Information and Media Technologies 1 (1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

Regular Paper

TCP Enhancement Using Recovery of Lost Retransmissions for NewReno TCP

Motoharu Miyake[†] and Hiroshi Inamura[†]

Conventional TCP fails to achieve optimal TCP performance since it does not handle well the loss of retransmitted segments, generated by the fast retransmit algorithm and the response of partial ACK under fast recovery. This paper introduces an algorithm to recover these lost retransmissions for NewReno TCP and details the steps to implement it. It provides careful retransmission by considering the loss of unacknowledged segments. The algorithm is followed by two options for restoring the congestion control state; both reduce the moderate transmission rate which mitigates network congestion. *ns*2 simulations show that the algorithm can overcome the loss of retransmitted segments. Moreover, it also suppresses the unnecessary throughput degradation more effectively than is possible with the recovery of lost retransmissions in Reno TCP and equals the performance offered by the SACK-based algorithm. The two options for restoring the congestion control state are also shown to offer adequate performance under retransmitted segment loss.

1. Introduction

Internet access via cellular phones using 3G high-speed mobile communications has become popular around the world. TCP streams are widely used for data communication across wired and wireless links.

In order to provide reliable transfer, one of the global telecom systems for the IMT2000 3G mobile communication standard, wideband code division multiple access (WCDMA), uses radio link control (RLC), a selective repeat and sliding window auto repeat request (ARQ) scheme. The ARQ mechanism in WCDMA supports a packet service with a negligibly small probability of undetected errors due to RLC frame retransmission [1)]. However, the delay jitter caused by error recovery can lead to unexpected increases in round trip time (RTT). Moreover, inter-system handover is likely to trigger a significant delay spike, and can result in data loss [2)].

The weaknesses of wireless links raise many issues when trying to maximize the efficiency of TCP transmission. In particular, segment loss may occur more readily in wireless links than wired links. Key goals are to strengthen the segment loss tolerance while both minimizing unnecessary throughput degradation and mitigating network congestion.

The fast retransmit algorithm is described in RFC2581 [3)]. It retransmits the first unacknowledged segment, when the sender detects segment loss upon the receipt of the third duplicate ACK. Moreover, it reduces the congestion window and the slow start threshold. The sender receives duplicate ACKs when the receiver determines that a segment has been received out-of-order as indicated by the packet sequence number. The fast retransmit algorithm uses this information to realize more effective loss recovery than is possible with the retransmission timeout technique. However, it does not work well if the congestion window is small or a large number of segments are lost in a single transmission widow, because the sender may fail to receive the third duplicate ACK.

To avoid this situation, Limited Transmit [4)] and Early Retransmit [5)] were proposed. Limited Transmit allows a new data segment to be sent in response to each of the first two duplicate ACKs. Early Retransmit changes the duplicate ACK threshold that triggers fast retransmission in special circumstances. Although the probability of segment retransmission is increased without entering costly retransmission timeout, loss of the retransmitted segment leads to retransmission timeout [6)]. The timeout is unavoidable even if the sender uses the above algorithms in Reno TCP [7)], NewReno TCP [8)], and SACK [9),10)]. Network studies show that about 5% of timeouts are caused by the loss of retransmitted segments [11)].

To strengthen TCP against segment loss, an approach to the recovery of lost retransmissions

† NTT DoCoMo, Inc.

Information and Media Technologies 1 (1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)

that uses TCP's ACK-clock was proposed[11]. We call this method the loss recovery algorithm hereafter. Duplicate acknowledgment counting (DAC) was proposed as the loss recovery algorithm of NewReno and its loss recovery probability has been analyzed[12],[13]. In the event of the recovery of lost retransmissions, the DAC has a problem in that it performs unauthorized segment transmission exceeding the TCP congestion control requirement[3]. A revised algorithm was proposed in Ref. 14), however it does not provide adequate congestion control under fast recovery.

This paper explains the steps needed to implement the loss recovery algorithm in detail. We also modified the process used in DAC[12],[13] to recover lost retransmissions to permit the fast recovery and loss recovery algorithms to cooperate. The algorithm provides careful retransmission by considering the loss of the unacknowledged segments that were retransmitted by the fast retransmit algorithm and the response to partial ACK under fast recovery. Next, this paper proposes two options for restoring the congestion control state. Both options share the goal of mitigating network congestion by reducing the moderate transmission rate.

The modified loss recovery algorithm for NewReno TCP isimplemented in the *ns*2 simulator[15], together with the options, to confirm its performance. The implementation is also useful in evaluating individual phenomenon, such as outstanding segments and congestion window behavior. Results collected from the *ns*2 simulator show that the loss recovery algorithm can overcome the loss of retransmitted segments in both fast retransmit and in response to a partial acknowledgment. Moreover, it also avoids the unnecessary throughput degradation more effectively than the loss recovery algorithm for Reno TCP and equals the performance offered by the algorithm that uses the SACK option. Our options for restoring the congestion control state also show adequate performance under retransmitted segment loss.

2. Related Works

In order to strengthen TCP against segment loss, Lin and Kung originally proposed a loss recovery algorithm that uses TCP's ACK-clock[11]. It retransmits the first unacknowledged segment raised by the fast retransmit algorithm if the number of duplicate ACKs under the fast retransmit/recovery phase reaches the number of outstanding segments plus duplicate ACK threshold. In the case of multiple segment loss, the sender is not well handled in Reno TCP, even if it uses the loss recovery algorithm. In this case, using the SACK option to offset the loss of the retransmitted segment is a useful approach[11]. If the connection between sender and receiver permits the SACK option, the right edge in the SACK block in the sender can show the correct timing clearly without resorting to any heuristics. The SACK option is standardized as RFC2018[9] and RFC3517[10], and it is implemented in most operating systems. Unfortunately, it may add a 10 to 34 byte overhead due to the use of SACK blocks which are included in ACKs (40 bytes) until an acceptable ACK, and receiver modification in some cases, such as Internet access via PCs, PDAs, and cellular phones.

