

Experts Framework on the Prediction of Round Trip Time, to Improve TCP/IP Performance.

Project Final Report

Name	Student ID	Email
Bruno Astuto Arouche Nunes	3811226	bnunes@ucsc.edu

Introduction

Estimating the RTT (Round Trip Time) is of great importance in the context of performance of networking protocols based on TCP (Transmission Control Protocol). In a scenario where we have two communicating nodes (sender and receiver), the RTT quantity is defined as the amount of time the sender measures from the moment it sends the packet to the moment it receives the acknowledgement packet of that particular packet sent. Since TCP is a reliable protocol, the TCP sender must know if the packet just sent reached the destination correctly. This is done by the acknowledgement packet, called ACK from now on in this text, and it is sent by the receiver to confirm a successful transmission. The sender will timeout if it does not receive the ACK from the receiver within a given period of time that we will call here a timeout. If a timeout occurs, then the sender will assume that the packet not acknowledge was lost and it will retransmit it.

As said before TCP is a reliable protocol, and it takes care of controlling congestion in the network. It does that by expanding and shrinking the congestion window. The congestion window determines the number of bytes/packets that can be outstanding at any time. This is a means of stopping the link between two places from getting overloaded with too much traffic. The size of this window is calculated by estimating how much congestion there is between the two places. In our case, this estimation is done by looking at the RTT and setting a proper timeout. Once this size of the congestion window is calculated that is the maximum number of bytes that can be transmitted without acknowledgment that they have been received. Basically the size of the window, to a large degree, controls the speed of transmission as transmission pauses until there is acknowledgment.

This means that, if we set the sender timeout to be too large, we may wait a lot of time and some times don't receive an ACK, or wait too little and timeout before having the opportunity to receive the incoming ACK. In the first case we will lose time (or bandwidth) by just waiting idle, when we could be using this time with useful tasks, like retransmitting the packets not acknowledged. In the second case, we would lose time by retransmitting packets that were in fact successfully received by the receiver, but we did not wait enough time to receive the corresponding ACKs.

Objective

The main goal of this project is to implement algorithms from the Experts' Framework [1] and use them to predict the RTT of a TCP packet. The standard TCP RTT predictor uses the Jacobson's method to compute the estimation of RTT for the next transmission. This method will be described in the following sections.

The new RTT estimator proposed here, based on algorithms from the expert framework, will be compared against the estimator on the standard, based on two defined metrics. The results reported here show that the proposed algorithm can predict the RTT value on a much more accurate way and also spend less time transmitting the same amount of packets by setting more appropriate timeout values.

It is also our primary goal to study modifications in the shared algorithm and try to propose some new improvement. The modifications here concern basically the expert guesses of what the next RRT value would be and how to set these values.

Problem Description

The condition of the network will tell us how long should we wait for an ACK (the timeout value). A good estimation of this condition is the RTT value, which should be large for congested links and small otherwise. Thus, estimating accurately the RTT allow us to set "good" timeout values, decreasing our time (or bandwidth) loss. Here we define our loss as time wasted waiting for an acknowledgement or retransmitting a packet. Normally we would want to define this loss as bandwidth used in these two cases, but this would imply knowledge about the actual link an

available bandwidth used during the data collection to build the dataset used in this evaluation. For the sake of abstraction, we prefer to define our loss in terms of “seconds of data transfer” used. One “second of data transfer” is the amount of data transmitted or the bandwidth used by the system in one second. We are interested to measure the amount of “second of data transfer” spent waiting for a timeout and/or used in a retransmission. Using “second of data transfer” allows us to discuss network performance while avoiding bandwidth and transmission rates conversion factors.

We use the term “retransmission cost” to define to the total cost of retransmitting a packet. We measure this cost in seconds of data transfer, meaning that if retransmission consumes t seconds, we would be able to transmit as many bytes of new information as the available bandwidth allows us to transmit in t seconds.

Topics Considered

- Compare performance of the algorithms of interest under the original data set and the permuted data set. We expect to see improved accuracy when running the algorithms in the original dataset, given the correlation between RTT samples. This correlation is lost when permuting the data in time.
- Derivation of the weight's updates:
 - use of different updates = Loss Update x Mixing Update
 - use different mixing schemes [1], as Start Vector, Uniform Past and Decaying past;
 - impact of sharing [4] on the exponential updates;
- Tune model parameters: learning rate for Loss update and experts spacing;
 - Comparing Algorithms results with the results of using the standard TCP RTT predictor, Jacobson's algorithm.

Dataset

In the following we present the dataset used in this work. In order to get a dataset with real RTT measurements we used the tcpdump tool (a network packet capture tool) [5] to capture traffic of a

common TCP application (i.e. file transfer).

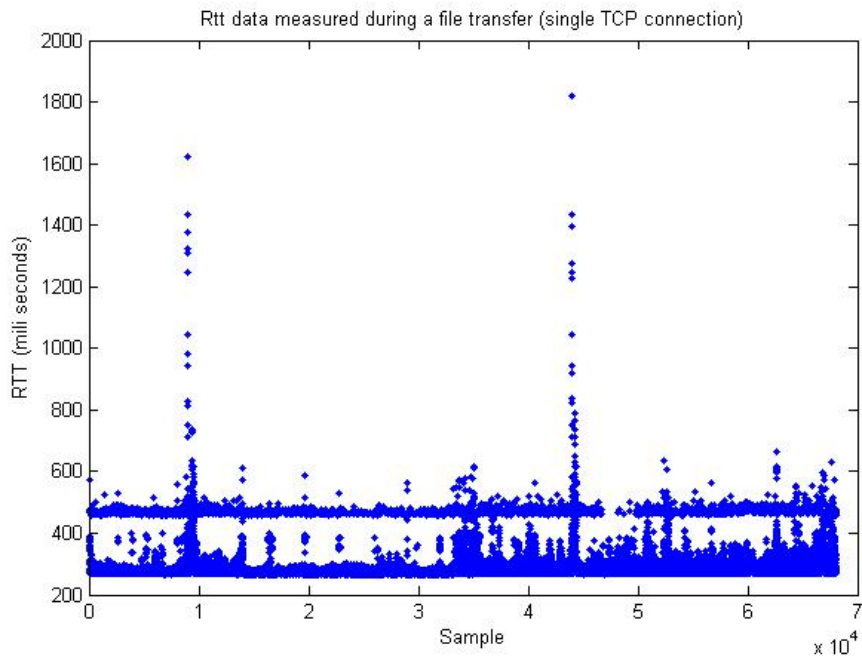


Figure 1: Dataset used in the experiments;

