

# Temporal Network Analysis for Predictive Routing Table Optimization

Yujia Jin, Mel Tsai  
{yujia,mtsai}@eecs.Berkeley.edu  
CS268 Final Project, May 2001  
Professor: Ion Stoica

## Abstract

*Routing algorithms currently deployed in today's networks represent an aging group of standards. In particular, they are often driven by concerns such as contractual obligations, economic factors such as cost minimization, distance, and addressing local concerns. These factors are only loosely correlated with good end-to-end performance. While there are many new proposals for improved routing algorithms, in general they can only react to sub-optimal network conditions and balance load poorly. We propose a new scalable routing algorithm that attempts to temporally analyze network conditions using available performance metrics and predictively optimize routing decisions. Predicting network congestion allows us to generate routing tables and make decisions that can altogether circumvent the congestion and balance load. The algorithm has a fine-tuning mechanism when predictions are inaccurate, and we present the results of our experiments using the Berkeley ns2 simulation package.*

## Introduction

The original architects of Internet routing protocols could not have foreseen the explosive growth in today's network usage. These aging standards have become a bottleneck towards efficient growth and scaling of network resources. The problem will only be intensified as networks continue to grow at such "Super-Moore's law" rates, with no end in sight. Savage [S+99] et. al. has reported that today's routing algorithms and policies are rarely driven by the desire for good end-to-end performance, but rather through characteristics such as contractual obligations, economic factors such as cost minimization, load balancing, distance (e.g. hop count), and addressing local concerns. Together, these factors only loosely contribute towards good end-to-end performance. Through their experiments, Savage reports that 30 to 80% of current end-to-end paths are sub-optimal, and in those cases, better alternative paths exist. In their experiments, we clarify that "better" alternate paths are defined to have achieve superior performance in terms of bandwidth, latency, and/or loss-rates.

The need for new standards and protocols for efficient routing in the Internet is clear. In this paper, we begin by discussing the current protocols and methodologies in which current Internet routing is based. Then, we describe some existing work on new algorithms that attempt improve on the weaknesses in current algorithms. Specifically, we present the *Q-Routing* algorithm, an algorithm that attempts to optimize paths taken through a network by optimizing aggregate end-to-end latency through a *reinforcement learning* approach. Next, we describe a new hybrid routing algorithm that builds upon Q-Routing. Q-Routing attempts to optimize the end-to-end performance of

network paths and balance load across links by using *learned* information. However, in our approach, we attempt to use past information to *predictively* optimize network paths based on available performance measurements such as bandwidth and latency. This algorithm has fundamental advantages over current proposals that, in general, are *reactive* to sub-optimal routing conditions. Our algorithm is scalable, has reasonable computational requirements, and converges quickly. Finally, we present the results of our experiments using the Berkeley ns2 network simulator package, our conclusions, and comment on possible future work.

## Related Work

The basic problem statement involved in network routing is "how do I optimally route packets from A to B through a network of interconnected nodes." *Routing algorithms* define the set of parameters and characteristics under which a particular node makes decisions. Specifically, they decide on which "next-hop" a particular packet destined to B should be routed. The most commonly deployed routing algorithms in today's networks are now over 20 years old. They were developed during a time when it was inconceivable that networks would reach topology sizes and bandwidth achievable today. In this section we present the basic classes of today's routing algorithms, along with their weaknesses.

Most of today's routing algorithms can be classified in to one of two approaches: Distance Vector (DV) or link-state protocols. In link-state protocols, information about the current state of the entire network is distributed among the nodes (or to a centralized location), and routing decisions are based upon this global information. Link-state protocols allow the possibility to find the optimal path through a network according to the performance metrics being considered, but suffer tremendously from control-plane issues. Ideally, the goal is to have an algorithm that can find near-optimal paths through a network, but without the global state required for link-state protocols.

Distance vector protocols generally attempt to optimize paths by making *local* decisions based on *locally* disseminated information. The common example is the Bellman Ford algorithm. In this algorithm, each node in the network maintains a complete table that hashes (for every possible destination) a potential next-hop, and the cost (a *hop count*) metric associated with making that decision. Packet routing decisions are made by simply choosing the next-hop that will result in the shortest path to the destination. In DV, each node periodically broadcasts hop count information to its neighbors. The neighboring nodes then update their own tables based on this

new hop information by adding the next-hop cost of the link to the broadcasted value. In this manner, the complete cost information to each node is kept relatively up-to-date with changes in routing topology. Note that for the remainder of this paper, we use the terms “Distance Vector”, “DV,” and the “Bellman Ford” algorithm synonymously.