Duplicate acknowledgment counting (DAC) was proposed as a loss recovery algorithm for NewReno and its loss recovery probability has been analyzed[12],[13]. In general, SACK needs one RTT (Round Trip Time) to recover all losses in a single transmission window, whereas NewReno requires L $\times$ RTT to finish, where L is the number of lost segments. NewReno TCP, however, overcomes multiple segment loss without any overhead, and it can support existing receiver side equipment without any modification. The above papers mainly focused on deriving the loss recovery probability, so provided insufficient detail, such as integration with the fast recovery and loss recovery algorithms; congestion control was not considered either. In the case of recovering lost retransmissions, the DAC has a problem in that it performs unauthorized segment transmission; that is, two segments, a new segment and an unacknowledged segment are transmitted due to the lack of cooperation between the fast recovery and loss recovery algorithms. Segment transmission using the fast recovery algorithm is allowed by duplicate ACK arrival since the receipt of duplicate ACK indicates that prior segments have already left the network. Thus, the transmission of the above segments in DAC is too aggressive to be used, and it breaks the TCP congestion control requirement[3].

A revised algorithm was proposed in 14), however it does not provide adequate congestion control under fast recovery. Whenever the algorithm detects the loss of retransmitted segments, the congestion window and slow start threshold are lowered twice, independently of the fast recovery algorithm, even if the congestion window is artificially inflated for segment transmission. This means that it restricts the use of artificial congestion window inflation and segment transmission, both of which are required by the fast recovery algorithm. If the sender faces the loss of multiple segments and retransmitted segment loss, it transmits only one segment in response to one partial ACK per RTT after recovery of the lost retransmission.

3. Loss Recovery Algorithm for NewReno TCP

In this section, we explain the loss recovery algorithm for NewReno TCP and the steps for implementing it in detail. It is followed by two options for restoring the congestion control state of the algorithm.

3.1 Recovery of Lost Retransmission Steps

NewReno TCP well handles the arrival of a partial ACK during fast recovery and allows segment retransmission under multiple segment loss[8]. The loss recovery algorithm for NewReno TCP is proof against failure to identify the loss of the retransmitted segment raised by the fast retransmit algorithm and the response to a partial acknowledgment during fast recovery. It also provides careful retransmission and avoids any increase in the overhead in the receiver.

If the number of duplicate ACKs is equal to the number of outstanding segments plus duplicate ACK threshold, the sender retransmits the first unacknowledged segment transmitted by the fast retransmit algorithm as in the loss recovery algorithm for Reno TCP. Moreover, it monitors the relationship between the number of duplicate ACKs and the number of transmitted segments during fast recovery, and retransmits the unacknowledged segment raised by a response to a partial ACK. We clearly show that the duplicate ACKs represent not only the response to the original segment, but also the response to the segment raised by the fast recovery algorithm. The algorithm provides careful retransmission by considering the loss of unacknowledged segments that were retransmitted by the fast retransmit algorithm and the response to partial ACK under fast recovery. Next, the sender waits for an acceptable ACK while sending the next new segment in response to the duplicate ACK. As a result, the sender increases the probability of segment retransmission without entering costly retransmission timeout.

```
Step (1) When congestion occurs (the third duplicate ACK is received):
    Store recovery (highest sequence number),
        $\texttt{cumulativeFS} \leftarrow \texttt{FlightSize}$,
        $\texttt{deltaFS} \leftarrow 0$,
    retransmit the first unacknowledged segment.

Step (2) Set the parameters as follows:
    $\texttt{ssthresh} \leftarrow \max(\texttt{FlightSize}/2,\ 2 \times SMSS)$
    $\texttt{cwnd} \leftarrow \texttt{ssthresh} + SMSS \times \texttt{DupThresh}$.

Step (3) Wait for the arrival of either an acceptable ACK or a duplicate ACK:
    update the variable dupacks, and
    proceed to step (4)

Step (4) If an acceptable ACK whose sequence number covers recovery arrives
        then set
            $\texttt{cwnd} \leftarrow \texttt{ssthresh}$,
        and proceed to step (DONE),
    else if partial ACK arrives, set
        $\texttt{SND.NXT} \leftarrow \texttt{SND.UNA}$,
        $\texttt{cumulativeFS} \leftarrow \texttt{deltaFS} + 1$,
        $\texttt{deltaFS} \leftarrow 0$,
        $\texttt{dupacks} \leftarrow 0$, and
       retransmit the first unacknowledged segment,
       and proceed to step (3).
    else
        proceed to step (5).

Step (5) If the sender is satisfied with the condition
        $\texttt{cumulativeFS} == \texttt{dupacks}$
      then retransmit the first unacknowledged
      segment,
        $\texttt{deltaFS} \leftarrow 0,\ \texttt{dupacks} \leftarrow 0$,
      and proceed to step (3).
      Note that this step never increment cwnd, and
      transmits only the unacknowledged segment.
    else proceed to step (6).

Step (6) For each additional duplicate ACK received,
        increment cwnd by $SMSS$,
    If allowed by the new cwnd and RWIN
        transmit a segment and increment deltaFS.
    proceed to step (3).

Step (DONE) Leave the fast recovery and loss recovery algorithms.
```

Fig. 1 The loss recovery algorithm for NewReno TCP.

Figure 1 shows the congestion control and segment retransmission steps for NewReno TCP, which are included in the fast retrans-

Information and Media Technologies 1(1): 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

mit, fast recovery, and loss recovery algorithms. In Fig. 1, `FlightSize`, `recovery`, `SND.NXT`, and `SND.UNA` are sender state variables. `FlightSize` is the number of outstanding segments in the network, `recovery` is the highest sequence number transmitted after congestion occurs, `SND.NXT` is the segment sequence number of the next segment the TCP sender will (re-)transmit, and `SND.UNA` is the sequence number of the unacknowledged segment. `DupThresh` is the duplicate acknowledgment threshold; currently specified as the fixed value of three [3]. `RWIN` and $SMSS$ are the value of the receiver's advertised window and the size of the largest segment that the sender can transmit, respectively.