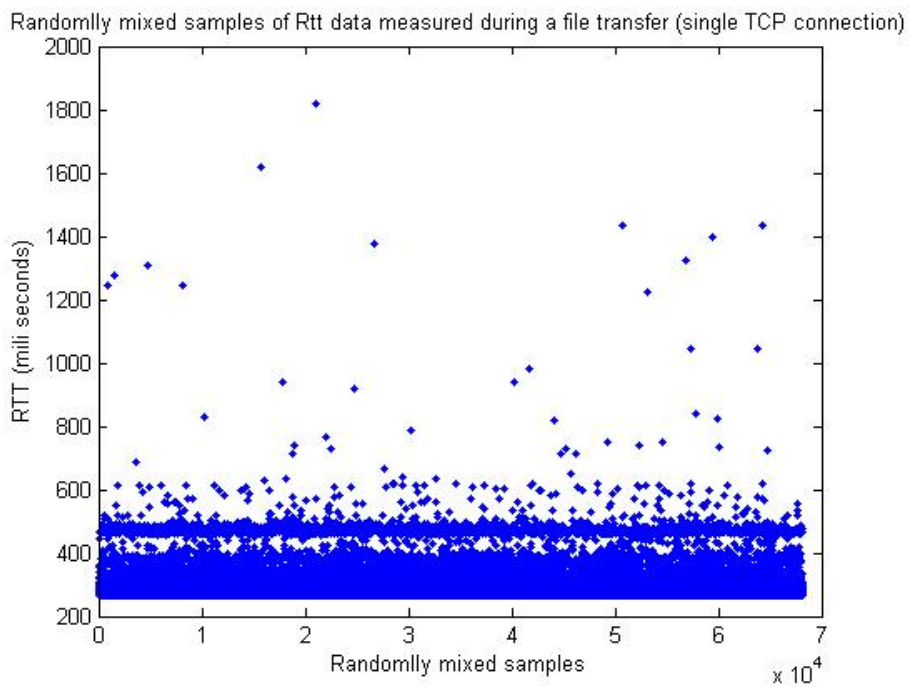


Figure 2: Randomized Dataset;

We used the tcpdump trace as an input to another software called tcptrace [6] which is a tool for analysis of tcpdump files. With tcptrace we were able to extract RTT values and build the data set.

The dataset generated contains approximately 70.000 RTT measurements. Figure 1 presents the plot of the dataset to be used. Figure 2 presents the same data permuted.

Standard TCP

The standard TCP implementation uses an adaptative scheme for setting the timeout value, according to an estimation of the RTT value of the next trial. This is done by averaging the RTTs measured over a number of segments. Two average schemes are possible, simple average and exponentiated average:

- Simple average: average the RTTs over a number of segments;
- Exponentiate average: later segments have more weight;

In this work we focus on the exponential average, which is used in the standard implementation of TCP for timeout estimation and is known as Jacobson's Algorithm:

Jacobson's Algorithm

The prediction in the standard TCP implementation is called Smoothed Round-Trip Time (srtt), is described in [2]. In Figure 3 we can see how the algorithm computes the srtt and Retransmission TimeOut (RTO) at trial k. The constant $\alpha = 7/8$ is the default value used in the standard implementation. Later, for the computation of the RTO at trial k the standard deviation is used to increase the timeout value a little bit, where the value of K is usually equal to 4. In this figure, RTT_k is the RTT value measured in trial k and used to compute srtt for trial k+1.

$$\begin{aligned}
 srtt_{k+1} &= \frac{7}{8}srtt_k + \frac{1}{8}RTT_k \\
 DevRTT_{k+1} &= \frac{3}{4}DevRTT_k + \frac{1}{4}|srtt_k - RTT_k| \\
 RTO_k &= \max(srtt_k + K \times DevRTT_k, MinRTO).
 \end{aligned}$$

Figure 3: Jacobson's algorithm for computing the Retransmission TimeOut (RTO) and the smoothed RTT (srtt = RTT prediction)

The algorithm gives greater weight to more recent values as shown by this expansion:

$$\text{srtt}(k+1) = (1-\alpha)\text{RTT}(k) + \alpha(1-\alpha)\text{RTT}(k-1) + \alpha^2(1-\alpha)\text{RTT}(k-2) + \dots + \alpha^{k-1}(1-\alpha)\text{RTT}(1)$$

α and $(1-\alpha) < 1$, so successive terms get smaller.

Therefore, the cost in time of using this algorithm is given as follows for trial t :

- Time spent with Jacobson's algorithm =
 - RTT_t , if $\text{RTT}_t < \text{RTO}_t$
 - $\text{RTO}_t + \text{re_tx_cost}_t$, if $\text{RTT}_t \geq \text{RTO}_t$;

where re_tx_cost_t is the time that costs for the system to retransmit a segment, and RTT_t is the measured RTT_t in the current trial.

For practical reasons, we used $\text{re_tx_cost}_t = \text{RTT}_{t-1}$. This means that the retransmission cost for trial t is equal to the RTT measured in the previous trial. This was done because we needed to guess what the re_tx_cost would be and we know that it is different from trial to trial, since it also depends on the channel current status and concurrent connections. Since we do not have this information, we decided to use a practical approach making $\text{re_tx_cost}_t = \text{RTT}_{t-1}$.

Optimal Algorithm

To be able to evaluate the algorithm we need to compare its results with a supposed optimal algorithm. To evaluate the accuracy of the timeout prediction we imagined an algorithm that “knows” exactly how to behave to get the optimal performance, by “looking” at the future and deciding what is the best action to take. The metric used, as mentioned before, is “second of data transfer” which is the amount of data transmitted by the system in one second. We define this as follows:

- Time spent with optimal algorithm =
 - RTT_t , if $RTT_t < re_tx_cost_t$;
 - $re_tx_cost_t$, if $RTT_t \geq re_tx_cost_t$;

RTT Last Trial

This algorithm is the simplest one can think of, that is to assume that the current RTT measurement will be the same as the previous trial. This was implemented so we could have a bad predictor to compare with.

