handling of parameters provided by the scenario being simulated.

Handshaking and connection termination are performed by a simple state machine as shown in Figure 15 (More accurate state diagrams for the TRUMP protocol are given in Section B.5).

Initially, TRUMP performs a connection handshake by transmitting connection setup packets once a second to the destination until a special acknowledgment (SACK) packet is received. Handshaking can be disabled from the input scenario file as well. In either case, TRUMP moves to the data transmission state, transmitting data packets to the destination. When all have been acknowledged, TRUMP moves to the connection teardown state; teardown packets are transmitted once a second to the destination until a teardown packet is returned by the destination. At this point, data transmission is complete, and the TRUMP source typically stops. An input scenario parameter can request that TRUMP redo the connection after a certain amount of delay.

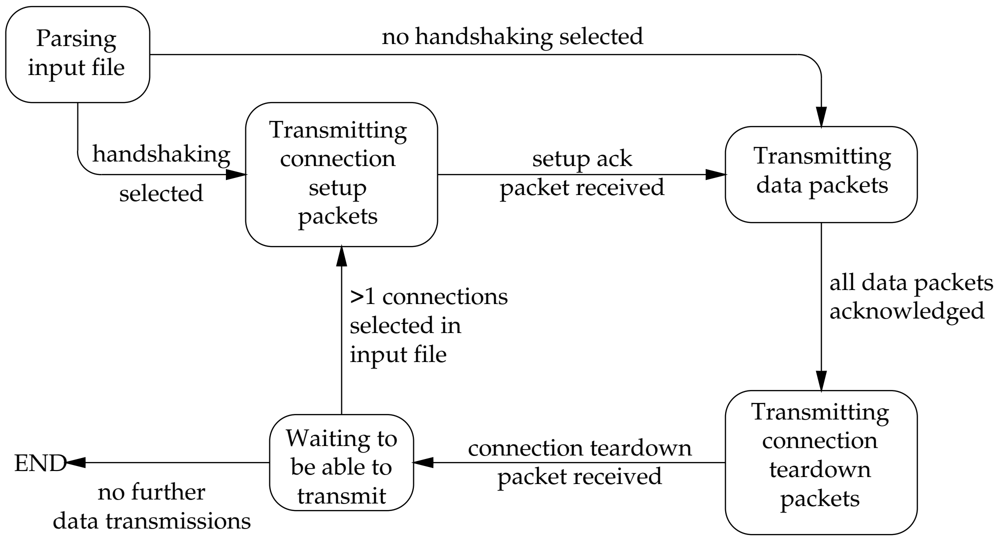

Figure 15: State Diagram for TRUMP Implementation

TRUMP's rate-based flow control is essentially controlled by the **Return_Rate** fields in packets from the destination. The `alter_rate()` function takes the **Return_Rate** fields in ACK and SACK packets and sets the transmission rate accordingly, ensuring that one round-trip's time passes between any rate change. Rate Quench packets lower the transmission rate immediately. Packets are admitted into the network evenly spaced by a delay calculated by Equation 4.1.

There are some cases when TRUMP's rate is not controlled by **Return_Rate** fields. In the connection setup and teardown states, TRUMP transmits acknowledgment-sized packets once a second, and so the transmission rate is known. When handshaking is disabled, TRUMP must begin data transmission with no appropriate rate information; a parameter from the input file sets the initial rate.

The choice of which packet to (re)transmit is performed by `get_seq_no()`. Packets are kept on several queues: the outstanding queue holds those packets which have been transmitted by have not yet been acknowledged; the lost queue holds those packets which have been transmitted and are known to be lost. The queues are ordered by time of transmission. `get_seq_no()` selects the next packet to (re)transmit as follows:

- If any packets are in the lost queue, choose the oldest one (and move it to the newest end of the outstanding queue).
- Otherwise, try to send a new packet. Choose the next unused sequence number, and place a copy of the packet on the newest end of the outstanding queue.
- If there are no new packets left to send, choose the oldest packet on the outstanding queue, and move it to the newest end of the outstanding queue.
- If there no new packets left to send, and nothing in either of the outstanding or lost queues, indicate that data transmission is complete.

Acknowledgment reception is handled by `find_outstanding_pkts()`, which removes packets from the outstanding queue, and discards them if successfully received, or moves them to the newest end of the lost queue if they were not successfully received.

In order to allow TRUMP's behaviour to be controlled from the input network scenario file, rather than via recompilation, TRUMP accepts many parameters from the input file.

Some of these parameters are expressed using the original REAL NetLanguage, and others are expressed using my *params* modification to REAL and are parsed by `parse_opts()`. The parameters accepted by TRUMP, and their influence on TRUMP, are given below.

dest: The number of the TRUMP destination node.

start_time: The time in seconds when TRUMP starts transmitting.

num_pkts: The number of packets to transmit.

peak: The desired transmission rate in bits per second. The default value of 0.0 represents ∞.

noretx: If specified, TRUMP does not retransmit lost segments. The default is to retransmit lost segments.

noshake: If specified, TRUMP does not perform connection setup handshaking, and the **initrate** parameter must specify the initial transmission rate in bits per second. The default is to perform connection setup handshaking.

restart low high: If specified, TRUMP will re-perform the connection and data transmission. TRUMP waits a random delay in the range [low:high] before restarting.

pktsize low high: If specified, packet sizes are spread evenly in the range [low:high].