The loss recovery algorithm provides careful retransmission against the response to partial acknowledgment during fast recovery, and strengthens the segment loss tolerance. It needs only small modification of the sender side TCP, so it can support existing receiver side equipment, such as PCs, PDAs, and Internet access cellular phones, without any modification. The difference from the loss recovery algorithm for Reno TCP and SACK is mainly the trigger of the unacknowledged segment transmission in steps (4) and (5). Moreover, we also modified the recovery of lost retransmissions in the DAC [12),13)] to permit the cooperation of the fast recovery and loss recovery algorithms; Step (5) in Fig. 1 does not increment `cwnd` and suppresses new segment transmission from the fast recovery algorithm.

The recovery of lost retransmissions also depends on the number of duplicate ACKs and might lose the chance to retransmit, the same as the loss recovery algorithm for Reno TCP, if a large number of duplicate ACKs are lost. In contrast, it will trigger unnecessary segment transmission of the first unacknowledged segment if a malicious receiver generates duplicate ACKs. However, a sender using the algorithm never experiences unnecessary throughput degradation caused by false congestion control. In that case, the fast recovery algorithm in conventional TCP also allows transmission of the next segment. Thus, the algorithm offers conservative behavior as does conventional TCP.

3.2 Restoring the Congestion Control State

In the previous subsection, we described a method that restores the congestion control state during the fast retransmit/recovery and loss recovery; it proceeds as follows:

(I) `ssthresh` = `cwnd_old`/2,
`cwnd` = `ssthresh`

where `cwnd_old` represents the value of `cwnd` when the third duplicate ACK was received at the sender. It keeps the congestion window size reasonably small, because segment retransmission by the fast retransmit and loss recovery algorithms are done within a single transmission window.

In order to make the process more conservative, we propose the following two options (II and III) for restoring the congestion control state:

(II) `ssthresh` = `cwnd_old`/4,
`cwnd` = `ssthresh`

(III) `ssthresh` = `cwnd_old`/2,
`cwnd` = 1

In the congestion control of TCP, the cooperation of end hosts and the response to congestion signals (i.e., dropped segments) by AIMD (Additive Increase and Multiplicative Decrease) prevents congestion collapse. Since the congestion control at the loss of retransmitted segments is not well considered, options (II) and (III) are proposed based on the regulation in RFC2581.

From the point of view of RFC2581[☆], `cwnd` and `ssthresh` are lowered twice in Option (II). When (II) is applied, the fast retransmit and loss recovery algorithms are independent, and congestion control is performed on each. According to Step (1) and (2) in Fig. 1, the fast retransmit algorithm uses the arrival of the third duplicate ACK as an indication of segment loss. It retransmits the first unacknowledged segment and sets parameters, `cwnd` and `ssthresh` using the process in option (I). In addition to the fast retransmit algorithm, the loss recovery algorithm sets `ssthresh` which is lowered twice. To avoid excessive reduction in `ssthresh`, the proposed algorithm reduces the parameter only once during loss recovery. If the acceptable ACK whose sequence number covers "`SDN.HIGH`" arrives, the sender sets `cwnd` to `ssthresh`. As per the above process, the loss

☆ RFC2581 mentions the application of congestion control under retransmitted segment loss as follows:
Loss in two successive windows of data, or the loss of a retransmission, should be taken as two indications of congestion and, therefore, cwnd (and ssthresh) MUST be lowered twice in this case.

```
Step (5) If the sender is satisfied with the condition
        cumulativeFS == dupacks
    then retransmit the first unacknowledged segment,
        deltaFS ← 0,  dupacks ← 0,
        ssthresh ← max(cwnd_old/2, 2 × SMSS)
      and proceed to step (3).
      Note that this step never increments cwnd, and
      transmits only the unacknowledged segment.
      else proceed to step (6).

      Note that ssthresh is reduced only once
      in the loss recovery algorithm.
```

Fig. 2 The loss recovery algorithm with (II): ssthresh=cwnd_old/4, cwnd=ssthresh.

```
Step (4) If an acceptable ACK whose sequence number
covers recovery arrives and segment retransmission is
experienced in Step(5)
        then set
            cwnd ← 1,
        and proceed to step (DONE),
    else if partial ACK arrives, set
        SND.NXT ← SND.UNA,
        cumulativeFS ← deltaFS + 1,
        deltaFS ← 0,
        dupacks ← 0, and
        retransmit the first unacknowledged segment
    else
        proceed to step (5).
```

Fig. 3 The loss recovery algorithm with (III): ssthresh=cwnd_old/2, cwnd=1.

recovery algorithm with option (II) can cooperate with the fast recovery algorithm. **Figure 2** shows this modification of Fig. 1 for option (II). The difference from Step (5) in Fig. 1 is only the calculation of the sender state variable, ssthresh. The comparison between cwnd_old/2 and $2 \times SMSS$ is the same as for the fast retransmit algorithm.

The loss recovery algorithm avoids the retransmission timeout caused by loss of retransmitted segments in NewReno TCP. Upon timeout, cwnd is set to 1 full-sized segment, while ssthresh is set to one half of the previous cwnd. After that, the sender uses the slow start algorithm, which controls the number of outstanding segments being injected into the networks. The increase in slow start continues until ssthresh is reached, at which point it is replaced by the congestion avoidance algorithm. By introducing this conventional congestion control, i.e., the retransmission timeout, to option (III), congestion control state restoration is made very conservative. **Figure 3** shows the modification of Fig. 1 for option (III). The difference from Step (4) in Fig. 1 is the change in sender state variable cwnd. The load on the networks under a retransmission timeout is actually increased due to the large number of unnecessary retransmissions [16]. It is expected that option (III) will mitigate network congestion by reducing the overall transmission rate. Accordingly, option (III) is better for handling severe congestion states.

4. Performance Evaluation

In the case of duplicate ACK arrival following fast retransmit and retransmitted segment loss, the sender using the loss recovery algorithm transmits the next new segments as well as retransmitting the unacknowledged segment. However, the algorithm shows improvement only for corner cases. That is, if the fast retransmit does not occur or duplicate ACKs do not arrive, the algorithm is not triggered and so provides no improvement in the connection's throughput. It is difficult to define a realistic wireless communication model that includes fast retransmit and retransmitted segment loss for throughput evaluations. Accordingly, this paper generated several scenarios and examined the time-sequence, congestion window size, and its throughput in the face of duplicate ACKs arriving after fast retransmit and the case of retransmitted segment loss.