- Time spent with Last-Trial algorithm =
 - RTT_t , if $RTT_t < RTT_{t-1}$;
 - $RTT_{t-1} + re_tx_cost_t$, if $RTT_t \geq RTT_{t-1}$;

Machine Learning Algorithms

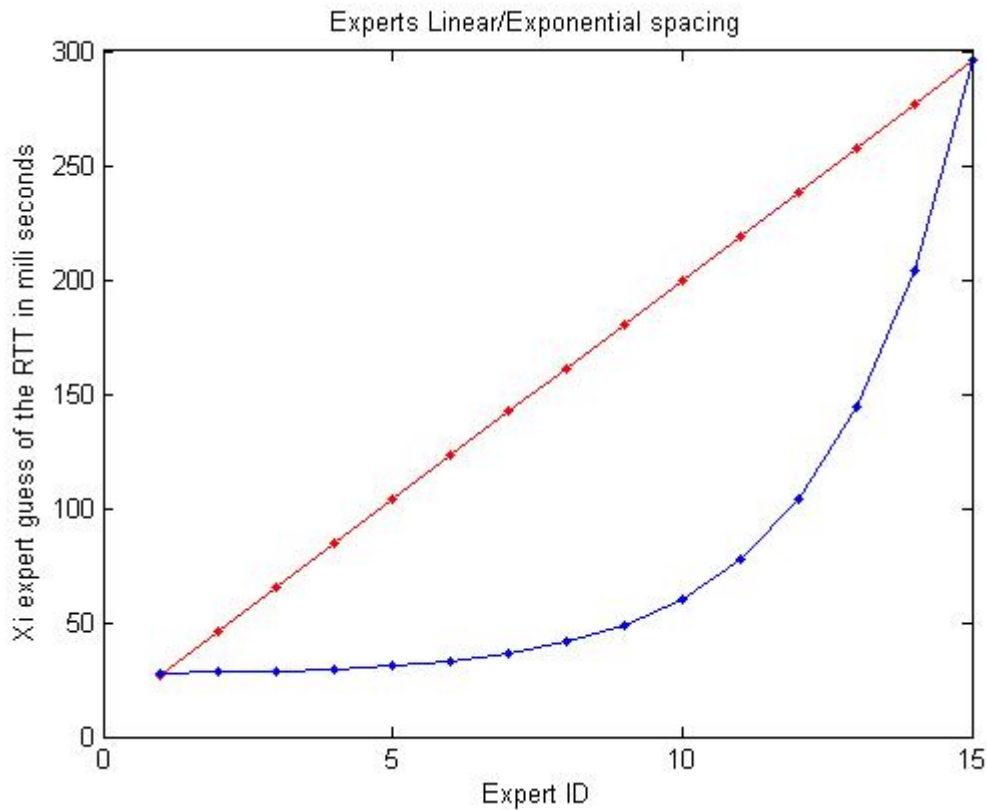
In this section we present the machine learning algorithms evaluated in this work, used to predict the RTT values of the dataset and compared with the standard TCP predictor. Here we describe 2 sets of algorithms inside the Experts Framework [1], the Share algorithm and the Mixing algorithms.

Experts Spacing

In this work we used 2 different sets of spacing between experts: Linear, Exponential.

- **Linear:** Experts are spaced according to a linear function, where expert X_n guesses that the RTT will be equal to the $re_tx_cost_t$ and the first expert X_1 guesses that RTT will be equal to the $0.1 * re_tx_cost_t$. Intermediate experts obey a linear function.
- **Exponential:** Same as Linear, but intermediate experts obey an exponential function.

Since $re_tx_cost_t$ changes for every t X_i must be recomputed every trial.



The figure above shows both expert spacing schemes used in the experiments presented in this work.

Sharing Update

Algorithms of this family receive as input a set of “experts”, other algorithms that make predictions. On each trial, each expert makes a prediction, and the share algorithm combines this prediction in a way that minimizes the error over a sequence of trials.

Here, we denote the predictions of the experts as X_i for $i=\{1, \dots, n\}$ where n is the maximum number of experts used by the algorithm. In our experiments we used $n=15$ experts. The weights of the experts, denoted by W_i , are initially set to $1/n$. We use $\text{Loss}(X_i)$ to denote the loss of expert i on a given trial.

The sharing algorithm works as follows on each trial:

1. time_out computed is equal to the weighted average of the experts guesses:

$$\text{time_out} = (\sum_i W_i X_i) / (\sum_i W_i)$$

2. Compute the Loss of each expert X_i :

- Time spent with Share algorithm = $\text{Time_Share}_t =$
 - RTT_t , if $\text{RTT}_t < \text{time_out}_t$;
 - $\text{time_out}_t + \text{re_tx_cost}_t$, if $\text{RTT}_t \geq \text{RTT}_{t-1}$;
- And the Loss is:
 - $\text{Loss}(X_i) = (\text{Time_Share}_t - \text{RTT}_t) / \text{re_tx_cost}_t$;

3. Updating the weights, reducing the weights of poorly performing experts:

$$W_i' = W_i * e^{-\eta \text{Loss}(X_i)}$$

4. Sharing the weights:

$$\text{pool} = \sum_i W_i' (1 - (1 - \alpha)^{\text{Loss}(X_i)}),$$

$$W_i'' = (1 - \alpha)^{\text{Loss}(X_i)} * W_i' + (1/n)\text{pool};$$

W_i'' = new weight for expert i at next trial.

Figure 4 shows the results for the performance over the data set for the optimal algorithm, the last trial, the standard TCP Jacobson and the Shared algorithm. We also tried to modify the shared algorithm actually, not to share at all. We see poor performance of the expert framework for negative learning rates. Figure 5 and 6 show the weights for the share algorithm for learning rates -10 and 10 respectively. We notice that for the negative learning rate, we have one predominant expert, while for the positive learning rate several experts contribute to the prediction. We believe that is the reason for the better performance of the positive learning rates.

