TICP: Transport Information Collection Protocol

Chadi Barakat, Mohammad Malli
INRIA - Planète research group
Sophia Antipolis - France
{cbarakat, mmalli}@sophia.inria.fr

Naomichi Nonaka
Hitachi, Ltd.
Systems Development Laboratory
nonaka@sdl.hitachi.co.jp

***Abstract*— We present and validate TICP, a TCP-friendly reliable transport protocol to collect information from a large number of sources spread over the Internet. TICP is a stand-alone protocol that can be used by any application requiring the reliable collection of information. It ensures two main functions: (i) the information arrives at the collector entirely and correctly, (ii) the implosion at the collector and the congestion of the network are avoided. The congestion control in TICP is done by having the collector probe the sources at a rate function of network conditions. The probing rate increases and decreases in a way similar to how TCP adapts its congestion window. We implement TICP in ns-2 and validate its performance. In particular, we show how efficient TICP is in quickly and reliably collecting information from a large number of sources, while avoiding network congestion and being fair with competing traffic.**

***Résumé*—Nous présentons et validons TICP, un protocole transport fiable, courtois avec TCP, qui sert à collecter des informations d'un grand nombre de sources distribuées à travers l'Internet. TICP est un protocole indépendant pouvant être utilisé par toute application demandant une collecte fiable d'informations. Notre protocole assure que les deux objectifs suivants soient réalisés: (i) l'information à collecter arrive entièrement et correctement au receveur, et (ii) l'implosion du receveur et la congestion du réseau sont évités. Le contrôle de congestion dans TICP est basé sur le receveur sondant les sources à une vitesse fonction de l'état du reseau afin qu'elles renvoient leurs informations. TICP est implémenté et validé dans ns-2. Nous montrons par des simulations intensives l'efficacité de TICP dans une collecte rapide et fiable d'informations tout en évitant la congestion du réseau et restant courtois avec le trafic concurrent généré par les autres applications.**

I. Introduction

This paper describes TICP, a TCP-friendly reliable transport protocol for collecting information from a large number of sources spread over the Internet. TICP stands for Transport Information Collection Protocol. It allows an entire collection of the data and avoids congestion by using efficiently the available network resources and being fair with the competing TCP traffic of other applications. Applications using TCP are known to form the majority of Internet traffic [18], hence it is important for a new transport protocol like TICP to be friendly with them [8]. The originality of TICP is in the new service it provides and in the efficient, yet simple, functions it supports.

TICP is general since it does not impose any constraint on the type of the collected information. In particular, this information does not need to be filterable or addable. The protocol ensures that the information is entirely collected from the sources. This generality widens the spectrum of applications of the protocol. One example could be the case of a collector that wants to know if and which sources have well received a certain document. Other examples of data to collect include the quality of TV or video reception (who is not receiving a good quality), the measurements from network monitoring devices, the weather (the temperature at different points of the globe), the declaration on revenues, the census of the population, the results of a vote, etc. The protocol can be occasionally used as when the information is collected at the end of a working day. It can also be frequently used as when the collector decides to know the quality of reception during a multimedia broadcast session (loss rate, average delay).

The TICP collector sends request packets to the sources asking them to send their reports containing the data. The main challenge is that the available bandwidth in the network is limited and that the number of sources can be very large (thousands or more). The limitation of bandwidth is more pronounced in the direction sources-collector, where the volume of reports is in general larger than the volume of request packets[1]. We cannot ask many sources to send their reports at the same time, otherwise the network could become congested. Request packets sent by the collector and reports sent by sources may also result in an aggressive traffic that can be harmful to other applications. Therefore, TICP has to implement a congestion control function that makes the collection traffic (in both directions) not congest the network and not harm the other applications. Also, TICP has to implement an error control function. Reports sent by sources and lost in the network have to be retransmitted in an efficient way that minimizes the collection session duration.

The congestion control part of TICP is inspired from that of TCP [1], [11], and this is for the main purpose to make TICP friendly with the TCP protocol while being easy to implement. A window-based congestion control algorithm is introduced into TICP to decide on how many sources the collector can probe at a certain time. This algorithm is designed to handle the congestion of the network that may be caused by the probes in the direction collector-sources or by the collected data in the direction sources-collector. We call the first direction *forward* and the second one *reverse*, see Figure 2. We also want multiple TICP sessions 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 collection session duration. We explain in this paper the different functions of our

[1]One can see request packets as ACKs in the context of TCP while reports can be seen as data packets.

protocol. We also present a validation of its performance using an implementation that has been done in ns-2, the network simulator [14].

In the next section, we outline the related literature and explain the originality of our protocol. Section III describes the protocol. Each subsection in Section III describes one function of the protocol, and Section IV puts all functions together in one algorithm. Section V discusses the fairness of our protocol with TCP. In Section VI, 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.

II. Related work

The collection of information has been studied in the literature in different contexts. We discuss in this section the three most relevant to our work: reliable multicast, counting the number of sources and sensor networks.

In reliable multicast, sources that did not receive a packet send a NACK asking the collector for a retransmission of the packet (we keep the terms collector and sources even though the collector in this context is transmitting data and the sources are listening). Many NACKs may cause a congestion in the network or at the collector. 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 another source has already sent a NACK for the same packet, since the collector will retransmit anyway the lost packet to all members of the multicast session. The aggregation of NACKs can be done either along a tree that connects all sources or using multicast itself. In [12], it is proposed that leaf sources send their NACKs to a parent source (called designated receiver), which aggregates this information and sends it to its parent until it reaches the collector. In [9], a source waits for a random time before sending a NACK, and listens at the same time if another source has sent a NACK for the same packet. If so, the former source cancels its request, otherwise it sends it when the timer expires. Another approach for NACK aggregation is to use the principle of active networks to program nodes of a multicast tree as advocated by [6].

Counting the number of sources in a multicast session requires that each source sends an "I am here" message to the collector. Sending all these messages is not feasible when the number of sources is large. However, given that the messages are identical, the collector can only ask a subset of sources (say 10%) to send their messages, and try to infer the total number of sources from the number of messages received. Different works have studied such counting scheme, and the selection of the subset of sources is usually done with a *message transmission probability* communicated to all sources by the collector. Some works have considered the case of a fixed population of sources [10], [13], and others have considered the case of a variable population [3], [4]. In [5], the counting of sources is done by the collector using keys instead of transmission probability. The collector sends different sets of keys and only sources whose keys are in a set answer the collector, so the collector can get an information on the number of sources without asking everyone to transmit a message.