8.4 Implementation of RBCC

The implementation of RBCC consists of 630 lines of C code in the file `router/rbcc.c`. This file performs the operations of a router, as well as containing the RBCC algorithm.

The file is long and complicated due to the large number of operations it must perform: standard router operations, and RBCC operations. The RBCC/router code performs:

- parsing of the input router parameters,
- identification of new traffic flows,
- flushing of idle or terminated flows,
- deciding to update the Allocated Bandwidth Table (ABT) for packets going from source to destination,
- deciding to update the ABT for packets going from destination to source,
- recalculation of the ABT,
- sending Rate Quench packets,
- queueing/dropping of packets on the output buffer,
- determination of the immediate next route for each packet,
- transmission of packets to the next router or the destination,
- and changing routes.

Router operations will not be discussed in detail, as most of the code was taken directly from the standard REAL router code in `router/router.c`. The only queueing mechanism used in the RBCC implementation is First-Come-First-Served. All of the existing REAL packet dropping mechanisms, enumerated in Section 8.1, are available.

Like TRUMP, RBCC's behaviour can be controlled from the input network scenario file via parameters, and again some parameters are expressed using the original REAL Net-Language, and others are expressed using my *params* modification to REAL and are parsed by `rbcc_parse_options()`. The parameters accepted by RBCC are given below.

decongestion_mechanism: This selects the packet dropping scheme for when a packet cannot be queued.

policy: The queueing mechanism. Only FCFS can be chosen.

norev: If set, RBCC does not use **Return** packet fields to update the ABTs. The default is to use the **Return** fields.

quench: If set, Rate Quench packets are transmitted as described below. The default is to not send Rate Quench packets.

thresh pkts scale: If set, RBCC will scale the available bandwidth by the given scale if the buffer occupancy is higher than the given number of packets. The default is to do no scaling.

route dest via node at time: If given, the route to the named destination is changed at the given time. A number of route changes can be expressed for each router. The standard REAL router code has been altered to accept this parameter as well.

RBCC checks all setup, setup ack, data and ack packets to see if a new flow is passing through the router. New flows cause the appropriate ABT to be recalculated, in `add_new_conversation()`. Packets which terminate traffic flows cause them to be deleted from the ABTs, and the affected ABTs recalculated.

As route changes may cause 'lost' flows[3] to go undetected, RBCC checks for idle flows every 5 seconds, and flushes them from the ABT with the `flush_lru_conv()` function. This is a very simplistic method to detect lost traffic flows. As each router knows the current data rate of each flow, it should be able to estimate the time when the next packet for the flow will arrive. However, rate changes at the source, packet size changes, short-term congestion, and any associated time variances will render this estimation less accurate. The implementation of RBCC, therefore, errs on the side of simplicity, and checks for idle flows every 5 seconds.

The implementation of RBCC in REAL caches the congestion fields from the last packet seen of each traffic flow. When a new packet arrives, `fields_not_cached()` determines if the cached fields match the new packet's congestion fields; if they do, the ABTs for the traffic flow do not need to be updated.

If the cached fields do not match the packet's fields, RBCC must decide if the ABTs do need to be updated. This is performed by two functions, `do_forward_update_decisions()` and `do_reverse_update_decisions()`. The decision heuristics given in Section 6.2.2 were derived from the implementation's source code. These two functions do not update the ABTs themselves; they only modify the traffic flow information in the ABTs, and indicate that an update is required.

The RBCC update algorithm, given in Section 6.2.3, is contained in the function `recalc_rate_table()`, and again the algorithm given there was derived from the implementation's source code.

[3] i.e flows no longer crossing through the router.

Packets may have their **Rate** and **Bottle_Node** fields updated, reflecting the sustainable rates set in the appropriate ABT. This is performed before the packet is dropped or queued by the `queue_or_drop()` function.

If a data packet causes a flow's rate to be lowered in a router's ABT, a Rate Quench packet is returned to the source with the **Return Rate** field set to the flow's new rate. At present, RBCC routers forward Rate Quench packets without examining their contents, although utilisation of the **Return Rate** field therein may help improve congestion control.

Section 11.3 describes how the use of Rate Quench packets affects the congestion control of the framework.

8.5 Simulation of TCP in REAL

Simulation of my proposed congestion control framework, using TRUMP and RBCC, should indicate its success or otherwise in controlling network congestion. As a reference, each network scenario simulated with TRUMP and RBCC was also simulated using both TCP Reno and TCP Vegas as the transport protocol. This should show if TRUMP and RBCC provide better congestion control than the most commonly used end-to-end window-based congestion control scheme.

