TICP: TCP-friendly Information Collecting Protocol

Chadi Barakat, Naomichi Nonaka

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

TICP: TCP-friendly Information Collecting Protocol

Chadi Barakat — Naomichi Nonaka

N° 4807

April 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4807--FR+ENG

TICP: TCP-friendly Information Collecting Protocol

Chadi Barakat* , Naomichi Nonaka†

Thème 1 — Réseaux et systèmes
Projet PLANETE

Rapport de recherche n° 4807 — April 2003 — 29 pages

Abstract: We present and validate TICP, a TCP-friendly reliable transport protocol to collect information from a large number of receivers spread over the Internet. TICP is a stand-alone protocol that can be used by any application requiring the collection of information: quality of reception in a multicast session, numbering of population, weather monitoring, etc. The protocol does not impose any constraint on the nature of the collected information. It ensures two main things: (i) the information to collect arrives entirely and correctly at the source where it is stored to be treated later, and (ii) the implosion at the source and the congestion of the network are avoided by controlling the rate at which receivers send their information. The congestion control part of TICP is designed with the main objective to be friendly with applications using TCP. We implement TICP in ns-2, and we validate that it allows to quickly and reliably collect information from receivers, while avoiding network congestion and being fair with competing traffic.

Key-words: TICP, collecting information, TCP-friendliness, congestion and error control, multicast.

* Chadi Barakat is with INRIA - Sophia Antipolis, Planète research group, email: Chadi.Barakat@sophia.inria.fr

† Naomichi Nonaka is with Hitachi, Ltd., Systems Development Laboratory, email: nonaka@sdl.hitachi.co.jp

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

TICP: Un protocole TCP-friendly pour la collecte d'informations

Résumé : Nous présentons et validons TICP, un protocole transport TCP-friendly fiable pour collecter des informations d'un grand nombre de récepteurs distribués à travers l'Internet. TICP est un protocole indépendant qui peut être utilisé par toute application demandant la collecte d'informations, par exemple, collecter la qualité de la réception dans une session multimédia multipoint, recenser une population, contrôler le temps, etc. Notre protocole TICP n'impose aucune contrainte sur la nature de l'information à collecter. Parmi d'autres, le protocole assure que les deux objectifs suivants soient réalisés: (i) l'information à collecter arrive entièrement à la source, et (ii) l'implosion du réseau est évitée, ceci se fait en contrôlant le débit auquel la source envoie ses demandes et les récepteurs envoient leurs informations. La partie contrôle de congestion du protocole est développée de façon à être équitable avec les applications utilisant TCP. Nous implémentons TICP dans le simulateur réseau ns-2, et nous montrons par simulations qu'il permet de collecter correctement et rapidement l'information désirée, tout en évitant la congestion du réseau, et tout en restant équitable avec le trafic concurrent (TICP ou TCP).

Mots-clés : TICP, collecte d'informations, équité avec TCP, contrôle de congestion et d'erreurs, multipoint.

1 Introduction

This paper describes TICP, a reliable transport protocol to be used for collecting information from a large number of receivers spread over the Internet. TICP stands for TCP-friendly Information Collecting Protocol. The information is collected by a single source, where it is stored and treated. The source reaches the receivers by multicast (point-to-multipoint). For example, one can consider the scenario where messages from the source are broadcasted to receivers through a satellite link. Receivers send their information to the source through the wired Internet using unicast (point-to-point). The protocol can be used to collect any kind of information from receivers. One example could be the case of a source that wants to know if (and which) receivers have well received a certain document, that the source has transmitted to the receivers before the collect-information session. Other examples include the quality of TV or video reception, the weather, the electronic vote, the numbering of a population, etc. The protocol can be occasionally used as when the source collects information at the end of a working day. It can be frequently used as what happens when the source desires to know the quality of reception during a multimedia broadcast (loss rate, average delay).

Our protocol is based on the following requirements. Some of these requirements are not really necessary and can be easily removed or alleviated.

- The source has a list of all receivers. This list can be the list of IP addresses used by receivers, the list of IDs identifying the receivers at the session level, the names of receiving machines, etc.
- Multicast is used to send messages to receivers. Multicast issues are not addressed in this paper. Only transport issues are addressed.
- Each receiver sends its information in unicast to the source when it receives a message from the source inviting it to send its information. We consider in this paper the case where the information of the receiver can be included in one packet, and we call that packet *the report of the receiver*. We leave the case of large reports for future research. The message of the source sent to a receiver is called *request message*. A *request packet* is a packet sent by the source that carries multiple request messages to different receivers. The source can put in one packet more than one request message. A receiver sends a report to the source if it receives a packet including a request message addressed to it.
- The source aims at receiving the entire information detained by receivers. The information of a receiver has to be received at least once by the source. The

source does not ask a receiver that has sent its information to send it a second time. The source asks however a receiver whose report was lost in the network to send it back.

- A receiver answers quickly the request messages of the source by sending its report. The same report can be sent multiple times if multiple request messages are received by one receiver.

The main challenge with such a transport protocol is that the available bandwidth in the network is limited and the number of receivers can be very large (thousands or more). The limitation of bandwidth is more pronounced on the reverse path, where the volume of reports is likely larger than the volume of request messages. First, we cannot ask many receivers to send their reports at the same time, otherwise the network will be congested. Request messages sent by the source and reports sent by receivers may also result in an aggressive traffic that is harmful to other applications. TICP has to implement some kind of congestion control so that the resulting traffic (in both directions) does not congest the network and does not harm the other applications. Second, TICP has to implement some kind of error control. Reports sent by receivers and lost in the network have to be resent in a way to minimize the session duration.

We explain in this paper the different functions of our protocol. We implement the protocol in ns-2, the network simulator [12], and we validate its performance. The congestion control part of TICP is inspired from that of TCP [1, 9], and this is for the purpose to make TICP friendly with the TCP protocol. Applications using TCP generate the majority of Internet traffic [14]. We also want multiple sources using TICP to share fairly the network resources and to be friendly with each other. The error control part of TICP is based on retransmissions and is developed with the main objective to minimize the session duration.

In the next section, we outline the related literature and we illustrate the originality of our protocol. Section 3 describes the protocol. Each subsection in Section 3 describes one function of the protocol, and explains how this function is implemented. In Section 4, we put all the functions together and we provide the main algorithm of our protocol. Section 5 discusses the fairness of our protocol with TCP. In Section 6, we present simulation results that validate the effectiveness of TICP in controlling the congestion of the network and in enforcing fairness. We end the paper with conclusions and perspectives on our future research on TICP.