4.1 Simulation Model

In this subsection, we implement the modified loss recovery algorithms for NewReno TCP as described in **Table 1** using the *ns*2 simulator (ns-2.26) [15]. It was originally developed in the VINT project, and aimed to build a network simulator that would allow the study of scale and protocol interaction in the context of current and future network protocols. We evaluate the algorithms in Table 1 using a communication model based on **Fig. 4**. The TCP sender in the wired network communicates with the receiver in the wireless network. Retransmitted segments may be lost more often than expected from just network congestion given the unstable nature of the wireless link.

Figure 5 shows the topology used in our experiments. The sender is connected to the BS via a 10 Mbps wired link with 20 ms delay, and the receiver is connected to the BS via a 384 kbps wireless link with 500 ms delay. The TCP segments of $SMSS = 1{,}460$ bytes are transmitted from the sender to the receiver. The BS has enough queue depth and does not

Information and Media Technologies 1(1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

Table 1 List of algorithms evaluated.

	Base-protocol	Fast retransmit /recovery	Loss recovery	Congestion control after retransmitted segment loss

Loss recovery for Reno	Reno	○	○	`ssthresh` = `cwnd_old/2`
				`cwnd` = `ssthresh`

Loss recovery for SACK	SACK	○	○	`ssthresh` = `cwnd_old/2`
				`cwnd` = `ssthresh`

Loss recovery for NewReno (I)	NewReno	○	○	`ssthresh` = `cwnd_old/2`
				`cwnd` = `ssthresh`

Loss recovery for NewReno (II)	NewReno	○	○	`ssthresh` = `cwnd_old/4`
				`cwnd` = `ssthresh`

Loss recovery for NewReno (III)	NewReno	○	○	`ssthresh` = `cwnd_old/2` `cwnd` = 1
NewReno	NewReno	○	×	`ssthresh` = 2☆ `cwnd` = 1

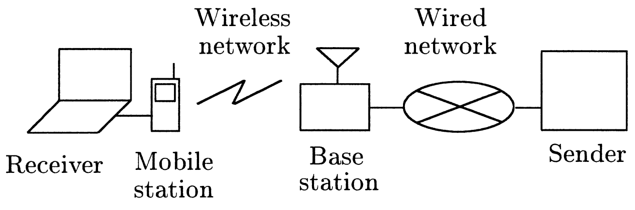


Fig. 4 System model.

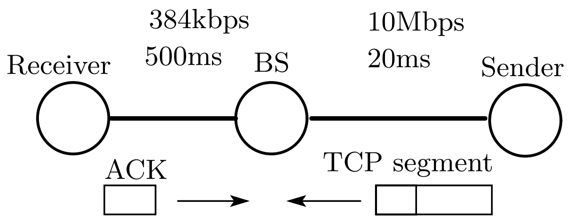


Fig. 5 Simulation model.

drop any original outstanding segments. In the following simulations, the wireless link drops three original segments (38, 42, and 46) and one of the retransmitted segments in the fast retransmit or in the response to a partial acknowledgment.

4.2 Loss Retransmission Evaluation

Figures 6, **7**, and **8** show the segment number versus time (simply called time-sequence graph) with three original segments and the first retransmitted segment loss. These graphs plot data collected from the *ns*2 simulator on the sender side. The graph convention used in the time-sequence graph is similar to that introduced in Ref. 17). The thick and thin (upper and lower) solid lines plot the segments, advertised window size, and acknowledgments, respectively. Symbol "R" shows the segment retransmitted by the fast retransmit algorithm, loss recovery algorithm, or retransmission timeout. A small tick on the bottom Acknowledgment line indicates that a duplicate ACK has been received. The slow start algorithm allows

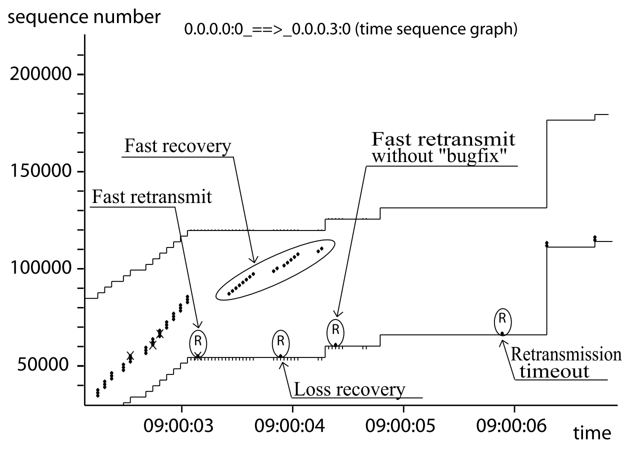


Fig. 6 Loss recovery for Reno TCP with the loss of three original and first retransmitted segments.

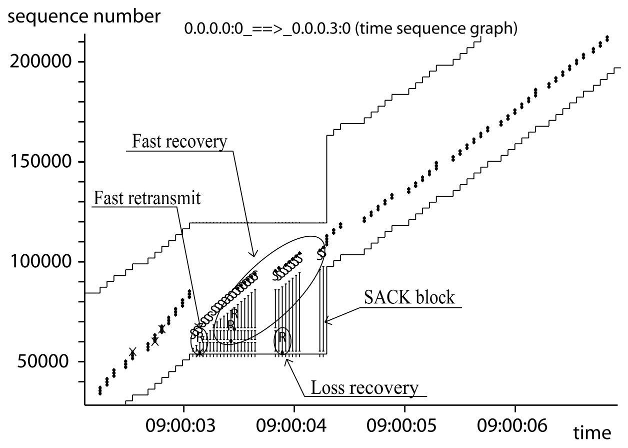


Fig. 7 Loss recovery for SACK with the loss of three original and first retransmitted segments.

two or three segments to be injected in response to each of acceptable ACKs into the network until 9:00:03.1, and the number of outstanding

☆ According to the congestion control for retransmission timeout (caused by retransmitted segment loss), `ssthresh` must be set to no more than the value given the following equation [3]:
$\mathtt{ssthresh} = \max(\mathtt{FlightSize}/2, 2 \times SMSS)$.

