Quality-of-Experience driven Acceleration of Thin Client Connections

Mayutan Arumaithurai, Jan Seedorf, Maurizio Dusi
NEC Laboratories Europe
Kurfuerstenanlage 36
Heidelberg
Germany
mayutan.arumaithurai@neclab.eu
jan.seedorf@neclab.eu
maurizio.dusi@neclab.eu

Edo Monticelli, Renato Lo Cigno
University of Trento
Via Belenzani 12
Trento
Italy
edo.monticelli@studenti.unitn.it
locigno@disi.unitn.it

***Abstract*—Thin-client based solutions allow users to connect to remote servers and access content that is running on these servers within a virtual PC. With the advent of cloud based solutions, thin-client deployments running on remote Data-centers are increasingly popular. Unfortunately, since the traffic has to traverse through the Internet, issues such as latency, packet drops, and potentially congestion are introduced, which in turn affect the Quality of Experience (QoE) for the user. Our objective is to provide preferential treatment to certain thin-client flows over the rest of the thin-client flows traversing the same intermediate node, based on the application that the user is using at a given point in time. This is not straightforward, as it is not easy to identify what actual application is running inside a given thin client session, given that thin client protocols essentially only send bitmaps of the desktop to the client and in addition are often encrypting traffic. Assuming that some statistical mechanisms for application identification for thin client connections exists, the challenge we address is how to exploit this information for QoE-driven scheduling.**

We present a scheme that allows the prioritization of thin-client flows that are serving delay-sensitive applications, i.e., prioritizing flows based on the specific QoE requirements of the *dynamically changing* individual applications running within *persistent* thin client traffic flows. Our solution is essentially a hybrid scheduling scheme that takes into account the dynamically changing delay and bandwidth requirements of *inner* (i.e., tunneled in the thin client protocol) applications to prioritize flows that are close to an application-dependent QoE threshold. Our evaluation based on a prototype implementation reveals that our algorithm is indeed effective in dynamically prioritizing persistent thin-client flows based on the dynamically changing *inner* applications running within the flows.

I. Introduction

Thin-client solutions allow users to connect to remote services and access content with a virtual PC as if they had physical access to the remote machine. Thin-client protocols, such as Microsoft *Remote Desktop Protocol (RDP)* and Citrix *High Definition user eXperience (HDX)*, allow graphical displays to be virtualized and served across a network to a client device, while application logic is executed on the server (e.g., editing documents or running multimedia applications).

Thin-client solutions were initially designed for LAN environments, where bandwidth and delays between users and their Virtual PCs are generally not an issue. But with the advent of cloud based solutions, thin-client deployments running on remote Data-centers is increasing in popularity. Unfortunately, since in this case the traffic has to traverse through a WAN, issues such as latency, packet drops, and potentially congestion are introduced, which in turn affect the Quality of Experience (QoE) for the user. As users perform interactive real-time activities through thin-client protocols such as watching video, VoIP, or web browsing, the responsiveness of the network becomes a crucial parameter to determine the QoE perceived by end-users when interacting remotely with their virtual PC. Moreover, the presence of one or more bottlenecks might have varying effects to different applications, e.g., the effect on RTT due to congestion might be tolerable for a user reading a document via the thin-client, but probably unacceptable for a user watching multimedia content.

We target the following (realistic) scenario: Multiple (possibly encrypted) thin client connections run through a middlebox. The applications running in individual flows (and correspondingly the QoE expectation and bandwidth requirements for the flows) as well as the RTT of each flow change *dynamically* over time. We assume that some sort of (most likely statistical) per-flow application identification is available to the middlebox (see [1]), such that it can estimate at each point in time the type of application running within each thin client flow. Knowing the kind of application that users are currently running on their virtual PC is necessary to determine the individual QoE that they perceive given the actual network conditions, and allows solutions such as preferential treatment based scheduling. However, the detection of the application running inside a thin-client protocol is a challenging task [1]: Thin-client connections are usually encrypted and run on the same port (rendering pattern-matching or port-based application identification unfeasible). Further, the data exchanged within a thin-client session consists only of video bitmaps of what appears on the remote screen, and does not follow

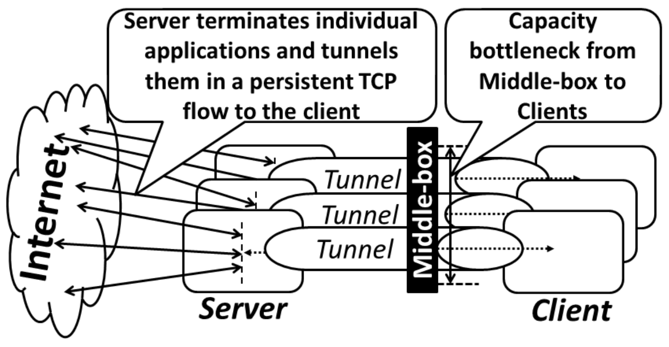

Server terminates individual applications and tunnels them in a persistent TCP flow to the client
Capacity bottleneck from Middle-box to Clients
Internet
Tunnel
Tunnel
Tunnel
Middle-box
Server
Client

Fig. 1. Targeted Scenario of Thin-Client Flow Scheduling

the request-response scheme of the specific protocol that the running application is using. Finally, most thin-client protocols are closed source, making it hard to use their properties to infer the current application running inside a connection. Still, machine-learning techniques exist that feature a low false-positive rate in classifying application types running in thin client sessions [1]. We assume the availability of such mechanisms for our work.