The filtering of messages at the sources is only possible since messages are identical. The problem will be much more complex if the collector decides to know, in addition to the number of sources or whether a packet is correctly received or not, some additional information that changes among sources, as the name or the preferences. Here, filtering the information is no longer possible and a protocol as the one we are proposing in this paper is absolutely needed.

The reliable collection of information has also its application in the context of sensor networks. Sensors are sources of information that wake up generally when an event happens and send information about this event to some collecting point called the sink. Some protocols exist in the literature for a reliable collection, e.g., [17], [19]. These protocols agree on that end-to-end transport solutions lead to poor performance given the noisy nature of wireless links connecting the sensors, and the absence of permanent routes caused by the intermittent wake up of sensors and their limited lifetime. Per-hop transport protocols have been advocated for sensor networks. The information is proposed to be reliably sent from one sensor to another until it reaches the sink, with retransmissions done on a hop-per-hop basis. Clearly, such solutions are not optimal in the wired Internet where permanent routes exist and are provided by the IP protocol. Moreover, the round-trip time in the Internet is usually in the order of hundreds of milliseconds and links are of good quality. All this make an end-to-end solution the most appropriate for the Internet, which is what TICP provides. Also note that losses in the Internet are mostly caused by congestion and that other traffic exists, so transport protocols have to implement end-to-end congestion avoidance and error control mechanisms to reduce the number of losses and to provide fairness. This is not the case in sensor networks where information collection is the major source of traffic.

III. 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 IV, 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 collector and reports of the sources, (ii) to enforce fairness, and (iii) to minimize the time necessary to get reports from all sources.

III.1. Clustering sources

TICP probes the sources to send their reports and controls the rate at which probes are sent so as not congest the network. For this congestion control to be effective, TICP resorts to a clustering of sources based on their proximity to each other and to the collector. Sources in a certain neighborhood are first probed, then the protocol moves to sources in another neighborhood, and so on. The idea is that close sources experience very probably the same network conditions on their paths to the collector, i.e., they are located behind the same bottleneck, and hence the loss of reports is an indication that the common bottleneck on the paths to these sources is congested and that the probing rate should be reduced. The absence of losses means that the common bottleneck is not congested and that the probing rate can be increased further. Without this clustering, TICP sees the Internet as one bottleneck link and fails in controlling the congestion at the bottlenecks close to the sources.

We suggest to do the clustering of sources using the binning method proposed in [16]. According to this method, each source must determine its bin by measuring its round-trip time (RTT) to a set of landmark points spread through the Internet. The bin number of a source is formed by putting together the measured RTTs – we refer to [16] for details[2]. One can see the bin number of a source as a vector describing its spatial coordinates. The list of sources is then ranked by the collector using the distances between the collector's bin and the sources's bins. The collector ranks the sources from those belonging to the nearest bin to those belonging to the farthest one. Sources belonging to the same bin are considered as belonging to the same cluster/neighborhood. Note that other methods can be used as well to cluster sources, as for example using the domain names, the geographical position of the sources, etc.

Clusters are probed in a round-robin way over the sorted list of bin numbers, from the closest cluster to the collector to the farthest one. The collector requests the report of a source when the turn arrives to its cluster. Congestion control in TICP limits the rate at which the sources in a cluster transmit their reports. As for protecting clusters from the probes sent to each other, we propose to do it either by using IP multicast with a multicast address associated with each cluster, by using unicast probes, or by relaying the probes at the TICP layer within each cluster. In our simulations, we use the IP multicast solution.

Probing clusters from the closest one to the farthest one ensures a smooth variation of TICP congestion control parameters, e.g., the probing rate. One can imagine a reset of these parameters when moving from one cluster to another. But, since the warming phase in TICP takes some time, we choose to do the transition without resetting the parameters and make TICP adapt them to the quality of the network path that connects the collector to the new cluster. As for starting by the clusters close to the collector, we choose this way for the purpose of collecting as many reports as possible at the beginning of the session, so that if the session stops for any reason, the collector has the majority of the information. This is also useful if the collected information is treated in real time at the collector.

III.2. Routing issues

The probes of the collector can reach the sources using different routing methods. For example, one can imagine the use of native IP multicast when available. This has a particular interest when the probes of the collector are broadcasted to the sources through a satellite link or any other medium with broadcast capabilities. Another possibility is to use IP unicast routing (point-to-point). A third possibility is to use application layer multicast, where a probe (destined for a set of sources) is first sent to some source, then forwarded by this source to its final destinations. In our simulations to validate the protocol, we use the IP multicast routing, where each cluster of sources has its own IP multicast address extracted from the bin number of the cluster. The study of the other routing methods is left for future research.

When receiving a probe requesting its report, a source sends its information to the collector using IP unicast. We consider in this paper the case where the report of a source can be included in one packet. We leave the case of large reports for future research[3]. The probe sent by the collector to a source is called *request message*. A *request packet* is a packet, sent by the collector, that carries multiple request messages to multiple sources. This is useful when multicast is used to deliver probes (either native IP multicast or application-layer multicast). In case of unicast in the forward direction, a request packet carries one request message. A source sends immediately a report to the collector if it receives a packet including a request message addressed to it.

III.3. Addressing

The TICP collector has a list of all sources and every source is distinguished by some ID at the TICP level. If native IP unicast is used in the forward direction to deliver the probes, the ID of a source has to be extended by its IP address or its host name.

Note that in case of multicast in the forward direction, the collector can encapsulate in one request packet more than one request message. These messages can be aggregated so as to reduce the request packet size. Possible aggregation techniques are the use of ID masks or hash functions.

III.4. Error recovery

Sources whose reports are lost need to be probed again for retransmission. We propose the following method that ensures that the collector gathers the maximum volume of information at the beginning of the session. The purpose is to reduce as much as possible the collection session duration. It is a round-robin probing method that works as follows:

- In a first round, the collector sends requests to all sources following the ranked list of their clusters (at a rate determined by the congestion control mechanism to be described later). It does not retransmit request messages to sources whose reports are (judged to be) lost. Note that the absence of a report from a source can be the result of the loss of the request itself rather than the loss of the report.
- In a second round, the collector sends requests to sources whose reports were not received in the first round.

[2]Here a quick example to illustrate how this works. Suppose that a source measures three RTTs to three landmarks. The RTTs are then mapped onto a certain scale, say for example from 1 to 10. This gives X_1, X_2, X_3. The bin number of the source is then $X_1X_2X_3$. A cluster is formed by sources belonging to the same bin.

[3]For a report of size X packets to be delivered by source Y, one possible solution that we are investigating is to substitute the source Y by X virtual sources of one packet each, and use our protocol with one packet sized information to realize the reliable collection.