2 Related work

Different works have addressed the problem of collecting information from receivers in a multicast session [2, 3, 7, 8, 11]. Usually, the collected information is identical for all receivers (e.g. negative ACK corresponding to one packet in a reliable multicast session, a reply message sent in response to a source "hello" message), so it can be filtered at the receiver side to avoid the problem of feedback implosion. By filtering we mean that the information sent by a receiver can be substituted by that sent by another receiver without altering the source behavior. Our work is different from previous works by the fact that receivers do not send the same information, and that the source requires the receipt of the entire information sent by all receivers. In our context, the information sent by receivers cannot be filtered at the receiver side.

Another difference between our work and the previous works is that our protocol is a stand-alone protocol that can be used in different scenarios. Our protocol does not impose any constraint on the nature of the collected information. The congestion control part of TICP does not require that the collected information be filterable, as it is the case with the previous works. The collected information can be general, for example the IDs of receivers that did not correctly receive a certain object. We can imagine many other information that can be collected from receivers, as the quality of TV reception, the weather, the electronic vote, the numbering of a population, etc.

The collection of information has been studied in the literature in two contexts: reliable multicast and counting the number of receivers. Both applications are somehow linked together, since some reliable multicast schemes require the knowledge of the number of receivers [7].

In reliable multicast, receivers that did not receive a packet send a NACK asking the source for a retransmission of the packet. Many NACKs may cause a congestion in the network or at the source. The problem is called "NACK implosion". But, the NACK information can be safely filtered; there is no need that a host sends a NACK if its neighbor has already sent a NACK for the same packet, since the source will retransmit the lost packet to all members of the multicast session. The aggregation of NACKs can be done either in the network by routers, or by receivers. In [7], a receiver waits for a random time before sending a NACK, and listens at the same time if another receiver has sent a NACK for the same packet. If so, the former receiver cancels its request, otherwise it sends it when the timer expires.

Counting the number of receivers in a multicast session requires that each receiver sends a "I am here" message to the source. Sending all these messages is not feasible when the number of receivers is large. However, given that the messages are identical,

the source can only ask a subset of receivers (say 10%) to send their messages, and try to infer the total number of receivers from the number of messages received. Different works have studied such counting scheme, and the selection of the subset of receivers is usually done with a *message transmission probability* detained by each receiver. Periodically, a receiver transmits the "I am here" message with this probability. Some works have considered the case of a fixed population of receivers [8, 11], and others have considered the case of a variable population [2, 3]. Clearly, this filtering of messages at the receivers is only possible since messages are identical. The problem will be much more complex if the source decides to know, in addition to the number of receivers, some additional information that changes between receivers, as the name or the preferences. Filtering the information is not possible in this latter case. A protocol as the one we are proposing in this paper is then absolutely needed.

3 Protocol description

We shall describe in this section the main functional blocks of our protocol. We also define the different variables and methods required to implement each block. In Section 4, the different blocks are grouped together in one algorithm. Note that the main purpose of our protocol is (i) to control the congestion that may be caused by requests of the source and reports of the receivers, (ii) to enforce fairness, and (iii) to minimize the time necessary to get reports from all receivers.

For the congestion control part, we choose to do it in a TCP-friendly way. Most of Internet traffic is carried by TCP [14], and being friendly with applications using TCP is a requirement for a stable and fair Internet [6]. Our main concern is that our protocol has to react to congestion in the network in the same way TCP does (by dividing its congestion window by two), and that it has to increase the volume of requests and reports in the network in a conservative way, so that competing TCP traffic does not suffer. TCP is known to implement an additive-increase multiplicative-decrease algorithm for adapting its congestion window as a function of network conditions [9]. The transitory phase of a TCP connection, called Slow Start, implements an exponential window increase algorithm. Our protocol implements similar algorithms to TCP. In Section 5, we discuss the issue of friendliness of our protocol with applications using TCP.

3.1 Filtering receivers

The source of the collect-information session sends request messages asking a certain number of receivers to send their reports. A request message is received by all receivers (at the IP level), but only those receivers to which the request is addressed (at the session level) send their reports. Filtering of requests is supposed to be done at the session level. This can be realized by sending the session IDs of receivers of interest in request packets. When receiving a request packet, the receiver looks at the list of IDs carried by the packet, and sends its report if its ID exists in this list. If its ID is not found, the receiver discards the request packet and does not send its report.

The IDs of receivers can be sent separately. They can also be aggregated by the source so as to reduce the size of request packets. The aggregation of IDs can be done in different ways. For example, instead of sending a set of contiguous IDs, the source can send the first and the last ID in the set (in the same way SACK does [10]). All receivers with the ID in the set respond by sending their reports. Aggregation can also be done by using masks, as IP addresses are aggregated in CIDR (Classless Inter-Domain Routing). Suppose that IDs are of 32 bits. The source can ask all receivers with IDs in the set X.Y.Z/24 to send their reports. This set covers all IDs with the highest-order three bytes (or 24 bits) equal to X.Y.Z. Bloom filters [4] based on hash functions can also be used to aggregate IDs in some bit mask that can be transmitted in request packets.

We suppose that such a mechanism for aggregating IDs exists. It is up to the source to choose the best aggregation method. If there is enough bandwidth on the forward path (e.g. a high speed dedicated satellite channel), the source can avoid aggregation by sending separately the IDs of the receivers of interest. Sending separately the IDs reduces the processing time at the source. However, it consumes more bandwidth and requires more CPU power at the receivers. When the list of IDs is long, a receiver needs more processing time to search the list and to decide whether to send its report or not.

3.2 Error recovery

The TICP source is interested in minimizing the time necessary to get reports from all receivers. Clearly, this objective can be only achieved if the source operates in the following way:

- In a first round, the source sends requests to all receivers (at a rate determined by the congestion control mechanism to be described later). It does not re-

transmit request messages to receivers whose reports are (supposed to be) lost. Note that the absence of a report from a receiver can be the result of the loss of the request itself rather than the loss of the report. Still in this case, the source does not retransmit its request.

- In a second round, the source sends requests to receivers whose reports were not received in the first round.
- In a third round, the source sends requests to receivers whose reports were not received in the first two rounds.
- The source continues sending requests in rounds until all reports are received (or the session is stopped by the source since its duration exceeds some allowed time).

The explanation for this behavior in rounds is simple. It is better to try new receivers rather than wasting time sending multiple requests to a receiver that is located behind a congested link. Multiple requests to the same receiver result at maximum in one report, however sending the same number of requests to different receivers may result in more than one report. Furthermore, the absence of a report is most probably a sign of network congestion. This congestion may be transitory, so it is better for the source to wait a little before retransmitting requests to reports that were not received, hopefully during this time the congestion disappears and the retransmitted requests and their corresponding reports succeed to get through.

The operation in rounds has another advantage, that of absorbing the excessive delay that some reports may experience. Between the transmission of a request in a round, and its retransmission in the next round, there is enough time for the report to arrive at the source (supposing that the report is simply delayed in the network and not lost). As we will explain later, the excessive delay of a report is considered by our protocol as a sign of network congestion. The delayed report is however not discarded when it arrives at the source; its content is considered in the same way as that of non-delayed reports.