Our objective is to design a QoE-driven, scalable per-flow scheduling at the middlebox by periodically applying a scheduling technique that takes into account the dynamically changing delay and bandwidth requirements of *inner* applications. The overall goal is to provide good QoE for multimedia content running in thin client sessions, by dynamically providing preferential treatment to certain flows over the rest of the thin-client flows traversing the same intermediate node (based on the dynamically changing QoE requirements and delay of the flows). One core challenge is thus the design of a scheme that balances the computation complexity of the scheduling algorithm with fast enough reaction to changing conditions within the network and the individual flows. We believe that this is the first work to propose an application identification based scheduling mechanism solution for thin-client flows.

Fig. 1 visualizes our general scenario: A middle-box can see tunneled flows only at the level of the tunneling thin client protocol (e.g., RDP) which is running over a transport protocol, e.g., TCP. Given that the middle-box can somehow estimate the higher-layer application running within each flow (which is dynamically changing within a persistent tunnel flow), it can apply scheduling on the transport layer in order to prioritize certain flows for a certain amount of time (depending on the dynamically changing application type running within a given tunnel flow). Note that the server terminates individual application flows to the Internet and tunnels the application into an existing thin client flow to the client. The middlebox needs to be placed before a downstream bottleneck (e.g., close to the access network of clients), and is assumed to handle in the order of hundreds thin client flows in parallel (as is realistic in current Virtual Desktop Infrastructure (VDI) deployments).

Our contribution is the design and evaluation of a scalable, per-flow middlebox scheduling scheme. Our solution enables to preferentially treat delay-sensitive thin-client connection compared to delay-insensitive ones. Our results demonstrate, that in cases where there is congestion on the downstream path, our approach results in significantly increased QoE at users. Section II describes our solution and proposed scheduling algorithm, followed by an overview of our prototype implementation in Section III. We present evaluation results of our approach in Section IV, discuss related work in Section V, and conclude the paper with a summary in Section VI.

II. A Solution Based on Delay Budgets

A. Assumptions and Requirements

Our solution assumes that a middlebox has some means of *application identification* for thin-client flows, i.e., it can estimate (with a certain probability) what actual type of application is currently running inside a persistent thin client flow at a given point in time (e.g. video streaming vs. webbrowsing). Our prototype (see Section III) integrates the algorithms developed in [1], which leverage machine learning techniques to identify the type of application running in a thin client flow based on IP-level features (e.g., packet size or packet inter-arrival times). However, our approach, which is described in detail in this section is orthogonal to the specific type of application identification mechanism used by the middlebox, as long as at each point in time the mechanism can provide an estimate with a reasonably low false positive rate.

B. Thin-client Scheduling

In order to provide preferential treatment to different thin-client flows traversing the same bottlenecks, we propose the use of a scheduling scheme, which is essentially a deadline-based solution based on a) per-flow Quality-of-Experience (QoE) threshold (changing dynamically as users change applications they execute within a thin client session), b) per-flow RTT (changing dynamically due to network conditions and congestion, see Section III for details on how the RTT is measured from a middlebox), c) per-flow bandwidth requirements of each flow (changing dynamically as users change applications their execute within a thin client session). In short, the scheduling component consists of frequently calculating the current *delay budget* available for each flow which is the difference between the QoE-threshold (i.e., the maximum tolerable RTT for the identified application currently running in the flow) and the flow's current RTT. The lower the delay-budget, the higher should be its priority. Therefore, in order to use our scheme, we need to have a means of estimating the current application running inside a thin client flow, the corresponding bandwidth requirements, and its RTT. Thin clients use protcols such as RDP to transmit data and thereby send a bitmap of the image of the screen, therefore even if a user is using multiple applications at the same time, the most delay-sensitive application takes precedence and the RDP flow accordingly adjusts its sending/refresh rate. E.g. if a user is watching Youtube and reading a word document, the refresh rate for Youtube is higher and therefore the rate at which RDP sends the image of the screen is proportional to the more delay-sensitive Youtube flow.

We can derive the following requirements for our target scenario and envisioned solution: 1) Scalability: It must be able to handle a large number of flows, 2) Effectiveness/Efficiency: It must provide a good throughput and good QoE for thin client users in scenarios with some congestion, ensuring that no flows are starved, 3) Complexity: Since the scheduling needs to be performed at line rates, it is essential that the computational complexity is kept low, 4) Flexibility: The scheduler must adapt to changing applications within flows, a varying number of flows, and varying network conditions.