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

The explanation for this behavior in rounds is simple: it is better to try new sources rather than wasting time sending multiple requests to a source that is located behind a congested link. Multiple requests to the same source result at maximum in one report, however sending the same number of requests to different sources may result in more than one report. Furthermore, the absence of a report is most probably a sign of network congestion. This congestion can be transitory, so it is better for the collector to wait a little before retransmitting requests to reports that were not received, with the hope that 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 corresponding report to arrive at the collector (supposing that the report is simply delayed in the network and not lost). As we will explain later, the excessive delay of reports is considered by our protocol as a sign of network congestion. The delayed reports are however not discarded when they arrive at the collector; their content is considered in the same way as that of any non-delayed report.

III.5. Flow control

To control the rate of requests and reports across the network, we consider a report-clocked window-based flow control mechanism similar to that of TCP [11]. This choice is driven by our major concern to make TICP efficient, scalable, and easy to implement.

The collector maintains one variable `cwnd` that indicates the maximum number of sources it can probe at the same time. We call it *congestion window*. If there is enough sources 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 collector decides that the number of expected reports is less than `cwnd`, and that new (or retransmitted) requests can be sent.

Two particular cases to be cited:

1) `cwnd=1`: The protocol operates in a stop-and-wait mode. The collector probes one source, waits for its report, probes another source, and so on. To avoid deadlock, the collector can take the decision that a report is lost if not received within a certain time, e.g., within an estimate of the round-trip time. A request to a new source (or to the same source if there is still only one source that did not send its report) is sent when this time elapses.
2) `cwnd=`∞: The collector probes all sources at once. It waits for a certain time, then decides that some reports were lost, and probes the corresponding sources again.

III.6. Congestion control

The TICP collector adapts its congestion window `cwnd` based on the observed loss rate of reports. We propose two algorithms: Slow Start and Congestion Avoidance.

Before describing the two algorithms, we suppose for instance that the collector implements 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, then increase it again.

III.6.1. Packet request size: In case of multicast routing in the forward direction, a TICP collector probes `RS` sources (`RS` ≥ 1) in each request packet. 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 collector is allowed to send request packets of size smaller than `RS` in the sole case when there is not enough sources to probe (this happens at the end of the session).

In case of unicast in the forward direction, we have two choices: either keep the value of `RS` greater than one, but send the `RS` request messages in separate small packets, or set `RS` to one. The drawback of setting `RS` to one is that it results in a slower increase in the congestion window. For this reason, we only focus in this paper on `RS` greater than one. One difference from the case of multicast is that in the case of unicast in the forward direction, since probes are sent in separate small packets, we do not need to wait until the window size of TICP allows the transmission of `RS` request messages before sending a request packet. A request message is sent in a separate small packet when the window allows.

III.6.2. Slow Start: The TICP collector starts the session by setting its congestion window to `RS` and sending `RS` request messages. Some time later, reports start to arrive. Some of them arrive before their deadline, others are delayed. We explain later what we mean by the deadline of a report and how to set it. For instance, a timely report indicates that the network is not congested and that the collector can go ahead in increasing its congestion window, so the collector increases its congestion window by one: `cwnd` $\leftarrow$ `cwnd + 1`. At the opposite, the collector does not increase its congestion window when delayed reports arrive since they are an indication of an imminent network congestion.

By applying the above update rule, the congestion window doubles during Slow Start every time all expected reports (of number `cwnd`) arrive. The growth of the congestion window continues until the network becomes congested. Here, the collector divides its congestion window by two and enters the Congestion Avoidance phase. Clearly, the objective of Slow Start is to gauge quickly (but not aggressively) the network capacity at the beginning of the session. If the network is not severely congested, the collector 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.

III.6.3. 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 collector increases slowly its congestion window `cwnd` in order to probe the network for more capacity. Upon each timely report, the congestion window is increased by the following amount: `cwnd` ← `cwnd + RS/cwnd`. With this rule, `cwnd` increases by `RS` when all expected reports (of number `cwnd`) arrive at the collector. This allows the collector to probe `RS` more sources. When congestion is detected, the congestion window `cwnd` is divided by two, and a new Congestion Avoidance phase is started. This behavior is similar to that of TCP during the congestion avoidance phase, where the window is slowly increased by one packet every round-trip time (when the number of expected acknowledgments arrive) [11] and is divided by two upon congestion.