3.3 Flow control

The main task of a TICP source is to control the rate at which requests and reports are injected into the network. The congestion of the network may appear on the return path from receivers to the source. It may also appear due to requests in the forward direction. To avoid network congestion, we consider a window-based flow

control mechanism similar to that of TCP [9]. The source detains one variable `cwnd` that indicates the maximum number of receivers it can probe at the same time. We call it *congestion window*. Two particular cases to be cited:

1. `cwnd=1`: The protocol operates in a stop-and-wait manner. The source probes one receiver, waits for its report, probes another receiver, and so on. To avoid deadlock, the source can take the decision that a report is lost if not received within a certain time, e.g. within a source estimate of the round-trip time. A request to a new receiver (or to the same receiver if there is still only one receiver that did not send its report) is sent when this time elapses.

2. `cwnd=`∞: The source probes all receivers at once. It waits for a certain time, then decides that some reports are lost, and probes the corresponding receivers again.

With this congestion window, TICP limits the number of request packets and reports in the network. At any moment, `cwnd` limits the number of receivers that were probed and whose reports were not received. If there is enough receivers to probe, `cwnd` is equal to the number of expected reports. New requests are transmitted only when the number of expected reports is less than the value allowed by `cwnd`. Later we explain how the source decides that the number of expected reports is less than `cwnd`, and that new (or retransmitted) requests can be transmitted.

3.4 Congestion control

To avoid the congestion of the network, the source adapts its congestion window `cwnd`. We propose two algorithms for adapting `cwnd`, similar to those of TCP: Slow Start and Congestion Avoidance.

Before describing the two algorithms, we suppose for instance that the source disposes of a mechanism to detect network congestion. We describe later our *congestion detection mechanism*. The principle of congestion control is then simple: increase the congestion window until the network becomes congested, back it off, and increase it again.

3.4.1 Packet request size

A TICP source probes `RS` receivers (`RS` ≥ 1) in each request packet; we allow request packets to carry the IDs of multiple receivers. This improves the efficiency of the network and reduces the number of request packets in the forward direction. `RS` also

serves as a lower bound on the congestion window. If it happens that `cwnd` becomes smaller than `RS`, it is reset to `RS`. The TICP source is allowed to send request packets of size smaller than `RS` in the sole case when there is not enough receivers to probe (this happens at the end of the session).

3.4.2 Slow Start

The TICP source starts the session by setting its congestion window to `RS`. This allows the source to send a request packet of size `RS` at the beginning of the session. Some time later, reports start to arrive. Upon each *good* report, the source increases its congestion window by one: `cwnd` $\leftarrow$ `cwnd + 1`. The source does not increase its congestion window when a *bad* report arrives. We explain later in details what we mean by *good* and *bad* reports. For instance, we note that the words *good* and *bad* have no relation with the content of reports. They are only used for congestion control. Both types of reports reach the source and their contents are good. A good report indicates that the network is not congested and that the source can go on in increasing its congestion window. A bad report indicates the opposite, and the source refrains from increasing its congestion window if the number of bad reports exceeds some threshold that we define later.

During Slow Start, the congestion window doubles when all expected reports (of number `cwnd`) arrive. The Slow Start phase continues until the network becomes congested. Here, the source divides its congestion window by two and enters the Congestion Avoidance phase. Clearly, the objective of Slow Start is to gauge quickly (and not aggressively) the network capacity at the beginning of the session. The source passes by Slow Start at the beginning of the session. If the network is not severely congested, the source will not come back to Slow Start. It comes back to Slow Start when a severe congestion appears. We call this severe congestion a *Timeout* event, and we explain later when it happens while describing our congestion detection mechanism.

3.4.3 Congestion avoidance

Congestion Avoidance follows Slow Start. It represents the steady state phase of TICP, whereas Slow Start represents the transitory phase. During Congestion Avoidance, the source increases slowly its congestion window `cwnd`. Upon each good report, the congestion window is increased by the following amount: `cwnd` $\leftarrow$ `cwnd + RS/cwnd`. With this rule, the congestion window increases by `RS` when all expected reports (of number `cwnd`) arrive at the source. This allows the source to probe `RS`

more receivers, which increases the sum of requests and reports in the network by `RS`. When congestion is detected, the congestion window `cwnd` is divided by two, and a new Congestion Avoidance phase is started. The Congestion Avoidance phase is similar to that of TCP where the window is slowly increased by one packet every round-trip time (when the number of expected acknowledgements arrive) [9].

3.4.4 Timeout

The network may become severely congested. We describe later how the source can detect such an event. Here, we only explain how the source reacts. The TICP source reacts by closing its congestion window `cwnd` to `RS`, and by resorting to a new Slow Start phase. Thus, in case of Timeout,

```
cwnd ← RS
return to Slow Start
```

3.5 Sliding the window and sending new requests

The source detains a variable that indicates the number of expected reports, or the sum of requests and reports flying in the network. This is also the number of receivers that were probed and whose reports were not received. We denote this variable by `pipe`. When a good report is received, the source decreases `pipe` by one. When `pipe` falls below `cwnd`, the source transmits a new request packet (or more) of size `RS`, clearly if the window allows. Therefore, when a good report arrives,

```
pipe ← pipe -1
if ((cwnd - pipe ) ≥ RS) do {
send a request packet of size RS
pipe ← pipe + RS }
until ((cwnd - pipe ) < RS)
```

For instance, note that when a bad report is received, the source simply tries to transmit new requests without changing the variables `pipe` and `cwnd`.

As explained in Section 3.2, the source probes the receivers in rounds. Consider the first round. When congestion control allows the source to inject new requests into the network, it probes receivers to whom it did not send any request. When the list of all receivers is crossed (from left to right) , the source comes back to the left, starts a new round, and transmits requests to receivers whose reports were not

received in the first round. The same thing applies when the second round ends, and so on. This continues until all reports are received.

This section mentions that the source sends requests when reports are received. In fact, there are other moments at which the source can send requests, if its window allows. Our protocol disposes of a timer, and when this timer expires, the source checks (as above) if the congestion window `cwnd` allows to probe new receivers, and if so, new requests (or retransmissions) are emitted. We explain in the next section this timer, which is an important component of TICP used for congestion detection.

3.6 Congestion detection mechanism

This mechanism forms an important part of our protocol. It can be designed in different ways. We choose to design it with a timer. This mechanism serves for different purposes. It serves to distinguish good reports from bad reports. It serves to decide if a report (or a request) is lost or not. It serves to slide the left-hand side of the congestion window. Finally, it serves to trigger the transmission of new requests, in the same way the arrivals of reports do. This mechanism is similar to the Retransmission Timer in TCP.