Information and Media Technologies 1 (1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

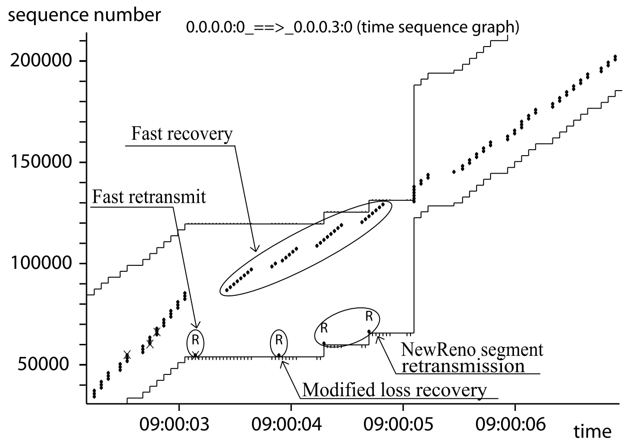


Fig. 8 Loss recovery for NewReno (I) with the loss of three original and first retransmitted segments.

segments increases exponentially. In contrast, a new segment is transmitted in response to each of the duplicate ACKs towards the advertising window using the fast recovery algorithm.

The loss of three original segments at 9:00:02.5 triggers the fast retransmit/recovery algorithms. The loss recovery algorithms for Reno TCP, SACK, NewReno (I) identify the loss of the first unacknowledged segment and retransmit the segment using the loss recovery algorithm at 9:00:3.9. New segments are continuously transmitted towards the advertising window using the fast recovery algorithm. In this case of multiple segment loss, the sender using the loss recovery algorithm for Reno TCP fails to handle the duplicate ACKs well, because Reno TCP uses "bugfix" ☆ 8) to restrict multiple fast retransmit. Even if it allows multiple fast retransmit, the sender still faces a shortage of duplicate ACKs. Thus, the sender enters a costly retransmission timeout; the congestion window size reduction leads to an unnecessary degradation in throughput.

In contrast, the loss recovery algorithm for SACK and NewReno (I) monitor the SACK block and the number of the duplicate ACKs, respectively, to detect the retransmitted segment loss. **Figure 9** shows the step for the loss recovery algorithm with SACK. The difference from the loss recovery algorithm for Reno TCP is not the count of duplicate ACKs but the evaluation in Fig. 9 performed as Step (5) in Fig. 1. If the right edge in a SACK block advances the

☆ The current Reno TCP uses "recovery" variable that records the highest sequence number for lost segment recovery, and avoids the performance problems caused by multiple fast retransmits. So, this modification is originally called "bugfix" in 18).

```
Step (5) If the SACK block reports as follows:
    Right edge in SACK block > recovery
  then retransmit the first unacknowledged segment,
  store recovery again, and proceed to step (3)
else proceed to step (6).
```

Fig. 9 The loss recovery algorithm for SACK.

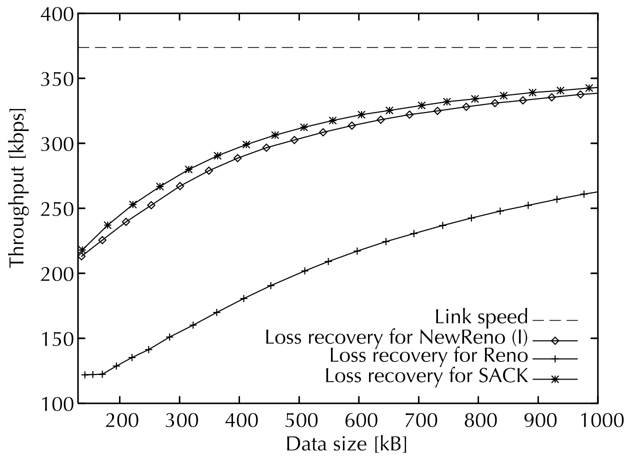
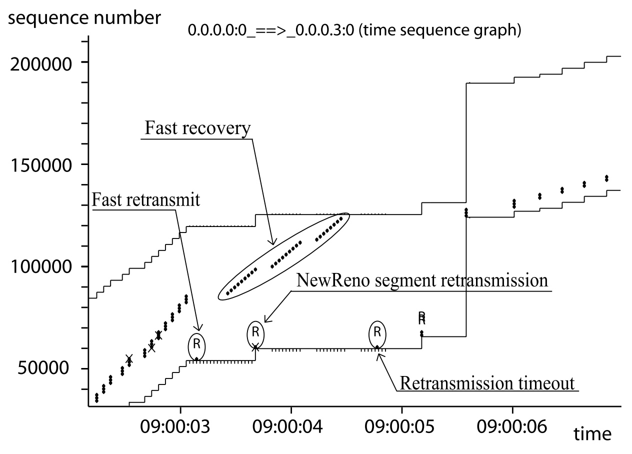


Fig. 10 Throughput comparison: Reno, SACK, and NewReno.

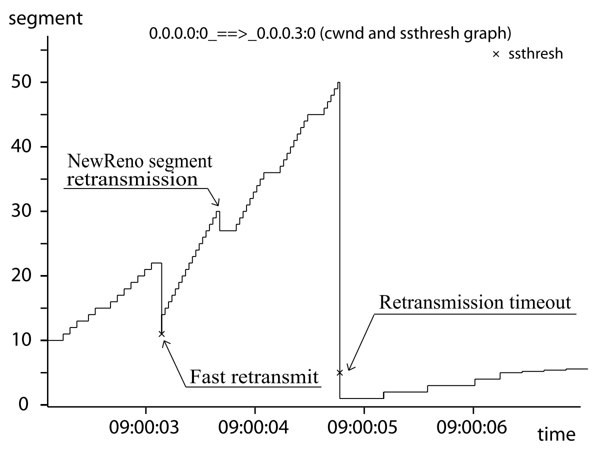
value `recovery`, the sender retransmits the first unacknowledged segment immediately. Next, the sender waits for an acceptable ACK while sending the next new segment in response to a duplicate ACK. In Figs. 7 and 8, the sender then retransmits the unacknowledged segment at 9:00:03.8, and avoids the costly retransmission timeout and subsequent congestion window size reduction. The loss recovery algorithm for SACK shows effective segment loss recovery at the cost of increasing the receiver overhead due to the use of the SACK algorithm. On the other hand, the loss recovery algorithm for NewReno (I) is based on the number of duplicate ACKs; that is, the sender waits for 0.8 seconds until the acceptable ACK arrives. However, it transmits the new segments in response to the duplicate ACK during fast recovery. This ensures that it keeps one half of the congestion window and offers the same throughput as the algorithm using SACK option without increasing receiver overhead.