While DV is used as the primary routing mechanism in today’s networks, it has a number of weaknesses. First it only makes use of local information, and therefore pathologies due to the local broadcasting can occur. In DV, a change in routing topology (e.g. due to a node becoming overloaded and timing out) can cause neighboring nodes to change the path in which they route packets to particular destinations. This, in turn, causes a new broadcast to each subsequent neighbor, etc. The control information being broadcast can quickly overload a network if the topology is rapidly changing.

For example, if the node or link responsible for the topology change (due to overload) is bypassed by its neighbors, the load it must process is immediately alleviated. Therefore, it can potentially recover from its overloaded state and resume normal operation. This, of course, will cause yet another broadcast of information, and its neighboring nodes will again choose it as the next-hop for certain destinations [LMJ97]. The newly broadcasted information can therefore cause a back-and-forth “route storm” that may never recover without some sort of update hysteresis scheme.

Control-plane issues aside, the primary weakness of concern to our work is the metric DV uses to optimize paths. Optimizing a path based on raw hop count will only loosely correlate with good end-to-end performance; the important observation being that hop count generally has little association with expected bandwidth or delay characteristics. In addition, using the shortest path as your primary metric will not balance load. We argue that any “optimal” routing algorithm must necessarily use such performance metrics to route packets.

There have been a number of proposed algorithms that attempt to use a richer set of available performance metrics to optimize paths through a network. We will not exhaustively list them these approaches here. Instead, in the next section we will present *Q-Routing*, which will serve as the basis for our approach. *Q-Routing* is an efficient algorithm that has the same flavor of DV, but instead optimizes the end-to-end *latency* of paths. From there we will highlight the weaknesses of *Q-Routing* and present our proposed predictive approach.

## Q-Routing

*Q-Routing*, first proposed in 1993 by Littman and Boyan [BL93], utilizes reinforcement learning to update its path cost metrics to choose the next-hop for packet routing. In *Q-Routing*, each node maintains a table of *Q-values* for every possible destination and every next-hop. A “*Q-value*” is simply the estimated aggregate latency a packet sent along the associated path will experience, from the current node to the destination.

If a particular node receives a packet destined for node B, it will peer into its *Q-valued* table and parse the table for entries showing B as a destination. From there it has the list of every possible next-hop that it could choose to route to B, along with an estimated delay value (cost). Using a greedy strategy, it simply chooses the next hop with the minimum aggregate path delay.

Figure 1 shows the basic equation and example diagram of *Q-Routing*. In the figure, packets destined for node F must be routed through the network of nodes. Node B must decide whether to choose node C or node D as its next hop for packets routed to F. It examines its *Q-valued* table and decides that choosing node C (with an aggregate delay of 56 milliseconds) will result in the “optimal” routing of packets to node F. In the reinforcement learning phase of *Q-Routing*, after each packet is routed through C, node C will broadcast back to node B with its latency metric for destination F. Node B will then update its *Q-value* table by computing a difference of its current value, the new estimated value from C, and the current delay estimate along the link from B to C. The computation is parameterized by a *learning rate*  $\eta_f$ , as shown in Figure 1. In this manner, *Q-values* are updated in a per-destination, reverse-path reactive fashion.

*Q-Routing* exhibits a number of desirable properties. First that the delay along a particular link can be accurately estimated, *Q-values* represent an accurate estimate of the aggregate delay for packets traveling along a path. Therefore, the routing decisions made by a node performing *Q-Routing* can fully optimize the end-to-end latency for packets.

Second, like DV, it uses only locally broadcasted information to make routing decisions, so the algorithm is scalable. Although it requires a table of  $O(n)$  size (where  $n$  is the number of destinations), the primary routing table that the fast-path routing mechanism uses can be much smaller, and the next-hop routing table can be computed from *Q-values* by a secondary processor.

Under increased loading of the network, *Q-Routing* eventually converges to near-optimal paths for packets being routed through a network. In their experiments with an imbalanced 6x6 grid of nodes, [BL93] reports that the load across the congested links becomes well-balanced after a period of initial *Q-value* exploration of the algorithm. A standard shortest-path algorithm (such as DV), however, cannot account for this imbalance and overloads the bottleneck links.