3.6.1 Round-trip time estimator

Our aim is to set the timer of our mechanism to an estimate of the round-trip time, using the samples of the round-trip time seen so far. This choice will be made clear later. We compute the value of the timer using estimates of the average round-trip time and of its variance. This computation is similar to what is done by TCP [13]. The difference from TCP is that in our case, the round-trip time mainly varies due to the presence of different receivers with different paths to the source, however in the case of TCP, the round-trip time varies due to the variation of queuing time in routers.

As with TCP, our protocol estimates the average round-trip time and its variance using Exponentially Weighted Moving Average algorithms. Let `srtt` and `rttvar` be the average and the mean deviation of the round-trip time. The source timestamps the requests and the receivers echo the timestamps in their reports. The source can then measure the round-trip time when reports arrive. Let `rtt` be a measurement of the round-trip time obtained when a report arrives. The source updates its estimates in the following way:

```
rttvar ← (3/4) rttvar + (1/4) |srtt - rtt|
srtt ← (7/8) srtt + (1/8) rtt
```

The value of the timer (`TO`) is then set to: `TO` $\leftarrow$ `srtt + 4 rttvar` . The coefficients of the estimator are taken equal to those of TCP retransmission timer, which has proven its efficiency in controlling the congestion of the Internet.

At the beginning of the session, `TO` can be set to a default value, for example 3 seconds. It can also be set to its value in past collect-information sessions. `srtt` can be set to the first round-trip time measurement, and `rttvar` to half this measurement. They can also be set to their values in past sessions.

With the value of `TO` set in this way, we are quite sure that reports corresponding to a request packet sent when the timer is scheduled, will arrive before the expiration of the timer, of course if these reports were not lost in the network. This is exactly the idea behind the retransmission timer of TCP [9]. A TCP packet not acknowledged before the expiration of the timer is a strong indication that the packet was lost. With `TO`, we have then a (dynamic) time window that allows to decide whether the network is congested or not, by simply computing the loss rate of reports expected to arrive during this time interval (between the scheduling of the timer and its expiration). The next sections explain the role of this timer in detecting the congestion (and the severe congestion) of the network.

3.6.2 Scheduling the timer

The timer is scheduled at the beginning of the session after the transmission of the first request. It is rescheduled (with a new value of `TO`) every time it expires.

3.6.3 Detecting network congestion

The idea is to compute the loss rate of reports expected to arrive during a time window equal to `TO`. The source compares this loss rate to two thresholds to decide whether the network is not congested, congested, or severely congested. The computation of the loss rate, and consequently, the adaptation of the congestion window, are done when the timer expires. This is done in the following way.

When the timer is scheduled, the source detains in one variable the number of reports to be received before the expiration of the timer. Denote this variable by `torecv`. Let `recv` be the number of good reports received between the scheduling of the timer and its expiration. The source makes the assumption that `(torecv - recv)` reports were lost in the network. It estimates the lost rate of reports to `1 - recv/torecv`. The network is considered as congested if this loss rate exceeds a certain threshold `CT` (Congestion Threshold). The congestion window is then divided by two. The network is considered as severely congested if the loss rate exceeds a

higher threshold `SCT > CT` (Severe Congestion Threshold). The congestion window is reset in this latter case to `RS`, and Slow Start is entered.

`CT` and `SCT` are two parameters of our protocol. They can be set to some default values, for example, to 10% for `CT` and to 90% for `SCT`. We set them as follows:

```
CT = min (0.1 , RS/cwnd)
SCT = max (0.9, (cwnd - RS) / cwnd)
```

The minimum and maximum functions in the expressions of `CT` and `SCT` are necessary to ensure that these thresholds do not take unrealistic values when the congestion window is of small size (close to `RS`). One can use other default values than 0.1 and 0.9. For large congestion windows, `CT` is equal to `RS/cwnd`, which means that congestion is concluded when more than `RS` reports are not received (resp. severe congestion is concluded when less than `RS` reports are received in a window). We recall that a report is not received if it is lost (resp. delayed), or if the corresponding request itself is lost (resp. delayed).

The way we set the two thresholds is compliant with TCP, which considers that the network is congested if one packet is lost. A TCP packet corresponds in our case to `RS` reports. The network is considered by TCP as severely congested when all packets are lost. This is reflected by the way we set our `SCT` parameter. With this setting of `CT` and `SCT`, our protocol is able to control the congestion in the forward and reverse directions in a TCP-friendly way. We explain this issue in Section 5. For instance, if we consider the forward direction, the loss of a request packet results in the loss of `RS` reports, which leads to a division of TICP congestion window by two, exactly the same reaction of TCP to the loss of a data packet. The loss of all request packets in the forward direction triggers a Timeout, a reset of the congestion window to `RS` and the call of Slow Start, which is similar to TCP behavior. The friendliness with TCP comes also from the fact that TICP increases its congestion window in the same way TCP does (during both Slow Start and Congestion Avoidance).

3.6.4 Good and bad reports

We come now to the formal definition of good and bad reports. A good report is a report received before its deadline. The deadline of a report is the expiration of the timer. After this deadline, the report is assumed to be lost, and will be considered by the source as a bad report if it arrives later at the source. A bad report is used to update the round-trip time. However, it is not used to increase the congestion window, nor to change the variable `pipe`.

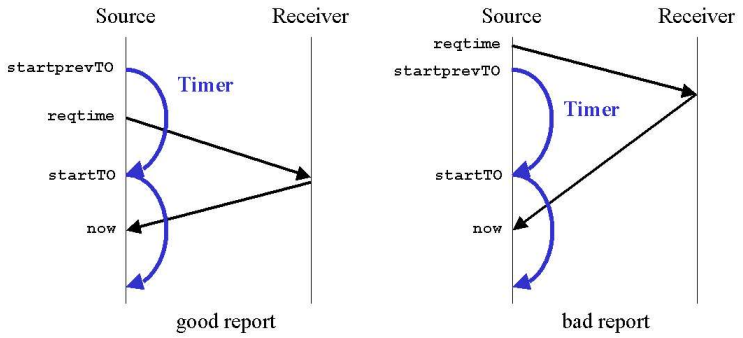

Figure 1: Good and bad reports

The content of a bad report is considered and added to the list of received information. The source does not ask a receiver that has sent a bad report to resend it in subsequent rounds. A report is only bad from congestion control point of view. In subsequent rounds, the source only sends requests to receivers whose reports did not arrive in previous rounds.

We explain now how a source can distinguish good reports from bad ones. This explanation is illustrated in Figure 1. Let `startTO` be the scheduling time of the timer. Let `startprevTO` be the last scheduling time of the timer. When a report is received, the source extracts from its header the timestamp echoed by the receiver, which indicates the time at which the corresponding request has been issued. Denote this time by `reqtime`. The report is good if and only if `startprevTO < reqtime`. The report is bad in the opposite case. In other words, a report is good if it is received before the expiration of the first timer that is scheduled after the transmission of the corresponding request.