III.6.4. Timeout: The network may become severely congested. We describe later how the collector can detect such an event. For now, the collector reacts to such an event 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
```

III.7. Sliding the window and sending new requests

In addition to `cwnd`, the TICP collector maintains a variable that indicates the number of expected reports, or the number of sources that have been probed and whose reports have not yet been received. We denote this variable by `pipe`.

When a timely report is received, the collector decreases `pipe` by one. When `pipe` falls below `cwnd`, the collector checks whether it can transmit a new request packet (or more) of size `RS`. If so, the request packet is immediately sent. For delayed reports, the collector simply tries to transmit new requests without changing the variables `pipe` and `cwnd`. Delayed reports are supposed to exceed the network capacity and so, they are not substituted before making sure that the network is not congested.

Generally, the TICP collector sends requests upon the receipt of reports (i.e., report-clocked transmission of requests). But, there are also other moments at which the collector can send requests, if its window allows. Indeed, TICP implements a timer for the purpose of report loss detection and calculation. When this timer expires, the collector checks (as above) if the congestion window `cwnd` allows to probe new sources, and if so, new (or retransmitted) requests are emitted. We explain in the next section this timer, which is an important component of TICP congestion and error control.

III.8. Congestion detection mechanism

This mechanism forms an important part of our protocol. It can be designed in different ways. We choose to build it upon a timer.

The mechanism serves four different purposes: (i) to set the deadline for reports and to distinguish those which are timely from those which are delayed. (ii) to decide if a report (or a request) is lost or not. (iii) to slide the left-hand side of the congestion window. Finally, (iv) to trigger the transmission of new requests, in the same way the arrivals of reports do.

III.8.1. Round-trip time estimator: We want 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. The timer mainly serves to decide when it is safe to consider an expected report as being lost. It sets a deadline by which a report should arrive at the collector if the network is running in good conditions. This allows to decide whether the network is congested or not by simply computing the loss rate of reports expected to arrive between the scheduling of the timer and its expiration.

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 [15]. The difference from TCP is that in our case, the round-trip time varies due to the presence of different sources with different paths to the collector, whereas in the case of TCP, the round-trip time mainly varies due to the variations of queuing time in routers. The source clustering principle is introduced into TICP for the purpose of smoothing the round-trip time variations caused by the difference in paths, and hence for making the round-trip time estimation effective.

TICP 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 collector timestamps the requests and the sources echo the timestamps in their reports. The collector 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 collector 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 ← srtt + 4.rttvar` . The coefficients of the estimator are taken equal to those of TCP retransmission timer, which have proven their effectiveness 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. `srtt` can be set to the first round-trip time measurement, and `rttvar` to half this measurement. The three variables can also be set to their values in past collection sessions.

III.8.2. 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.

III.8.3. Detecting network congestion: The idea is to compute the loss rate of reports expected to arrive during a time window equal to `TO`. The collector 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 happens in the following way.

When the timer is scheduled, the collector saves 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 timely reports received between the scheduling of the timer and its expiration. The collector makes the assumption that `(torecv - recv)` reports were lost in the network. It estimates the loss rate of reports to `1 - recv/torecv`. The network is considered as congested if the loss rate exceeds a certain threshold `CT` (Congestion Threshold). This triggers a division of the congestion window 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.

Set as above, `CT` is equal to `RS/cwnd` for large congestion windows, 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 at least one packet is lost, and severely congested when all packets are lost or delayed (i.e., they arrive after the expiration of the timer).[4] A TCP packet corresponds in our case to `RS` reports. With these values 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 further this issue in Section V. 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).

III.8.4. Timely vs. delayed reports: A timely report is a report received before its deadline. The deadline for the receipt of a report is given by the timer. A report not received before its deadline is assumed to be lost. If it arrives later than the deadline, it is considered to be delayed. A delayed report is used to update the round-trip time. However, it is not used

[4]We have in mind the new versions of TCP that do not necessarily timeout when less then three duplicate ACKs are received in a window, see [2].

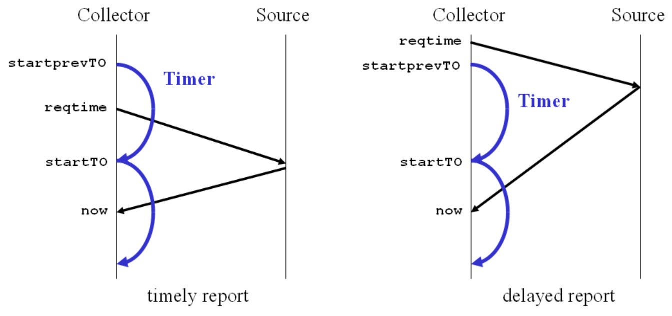


Fig. 1. The two types of reports

to increase the congestion window, nor to change the variable `pipe` (number of expected reports).

The content of a delayed report is considered and added to the list of received information. The collector does not ask a source that has sent a delayed report to resend it in subsequent rounds. A delayed report is only different from congestion control point of view.

We explain now how the deadline of a report is set. This explanation is illustrated in Figure 1. Let `startTO` be the scheduling time of the timer. Let `startprevTO` be the previous scheduling time of the timer. When a report is received, the collector extracts from its header the timestamp echoed by the source, which indicates the time by which the corresponding request has been issued. Denote this time by `reqtime`. The report is received on time if and only if `startprevTO < reqtime`. The report is delayed in the opposite case. In other words, a report is timely 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, `startprevTO` is set to `startTO` and `startTO` is set to the current time.

III.8.5. Sliding the left-hand side of the window when the timer expires: In Section III, we explained how the left-hand side of the window slides when timely reports arrive. This sliding is realized by decrementing the variable `pipe` by the number of timely reports. But, the variable `pipe` has also to be decremented when reports are concluded by the collector to be lost, otherwise we end up with a situation where `pipe` overestimates the real number of reports in the network, which drains out the network and blocks the protocol.

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

III.8.6. Updating the variable `torecv`*:* This variable indicates the number of reports to be received between the scheduling of a timer and its expiration. The collector expects to receive by the expiration of a timer all what have been sent during the active time of the last timer. Therefore, to update `torecv`, we need to introduce a new variable, which is the number of requests sent by the collector between the scheduling of the last timer and its expiration. Denote this latter variable by `sent`. When the timer expires, the collector sets `torecv` to `sent` and resets `sent` to 0.

III.8.7. Updating the variable `sent`*:* This variable indicates the number of requests sent between `startprevTO` and `startTO` (previous timer active time), and to be received before the current timer scheduled at time `startTO` expires. It is incremented every time new requests are transmitted. And it is reset to zero when the timer expires.

The variable `sent` has also to be decremented when a report arrives on time. For this kind of reports, we have two distinct cases: (i) `startprevTO` $<$ `reqtime` $\leq$ `startTO` (request sent during the previous timer active time), and (ii) `startTO` $<$ `reqtime` (request sent during the current timer). In the first case, `sent` is not decremented. In the second case, it is decremented by 1. This decrease is necessary since a good report satisfying (ii) must not be included in the number of reports to receive after the expiration of the current timer scheduled at `startTO`.

IV. 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 VI.

The collector starts the collection session by sending one request packet of size `RS` (that probes `RS` sources). It sets its variables as follows,

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

The collector 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 collector proceeds into the adaptation of its congestion window and the transmission of new requests. The congestion window is adapted if the report is arriving before its deadline. Requests are transmitted for all reports, if the window size allows.

```
if the report arrives on time
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 }