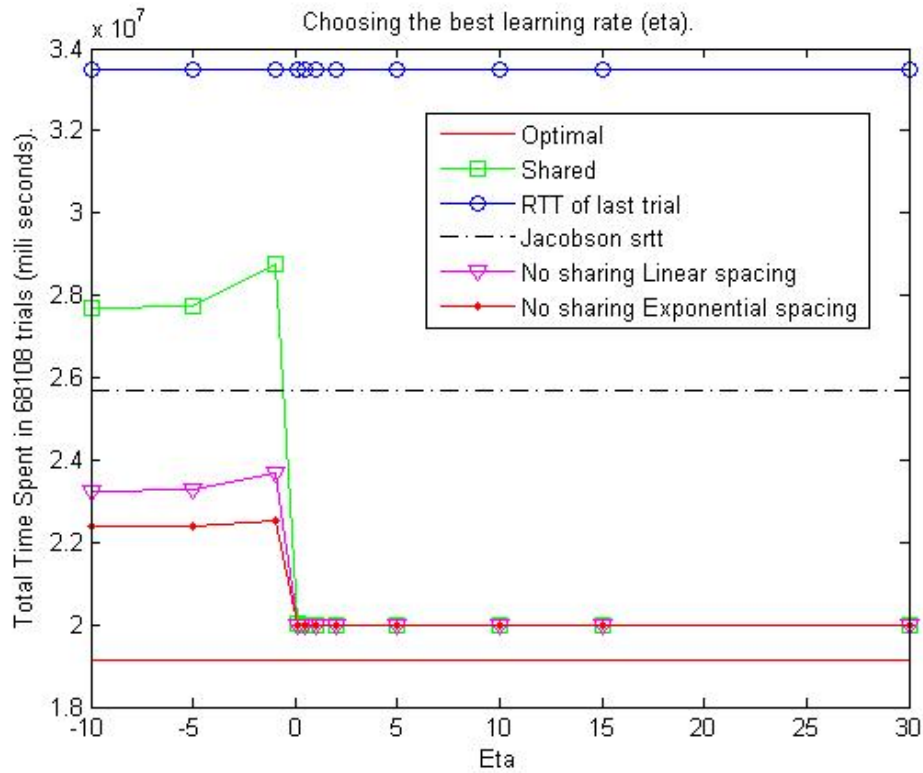


Figure 4: Results for Share and no-Share, against Last-RTT Jacobson and optimal;

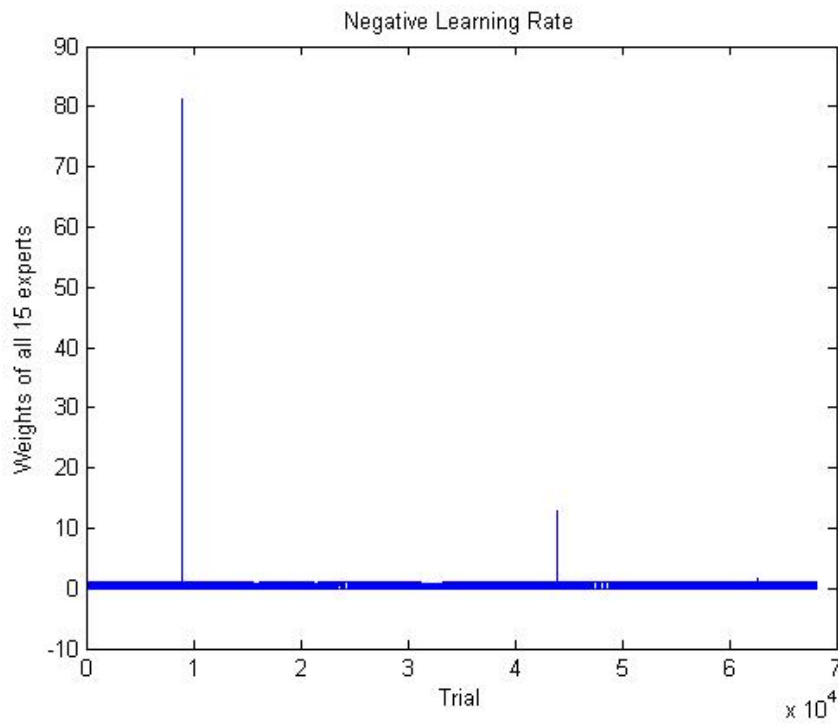


Figure 5: Negative Learning rates for shared algorithm linear expert spacing.

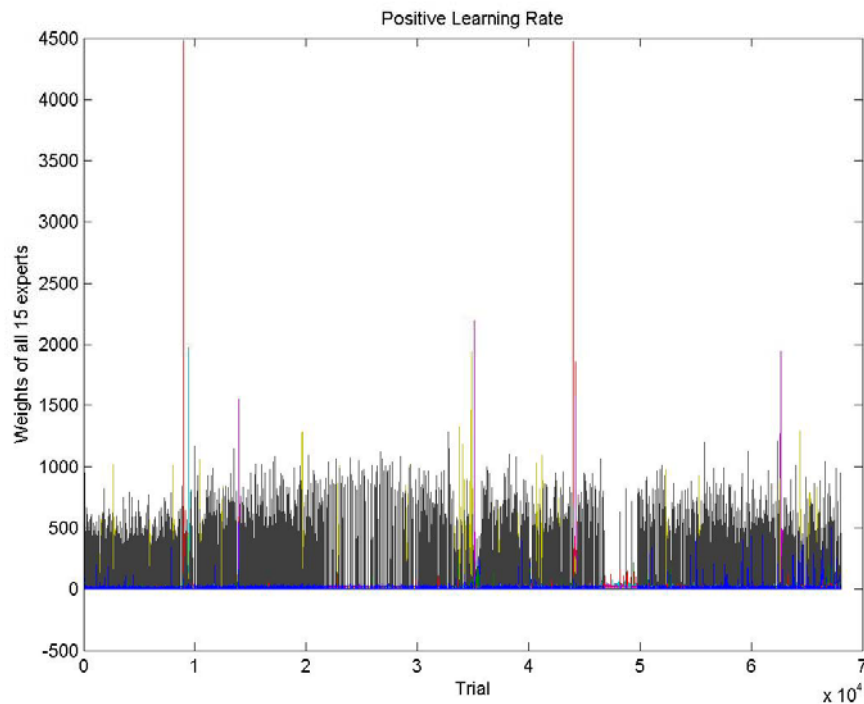


Figure 6: Positive Learning rates for shared algorithm linear expert spacing.