When the timer expires and before it is rescheduled, the above variables are updated in the following way:

```
startprevTO ← startTO
startTO ← now
```

3.6.5 Sliding the left-hand side of the window when the timer expires

In Section 3.5, we explained how the left-hand side of the window slides when good reports arrive. This sliding is realized by decrementing the variable `pipe` by the number of good reports. The variable `pipe` has also to be decremented when reports

are considered by the source to be lost, otherwise we end up with a situation where `pipe` overestimates the number of reports in the network, which drains out the network and stops the protocol.

The source decides that some reports are lost every time the timer expires. The number of (supposedly) lost reports is approximated by the source to `torecv - recv`. Therefore, when the timer expires, the source decrements its variable `pipe` by `pipe` $\leftarrow$ `pipe - (torecv - recv)` . If the congestion window allows, the source transmits then new requests of size `RS` and increases its variable `pipe` so as to account for these new transmissions.

3.6.6 Updating the variable `torecv`

We already saw that this variable indicates the number of reports to be received between the scheduling of a timer and its expiration. To update `torecv`, we need to introduce a new variable, which is the number of requests sent by the source between the scheduling of the timer and its expiration. Denote this latter variable by `sent`. When the timer expires, the source does the following:

```
torecv ← sent
sent ← 0
```

3.6.7 Updating the variable `sent`

This variable indicates the number of requests sent between `startprevTO` and `startTO`, and to be received before the timer scheduled at time `startTO` expires. The variable `sent` is incremented when new requests are transmitted, i.e. `sent` $\leftarrow$ `sent + number of request messages sent`. The number of request messages sent is in general equal to `RS` times the number of request packets sent.

The variable `sent` is reset to zero when the timer expires. It can also be decremented when a good report arrives. Indeed, when a good report arrives, we have two distinct cases: (i) `startprevTO` $<$ `reqtime` $\leq$ `startTO`, and (ii) `startTO` $<$ `reqtime`. In the first case, `sent` is not decremented. In the second case, it is decremented by 1, i.e. `sent` $\leftarrow$ `sent - 1`. This decrease is necessarily, since a good report satisfying (ii) must not be included in the number of reports to receive after the expiration of the timer scheduled at `startTO`. Recall that the variable `torecv` is set to `sent` when the timer expires.

4 Main algorithm

We group together in one algorithm the different functions and variables explained in the protocol description section. We implemented this algorithm into the network simulator `ns-2`, and we validated its performance. The results of the simulations are presented in Section 6.

The source starts the collect-information session by sending one request packet of size `RS` (that probes `RS` receivers). It sets its variables as follows,

```
cwnd ← RS
pipe ← RS
torecv ← RS
sent ← 0
recv ← 0
```

The source then schedules its timer with the following parameters,

```
TO ← default value, e.g.  3 seconds
startprevTO ← -1
startTO ← now
```

When a report arrives, the first thing to do is to update `srtt`, `rttvar`, and `TO`,

```
rtt ← measured round-trip time
rttvar ← (3/4) rttvar + (1/4) |srtt - rtt|
srtt ← (7/8) srtt + (1/8) rtt
TO ← srtt + 4 rttvar
```

Then the source proceeds to the adaption of its congestion window and the transmission of new requests. The congestion window is adapted if the report is good. Requests are transmitted for both types of reports.

```
if the report is good i.e.  (startprevTO < reqtime) {
if (Slow Start) cwnd ← cwnd + 1
if (Congestion Avoidance) cwnd ← cwnd + RS/cwnd

if (reqtime ≤ startTO) recv ← recv + 1
else sent ← sent - 1

pipe ← pipe - 1 }

send new request packets of size RS each (if the window allows)
pipe ← pipe + number of request messages sent
sent ← sent + number of request messages sent
```

Now, when the timer expires

```
if (CT ≤ (1 - recv / torecv) < SCT)
cwnd ← cwnd/2 (network is congested, stay in Congestion Avoidance)

if ((1 - recv / torecv) ≥ SCT)
cwnd ← RS (network is severely congested, go to Slow Start)

pipe ← pipe - (torecv - recv)

send new request packets of size RS each (if the window allows)
pipe ← pipe + number of request messages sent
sent ← sent + number of request messages sent

torecv ← sent
recv ← 0
sent ← 0

startprevTO ← startTO
startTO ← now

Reschedule the timer using the current value of TO
```

The source always crosses the list of receivers from left to right, and sends requests only to those it has never probed, or to those whose reports are (supposed) lost. The source does not resend a request to a receiver before taking the decision that its report

is lost. The algorithm stops when all reports are received (or when the duration of the session exceeds some allowed time).

5 Friendliness of our protocol with TCP traffic

Our protocol is developed with the main objective to be friendly with applications using the TCP protocol. The TCP-friendliness is also an indication on the intra-fairness of our protocol, i.e. fairness between multiple sessions using TICP. We illustrate this TCP-friendliness in two cases. We keep the study of the other cases for future research.

The first case is when the bottleneck exists in the forward direction, and all requests cross this bottleneck. The network on the return path is not congested. One can consider the example of a TICP source accessing the receivers via a slow satellite link, and the receivers sending back their reports via high speed terrestrial networks. Suppose that a TICP session shares the forward path with a TCP connection having approximately the same average round-trip time. Both the TICP session and the TCP connection increase their congestion windows in a similar way (linearly during Congestion Avoidance by roughly one packet every round-trip time). They both divide their congestion windows by two when one or more packets are dropped on the forward path (Section 3.6.3). Indeed, the loss of one or more request packets results in a report loss rate larger than `RS/cwnd`, which triggers our congestion detection mechanism. Our protocol achieves then the same throughput on the forward path as that of the TCP connection in terms of packets/s, of course if the network drops packets from both flows in the same way. The throughputs of the two flows are equal in terms of bits/s if the size of a request packet is that of a TCP packet. As for the throughput of TICP reports on the reverse path (receivers to source), this throughput depends on `RS` and the size of reports.

We study now the second case. The bottleneck exists on the return path and all reports cross this bottleneck. The forward path is not congested. Consider that a TCP connection shares the bottleneck with the reports sent by receivers, and that both the TCP connection and the TICP session have approximately the same average round-trip time. Our protocol divides its congestion window by 2 when more than `RS` reports are lost, and increases the number of reports in the network by `RS` reports when `cwnd` reports are received (Congestion Avoidance mode). The flow of reports behaves then approximately as an aggregate of `RS` TCP connections having a packet size equal to the size of a report. If the size of `RS` reports is equal to that of a normal TCP packet, the throughput of reports on the reverse path in bits/s will be then

comparable to that of the competing TCP connection. If we want the throughput of reports on the reverse path to be equal to that of `N` TCP connections, and given the size of TCP packets and that of TICP reports, one can find a value for `RS` that realizes this objective.