For both delayed and non delayed reports {
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 TO
```

The algorithm stops when all reports are received, or when the duration of the session exceeds some allocated time.

V. Friendliness of our protocol with TCP traffic

It is very important for a new transport protocol like TICP to share fairly the network resources with TCP [8], [18]. In particular, we want our protocol to fully utilize the available resources when it is operating alone in the network, but to back-off when there is a concurrent TCP traffic. At long time scale and for the same network conditions, TCP and TICP should obtain almost the same throughput. Note that the same reasons that make TICP friendly with TCP, make the TICP sessions friendly with each other. We illustrate this friendliness in the sequel and we validate it in the next section with simulations.

TCP-friendliness is to be verified under the same network conditions for both TICP and TCP. Let us consider a scenario where TICP collects information from a cluster of sources located behind the same bottleneck and having the same path characteristics to the collector. The concurrent TCP traffic is also assumed to cross the same bottleneck and to have the same path characteristics. We study two cases based on which direction the TCP traffic flows through the bottleneck.

The first case is when the congestion appears in the forward direction of the bottleneck. The congestion is caused by the TICP requests and the data packets of one concurrent TCP connection. The network on the return path is not congested. Both the TICP session and the TCP connection experience the same network conditions and react in the same way. They increase their congestion window at the same rate in the absence of congestion (linearly during Congestion Avoidance by roughly one packet every round-trip time), and they divide it by two when one or more packets are dropped on the forward path (Section III). Indeed, the loss of one or more request packets results in a report loss rate larger than `RS/cwnd`, which triggers TICP congestion detection mechanism and consequently, TICP window division by two. Our protocol achieves then the same throughput on the forward path as that of the competing TCP connection in terms of packets/s. 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.

We study now the case of a TCP traffic running in the reverse direction. The congestion on the reverse path is caused by the TICP reports and the data packets of one concurrent TCP connection. The forward path is not congested. The TICP collector 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. If the total size of `RS` reports is equal to that of a TCP data packet, the throughput of reports on the reverse path in bits/s becomes comparable to that of the competing TCP connection. If we want the throughput of reports on the reverse path to be on the order of the total throughput of `N` TCP connections, we need to set `RS` to `N` times the size of a TCP data packet divided by the size of a report.

Clearly, `RS` is an important parameter that decides the TCP-friendliness of our protocol. For example, let us define TCP-friendliness as realizing a throughput *equal* to that of a single concurrent TCP connection experiencing the same network path properties. 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 set so that the total size of `RS` reports is equal to that of one TCP data packet. The simulation results presented in the next section validate this choice.

VI. Validation of the protocol by simulation

We implement our protocol in the network simulator `ns-2` [14] and validate its performance under different scenarios. Our objective is to prove the effectiveness of TICP in controlling the congestion of the network and in fairly sharing the available resources with the competing traffic.

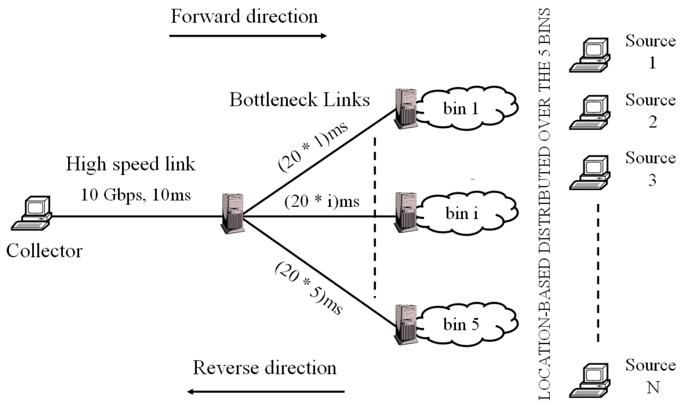


Fig. 2. Simulation testbed

VI.1. Simulation setup