We notice that the share algorithms present much better performance than regular TCP with Jacobson's algorithm, for positive learning rates. We also can see from Figure 4 that when we don't use sharing at all, we have a slightly better performance. This can be seen in Figure 7, where we zoomed around $\eta=2$. We can now also see how exponential spacing allows a slightly gain of 731 mili seconds total around $\eta=2$.

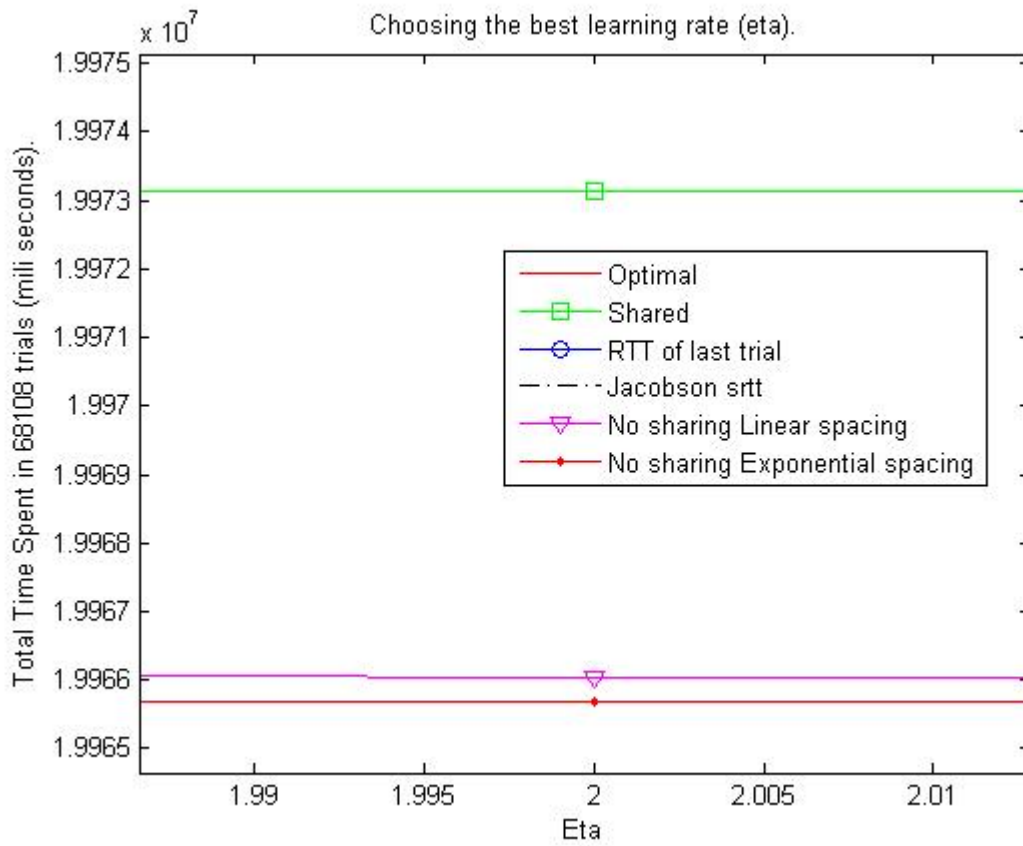


Figure 7: Figure 4 zoomed around eta=2, showing that no-sharing can perform better in our dataset.

If we compare Figure 1 with Figure 8, we can have an idea of how accurate is the prediction of the share algorithm for a learning rate of eta=2.

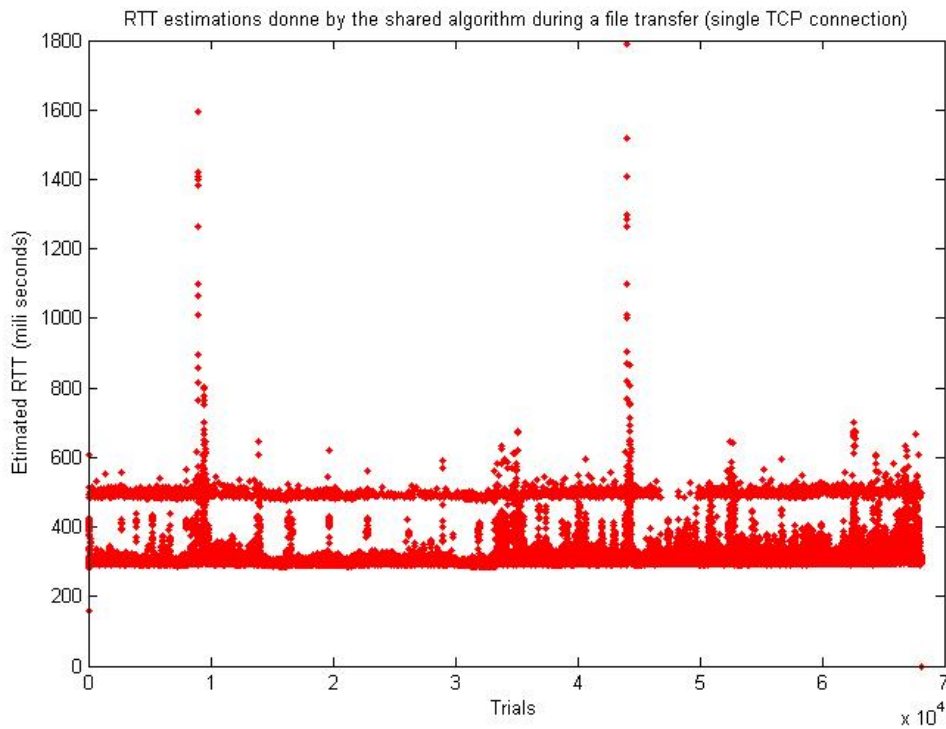


Figure 8: Predictions of the share algorithm (linear spacing) for the dataset presented at Figure1;