Unfortunately, *Q-Routing* also has weaknesses. While *Q-Routing* can balance the load across a network under a variety of conditions, it exhibits a *path hysteresis* problem in which it cannot fall back to the optimal unloaded state when network activity decreases. This is due to the way *Q-values* are updated in the network. In the basic framework, *Q-values* are only updated as a packet traverses a link, in which the upstream node receives the path latency metric from the downstream node. In this manner, once a path becomes sub-optimal according to the greedy exploration, the former path is no longer explored by the nodes, and will likely never return as an optimal next-hop candidate.

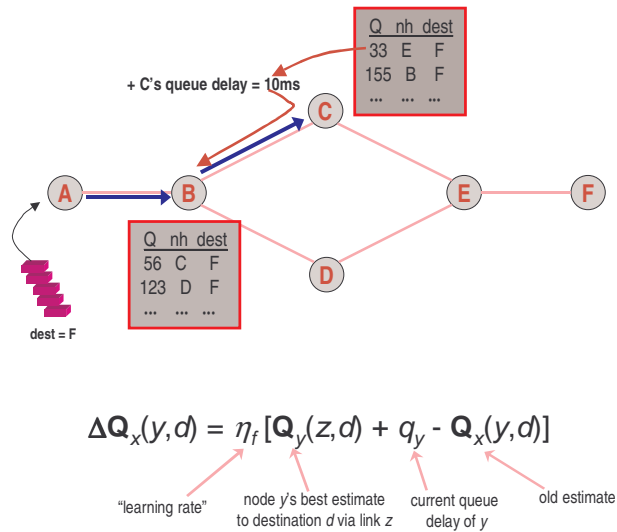


Figure 1 - The Basic Q-Routing Algorithm

## Our Approach

Our approach builds upon the Q-Routing framework, but attempts to alleviate the path hysteresis problems associated with the algorithm, as well as alleviate the slow convergence of the Q-Routing algorithm during periods of changing load conditions.

As stated before, the majority of routing algorithms simply *react* to sub-optimal network conditions in an attempt to alleviate congested links and “hotspots.” However, as reported by Savage in [S+99], large-scale network traffic patterns generally exhibit periodic time-of-day and time-of-week effects. The effects can also be seen on a yearly basis, and this has motivated work relating to self-similar traffic generation. Also, work reported in [Ryu97] attempts to accurately model Internet traffic according to fractal geometric principles, further indicating the periodicity inherent to today’s Internet traffic.

Intuitively, the periodic time-of-day etc. effects of large-scale traffic patterns make sense. It seems far more likely that the network might become highly loaded at 3:00pm at the height of a business day, as opposed to 1:00am on a Sunday morning. In addition to the social and business factors that may contribute to periodic internet traffic, there may also be other unforeseen sources of periodic traffic in the internet, but this work does not attempt to *explain* traffic patterns seen on the internet, but rather we attempt to *exploit* them.

If a routing algorithm could make use of statistically gathered past information, it is conceivable that such an algorithm could circumvent pathological or sub-optimal routing conditions altogether. This is in sharp contrast to reactive algorithms, which can only correct for such conditions.

## The Algorithm

At a high level, our algorithm uses the following approach. Each node in the network constantly gathers information about packets traversing its buffers. It seems prudent that this information is gathered by sampling a certain percentage of incoming or outgoing traffic, as gathering data on all packets could severely inhibit performance. This statistical information is sent to a secondary offline processor, which could even be a completely separate networked PC. This offline computer could perform all of the complex operations required by our algorithm, thus alleviating the problem of maintaining high computational requirements within the node.

Each node runs the standard reactive Q-Routing algorithm. However, similar to DV, the offline computer that is gathering traffic data will periodically broadcast a prediction to the neighboring nodes. This exact information contained in the prediction, and the method used to compute the prediction (the predictor) is described in detail later. But for now, we state that each node uses this prediction to update the Q-value table maintained for each destination according to the same formula used for standard reactive Q-value updates. Since this information is broadcasted, and not specifically updated by a packet traversing a link, the path hysteresis problem is eliminated. In addition, since the Q-value tables are supposedly being updated with future “expected” Q-values, the intuition is that each node can make routing decisions and balance load *before the congestion even begins*. In essence, depending on the quality and type of prediction, our algorithm converges instantaneously because it is not reacting.

Despite the characteristics that such an algorithm might have, finding a “perfect” predictor for network traffic would be a naïve proposition. This is where the original Q-Routing algorithm can help. Any prediction, however inaccurate or even

pathological, would be quickly corrected by the reactive updating of Q-values performed by the original greedy algorithm. In a sense, Q-Routing not only helps us correct for wildly inaccurate predictions, but it also has the ability to fine-tune predictions that were not perfect (which should be the common case). However, finding an accurate predictor is still an important goal.