We consider different simulation scenarios built over the network topology in Figure 2. All scenarios have in common the fact that one (sometimes two) TICP collector (located at Collector) probes a large number of sources (thousands) spread over 5 bins, or clusters, based on their network location. All sources of a bin are located behind the same bottleneck and share the same path properties when communicating with the collector. The collector joins the sources by Centralized Multicast [14], [7] where each cluster of sources has its own IP multicast address. The collector is connected to the Internet via a high speed link of 10 Gbps and 10ms one-way propagation delay. The 5 bins are connected to the Internet via low speed links that form the bottlenecks for both requests and reports. Bin i, $i = 1, \ldots, 5$, is connected to the Internet via a bottleneck link b_i of $1150 - 150.i$ kbps. The round-trip time (excluding queuing delay) between the collector and sources in bin i is set to $2.(10+20.i) = 20+40.i$ 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 each of the five bottleneck links are set to 20 packets and are of DropTail type. The sources have IDs ranging from 1 to N, where N is the total number of sources. The ID of a source matches its order in the location-based ranked list of sources. Bin 1 is the closest to the collector and contains sources with the smallest IDs, whereas bin 5 is the farthest and contains sources with the largest IDs.

We run four sets of simulations. The first set corresponds to a TICP session running alone in the network. In the second set, a TICP session shares the 5 bottlenecks with UDP traffic; we consider the two cases of UDP/CBR and UDP/Poisson. The objective of these first two sets is to show how well our protocol avoids network congestion and how efficiently it uses the available bandwidth. In the third set, a TICP session shares the 5 bottlenecks with TCP NewReno connections. The TCP connections run on the different bottleneck links and transfer each an infinite amount of data. They have a large receiver advertised window and packets of 1000 bytes. In the fourth set, two TICP sessions share the 5 bottlenecks, both sessions run in the same direction. The objective of the third and fourth sets of simulations is to illustrate the friendliness of TICP towards TCP traffic and other TICP sessions.

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

VI.2. Scenario without competing traffic

We consider a single TICP session that collects information from 10000 sources, where each 2000 sources are clustered in one bin. We run our protocol until all the information from sources is well received. First, we make the congestion appear in the forward direction of each bottleneck link 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. It shows the throughput of TICP requests at the collector. Figure 4 corresponds to the case where congestion is on the reverse path. It shows the throughput of TICP reports, also at the collector. The throughput is computed by averaging the number of bits transmitted over 1 second time intervals. We observe in the figures a descending staircase behavior, which is the result of the collector probing consecutively the bins from the closest one (bin 1) to the farthest one (bin 5). The peak at the end of the figures corresponds to probing rounds where lost reports are retransmitted.

It is clear how TICP adapts its probing rate to the bandwidth capacity in each direction and does not overload the buffers. This is reflected by the duration of the session, which is very close to its ideal duration 127 seconds, i.e. the duration achieved when the utilization of each bottleneck link is 100% and no packets are lost.

VI.3. Scenario with competing traffic

We run three sets of simulations to study the efficiency of our protocol, its TCP-friendliness, and its intra-protocol fairness (i.e., fairness among TICP sessions). First, we study the case where TICP shares the network with UDP and TCP traffic. Then, we consider the case of 2 TICP sessions running together in the network.

VI.3.1. TICP with UDP traffic: In this section, we run a single TICP session that collects information from 10000 sources, while a UDP traffic is running over all bottleneck links. We consider two scenarios, one running constant bit rate (CBR) traffic over UDP and another running Poisson traffic over UDP. In both cases, we set the average size of UDP packets to 1000 bytes and the average rate of UDP traffic on

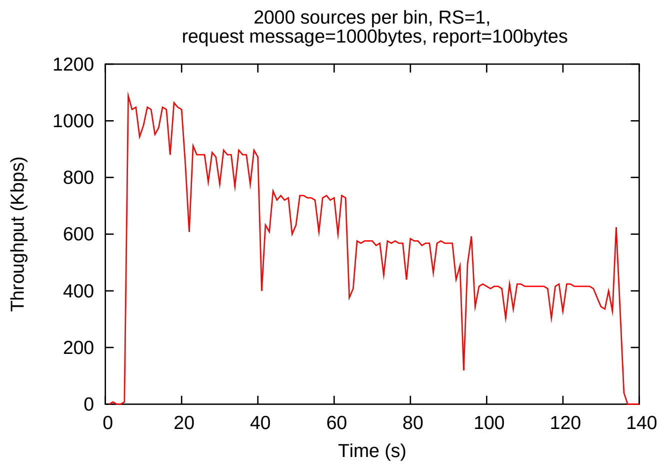


Fig. 3. One TICP session, congestion in the forward direction

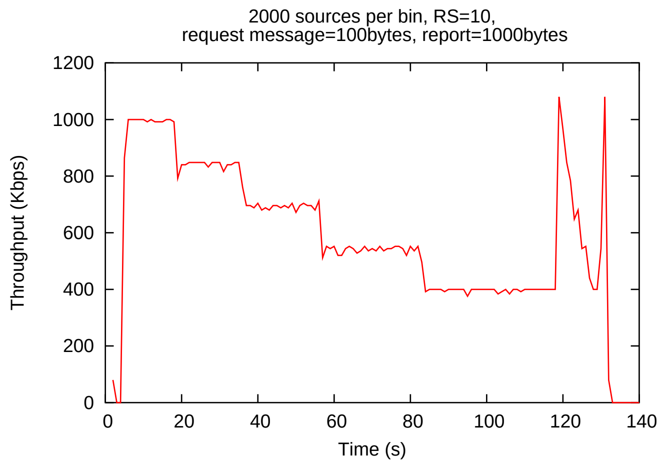
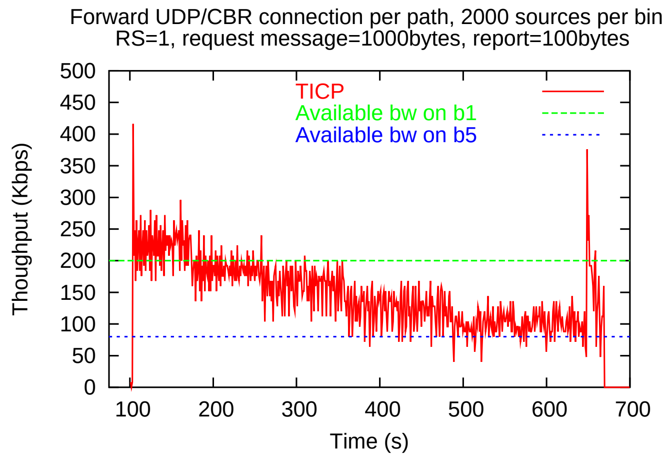


Fig. 4. One TICP session, congestion in the reverse direction

each bottleneck to 80% of its capacity, i.e., the rate of UDP traffic is 800 kbps on b_1, 680 kbps on b_2, 560 kbps on b_3, 440 kbps on b_4, and 320 kbps on b_5.

First, we launch the UDP traffic in the forward direction of each bottleneck link. We set RS to 1, the request message size to 1000 bytes, and the size of reports to a small value 100 bytes so that to remove any congestion from the reverse path. The UDP traffic starts at time 0, the TICP session starts at time 100 seconds. We plot in Figure 5 the throughput of TICP requests during the collection session for both UDP/CBR and UDP/Poisson background traffic. We can observe how well TICP adapts its probing rate to the available bandwidth at each bottleneck link, which in our case is equal to 20% of its total capacity. The staircase behavior still exists but is less pronounced due to the presence of the exogenous traffic.