Clearly, `RS` is an important parameter to decide the TCP-friendliness of our protocol. Let us define TCP-friendliness as realizing a throughput equal to that of a single concurrent TCP connection having the same average round-trip time. If we want TCP-friendliness in the forward direction, we have to choose `RS` so that the size of request packets is equal to that of TCP data packets. If we are concerned with TCP-friendliness in the reverse direction, `RS` has to be chosen so that the size of `RS` reports is equal to that of a TCP data packet. The simulation results presented in the next section illustrate the TCP-friendly feature of our protocol in the above two cases.

The situation becomes more complex when the TICP session crosses at the same time multiple bottlenecks. A TICP source measures the total loss rate in the network, and distributes its requests to receivers without taking into consideration their positions with respect to bottlenecks. Thus, some bottlenecks may see more reports than what a TCP connection would achieve through these bottlenecks. The TICP session may be less aggressive than TCP at other bottlenecks. TICP is a TCP-friendly protocol if we look at the network as a single bottleneck. The distribution of reports over bottlenecks may be unfriendly with TCP. A future research is to make TICP TCP-friendly everywhere. This will most probably require that the source considers the positions of receivers with respect to bottlenecks while sending requests. One possible solution to this problem is to divide receivers into regions, and to probe the regions in parallel or in sequence. By doing that, one can hope that receivers of one region will be behind the same bottleneck, which will avoid the problem of multi-bottleneck scenarios.

6 Validation of the protocol by simulation

We implement our protocol in the network simulator `ns-2` [12]. Then, we simulate different scenarios to prove the effectiveness of our protocol in controlling the congestion of the network, and in fairly sharing the available resources with competing traffic.

The first objective of a congestion control protocol is to optimize the utilization of network resources. This is equivalent to a high utilization and to a low loss rate. The low loss rate is synonymous to short queues in routers (short queuing delay).

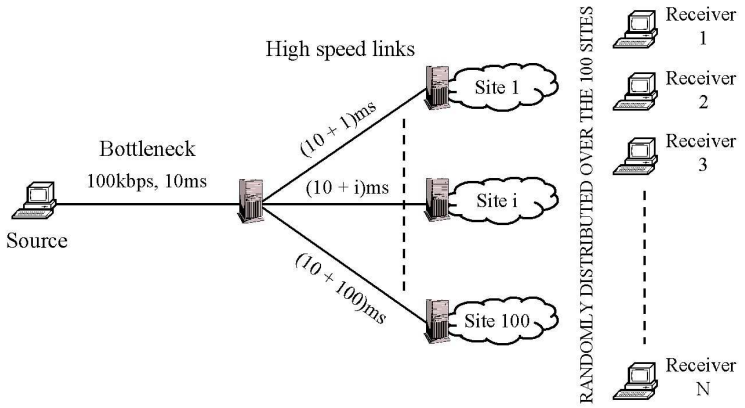

Figure 2: Simulation testbed

The tradeoff between utilization and loss rate can be illustrated by changing the congestion window `cwnd`. Consider a version of TICP where the congestion window is constant during all the session. When `cwnd` is small, the utilization is low, and the loss rate too. When `cwnd` increases, the utilization increases until approximately 100%, but the loss rate increases as well. There is an optimal congestion window that leads to a high utilization and to a low loss rate. The objective of a congestion control protocol is to operate around this optimal window. Above this optimal window, the network is congested, the utilization drops due to retransmissions, and the loss rate increases fast.

The second objective of a congestion control protocol is to fairly share the available bandwidth with competing traffic. A distributed protocol like TCP is known to distribute the available bandwidth among connections inversely proportional to their round-trip times and to the number of routers they cross [5]. When our protocol is used in presence of competing traffic (TCP or TICP traffic), it must achieve a fair share of the available network resources. The competing traffic must not be penalized by the presence of our collect-information session, and at the same time, our session must not obtain much less than its fair share. The following simulations validate that our protocol TICP realizes the above objectives.

6.1 Simulation scenario

We study different simulation scenarios, which are illustrated by the network in Figure 2. All scenarios have in common the fact that one (sometimes two) TICP source (located at Source) probes a large number of receivers (thousands) uniformly spread over 100 Internet sites. The source joins the receivers by Centralized Multicast, which is an implementation in ns-2 of PIM-SM. The source is connected to the Internet via a slow link of 100 kbps, which forms a potential bottleneck for both requests and reports. The 100 sites are connected to the Internet via high speed links that are always non-congested. The round-trip time (excluding queuing delay) between the source and receivers on site i, $i = 1, \ldots, 100$, is set to $2.(10 + 10 + i) = 40 + 2.i$ ms. This round-trip time covers a large number of Internet paths ranging from terrestrial links to satellite ones. Buffers at the two sides of the 100 kbps link are set to 20 packets and are of DropTail type. The receivers have IDs ranging from 0 to N, where N is the total number of receivers. The N IDs are randomly affected to the 100 sites.

We run three sets of simulations. The first set corresponds to a TICP session running alone in the network. The objective of this set of simulations is to show how well our protocol avoids network congestion and how efficiently it uses the available bandwidth. In the second set, two TICP sessions share the 100 kbps, both sessions run in the same direction. In the third set, a TICP session shares the 100 kbps link with a TCP NewReno connection. The two-way propagation delay between the source and the destination of the TCP connection is taken equal to 100ms, which is approximately the average round-trip time between the TICP source and the N receivers in the 100 sites. The TCP connection is used to transfer an infinite amount of data. It has a large receiver window and packets of size 1000 bytes. The objective of the second and third sets of simulations is to illustrate the TCP-friendliness feature of TICP.

We consider different values for `RS`, the size of request messages, and the size of reports. We change these values in order to move the congestion of the network between the forward and the backward paths, and to control the TCP-friendliness of our protocol, as discussed in Section 5. For a certain request message size `MS`, the TICP source sends request packets of size `RS.MS`. The request message to a receiver includes its ID plus some additional information that may help the receiver in preparing its report.