C. Scheduling Algorithm

Our goal is to exploit the knowledge of the individual application that runs in each RDP flow (as identified e.g. with machine learning) as well as the measured RTT and bandwidth requirements per flow for treating flows differently: Given that for each flow we a) can identify the application and b) measure its current RTT, we want to prioritize flows depending on how close they are to a *QoE threshold* (expressed as a fixed RTT value for each application type). In previous work, we have derived concrete values for such QoE thresholds based on extensive experiments with users [1]: 50 ms for video and 100 ms for data applications. Note that these QoE thresholds have been derived for (and are therefore specific) to thin client connections running over RDP (taking into account how users perceive the effect of delay on bitmap delivery, depending on individual applications executed).

Accordingly, assuming we have a set $F = f_1, f_2, ..., f_n$ of n flows that run through the envisioned scheduler, we define the *Delay Budget* (DB) for each flow at time t_j as follows:

$$DB(f_i, t_j) = QT(f_i, t_j) - RTT(f_i, t_j) \tag{1}$$

where $RTT(f_i, t_j)$ is the measured RTT and $QT(f_i, t_j)$ is the QoE threshold based on the identified application for flow f_i at time t_j. At any time our scheduler thus knows the Delay Budget for each flow. Intuitively, we would like to apply *Earliest Deadline First (EDF)* scheduling based on the Delay Budget to our problem. However, regular EDF is not directly applicable to our scenario for two reasons: First, when resource utilization is higher than 100% (as in our target scenario), EDF cannot guarantee that some flows will not starve [2]. Second, a per-packet computation of the Delay Budget is computationally not feasible and does not scale [3] [4].

Hence, we developed *QoE Driven thin client Scheduling (QDS)*, a combination of different queuing approaches (i.e., Weighted Fair Queuing (WFQ), Class Based Queuing (CBC), and Earliest Deadline First (EDF)) as follows. Flows get classified into m classes $C = c_1, c_2, ..., c_m$ (in general we assume less classes than flows, i.e., $m < n$) depending on their Delay Budget. Each class $c_k \in C$ has a maximum Delay Budget, $DB_{max}(c_k)$; classes are strictly ordered such that $k < h \rightarrow DB_{max}(c_k) < DB_{max}(c_h)$. Periodically, each flow f_i gets allocated to the class c_k with the lowest $DB_{max}(c_k)$ that is higher than $DB(f_i, t_j)$. We say $f_i \in c_k(t_j)$ if a flow f_i has been allocated to class c_k at time t_j.

In addition, we want to assign the outgoing bandwidth for queues depending on the bandwidth requirements of the flows allocated to them. Thus, assume further that —to large extent depending on its individual application type— each flow f_i has a certain bandwidth requirement. For instance, video streaming might require x bit/s downstream bandwidth whereas web browsing might only require y bit/s downstream bandwidth (where $x > y$). Our scheduler frequently measures and averages bandwidth requirements per flow and thus known at any time t_j the average bandwidth requirements for each flow $BR(f_i, t_j)$. Each class c_k then periodically gets assigned a weight $w(c_k, t_j)$ which is calculated based on the Delay Budgets of the flows in that class ($DB(f_i, t_j)$) and on the bandwidth requirements of the flows in that class ($BR(f_i, t_j)$) for all $f_i \in c_k(t_j)$. For each class there is a scheduling queue, and the outgoing bandwidth of each queue is proportional to the weight of its class.

If we want to schedule purely based on the Delay Budget, a simple algorithm would assign weights such that each class c_k would periodically get assigned a weight $w(c_k, t_j)$ which is proportional to the number of flows currently allocated to that class and inversely proportional to the Delay Budgets of all individual flows (counting negative DBs as zero):

$$w(c_k, t_j) = \sum_{f_i \in c_k(t_j)} 1 - \frac{max\left(0, DB\left(f_i, t_j\right)\right)}{\sum_{h=1}^{n} max\left(0, DB\left(f_h, t_j\right)\right)} \tag{2}$$

However, different application types may vary significantly with respect to their bandwidth requirements (e.g., a video flow might demand a bitrate of 5 Mbit/s were as a web-browsing application might only need 1 Mbit/s). Therefore, a more sophisticated algorithm assigns weights to classes not only based on the Delay Budgets of flows in the class, but in addition according to the bandwidth requirements of the individual flows. We propose the following algorithm, where α and β are configuration parameters that steer how much to weight the Bandwidth Requirements of flows and how much to prioritize flows with lower Delay Budget:

$$w(c_k, t_j) = \left(\sum_{f_i \in c_k(t_j)} \beta \times \frac{BR\left(f_i, t_j\right)}{min_{f_h \in F}\left(BR\left(f_h, t_j\right)\right)}\right) \times l\left(c_k\right) \tag{3}$$

where

$$l\left(c_k\right) = \begin{cases} 1 & if\ k = \left\lceil \frac{m}{2} \right\rceil \\ 1 + \alpha \times \left(\left\lceil \frac{m}{2} \right\rceil - k\right) & otherwise \end{cases} \tag{4}$$

Weights are then normalized over all weights (note that this normalization would also apply when using equation (2)):

$$w_{norm}(c_k, t_j) = \frac{w(c_k, t_j)}{\sum_{i=1,\ldots,m} w(c_i, t_j)} \tag{5}$$

For each class there is a scheduling queue, and the normalized weights are multiplied with the overall capacity of the congested downstream link to assign the outgoing bandwidth (e.g., in $Mbps$) of each outgoing queue proportionally to the weight of its class.