Next, we launch UDP traffic in the reverse direction of each bottleneck link together with setting RS to 10, the size of reports to 1000 bytes, and the request message size to a small value 100 bytes. With these values we are sure to remove any congestion from the forward direction. The UDP traffic starts



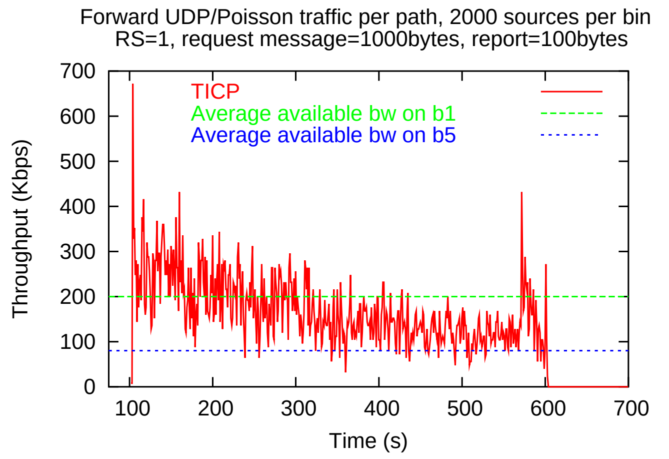
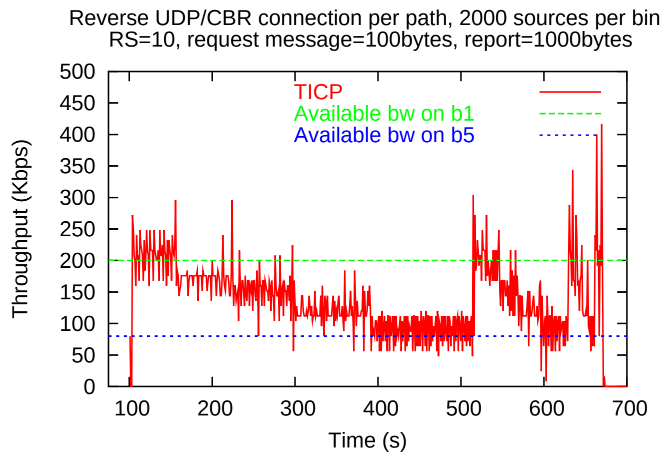


Fig. 5. One TICP session, congestion in the forward direction



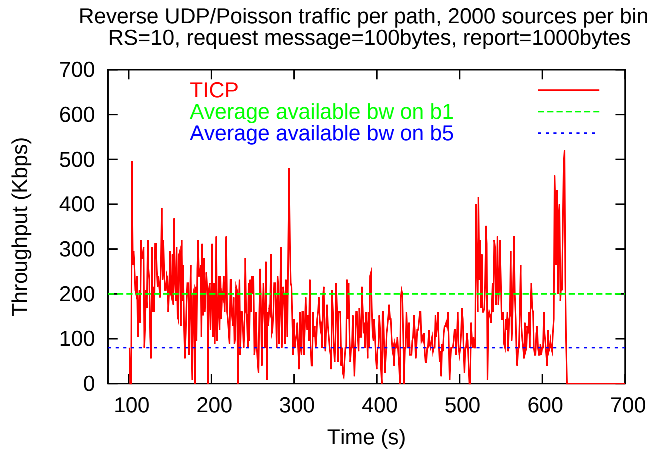


Fig. 6. One TICP session, congestion in the reverse direction

at time 0, the TICP session starts at time 100 seconds. Figure 6 plots the throughput of TICP reports. It shows how TICP can adapt the rate of collected reports to the available bandwidth at each bottleneck link. As we can verify in the figure, reports are gathered at a rate around 200 kbps from b_1, 170 kbps from b_2, 140 kbps from b_3, 110 kbps from b_4, and 80 kbps from b_5. We notice how more than two probing rounds are necessary to entirely collect reports.

The efficiency of TICP in controlling the congestion of the network can be seen from the duration of the session, which is around its ideal value 635 seconds. This is duration achieved if the available bandwidth of each bottleneck link was fully utilized and no packets were lost.

VI.3.2. TCP-friendliness of TICP: We consider one TICP session that collects information from 10000 sources and that shares the network resources with TCP connections having the same path properties (i.e., they share the same bottlenecks and have the same round-trip times). First, we run one TCP connection per bottleneck link in the forward direction. Second, we consider the same TCP connections but this time in the reverse direction. We want to check the TCP-friendliness of our protocol for both requests and reports.

For the forward direction case, we set RS to 1 and the request message size to 1000 bytes, which means that request packets have the same size as 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 paths. All TCP connections start at time 0. The TICP session starts at time 100 seconds. Figure 7 plots the throughput of the TICP requests averaged over 1 second time intervals and measured at the collector. The figure also plots the throughput of the two TCP connections running on links b_1 and b_5. The TCP throughput on the other links is not plotted for clarity of the presentation. We can observe how the arrival of the TICP probes into a cluster does not harm too much the existing TCP traffic. At the same time, the TICP session is not penalized by TCP. Both protocols manage to share fairly the network resources. When the TICP session ends, the TCP traffic increases its throughput again to fully utilize the available resources.

For the case of TCP traffic 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 paths. We plot in Figure 8 the throughput of TCP data packets and that of TICP reports, averaged over 1 second time intervals. We notice how our protocol is more aggressive than TCP. This is because the product of RS and report size is much larger than TCP data packet size (see discussion in Section V). The flow of reports behaves approximately as 10 long-lived TCP connections. TCP-friendliness of TICP can be improved by

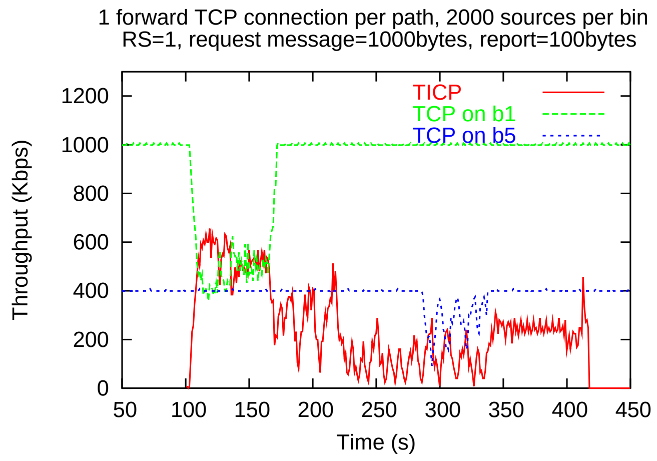


Fig. 7. One TICP session and one long-lived TCP connection per path, congestion in the forward direction

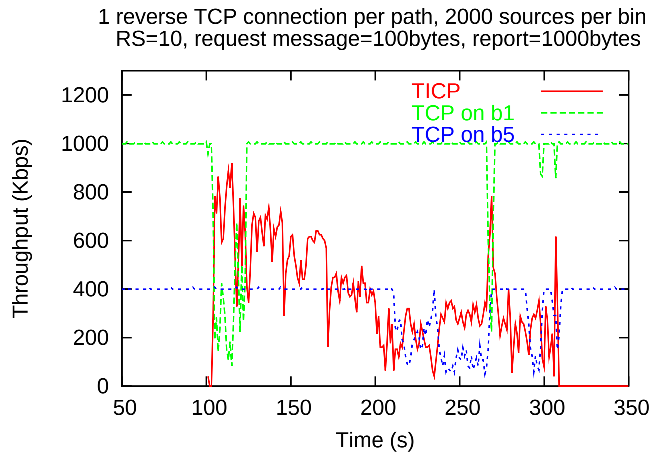
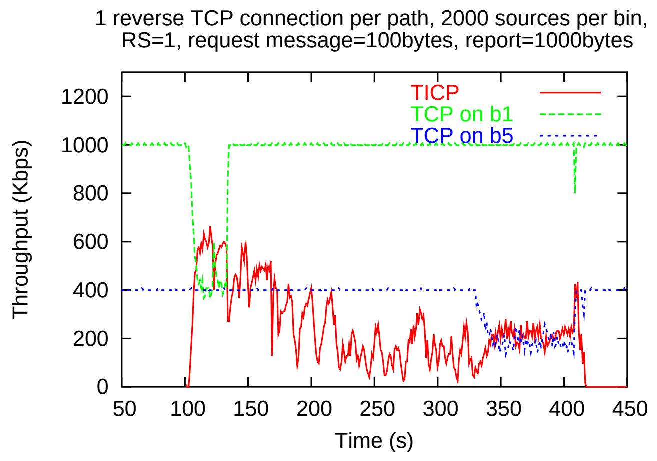


Fig. 8. One TICP session and one long-lived TCP connection per path, congestion in the reverse direction

reducing RS or the size of reports. 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 the TCP packet size of 1000 bytes. To prove that, we rerun the same simulation with RS equal to 1 and report size equal to 1000 bytes. We also rerun it for RS equal to 10 and report size equal to 100 bytes. In both cases, the product of RS and report size is equal to TCP packet size. Figure 9 shows the results 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 has to be collected from sources.

VI.3.3. Intra-protocol fairness of TICP: Here, we launch two TICP sessions that collects each reports from 10000 sources; every 2000 sources are clustered in one of the 5 bins. Both sessions have their collectors at Collector (see Figure 2). The first session starts at time 0. The second one starts at time 5 seconds. As before, we first allow congestion to appear in the forward direction by setting the size of request messages