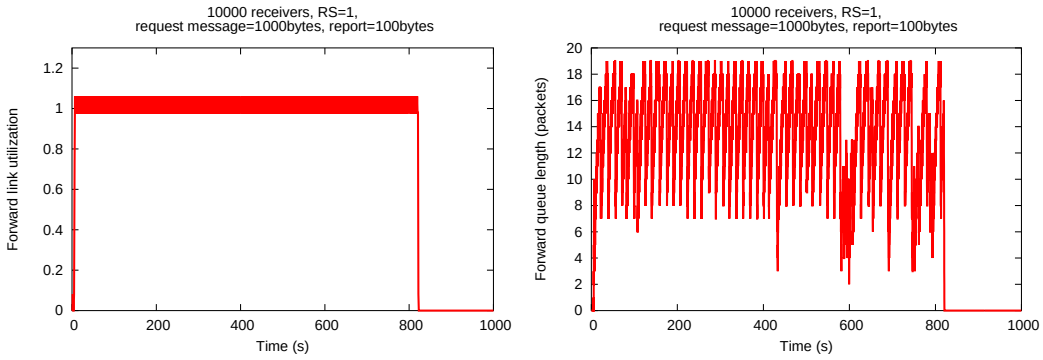

Figure 3: One TICP session, congestion in the forward direction

6.2 Scenario without competing traffic

We consider a single TICP session that collects information from 10000 receivers. We run our protocol until all the information from receivers is well received. First, we allow the congestion to appear in the forward direction by setting the size of request messages to a large value 1000 bytes, and the size of reports to a small value 100 bytes. Then, we move the congestion to the reverse direction by interchanging the sizes of request messages and reports. Concerning the value of `RS`, we set it to 1 in the first case and to 10 in the second case. The size of request packets is then constant in both cases and equal to 1000 bytes. Figure 3 corresponds to the case where congestion is on the forward path. This figure shows the utilization of the 100 kbps in the forward direction (source to receivers) and the queue length at its input. Figure 4 corresponds to the case where congestion is on the reverse path. This latter figure shows the utilization of the 100 kbps in the reverse direction (receivers to source) and the queue length at its input. The utilization is computed by averaging the number of bits transmitted over 1 second time intervals, then by normalizing this average by the link speed. Clearly, our protocol adapts to the available bandwidth in each direction of the 100 kbps link, and does not overload the buffers. This is illustrated by the duration of the session, which is very close to its ideal duration 800 seconds, i.e. the duration achieved when the utilization of the bottleneck link is 100% and no packets are lost.

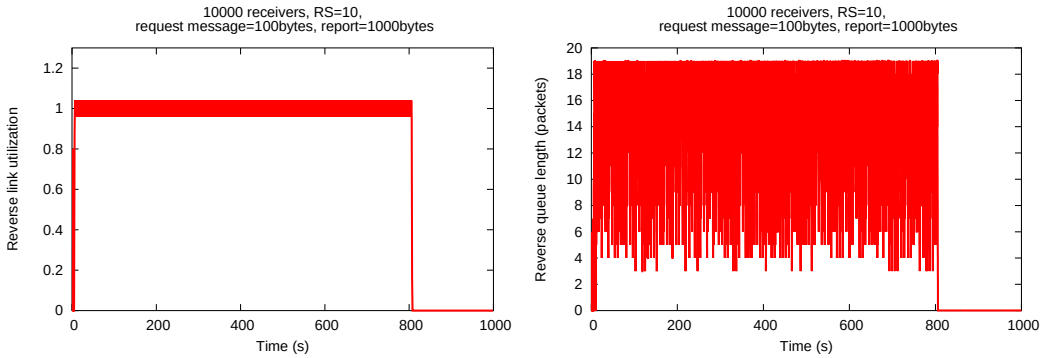

Figure 4: One TICP session, congestion in the reverse direction

6.3 Scenario with competing traffic

We run two sets of simulations to study the fairness of our protocol and its TCP-friendliness. First, we consider the case where 2 TICP sessions run together in the network, then we study the case where TICP shares the network with TCP.

6.3.1 Fairness of TICP

Two TICP sessions collect information from 5000 receivers each, spread uniformly over the 100 sites. The first session starts at time 0. The second one starts at time 250 seconds (before the end of the first one). As before, we allow first congestion to appear in the forward direction (source to receivers) by setting the size of request messages to 1000 bytes and the size of reports to 100 bytes, then we move congestion to the reverse direction (receivers to source) by interchanging the sizes of request messages and reports. We set `RS` to 1 in the first case and to 10 in the second case, which leads to request packets of constant size equal to 1000 bytes. When congestion is on the forward path, we plot the bandwidths consumed by requests of both sessions over the 100 kbps link. When congestion is on the reverse path, we plot the bandwidths consumed by reports of both sessions over the 100 kbps link. This gives rise to Figures 5 and 6. Consumed bandwidths are computed by averaging transmitted data over 1 second time intervals, then by normalizing to the link speed. As we see, before 250 seconds, the first session fully utilizes the link bandwidth. When the second session arrives, the bandwidth consumed by the first session is divided by two, and the second session consumes the other half of the link

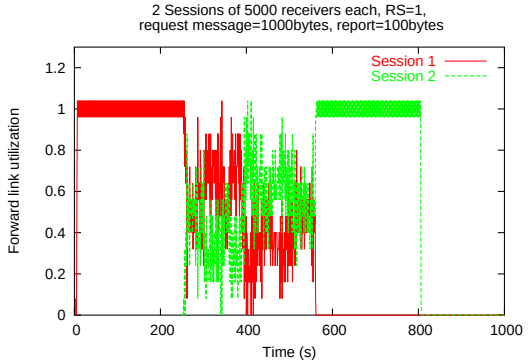

Figure 5: Two TICP sessions, congestion in the forward direction

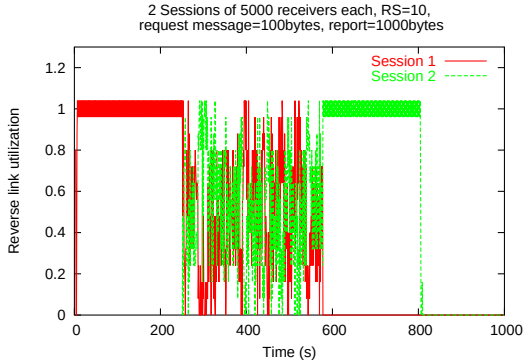

Figure 6: Two TICP sessions, congestion in the reverse direction

bandwidth. This continues until the end of the first session, where the second session sees its bandwidth multiplied by two, and it keeps fully utilizing the link bandwidth until its end at a time slightly longer than the ideal total duration of both sessions, i.e. 800 seconds.

6.3.2 TCP-friendliness of TICP

We consider now one TICP session that collects information from 10000 receivers spread over the 100 sites. The session shares the 100 kbps link with one long-lived TCP connection of round-trip time 100 ms. First, we run the TCP connection in

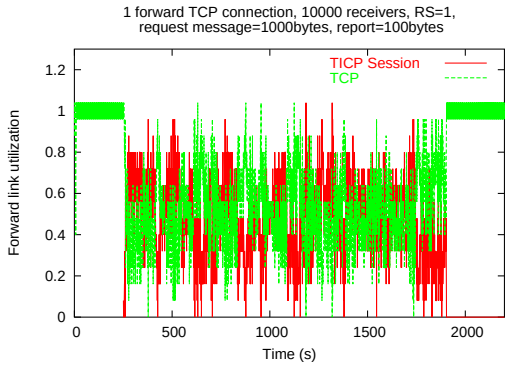

Figure 7: One TICP session and one long-lived TCP connection, congestion in the forward direction

the forward direction, and we let it share the 100 kbps link with request packets. Second, we consider the same TCP connection but this time in the reverse direction, and we let it share the 100 kbps with TICP reports. In this way, we are able to check the TCP-friendliness of our protocol in both directions.