The overall rationale behind our approach is as follows: Flows get allocated outgoing bandwidth proportionally to their bandwidth requirements (tunable with β), and flows get proportionally higher priority the closer they are to their QoE threshold (i.e. inversely proportional to their Delay Budget, tunable with α; this is conceptionally close to the idea of EDF). This is achieved by weighting (similar to WFQ). The concept of classes avoids to have a single scheduling queue per flow by grouping flows with similar Delay Budget together (similar to CBQ). Finally, by computing Delay Budgets, Bandwidth Requirements, and the weights of classes/queues periodically, the algorithm introduces acceptable computational overhead.

Key to the weight calculation and the assignment of weights to queues is to find the right frequency of the execution, i.e., balance the tradeoff between highly frequent computation and high accuracy with the overall load of the scheduling algorithm and scalability with number of flows to handle. Here lies the advantage of our approach: By emulating an EDF-style scheduling algorithm (based on the Delay Budget concept) with an adaptation and combination of class-based queuing and weighted fair queuing, it allows to execute an earliest deadline first scheme in intervals and not on a per-packet level. Nonetheless, the scheduling algorithm needs to be executed more often than the flows are expected to change their general properties in order to be effective. For instance, in our experimental evaluation (see Section IV), we execute the computation of weights and (re-)allocation of flows to classes periodically every $100ms$. Changes of flow properties within this interval (e.g. due to a user changing the application within a flow) are only detected (and handled) from the next algorithm execution point onward. This is not problematic because during this short timeframe a respective flow is not starved; the corresponding queue just gets a suboptimal outgoing bandwidth allocated for a short period of time.

To avoid that small fluctuations in consecutively measured RTT of a flow will potentially result in a flow being re-allocated to queuing classes very frequently (as a fluctuating RTT will result in a fluctuating Delay Budget), the average RTT gets computed using a *sliding-window* approach (see also Section III): the RTT of a flow f_i at time t_j used in the algorithm, $RTT(f_i, t_j)$, is always the average of a fixed number of directly preceeding, consecutive single RTT measurements. By using such a sliding window, small functuations in singular RTT measurements get smoothed out. At the same time, large fluctuations or continuous small changes in the same direction will have the desired effect of a change in the Delay Budget that possibly leads to a re-allocation of the flow to a different queuing class. Similar to RTT, also the bandwidth requirements of flows get computed using a sliding window (see further Section III). The small disadvantage of this sliding-window approach to computing averages is that it may take slightly longer for significant changes in flow properties (e.g. due to a change of the application running inside the flow) to propagate into effective change in queue allocation. As discussed previously, this only results in a slightly suboptimal outgoing bandwidth allocation for a short period of time.

III. Prototype Implementation

A. *Machine Learning Integration into 'BlockMon' Framework*

We have implemented a prototype middlebox of our thin client acceleration approach for Microsoft's *Remote Desktop Protocol (RDP)*. Our middlebox integrates previous work that developed mature machine learning techniques for RDP traffic [1] based on the Weka machine learning libraries[1]. In essence, we re-use the previously offline *RDP-trained* database to classify flows according to the type of application running in the thin client flow. At each point in time the machine learning estimates for a given flow if it is of application type video (e.g., video streaming applications), audio (e.g., VoIP), or data (e.g., web-browsing) [1].

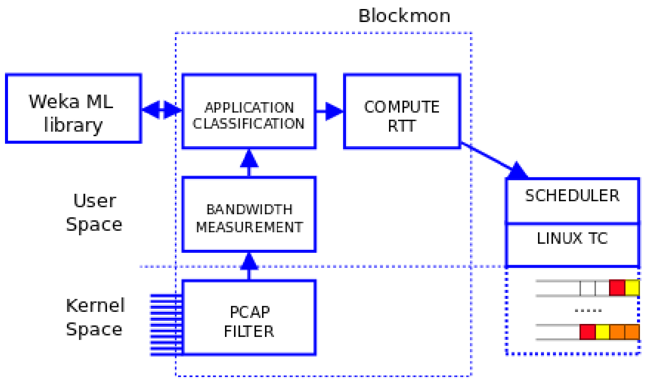


Fig. 2. Prototype of Thin-Client Acceleration Middlebox

In addition, we have integrated the machine-learning based application identification into our modular monitoring framework *BlockMon* [5]. This integration of all components into BlockMon improves speed in identification and classification of packets. Further, it enables a clean separation of components of our middlebox into BlockMon modules [5]. Figure 2 shows the overall architecture of our prototype and core BlockMon modules. The application identification also provides the bandwidth estimation we use in our algorithm (computing in a sliding-window the average bandwidth requirements per flow). Apart from the application identification, the PCAP filter (running in kernel space and obtaining packet headers) and the RTT computation (measuring the RTT of individual

[1]http://www.cs.waikato.ac.nz/ml/weka/

packets, and computing average RTT values per flow using a sliding window) are BlockMon modules we implemented for our prototype.