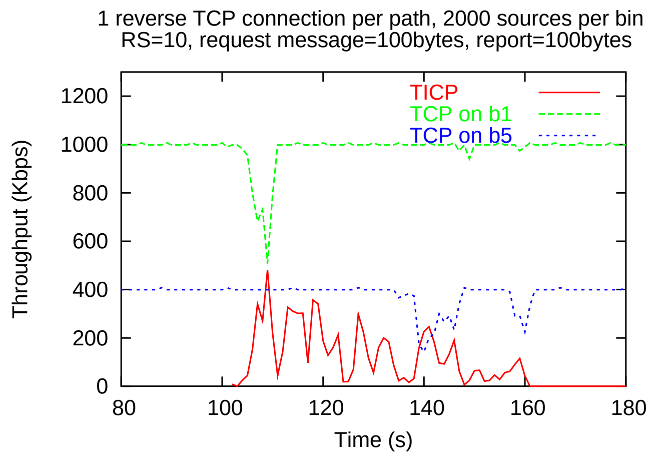


Fig. 9. One TICP session and one long-lived TCP connection per path, congestion in the reverse direction, better TCP-friendliness

to 1000 bytes and the size of reports to 100 bytes, then we move congestion to the reverse direction 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 paths, we plot the throughput of requests of both sessions measured at the collector and averaged over 1 second time intervals. When congestion is on the reverse paths, we plot the throughput of reports of both sessions. This gives rise to Figures 10 and 11. We can observe that before the 5th second, the first session fully utilizes the link bandwidth. When the second session arrives, the bandwidth utilized by the first session is divided by two, and the second session consumes the other half. Then, the first session moves to collecting information from the second bin leaving the bandwidth available at the first bottleneck to the second session. This behavior is repeated on each bottleneck link until the end of the sessions.

VII. Conclusions

We present in this paper TICP, a Transport Information Collection Protocol. A collector running TICP is able to collect the entire information from a large number of sources spread over the Internet. TICP provides a reliable data collection

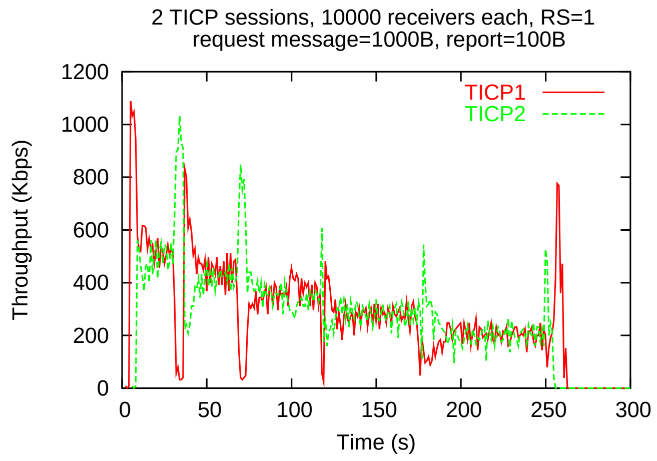


Fig. 10. Two TICP sessions, congestion in the forward direction

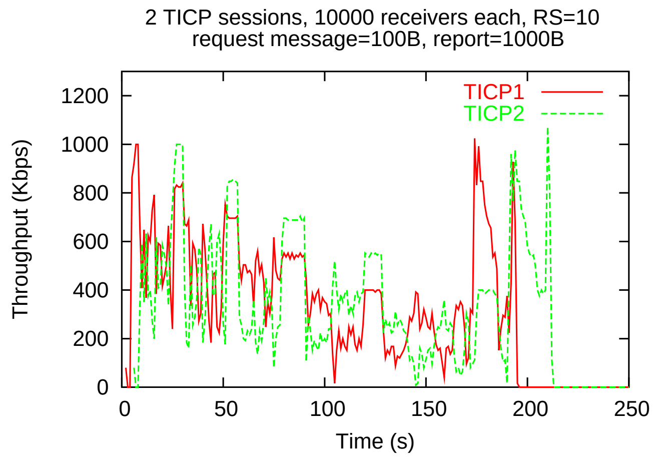


Fig. 11. Two TICP sessions, congestion in the reverse direction

service while controlling the congestion of the network and ensuring fairness towards other sessions running TICP, or other flows using the TCP protocol.

Our work on TICP can be extended in different directions. One extension is to consider the collection of large reports that cannot fit into one packet. Security is also an important issue in TICP. Sources must be sure that they are receiving request messages from the collector of the session, and the collector must be sure that the reports received arrive from the right sources. Malicious messages and reports alter the operation of TICP, in addition to corrupting the data to be passed to the application. Finally, we are intending to implement TICP and test its performance on a real network testbed.

References

[1] Allman (M.), Paxson (V.), Stevens (W.), TCP Congestion Control. *RFC 2581*, April 1999.

[2] Allman (M.), Balakrishnan (H.), Floyd (S.), Enhancing TCP's Loss Recovery Using Limited Transmit. *RFC 3042*, January 2001.

[3] Alouf (S.), Altman (E.), Nain (P.), Optimal on-line estimation of the size of a dynamic multicast group. *IEEE INFOCOM*, June 2002.

[4] Alouf (S.), Altman (E.), Barakat (C.), Nain (P.), Estimating Membership in a Multicast Session. *in proceedings of ACM SIGMETRICS*, June 2003.

[5] Bolot (J-C.), Turletti (T.), Wakeman (I.), Scalable feedback control for multicast video distribution in the Internet. *ACM SIGCOMM*, vol. 24, no 4, october 1994, pp. 58-67.

[6] Calvert (K. L.), Griffioen (J.), Mullins (B.), Sehgal (A.), Wen (S.), Concast: Design and Implementation of an Active Network Service. *IEEE Journal on Selected Area in Communications (JSAC)*, vol. 19, no. 3, March 2001.

[7] Estern (D.) et al., Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. *RFC 2362*, June 1998.

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

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

[10] Friedman (T.), Towsley (D.), Multicast session membership size estimation. *IEEE INFOCOM*, March 1999.

[11] Jacobson (V.), Congestion avoidance and control. *ACM SIGCOMM*, August 1988.

[12] Lin (J.C.), Paul (S.), RMTP: A Reliable Multicast Transport Protocol. *IEEE INFOCOM*, March 1996.

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

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

[15] Paxson (V.), Allman (M.), Computing TCP's Retransmission Timer. *RFC 2988*, November 2000.

[16] Ratnasamy (S.), Handly (M.), Karp (R.), Shenker (S.), Topologically-Aware Overlay Construction and Server Selection. *IEEE Infocom*, June 2002.

[17] Stann (F.), Heidemann (J.), RMST: Reliable Data Transport in Sensor Networks. *IEEE International Workshop on Sensor Net Protocols and Applications (SNPA)*, May 2003.

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

[19] Wan (C.), Campbell (A.), Krishnahmurthy (L.), PSFQ: A Reliable Transport Mechanism for Wireless Sensor Networks. *ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.