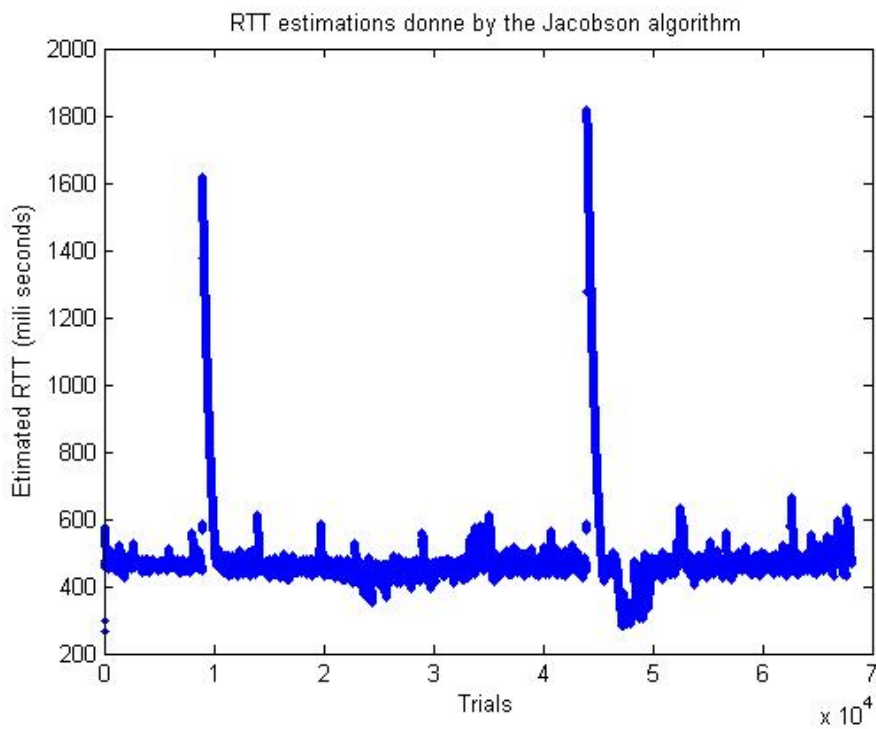


Figure 9: Predictions of Jacobson’s algorithm for the dataset presented at Figure1;

Mixing Update

For the mixing updates we used the same prediction function and loss function as we used in the sharing update algorithm, but now we changed the weights update to account for the past.

We implemented 3 different mixing schemes as described on [1]: Start Vector, Uniform Past and Decaying past. Figure 10 shows how the 3 different schemes mix the weights of previous trails for each expert.

Name	Coefficients
Static Experts	$\beta_{t+1}(t) = 1$ and $\beta_{t+1}(q) = 0$ for $0 \leq q < t$
Fixed-Share Update	$\beta_{t+1}(t) = 1 - \alpha$ and $\sum_{q=0}^{t-1} \beta_{t+1}(q) = \alpha$
<ul style="list-style-type: none"> • To Start Vector • To Past 	$\beta_{t+1}(0) = \alpha$
<ul style="list-style-type: none"> • Uniform Past • Decaying Past 	$\beta_{t+1}(q) = \alpha \frac{1}{t}$ for $0 \leq q < t$ $\beta_{t+1}(q) = \alpha \frac{1}{(t-q)^\gamma} \frac{1}{Z_t}$ for $0 \leq q < t$, with $Z_t = \sum_{q=0}^{t-1} \frac{1}{(t-q)^\gamma}$ and $\gamma \geq 0$

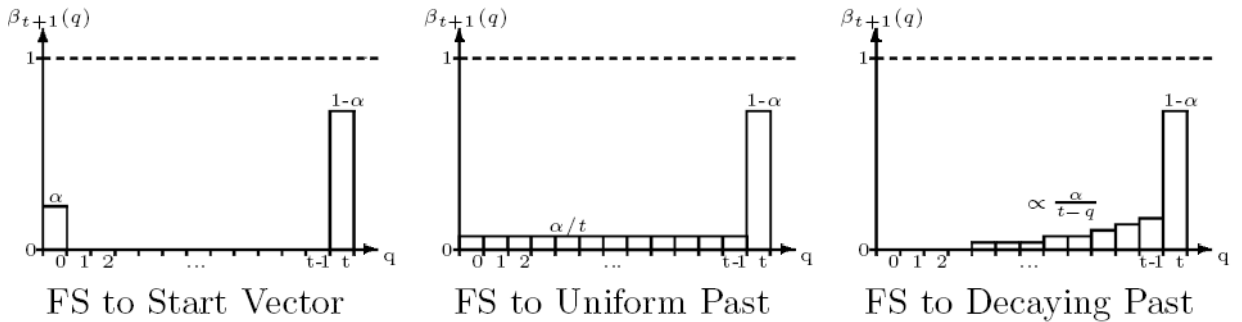


Figure 10: From [1], shows the different mixing schemes.

Where we first compute the Loss update, after receiving the correct value from the dataset y_t , as in:

$$v_{t,i}^m = \frac{v_{t,i} e^{-\eta L_{t,i}}}{\sum_{j=1}^n v_{t,j} e^{-\eta L_{t,j}}}, \quad \text{where } L_{t,i} = L(y_t, x_{t,i})$$

With the loss update, we are able to compute the mixing update, that is going to be used in our

prediction function. The mixing update is given by:

$$v_{t+1} = \sum_{q=0}^t \beta_{t+1}(q) v_q^m$$

Where v_q^m is the loss update already computed and $\beta_{t+1}(q)$ is the mixing coefficients, depending on what mixing scheme we want to apply, and q is the index of the weight that is also going to influence the new weight computation.

Figure 11 shows the optimal algorithm, the no-share-exponential-update algorithm, and the 3 mentioned mixing schemes for linear spacing of the experts and $\alpha=0.4$, which was the best alpha tested.