B. RTT Estimation

Measurement of RTT in a middlebox is not straightforward. The RTT module maintains a database for a sub-sample of packets that pass through it on a per flow basis and when the TCP ACK returns, it calculates the uni-directional RTT from the middlebox to the client. Similarly, it measures the RTT to the server and thus obtains combined RTT of the flow. In more detail, the RTT module for each flow maintains the last measured and the weighted average RTT. These values are updated everytime the pcap block captures an acknowledgment. The RTT measurement thus logically divides the (RDP client-server) communication into two halves, with respect to the point where it is run. The RTT is computed independently for the two halves and the complexive value is given by their addition: everytime one of the parts is updated, the overall value also is updated. To compute the RTT for one of the halves, whenever a TCP packet is captured, its sequence number and arrival time are stored in a list of 'unacked tcp packets'. Everytime a tcp acknowledgment is captured, the list of 'unacked tcp packets' in the opposite direction is searched for the corresponding sequence number. The RTT is then computed as the difference of the acknowledgment and the packet arrival times. Our actual implementation is more complex as it must also deal with a multitude of possible cases (connection initialization, termination, different window size, keepalive packets, and so forth).

Note that our RTT measurement algorithm only provides an estimation of the RTT: if the traffic is strongly asymmetric, i.e. the majority of packets are being sent in one direction, then one of the halves is updated much more frequently, possibly leading to inaccuracy of the estimated RTT. This is indeed the case with the RDP traffic we measure in our prototype, which upon a scarce user interaction generates much more traffic in the server to client direction than in the reverse one. To account for this asymmetric traffic, we measured the RTT close to the server in our experiments. A real-world solution —being likely deployed close to clients— would need to find a more sophisticated solution for coping with asymetric traffic when estimating RTT.

C. Scheduling Algorithm

The scheduling algorithm (as described in Section II) has been implemented in user space with the *Linux Traffic Control* API[2]. It receives per-flow measurements and application identification from the BlockMon modules. Our implementation makes use of the existing implementation of *Hierarchical Token Bucket (HTB)* (which provides basic functionalities such as class based queuing and allocating weights (i.e., bandwidth) to outgoing queues). The scheduler frequently calculates the delay budget for each flow, and —using HTB primitives— assigns flows to classes and computed weights to queues (according to equations (3), (4), (5)).

IV. Experimental Evaluation

A. Testbed Setup

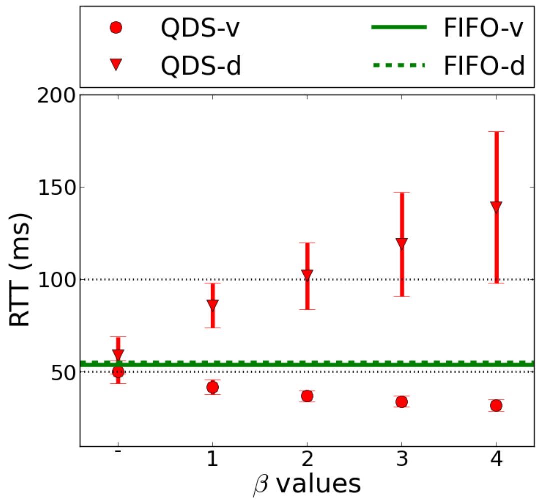
We set up a local testbed to emulate a scenario with a large number of thin-clients, their traffic passing through a bottleneck. Our testbed consists of our middlebox prototype placed on the path from several thin clients to multiple RDP servers (in principle as shown in Fig. 1). The RDP servers run Windows Server 2008; both the middlebox and client are standard Linux machines with Debian squeeze (stable). To emulate a WAN environment with realistic delays and congestion, we deployed another host between the middlebox and the clients running *netem*[3]. The netem device introduces a normally distributed random (initial) delay between 5-10 ms per flow for each run. N clients establish each a single RDP connection to one of $S = N/2$ servers (one server per two clients). Inside each RDP session runs an application of type either v (watching a *video* stream) or d (*data* application, i.e., viewing websites / editing documents), such that there are in total $N = V + D$ (denoted below as $[V : D]$) RDP flows, where V is the total number of video flows and D is the total number of data flows. The flows are set up such that the congestion caused is 20% above the overall allocated bandwidth from middlebox to clients (denoted with c). In each experiment, we started V 5 Mbit/s video flows and D 1 Mbit/s data flows in parallel and emulated an overall downstream capacity limit c from the middlebox to the clients. We emulated user behaviour by scripting the automatic execution of a video/data type of application with Windows login scripts. The bandwidth limit c is varied from 10 Mbit/s to 30 Mbit/s and each simulation run last 60s. In the case of different flow combinations — see Fig. 4(a) — the simulation runtime is longer, as each single application type lasts for 60s. For all our experiments, we executed 10 runs each for the same setting, and present below the average and standard deviation. The QDS scheduling uses $m = 10$ classes/queues (with $DB_{max}(c_i)$ increasing from 0 to $100ms$ in $10ms$ steps among the classes), and updates the weights of the queues every $100ms$.