Figure 10 shows the relationship between data size and throughput among the loss recovery algorithms for Reno TCP, SACK, and NewReno TCP. It also shows the scenario in Figs. 6, 7, and 8. The wireless link drops three original segments and one of the retransmitted segments in the fast retransmit or in the response to a partial acknowledgment, and

Information and Media Technologies 1 (1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

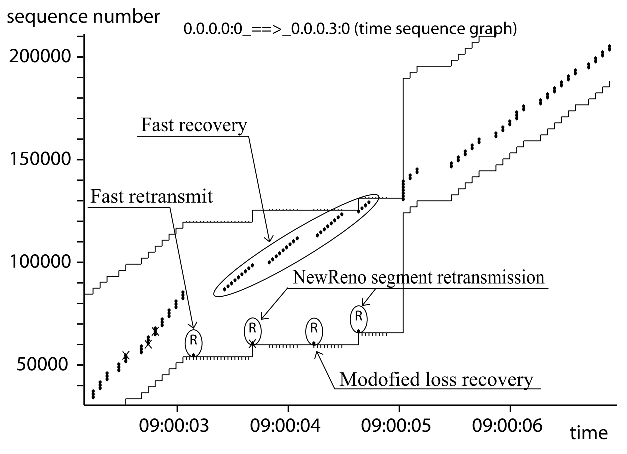


(a) Time-sequence graph

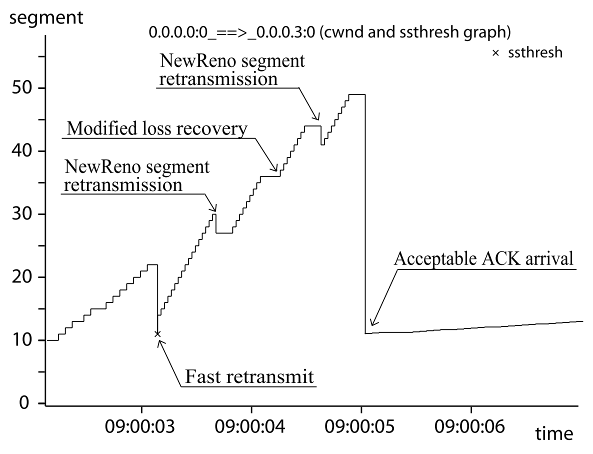


(b) Sender state variables, `cwnd` and `ssthresh`

Fig. 11 Loss retransmission for the original NewReno TCP with the loss of three original and second retransmitted segments.



(a) Time-sequence graph



(b) Sender state variables, `cwnd` and `ssthresh`

Fig. 12 Loss recovery for NewReno (I) with the loss of three original and second retransmitted segments.

never generates segments loss after that. The throughput is obtained by dividing the amount of data shown in the x-axis by the transmission time including the slow start and the loss recovery periods. The wireless link speed includes the overheads of TCP/IP header information as given by the following equation:

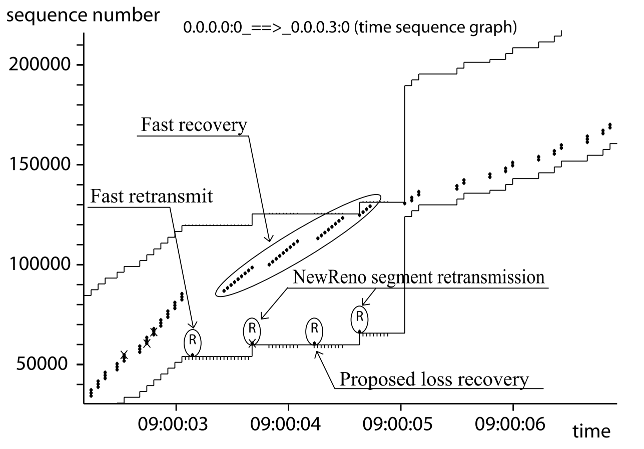
$$\text{Link speed} = 384 \text{ [kbps]} \times \frac{1{,}460 \text{ [bytes]}}{1{,}500 \text{ [bytes]}} = 373.8 \text{ [kbps]}.$$

In Fig. 10, Reno TCP needs excessive data size to recover the throughput. If the sender uses the Keep-Alive extension to HTTP [19)] for long-lived HTTP sessions, it takes much more time for sending than the others after loss recovery. In contrast, NewReno (I) and SACK offer superior performance to Reno TCP, and they effectively approach the wireless link speed by increasing the amount of data. NewReno (I) offers performance comparable to that of SACK, even though it takes additional time to recover from the second and third segment loss and waits for the arrival of duplicate ACKs (beyond the number specified by the duplicate ACK threshold) to detect retransmitted segment loss. Moreover, SACK and NewReno (I) utilize the wireless link speed efficiently even though the amount of data transferred is relatively small.