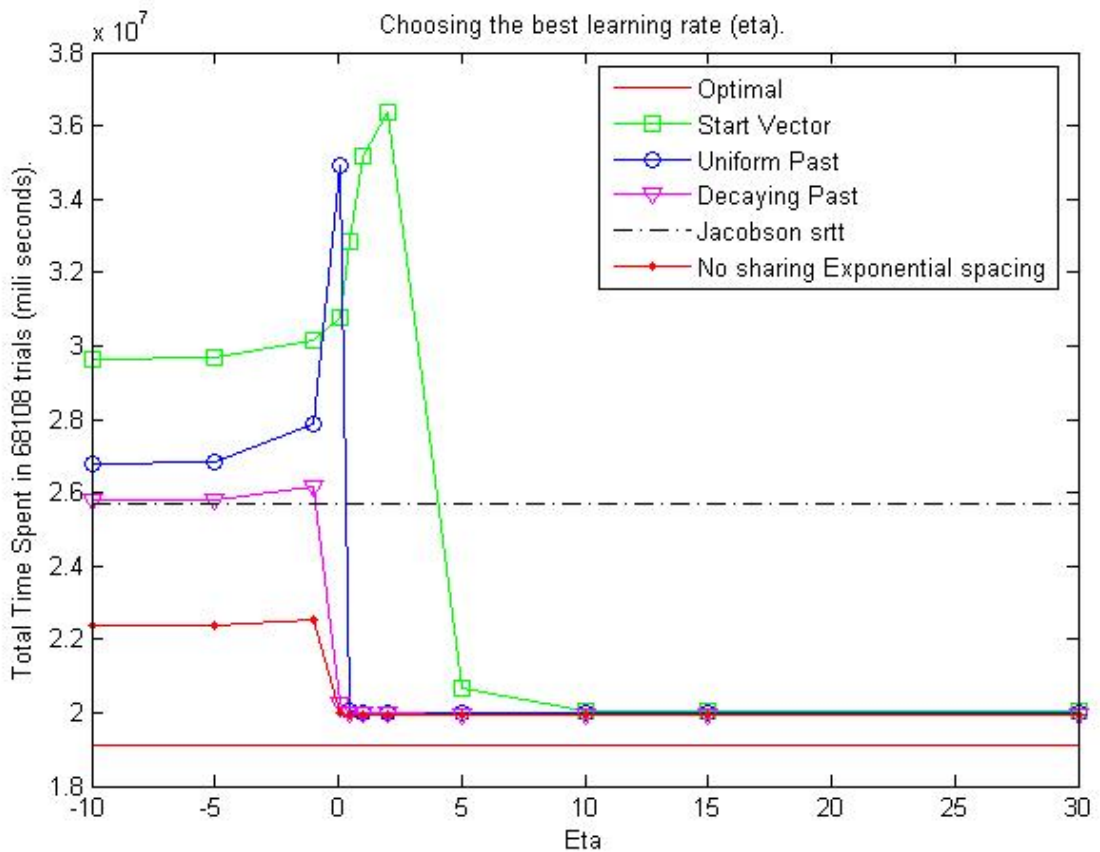


Figure 11: Mixing schemes against No-share exponential update and Jacobson’s algorithm.

Figure 12 we can see the same figure zoomed around eta=15, that shows that decaying past is the best of the mixing schemes in the case of our application but that the simple exponential update still performs better, even without any sharing.

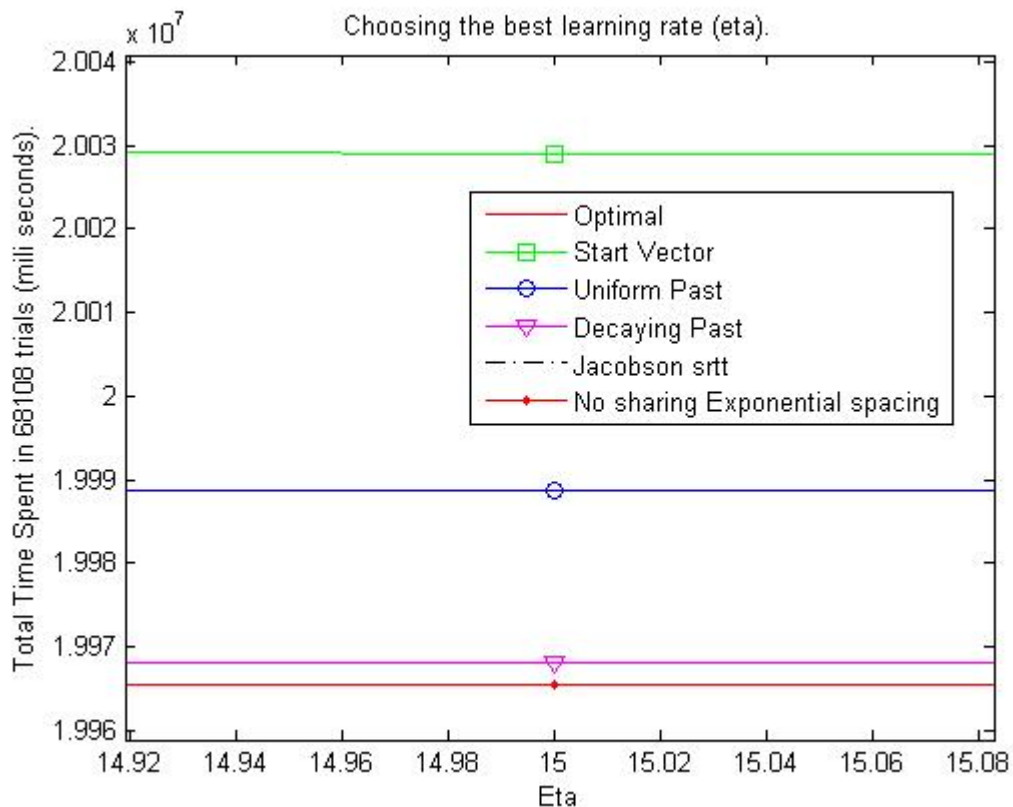


Figure12: Mixing schemes against No-share exponential update zoomed around eta=15.

When we look at the results in Figure 11 we may ask ourselves, why Start Vector and Uniform past have so poor performance for small etas and they get better when using large learning rates? Looking at figures 13 and 14 we can see the weights behavior over time for the SV mixing scheme. We see at Figure 13 the small learning rate, that the updates are smaller and for that reason the weights do not change much. This way, experts with smaller guesses still can influence in the decision, bringing the value of the timeout down, making it more likely to have an undershoot (guess a timeout smaller than the real RTT value), generating a retransmission and being penalized for that. In figure 14, in the other hand, we see only 2 experts influencing the timeout guess. These are the 2 experts with the larger values (expert 14 and 15, since n=15). This mitigates the risk of having an undershoot.