In the next sections, we describe the predictors used in our predictive algorithm and present some experimental results.

## Predictors

As already stated, the computation of our predictors can be run on a separate processor, apart from the main router, to achieve minimum impact on the router's speed.

In choosing the predictor within this framework, we note the following basic requirements:

- The predictor must be *accurate*.  
  
Inaccurate predictions can potentially cause greater congestion with unnecessary routing updates.
- It must produce predictions according to the *periodicity* of the traffic within the allotted computation time.

Our intuition leads us to believe that an accurate period would be on the scale of hours. This is the time scale that we expect real-world traffic to follow, and therefore it should not take one week of computation time to produce an hour's prediction.

The first requirement is challenging in general, but because we are predicting values for each link locally, the predictor can be tuned locally according to the observed patterns. The second requirement allows us to choose relatively complex predictors. In our experiments we evaluated three different predictors: low pass filtering, bin predictors, and predictors based on support vector machines.

### Low Pass Predictor

Intuitively, it seems reasonable that predicting the near future based on recently observed data (including present data) can give us a quick and relatively accurate prediction. Therefore, using a *low-pass* predictor as the average of the past  $n$  observed performance metric values might give us good results. The performance metric, in our specific case, is the latency experienced due to buffer length and link transit time, although this predictor is easily extended to support bandwidth characteristics.

The low-pass predictor essentially filters away any high frequency activities in the observed data. This predictor assumes the variance in the traffic pattern is low with respect to the time scale of the prediction, with the exception of some high frequency jitter.

### Bin Predictor

The bin predictor is based on the assumption that there is a regular periodic pattern present in the traffic pattern. For example, for every week, traffic patterns observed for the weekdays are likely to be similar to each other, and in addition, they should experience higher loading than during weekends. Also, it seems logical that daytime traffic should be generally greater than nighttime traffic. Therefore, we can make better predictions by using multiple periods of sample data. For the case of real world traffic, it seems sufficient to collect multiple weeks of data.

In bin prediction, we simply predict by using all the past data with the same day and time. For example, if the present time and day is Friday at 4:00pm, the bin predictor computes the exponential weighted average of the last  $n$  window periods of 4:00pm on Fridays. While we can potentially create a more complex predictor based on binned information, due to issues of temporal locality but we believe the EWA is a good initial candidate.

### SVM Regression Predictor

Both the low pass and bin predictors are relatively simple and have low computational requirements. We can therefore perhaps take advantage of greater processing power by exploring more complex predictors. These predictors can potentially use complex models of statistical regression to achieve much more accurate predictions. The Support Vector Machine (SVM) is one such tool that allows us to compute these complex predictions. SVM is a relatively new tool developed by Vapnik at AT&T [CV95]. It was originally developed for data classification purposes, but was later extended to support regression analysis.

In SVM, sample data is translated into a higher dimension by a kernel function in order to compute a simple hyper-plane solution in that higher dimension. The key to successful analysis using SVM is choosing the right kernel for the problem such that a good hyper-plane solution exists.

There are many attributes that the regression analysis can use to produce a prediction. Using SVM, we could support attributes such as average packet delay, average link bandwidth, average packet size, packet types along with the corresponding sequence numbers, TCP packet flags, TCP packet sequence number, packet drop ratio, source and destination address for the flows, time, day, and date. These attributes can be extended to include these values for the past  $n$  time intervals to form a sequence. This extension allows temporal behavior modeling.

## Experiments

Ideally, to evaluate our proposed algorithm, it seems prudent to use real-world observed traffic data to test the performance of our algorithm. However, we found a surprising lack of good long-term data on the web. In addition, once our algorithm begins operating on a simulated network of nodes, when simply

piping observed traffic data into the simulator, we lose the interactivity required to fully test our algorithm. For example, our algorithm might eliminate an observed hot spot in a web traffic trace, and from then on the rest of the traffic trace is invalid because our algorithm would have *affected* the observed results.

Due to this limitation, as well as limitations in simulator speed (explained later), we chose synthetic traffic generation as our source of traffic data. We evaluated our algorithm using synthetic data traces. The synthetic data traces contained hundreds of seconds of data (with 100ms intervals). The traces include the bandwidth information for a link. Again, due to issues with simulator speed, our evaluation was performed in Berkeley’s ns2 on small topologies.