B. Results

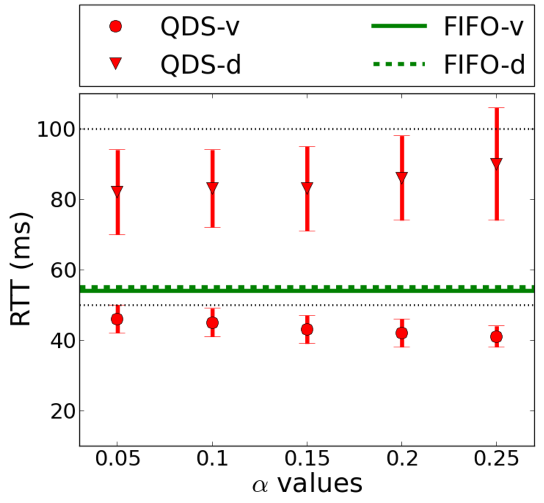
In the absence of any other reference algorithm directly suitable for our scenario —i.e., exploiting application identification for per-flow thin client scheduling (see further Section V)— we compare our solution, QDS, to FIFO. Any other reference algorithm would either need to use application identification for scheduling or could only perform very poorly with respect to treating flows differently according to their individual RTT (not knowing their expected QoE threshold). We

[2]http://linux.die.net/man/8/tc

[3]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

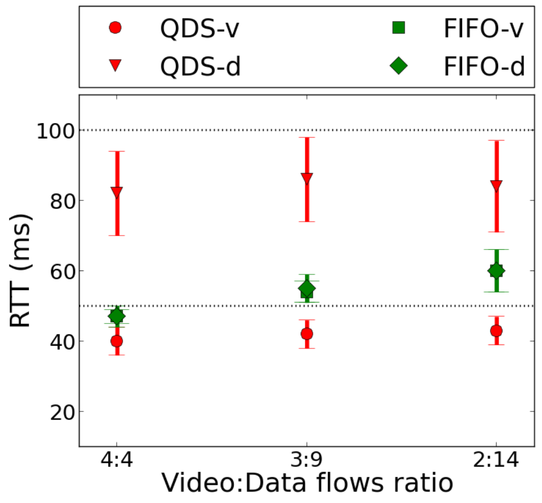


(a) α=0.2, c=20, Mbit/s,V:D=3 : 9

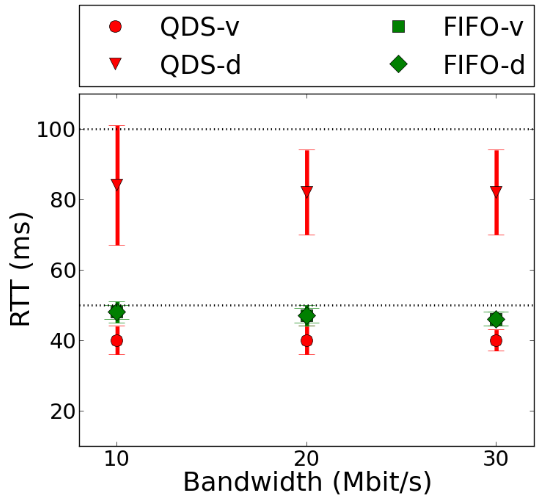


(b) β=1, c=20, Mbit/s, V:D=3 : 9

Fig. 3. Study of the impact of α and β for QDS.



(a) c=20, Mbit/s, α=0.2, β=1



(b) V=D, α=0.2, β=1

Fig. 4. Scalability of QDS for varying V:D ratio, bandwidth.

thus considered a comparison with other existing algorithms (besides FIFO) not fair for our target scenario.

Fig. 3 and Fig. 4 shows the average RTT (and standard deviation) we obtained —for video and data flows, respectively— for FIFO and QDS under different settings. Fig. 3(a) compares different values for β for a congested downstream link of $c = 20$ Mbit/s with 3 video flows and 9 data flows (3 : 9) for a fixed value of $\alpha = 0.2$. The first value on the x-axis of $\beta =$ ‘ $-$ ‘ indicates that bandwidth requirements were not used at all, i.e., in this case the weight for each class was only computed according to equation (4). The values for FIFO are displayed as green lines, essentially being the same for video and data as one would expect without application-dependent prioritization; dotted grey lines show the QoE threshold for video (50 ms) and data (100 ms). It can be observed that with increasing values of β, one can prioritize video over data flows more and more, possibly at the expense of data flows being over their QoE threshold (100 ms). This result shows that our algorithm effectively splits video and data flows into different queues (according to their delay budget), and in addition the allocation of weights to queues based on bandwidth requirements throttles the overall bandwidth that data flows can consume. In summary, $\beta = 1$ performs best, in the sense that this setting is able to “push” (compared to FIFO) the average RTT for video flows below the QoE threshold of 50 ms, while data flows are at the same time on average below their QoE threshold of 100 ms. Consequently, Fig. 3(b) compares different values for α for the same setting with a fixed value of $\beta = 1$. It can be seen that with increasing values of α, the average RTT for video flows becomes smaller, while the average RTT for data flows slightly increases. This is to be expected as with increasing α the flows with lower delay budget get prioritized more (i.e., mostly the video flows as they have a smaller QT than the data flows). Overall, in most of our experiments $\alpha = 0.2$ and $\beta = 1$ obtained the best tradeoff for getting both video and data flows below their QoE threshold.