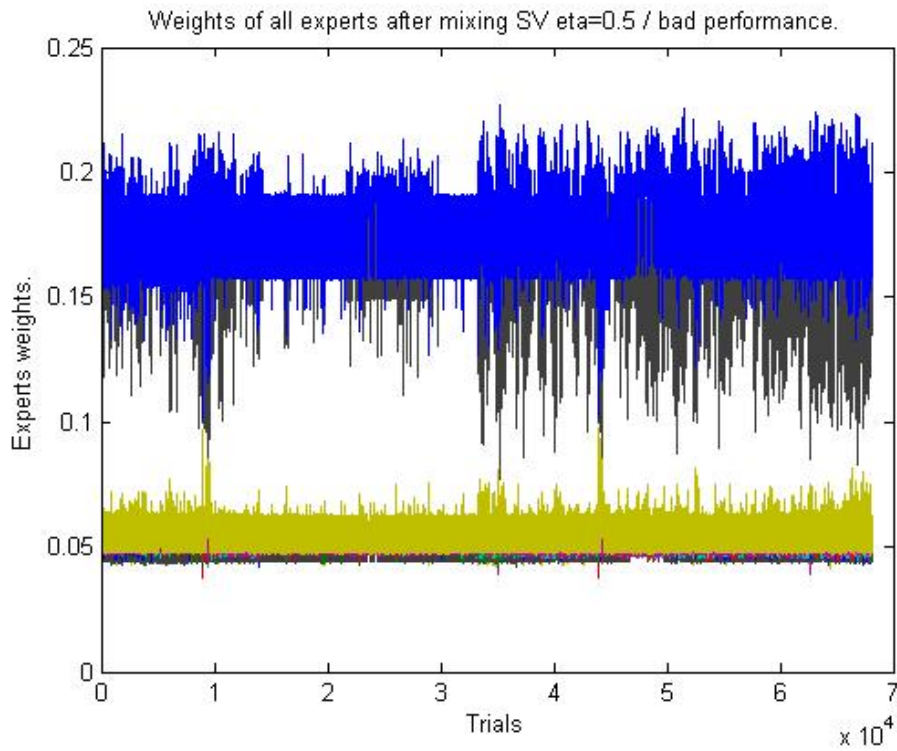


Figure 13: weights over time and small learning rate for SV mixing scheme

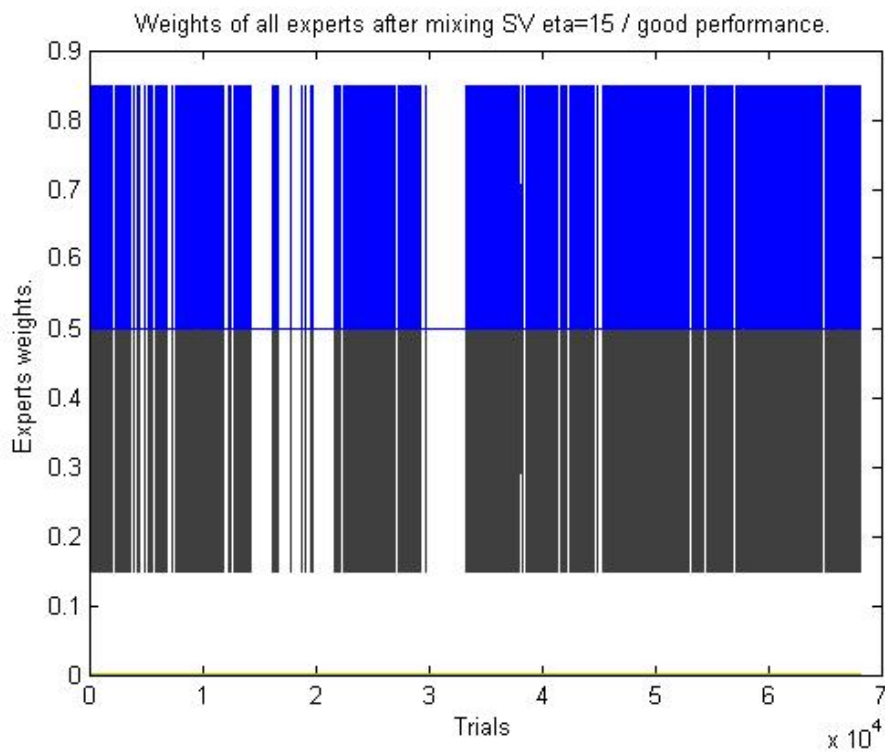


Figure 14: weights over time and large learning rate for SV mixing scheme

Accuracy of Predictions

We now investigate the accuracy of the prediction itself, not the extra time spent on transmission, but the difference between the predictions of the algorithms applied here and the real dataset. We have the prediction of all algorithms over all trails. Table 1 presents the sum of the absolute value of the difference between the predictions and the dataset over all trials. For the Mixing and exponential updates, we are going to use the results with learning rate $\eta=15$, since it is the best result for all cases. In summary we have in this table **Total_Loss = sum(abs(data-algorithm))**. Curiously, Mixing with Start Vector was the winner with the lowest absolute loss of all the algorithms studied.

Total Loss of Algorithm:						
Share Lin	Jacobson	No share lin.	NoShareExp	MIX_sv	MIX_uniform	MIX_decay
2.3580e+006	3.1729e+006	2.3979e+006	2.3982e+006	1.9938e+006	2.2346e+006	2.3074e+006

Table 1: Accuracy of the algorithms

Future work

Next step would be run the algorithms over different datasets. This was not done on this work due to time constrains. Also, it would be interesting to study the online proprieties of the data by running the algorithms on permutations of the data set, to measure the impact of lost data correlation.

Giving continuity to this work, we shall investigate the impact of the algorithms studied here on the throughput of a TCP connection. In order to do this, we should implement the studied algorithms on a network simulator like Qualnet [7] and evaluate the performance of the network over different scenarios.

References

- [1] "Tracking a Small Set of Experts by Mixing Past Posteriors", Olivier Bousquet and Manfred K. Warmuth.
- [2] "A New TCP for Persistent Packet Reordering", Katia Obraczka, *et. al.*

[3] "Adaptative disk spin-down for mobile computers", D. P. Helmbold and D. E. Long and T. L. Sconyers and B. Sherrod;

[4] "Tracking the Best Expert", MARK HERBSTER AND MANFRED K. WARMUTH

[5] "tcpdump", <http://www.tcpdump.org/>

[6] "tcptrace", <http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html>

[7] "Qualnet" Network simulator - <http://www.scalable-networks.com/>