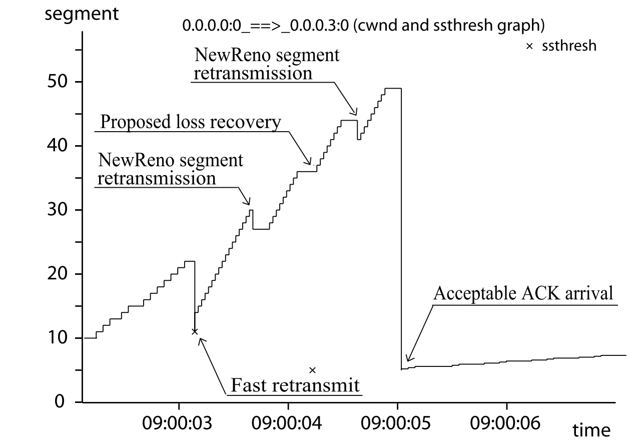
4.3 Congestion Control State Evaluation

Figures 11 and **12** plot the time-sequence graph and sender state variable, `cwnd` and `ssthresh` with the loss of three original segments and the second retransmitted segment. In Fig. 11, the sender using the original NewReno TCP [8)], i.e., without the loss recovery algorithm, enters a costly retransmission timeout at 9:00:04.8, and the sender state variables are reduced, `cwnd` = 1 and `ssthresh` = 5. In Fig. 12, the sender using the loss recovery algorithm for NewReno (I) monitors the relationship between the number of duplicate ACKs and transmitted segments during fast recovery, so it retransmits the unacknowledged segment

Information and Media Technologies 1 (1) : 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)
© Information Processing Society of Japan

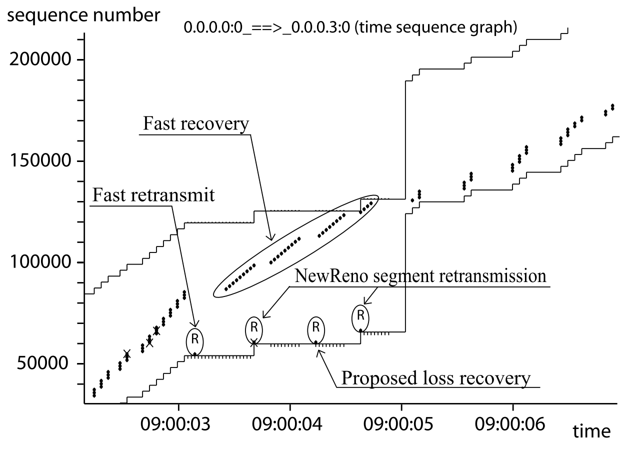


(a) Time-sequence graph

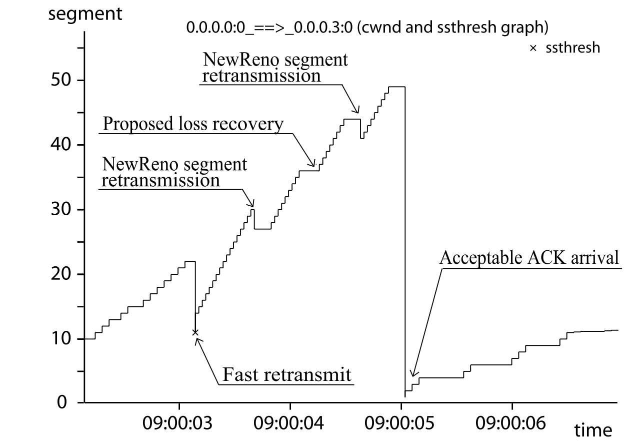


(b) Sender state variables, `cwnd` and `ssthresh`

Fig. 13 Loss recovery for NewReno (II): `ssthresh=cwnd_old/4, cwnd=ssthresh`.



(a) Time-sequence graph



(b) Sender state variables, `cwnd` and `ssthresh`

Fig. 14 Loss recovery for NewReno (III): `ssthresh=cwnd_old/2, cwnd=1`.

at 9:00:04.2. That is, the loss recovery algorithm for NewReno (I) avoids the costly retransmission timeout even if it loses the second retransmitted segment that was in response to the partial ACK. Moreover, it keeps one half of `cwnd` and `ssthresh` during fast recovery and loss recovery, and avoids the unnecessary throughput degradation.

Figures 13 and **14** plot the time-sequence graph and sender state variable of the loss recovery algorithm for NewReno TCP, which uses conservative restoration of the congestion control states, (II) and (III). In Fig. 13, the sender using the loss recovery algorithm for NewReno (II) retransmits the unacknowledged segment at 9:00:04.2, and it sets `ssthresh` $= 5$ by Step (5) in Fig. 2, so the congestion window is still increased by $SMSS$. That is, the new segment can be transmitted in response to each duplicate ACK using the fast recovery algorithm. When the sender receives an acceptable ACK which covers more than “recovery” at 09:00:05.0, `cwnd` is set to `ssthresh` $= 5$. After that, the fast recovery and loss recovery algorithms are replaced by the congestion avoidance algorithm. In Fig. 14, the sender using the loss recovery algorithm for NewReno (III) retransmits the unacknowledged segment at 9:00:04.2 and sets parameters, `cwnd` and `ssthresh`, the same as with option (I). When the sender receives an acceptable ACK that covers more than “recovery” at 09:00:05.0, `cwnd` is set to 1 by Step (4) in Fig. 3. After that, the sender transmits new segments using the slow start algorithm while `cwnd` is less equal than `ssthresh` ($= 11$) and then is replaced by the congestion avoidance algorithm. By using the slow start algorithm, option (III) reduces the transmission rate more than option (II) and so better mitigates network congestion.

Figure 15 shows the relationship between data size and throughput for the loss recovery algorithms for NewReno (I), (II), and (III) and the original NewReno TCP. It also shows the scenarios in Figs. 11, 12, 13, and 14. The original NewReno TCP needs excessive time to

Information and Media Technologies 1(1): 615–625 (2006)
reprinted from: IPSJ Digital Courier 1: 370–380 (2005)

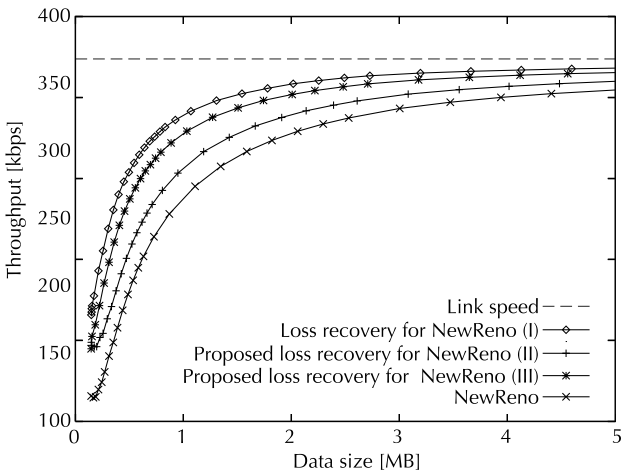


Fig. 15 Throughput comparison between the proposed and original NewReno.