For our experiments, we chose two different types of synthetic traffic generators, and therefore we had two traces for each simulation. The first traffic workload was generated by a web traffic generator module (a package added to ns2 by a 3<sup>rd</sup> party) with 10 client-server “sessions.” This web traffic generator module has been shown to produce realistic self-similar traffic, which is key to our experiment.

For our second traffic workload, we created a synthetic periodic traffic generator. This generator produces sinusoidal traffic patterns with a period of 100 seconds. In our experiments, we modeled 4 different sessions of such flows with the source and destination pairs randomly distributed among the nodes in the topology. The periodic traffic generator serves to test our predictors’ ability to track and account for repeating patterns in observed data.

## Predictor Evaluation

Before going to our algorithm and evaluating our predictors in realistic simulations, we performed some functional experiments on the predictors themselves to gain intuition about their strengths and weaknesses.

To evaluate the predictors, we fed each of them with a large number of single-link observed data points from our traffic generators running on a 2x3 grid of nodes. Then, we allowed each predictor to predict the next 4 data points in the traces are used for test. Therefore, all previous data points are used for training. The evaluation criteria for each predictor was the average absolute error from the “real” data they were able to achieve.

Obviously, in these experiments the necessary feedback in the real-world experiments is absent. However, the ability for a predictor to predict the next four data points when fed training data is a strong indicator about the resulting performance of the predictor. Another concession is that in these experiments, rather than using delay metrics (which our final experiment uses), we evaluated results using measured bandwidth. Nonetheless, extending the results of our predictor evaluation (which uses bandwidth) to our experiment (which utilizes latency) is not necessarily a big leap, as it can be shown that latency and bandwidth are often intimately related in a network. In any case, we believe that this test should give us a good estimation on the real-world performance of the predictors.

The functional SVM predictor uses time and the past  $n$  bandwidth values as function parameters. We tried four different values for  $n$ : 5, 10, 20, and 50. In each case the kernel is searched systematically to find the optimal choice. For SVM, we tried the following five common kernels:

<i>Kernel</i>	<i>Function</i>	*
dot	$k(x,y) = x \cdot y$	
polynomial	$k(x,y) = (x \cdot y + 1)^d$	$d$
radial	$k(x,y) = \exp(-\gamma \cdot  x-y  \cdot 2)$	$\gamma$
neural	$k(x,y) = \tanh(ax \cdot y + b)$	$a, b$
anova	$k(x,y) = (\exp(-\gamma(x_i, y_i)))^d$	$\gamma, d$

\*function parameters

The parameter values were searched in logarithmic fashion from .001 to 100 to minimize loss in the training set. Once the optimal parameters were chosen for each kernel type, prediction for each point is made by training on all previous data points.

<i>Kernel / n</i>	<b>5</b>	<b>10</b>	<b>20</b>	<b>50</b>
dot	46.15	47.07	47.58	54.34
polynomial	58.38	50.00	48.75	116.74
radial	58.37	61.74	61.69	62.29
neural	55.05	59.44	58.86	61.61
anova	45.45	45.40	50.12	50.12

**Average Absolute Error (%), Self-Similar Traffic**

<i>Kernel / n</i>	<b>5</b>	<b>10</b>	<b>20</b>	<b>50</b>
dot	0.81	11.23	34.04	0.24
polynomial	4.00	3.99	0.49	16.55
radial	4.00	4.78	1.77	13.20
neural	1.77	1.69	4.31	7.66
anova	0.89	1.99	4.24	0.54

**Average Absolute Error (%), Periodic Traffic**

The tables above show the performance achieved by each kernel type with the best parameters and with the four different values for  $n$ . Unfortunately, for self-similar traffic, all kernels predicted poorly. The best result is 45.40% by the anova kernel (with  $\gamma = 1$ ,  $d = 1$ , and  $n = 10$ ).

As expected, for periodic traffic all kernels predicted much better. The best result is .49% by the polynomial kernel (with  $d = 2$ ). This result indicates that the variable  $n$  does not affect the prediction in a consistent way. It seems intuitive that increasing  $n$  (and therefore the amount of training data) would improve predictor accuracy since a longer  $n$  will include all information that is included in the shorter  $n$ . However, for self-similar traffic, the opposite is true. Our only conclusion is that the extra information included in a larger training dataset is not useful here and it only serves to confuse the predictor kernel in SVM.