For the forward direction, we set `RS` to 1 and the request message size to 1000 bytes, which means that request packets have the same size as that of TCP packets (1000 bytes). We set the size of reports to a small value 100 bytes in order to remove any congestion from the reverse path. The TCP connection starts at time 0, the TICP session starts at time 250 seconds. We plot the bandwidths consumed by TCP packets and TICP requests in Figure 7. Consumed bandwidths are normalized by the link speed. During the lifetime of the TICP session, the 100 kbps link bandwidth is almost fairly shared between both flows (TCP and TICP requests); the arrival of the TICP session does not penalize the TCP connection, and the TCP connection does not consume more than its fair share of the available bandwidth. When the TICP session is off, the TCP session fully utilizes the available bandwidth.

Then, we run the TCP connection in the reverse direction. We set `RS` to 10 and the size of reports to 1000 bytes. The size of request messages is set to a small value 100 bytes in order to remove any congestion from the forward path. We plot in Figure 8 the bandwidth consumed by TCP data packets and that consumed by reports over the 100 kbps link. The consumed bandwidths are averaged over 1 second intervals and are normalized by the link speed. We notice how our protocol is more aggressive than TCP because the product of `RS` and report size is much larger than

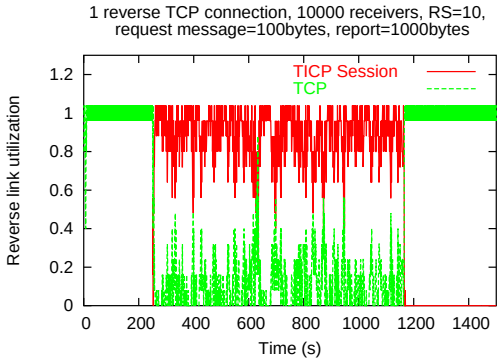

Figure 8: One TICP session and one long-lived TCP connection, congestion in the reverse direction

TCP packet size (see discussion in Section 5). The flow of reports behaves approximately as 10 long-lived TCP connections. The TCP-friendliness of our protocol can be improved by reducing `RS` or the size of reports, this if we define TCP-friendliness as realizing a throughput equal to that of a TCP connection running in the same network conditions. One should expect a rate of TICP reports close to that of TCP when the product of `RS` and the report size is equal to TCP packet size 1000 bytes. To prove that, we rerun the same simulation as that in Figure 8, but this time with `RS` equal to 1 - report size equal to 1000 bytes, and `RS` equal to 10 - report size equal to 100 bytes. In both cases, the product of `RS` and report size is equal to TCP packet size. We plot the results in Figures 9, where it is clear that our protocol is more TCP-friendly than in Figure 8. We also notice that when we reduce the report size to 100 bytes, the duration of the TICP session is shorter since less information is to be collected from receivers.

7 Conclusions

We present in this paper TICP, a TCP-friendly Information Collecting Protocol. A source running TICP is able to collect the entire information from a large number of receivers spread over the Internet. TICP provides a reliable data collection service, while controlling the congestion of the network and ensuring fairness with other sessions using TICP, or with other flows using the TCP protocol.

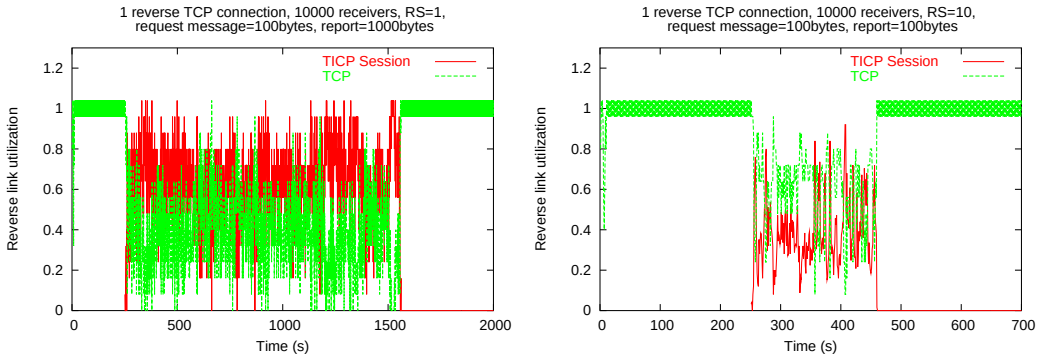

Figure 9: One TICP session and one long-lived TCP connection, congestion in the reverse direction, better TCP-friendliness

Our work on TICP can be extended in different directions. A first extension is to cope with situations where multiple bottlenecks exist at the same time in the network. Another extension is to consider the collection of large reports that cannot fit in one packet. We are also intending to implement TICP and to test its performance on real network testbeds.

References

[1] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", *RFC 2581*, Apr. 1999.

[2] S. Alouf, E. Altman, P. Nain, "Optimal on-line estimation of the size of a dynamic multicast group", *IEEE INFOCOM*, Jun. 2002.

[3] S. Alouf, E. Altman, C. Barakat, P. Nain, "Estimating Membership in a Multicast Session", *to appear in proceedings of ACM SIGMETRICS*, Jun. 2003.

[4] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors", *Communications of the ACM*, 13(7): 422-426, July 1970.

[5] S. Floyd, “Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic”, *ACM Computer Communication Review*, vol. 21, no. 5, pp. 30-47, Oct. 1991.

[6] S. Floyd, K. Fall, "Promoting the Use of End-To-End Congestion Control in the Internet", *IEEE/ACM Transactions in Networking*, vol. 7, no. 4, pp. 458-472, Aug. 1999.

[7] S. Floyd, V. Jacobson, S. McCanne, C. Liu, L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing", *ACM SIGCOMM*, Aug. 1995.

[8] T. Friedman, D. Towsley, "Multicast session membership size estimation", *IEEE INFOCOM*, Mar. 1999.

[9] V. Jacobson, "Congestion avoidance and control", *ACM SIGCOMM*, Aug. 1988.

[10] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options", *RFC 2018*, Oct. 1996.

[11] J. Nonnenmacher, E. Biersack, "Scalable feedback for large groups", *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 375-386, Jun. 1999.

[12] The LBNL Network Simulator, *ns*, http://www.isi.edu/nsnam/ns/

[13] V. Paxson, M. Allman, "Computing TCP's Retransmission Timer", *Internet Draft*, work in progress, Apr. 2000.

[14] K. Thompson, G.J. Miller, R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics", *IEEE Network*, vol. 11, no. 6, pp. 10-23, Nov. 1997.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399