Fig. 4(a) shows how our algorithm can adapt to variations of application types running in the flows for $c = 20\,\text{Mbit/s}$. We varied the ratio between video and data flows (i.e., $V : D$) over time to show that our algorithm can successfully maintain application-dependent QoE thresholds, independent of the individual applications running in flows. Note that the ratios $V : D$ are not chosen randomly, but such that the overall bandwidth requirements of the flows equal $1.2 \times c$, i.e., 20% congestion. E.g., 2 video flows at 5 Mbit/s and 14 data flows at 1 Mbit/s should cause the same congestion over a $c = 20\,\text{Mbit/s}$ downstream link as 4 video flows at 5 Mbit/s and 4 data flows at 1 Mbit/s. The results demonstrate that our algorithm is successful in achieving its objectives as application types running in the flows change dynamically. On the other hand, the latency for video (and data) flows increases in the case of FIFO as the total number of TCP flows increases and TCP dynamics aim for fair share among the flows. Fig. 4(b) shows how our solution scales up with the size of the capacity limit c: We ran the same number of video and data flows and varied the overall downstream bandwidth limit ($c = 10, 20, 30$). Again, we scaled the ratio of video and data flows according to c (i.e., 2:2, 4:4, and 6:6). The results demonstrate that our algorithm scales well with changing overall downstream capacity limit (for other V:D ratios we obtained similar results). Note that QDS performs better than FIFO and as observed in Fig. 4(a), because when the ratio of video to data changes and as the total number of flows increases, video flows experience higher latency.

In addition to the average RTT values depicted in Fig. 3 and Fig. 4, Table I displays the percentage of packets that arrive within the QoE threshold for the given application type, and the percentage of packets that arrive less than 20% late (i.e., within 60 ms for video flows and within 120 ms for data flows). Note that the QT values we use (obtained in [1]), do not constitute strict thresholds such that an RTT slightly over a given QT implies bad quality for the user. Instead, packets that arrive slightly after the QT will only have a small effect on QoE perceived by the user. The results show that QDS can significantly decrease the number of packets that arrive after the QoE threshold (compared to FIFO). Concretely, in a scenario with congestion (120% overall bitrate of flows vs. c), our approach can ensure that 80% (77%) of packets arrive within $1.2QT$ for $V : D = 3 : 9$ (respectively for $V : D = 2 : 14$).

Finally, we verified that our prototype implementation can schedule in the order of 100 parallel thin client flows with very low computational load on commodity hardware (i.e. a regular dual-core Linux host). We are thus confident that our solution scales computationally well for realistic real-world thin client deployments.

V. Related Work

There exists a vast amount of literature on various scheduling algorithms such as Weighted Fair Queuing (WFQ), Hierarchical Token Bucket (HTB), (Hierarchical) Packet Fair Queuing (PFQ) [6] [7], Deficit Round Robin (DRR) and many others. However, most scheduling algorithms are designed for scheduling a large amount of packets at line rate (e.g., on backbone routers). In contrary, our work aims at *per-flow* level scheduling (i.e., taking individual characteristics of each flow into account) which usually only scales up to the order of hundreds of flows and can therefore only be applied at access routers (or to a dedicated subset of all flows going through a router). Another distinctiveness of our approach is that we apply *deadline-based* scheduling: The current RTT and the (dynamically changing) QoE deadline of each flow is taken into account in scheduling. The majority of existing queuing mechanisms just ensures that a certain assigned rate is satisfied. Therefore, we will limit our state-of-the-art discussion to per flow, deadline-based scheduling approaches.

TABLE I. **% OF VIDEO (V), DATA (D) PACKETS THAT ARE BELOW A CERTAIN QOE THRESHOLD (QT) FOR DIFFERENT SCHEDULING ALGORITHMS (C [V:D]), α = 0.2, β = 1**

Scheduling	% of V $\leq$ QT	% of D $\leq$ QT	% of V $\leq$ QT+20%	% of D $\leq$ QT+20%
FIFO (20 [3:9])	29	100	68	100
QDS (20 [3:9])	68	73	80	80
FIFO (20 [2:14])	19	100	46	100
QDS (20 [2:14])	64	74	77	82

Several proposals have been made on using *Early Deadline First (EDF)* scheduling in networking. Usually, a delay budget is calculated for each flow based on the flow's RTT and the maximum RTT that a flow can have without affecting the user's QoE [8] [9]. The packets in the scheduler are served based on the increasing order of their deadline. The authors in [2] prove that in a deterministic setting, EDF is the optimal scheduling policy at a single switch. However, the computation complexity of EDF is very high [3], mostly since a search operation is required whenever a new packet arrives at the scheduler (for sorting packets according to their deadlines) [4]. To reduce the computational load of EDF, Liebeherr et al. propose *Rotating Priority Queue (RPQ)* [4]. This algorithm approximates EDF without requiring queued packets to be sorted. Incoming packets are assigned to a set of ordered FIFO queues based on their delay-budget, and queues rotate in sending packets such that queues with lower delay-budget flows get allocated more outgoing bandwidth. A downside of this approach is that it may result in packets being sent late or out of order due to residue packets that are left in a sending queue for a whole rotation cycle. This could have an adverse affect on the QoE for those flows.