recover the throughput because of the reduction in the sender state variables due to the retransmission timeout. In contrast, the loss recovery algorithms for NewReno (II) and (III) offer better throughput than the original, and more effectively utilize the wireless link speed even if the sender transmits a small amount of data. In Fig. 15, option (III) for the loss recovery algorithm realizes throughput nearer to those of option (I) than option (II). Although the sender set `cwnd` = 1 to avoid network congestion states, one half of `ssthresh` was kept during first recovery and loss recovery and realizes throughput comparable to that of option (I). Therefore, applying option (III) seems to be the most attractive choice in this simulation model.

5. Conclusions

In this paper, we introduced a loss recovery algorithm for NewReno TCP and detailed the steps needed for its implementation. The loss recovery algorithm allows careful retransmission by considering segment loss without incurring the overhead of the SACK algorithm; receiver side modification is not needed. Simulation results allow us to draw the following conclusions:

(1) A sender using the loss recovery algorithm can overcome the loss of retransmitted segments generated by either fast retransmit or in response to a partial acknowledgment.

(2) A sender using the loss recovery algorithm for NewReno avoids unnecessary throughput degradation, and its performance is comparable to that of the SACK-based algorithm.

(3) The options proposed here for restoration of the congestion control state in the loss recovery algorithms offer better throughput than the original NewReno TCP.

(4) Option (III) for the loss recovery algorithm realizes throughput comparable to that of option (I) while mitigating network congestion by reducing the transmission rate.

Acknowledgments The authors would like to thank Yukio Atsumi and Daikichi Osuga for several useful discussions and comments.

References

1) 3GPP: 3G TS 25.322 v.3.5.0, RLC Protocol Specification (2000).
2) Inamura, H., Montenegro, G., Ludwig, R., Gurtov, A. and Khafizov, F.: TCP over Second (2.5G) and Third (3G) Generation Wireless Networks, RFC3481 (2003).
3) Allman, M., Paxson, V. and Stevens, W.: TCP Congestion Control, RFC2581 (1999).
4) Allman, M., Balakrishnan, H. and Floyd, S.: Enhancing TCP's Loss Recovery Using Limited Transmit, RFC3042 (2001).
5) Allman, M., Avrachenkov, K., Ayesta, U. and Blanton, J.: Early Retransmit for TCP and SCTP, draft-allman-tcp-early-rexmt-03.txt (expired) (2003).
6) Fall, K. and Floyd, S.: Simulation-based comparisons of Tahoe, Reno and SACK TCP, *Computer Communication Review*, Vol.26, No.3 (1996).
7) Stevens, W.R.: *TCP/IP Illustrated, Volume 1 The Protocols*, ADDISON-WESLEY (2001).
8) Floyd, S., Henderson, T. and Gurtov, A.: The NewReno Modification to TCP's Fast Recovery Algorithm, RFC3782 (2004).
9) Mathis, M., Mahdavi, J., Floyd, S. and Romanow, A.: TCP Selective Acknowledgment Options, RFC2018 (1996).
10) Blanton, E., Allman, M., Fall, K. and Wang, L.: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP, RFC3517 (2003).
11) Lin, D. and Kung, H.T.: TCP Fast Recovery Strategies: Analysis and Improvements, *Proc. IEEE INFOCOM 98* (1998).
12) Kim, B. and Lee, J.: Retransmission Loss Recovery by Duplicate Acknowledgement Counting, *IEEE Communications Letters*, Vol.8, No.1 (2004).
13) Kim, B.: *Performance Analysis of TCP Loss Recovery*, The Graduate School Yonsei University (2003).
14) Kim, D., Kim, B., Han, J. and Lee, J.: En-

hancements to the Fast Recovery Algorithm of TCP NewReno, *Lecture Notes in Computer Science* (*LNCS*), No.3090 (2004).

15) Fall, K. and Varadhan, K.: ns Note and documentation, *The VINT Project* (*UC Berkeley, LBL, USC/ISI, and Xerox PARC*) (2003).

16) Gurtov, A. and Ludwig, R.: Responding to spurious timeouts in TCP, *Proc. IEEE INFOCOM'03*, Vol.3 (2003).

17) Hoe, J.: Improving the Start-up Behavior of a Congestion Control Scheme for TCP, *ACM SIGCOMM* (1996).

18) Floyd, S.: Presentation to the TCPIMPL Working Group, Revisions to RFC 2001 (1998).
ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.ps
ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.pdf

19) Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T.: Hypertext Transfer Protocol—HTTP/1.1, RFC2616 (1999).

(Received February 3, 2005)
(Accepted July 4, 2005)
(Released September 7, 2005)

Motoharu Miyake was born in 1973. He received the B.S. and M.S. degrees in Electronic Engineering from Tokyo University of Engineering, Tokyo, Japan in 1995 and 1997, respectively. He received the Ph.D degree in Electrical and Electronic Engineering from Tokyo Institute of Technology, Tokyo, Japan in 2000. Since 2000, he has been working in the Multimedia Laboratories at NTT DoCoMo, Inc., Yokosuka, Japan. His current research interest is a transport protocol for wireless communication. He is a member of IEICE.

Hiroshi Inamura was born in 1965. He received B.S. and M.S. degrees from Keio University, Japan, in 1998 and 1990, respectively. He joined NTT in 1990 and is working in the area of distributed systems, especially distributed file systems. From 1994 to 1995, he was a visiting researcher at the Department of Computer Science, Carnegie Mellon University. From 1999 he worked for Multimedia Laboratories at NTT DoCoMo, Inc. His research interests include transport protocol issues and solutions for wireless systems. He is a member of IPSJ, IEICE, and ACM.