As distributed, the REAL 4.0 simulator provides two versions of TCP, TCP Tahoe and TCP Reno, as described in the user's manual:

> The [two TCP] sources implement TCP flow control with the window adjustment and other modifications described by V. Jacobson, M. Karels, P. Karn and C. Partridge [Jacobson 88] [Karn & Partridge 87].
>
> **jk_tahoe:** This implements the flow control in 4.3BSD-Tahoe. The window size can increase or decrease — decreases cause the window to be shut down to 1. Duplicate acks cause retransmission.
>
> **jk_reno:** This incorporates further modifications made to TCP by V. Jacobson (as of Aug '90). When duplicate acks are seen, the window shuts down to half its previous value, not to 1.

As noted in Section 2.3.4, TCP Reno added Fast Recovery, TCP header prediction and delayed acks to TCP. The REAL version also faithfully implements the coarse timers from the 4.3BSD-Reno operating system implementation of TCP. TCP Reno was chosen over TCP Tahoe as a reference congestion control scheme for simulations, as the modification by Jacobson give Reno much better congestion control than Tahoe.

Since REAL 4.0 was made available, further enhancements to TCP have been proposed. Known as TCP Vegas, they are described in [Brakmo *et al* 94] and [Brakmo & Peterson 95], with several followup papers. The author of REAL made available an implementation of TCP Vegas for the then-unreleased REAL version 5; this implementation was written by Omar Hellal, and its use is described in [Ait-Hellal & Altman 96].

With help from Yoshifumi Nishida, Hellal's implementation was 'back-ported' to REAL 4.0. The port of Hellal's TCP Vegas code to REAL 4.0 has been verified by Hellal and by comparison with the results given in [Brakmo *et al* 94] and [Brakmo & Peterson 95].

TCP Vegas was also chosen as a reference congestion control scheme for simulations, as some of the enhancements in Vegas are claimed to improve TCP's congestion control.

8.6 Measurements Obtained by The REAL Simulator

In the next chapter, I will discuss the results obtained by simulating my proposed rate-based congestion control framework on the REAL network simulator. Firstly, I wish to outline the types of results produced by the simulator, and their relevance to the detection and control of network congestion.

The types of results produced by REAL are as follows:

- Packet sequence numbers and their transmission time at the source.
- Packet sequence numbers and their reception time at the destination.
- Packets dropped and their drop time at each router.
- Windows size for TCP sources, and the number of outstanding packets for TRUMP.
- The end-to-end delay between source and destination.
- The round-trip time between source and destination.
- The actual link utilisation for all links.
- The sustainable rate measured by RBCC as seen at the destination and at the source.
- The number of packets queued on each router's output interface.
- The RBCC sum allocated bandwidth for each router's output interface.

8.6.1 Indicators of Network Congestion

There are several indicators of network congestion:

- Packet loss as seen by a source,
- Increasing end-to-end delays and round-trip times as seen by destinations and sources respectively,
- High buffer occupancy in routers,
- Packet loss in routers, and
- A higher offered load to a link than its bandwidth capacity.

The first two indicators cannot be used by themselves as congestion indicators: packet loss is caused by both network congestion and bit errors; increasing round-trip time is caused by network congestion and lower available bandwidth to the source (competing sources). Window-based transport protocols such as TCP, however, do use these as congestion indicators. The last three indicators are reliable congestion indicators.

One of the best indicators of network congestion is a router's average queue lengths for its output interfaces. If a queue has an average length (in packets) less than 1, that interface is underloaded. If a queue has an average length greater than 1, then it is receiving more packets than it can transmit, and is overloaded. A queue length of 1 indicates an optimally loaded interface. As a queue grows in length, packets suffer delays in retransmission at the router, increasing end-to-end delays and round-trip time. All queues are finite in length, and a router must drop packets when there is no space left to queue incoming packets.

8.6.2 Other Indicators of Network Performance

Although not strictly related to network congestion, these factors show how well a transport protocol is using the resources on the network.

Packet transmission times can be used to infer instantaneous and average data rates. A source which successfully completes its transmission before an equivalent source using a different transport protocol has made better use of network resources.

End-to-end and round-trip times with high variance cause problems for protocols which use round-trip time prediction. Smooth end-to-end times are important for time-sensitive data transmission such as voice and video.

8.7 Summary

In order to measure the congestion control performance of the proposed rated-based framework, it has been implemented in the REAL network simulator. This simulator was designed to study flow and congestion control in connectionless packet-switched networks, and provides several end-to-end flow & congestion control mechanisms, as well as several router-level packet queueing and packet dropping mechanisms.

The two new functional elements of the rate-based framework, the TRUMP Transport protocol and the RBCC Sustainable Rate Measurement function, have been implemented within the REAL simulator. As well, the proposed modifications to TCP's congestion control, known as TCP Vegas, have also been implemented within the REAL simulator. The next three chapters examine the congestion control performance of the the rate-based framework, using TRUMP and RBCC, for several network scenarios. The congestion control performance of two end-to-end based systems, TCP Reno and TCP Vegas, will also be examined, and compared with the framework's performance.