The results with predicting self-similar traffic are poor, and this is unfortunate since these data represent a widely accepted model of real-world Internet traffic. Our results for periodic prediction were better, but they are less realistic. Nonetheless, our initial experiment is still valuable. Simply looking at graphs of observed daily and weekly traffic measurements (at various sites and links) reveals obvious daily and weekly periodic traffic

patterns. Looking at the self-similar traffic generated from ns2 reveals no visibly obvious periodic patterns, surely nothing as obvious as real-world observations show.

Therefore, it seems likely that SVM can successfully predict and capture these patterns. In addition, self-similar traffic is generally used to model traffic on small time scale. So the 400 seconds of self-similar traffic used here may not contain enough information to reflect the daily and weekly traffic patterns. Therefore the result may not be significant since the SVM didn't get enough "experience" to learn the pattern. Further, we are not interested in predicting events on small time scale. Therefore the failure to predict accurately on a small time scale is irrelevant as long as predictions are accurate on the time scale of interest (such as hour-long behavior). Finally, this result may represent a lower bound on what is capable of SVM, since only bandwidth is used for attributes. If the other network attributes are used in combination with bandwidth, the prediction may be greatly improved.

One important lesson learned from using SVM in these predictor evaluations is related to the computational complexity of the kernels. Excluding the parameter optimization, the training and prediction stage of SVM for the largest test case here (about 4000 data points, each with about 50 attributes) required approximately 30 minutes to compute on a Pentium-III at 800 MHz (1 GB of system memory). This experiment shows that storing data on hour-long intervals with standard PC equipment can allow us to analyze up to 4000 hours (166 days) and still train and predict at hour-long intervals. This should be long enough to explore weekly and daily traffic patterns as required by our algorithm.

The SVM parameter optimization can be done offline and separately from the prediction, since they are not likely to vary greatly with time. If more data points are needed, the training stage can be done less frequently than once per hour.

After SVM, we evaluated the low-pass and bin predictors. In our evaluation, the low-pass predictor simply set to average the past 4 intervals. The bin predictor is adjusted to reflect the 100-second periods that were programmed into the generators. Four periods of data are used for prediction (initialized to zero). We used a value of 0.5 for the exponential weighed average constant. The following table shows the test results of these predictors, along with the best results achieved by SVM:

<i>traffic/predictor</i>	<b>SVM</b>	<b>Bin</b>	<b>L-P</b>
Self-Similar	45.40	38.76	45.07
Periodic	0.49	30.51	0.83

**SVM vs. Ad Hoc Methods (Ave % Absolute Error)**

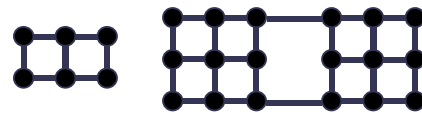
By examining these results, accurate predictions seem to be difficult. Again, these results are based on synthetic data, which may lack the important characteristics that exist in real-world data. For future work, this evaluation should be repeated with real world traces. Also from the result, with the exception of the bin predictor for periodic traffic, the simpler predictors did just as well as the complex offline SVM predictor. This seems to imply that simplicity is the best principle. However, we still believe SVM predictor has more potential since it can be modified to simultaneously use many other prediction metrics besides bandwidth and delay.

## Algorithm Evaluation

For our experiments we chose the Berkeley ns2 network simulator package. This package is written in a mix of object-oriented Tcl, with C++ modules written for certain simulator kernels to increase speed. The ns2 package allows us to string together network topologies and implement all our algorithms and predictors. Our Q-Routing framework was written primarily in Tcl, with some parts of the predictors written as a C++ module.

Nonetheless, despite having very fast computing resources at our disposal, ns2 proved to be exceedingly slow and had difficulty handling even moderate size topologies at the simulation times we wanted. Implementing topologies larger than 2x3 or 3x3 proved to be nearly impossible.

Therefore, we simulated a small topology that might reveal our algorithm's strengths and weaknesses along with our "real-world" traffic generators. Specifically, the topologies we simulated were a 2x3 grid and two 3x3 grids connected by 2 links (as shown in Figure 2). In the 2x3 grid topology, each link was set to achieve 1.5 Mbit/s. In the dual 3x3 grid case, each link is 0.5 Mbit/s.