Hierarchical Fair Service Curve based queuing [10] proposes a service curve based QoS model, which defines both delay and bandwidth based requirements of a class, to include fairness. The algorithm ensures that flows are serviced according to the assigned service curve. Though in principle this design is similar to our approach, our solution can provide preferential treatment to any flow that is running short on its delay budget. I.e if a data flow has a lower delay budget than a video flow, it will receive preferential treatment.

Finally, similar to our solution, there exist approaches (e.g. [11]) that dynamically adjust weights for queues based on packet arrival rates. However, it is important to realize that these approaches do not re-allocate flows to queues dynamically and do not re-compute weights based on a per-flow QoE threshold that changes dynamically. Instead, existing approaches are mostly based on a *DiffServ*-like model, where the allocation of flow to a queuing class and its QoE threshold is static over time.

In [1], we proposed statistical mechanisms for application identification within thin client connections. In this work, we use this application identification mechanism to provide preferential treatment to delay-sensitive flows, but the proposed solution is designed to work work with any application identification mechanism.

In summary, to the best of our knowledge, there exists no scheduling algorithm designed for the specific (but very relevant in practice) scenario that is the scope of our work: Thin client connections run through scheduler-middlebox where the application running in each flow (and correspondingly the QoE expectation and bandwidth requirements for the flow) as well as the RTT of each flow changes over time and has to be somehow re-estimated frequently and used in a timely, scalable manner by the scheduling algorithm. Prior work considers scenarios and solutions where either the application running in an individual flow is not changing over time (i.e., when the user starts a new application, a new TCP/UDP session is started), therefore flows are usually assigned to a certain class for their lifetime. Our contribution is to provide a solution for preferential treatment to certain delay-sensitive thin client flows by identifying applications in flows dynamically and frequently re-assigning flows to classes and weights to queues in a scalable deadline-based scheduler.

VI. Conclusion

In this paper, we presented the design and evaluation of a scheduling scheme for QoE-driven per-flow scheduling of thin client connections. Our approach aims at handling the dynamicity of application types running inside thin client sessions as well as changing network conditions in a scalable way. The results we obtained through our prototype implementation confirm that —in a scenario with downstream congestion— our solution is effective in allocating outgoing bandwidth to thin client connections depending on the QoE requirements of individual flows. Overall, our approach can ensure that a much higher percentage of users will experience good QoE for the specific applications they run in a virtual desktop infrastructure.

As future work, we intend to study our approach for thin client protocols besides RDP, and potentially also for other, non thin-client tunneling protocols (e.g. such as SSH). Further, we plan to validate the effectiveness of our solution in a larger variety of scenarios (e.g., with other types of applications running in flows and with different degrees of congestion).

Acknowledgement

This work has been partially supported by the mPlane project (mPlane: an Intelligent Measurement Plane for Future Network and Application Management), a research project supported by the European Commission under its 7th Framework Program (contract no. 318627). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the mPlane project or the European Commission.

References

[1] M. Dusi, S. Napolitano, S. Niccolini, and S. Longo, “A closer look at thin-client connections: statistical application identification for qoe detection,” *IEEE Communications Magazine*, vol. 50, no. 11, pp. 195–202, November 2012.

[2] L. Georgiadis, R. Guerin, and A. Parekh, “Optimal multiplexing on a single link: delay and buffer requirements,” *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1518 –1535, September 1997.

[3] V. Sivaraman and F. Chiussi, “Providing end-to-end statistical delay guarantees with earliest deadline first scheduling and per-hop traffic shaping,” in *In Proc. of IEEE INFOCOM 2000*, 2000, pp. 631–640.

[4] J. Liebeherr, D. Wrege, and D. Ferrari, “Exact admission control for networks with a bounded delay service,” *IEEE/ACM Transactions on Networking*, vol. 4, no. 6, pp. 885 –901, December 1996.

[5] A. di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi, “Blockmon: Toward high-speed composable network traffic measurement,” in *IEEE Infocom mini-Conf*, 2013.

[6] S. Golestani, “A self-clocked fair queueing scheme for broadband applications,” in *INFOCOM ’94. Networking for Global Communications., 13th Proceedings IEEE*, June 1994, pp. 636 –646 vol.2.

[7] J. C. R. Bennett and H. Zhang, “Hierarchical packet fair queueing algorithms,” in *IEEE/ACM Trans. Netw.*, 1997, pp. 143–156.

[8] D. Ferrari and D. C. Verma, “A scheme for real-time channel establishment in wide-area networks,” *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[9] D. C. Verma, H. Zhang, and D. Ferrari, “Guaranteeing delay jitter bounds in packet switching networks,” *TRICOMM*, pp. 35–46, April 1991.

[10] I. Stoica, H. Zhang, and T. S. E. Ng, “A hierarchical fair service curve algorithm for link-sharing, real-time and priority services,” in *ACM SIGCOMM*, 1997.

[11] F. Baker, S. Barbara, and Q. Ma, “Dynamic weighted resource sharing,” US PATENT 6,775,231 B1, April 2004.