**Figure 2 - Simulated Topologies**

For each topology, we used the traffic generator setup as described above in the Evaluation section. For each simulation we tested *four* different algorithms: distance vector (bellman ford) as our control algorithm, standard Q-Routing, our Q-Routing algorithm with the simple predictor, and Q-Routing with the bin predictor. Since the experiments using the SVM predictor were performed offline (we found no way to hook SVM into ns2 to perform interactive simulations, and this is left for future work), we did not do an overall algorithm test using our complex SVM predictors. In addition, in our initial exploration we found that the estimated performance of the SVM predictor was similar to the low-pass and bin predictors. Therefore the performance of the low-pass and bin predictors may be indicative of the resulting performance of SVM.

The first set of simulations were performed on the 2x3 grid of nodes. The 10-session, self-similar web traffic generator was simulated (with random source and destination nodes) for 400 seconds. The periodic traffic generator (again, with random source and destination nodes) was also simulated for 400 seconds. Simulating one algorithm on the 2x3 grid (with one type of generator) required approximately 30 minutes to complete on our test bed. Not including data analysis generation, our 2x3 grid experiments require 4 hours of computation for a complete test of all algorithms.

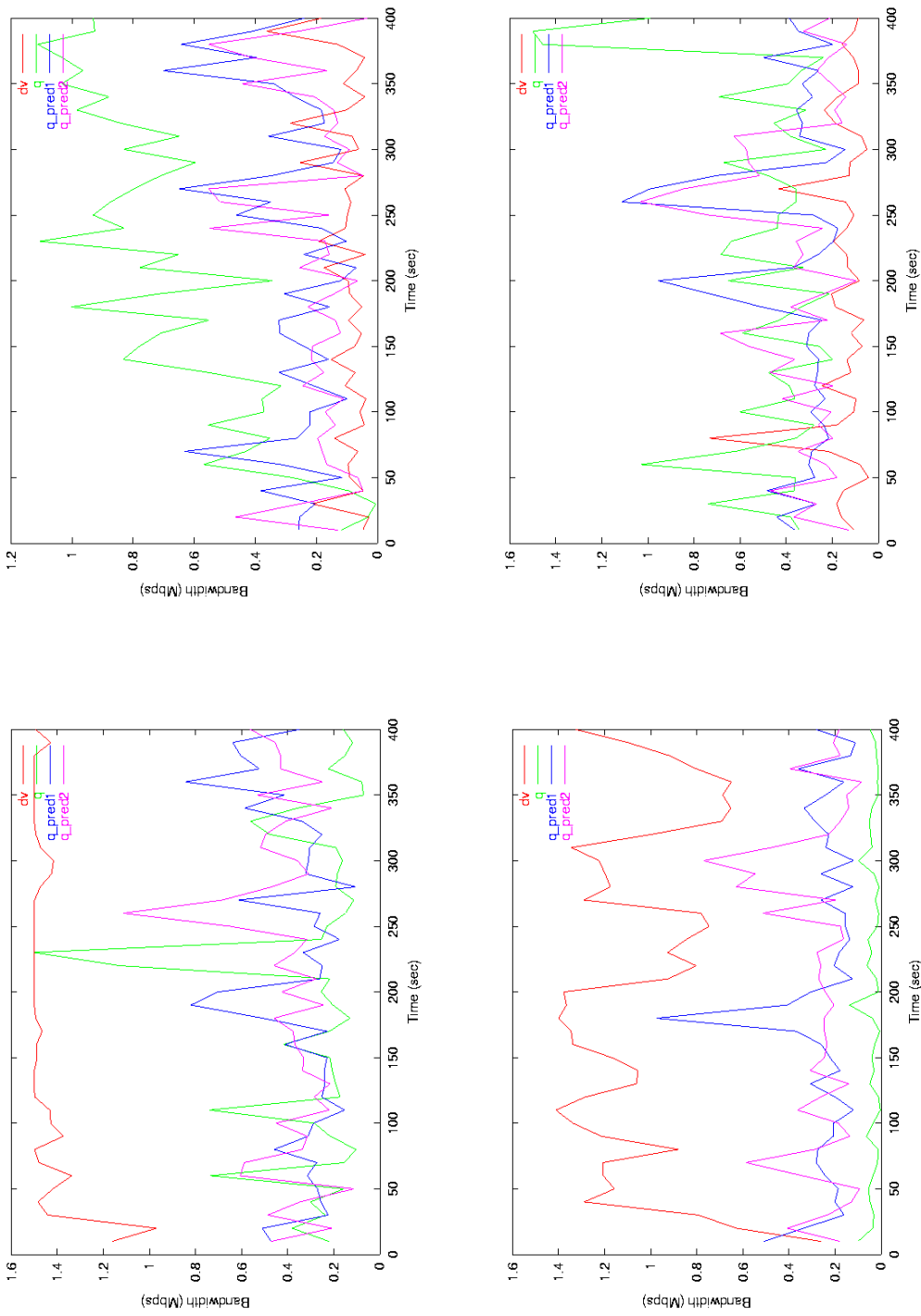


Figure 3 – Self-Similar Web Traffic Traces on Representative links, 2x3 grid  
(Graphs 1-4, clockwise from bottom left)

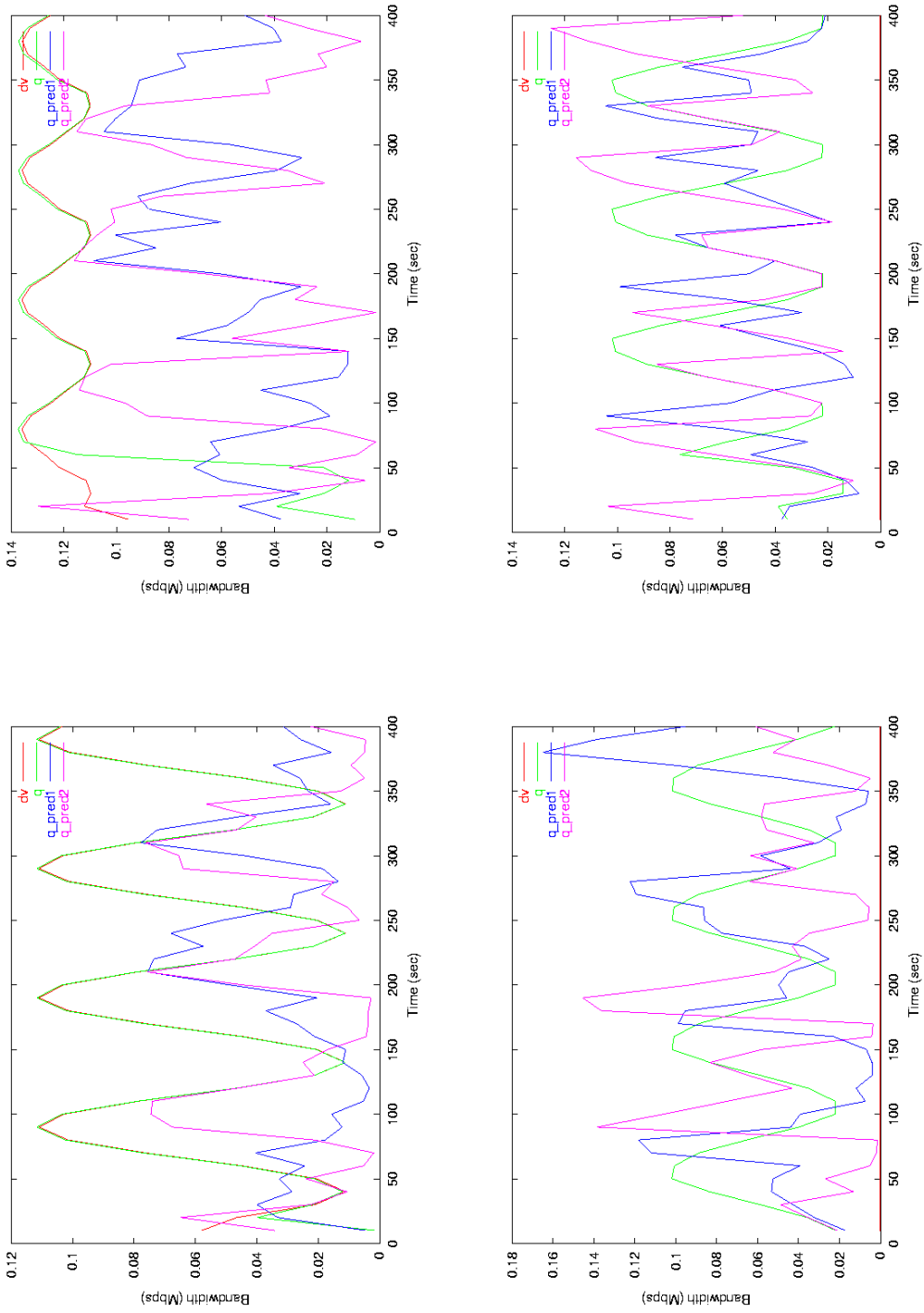
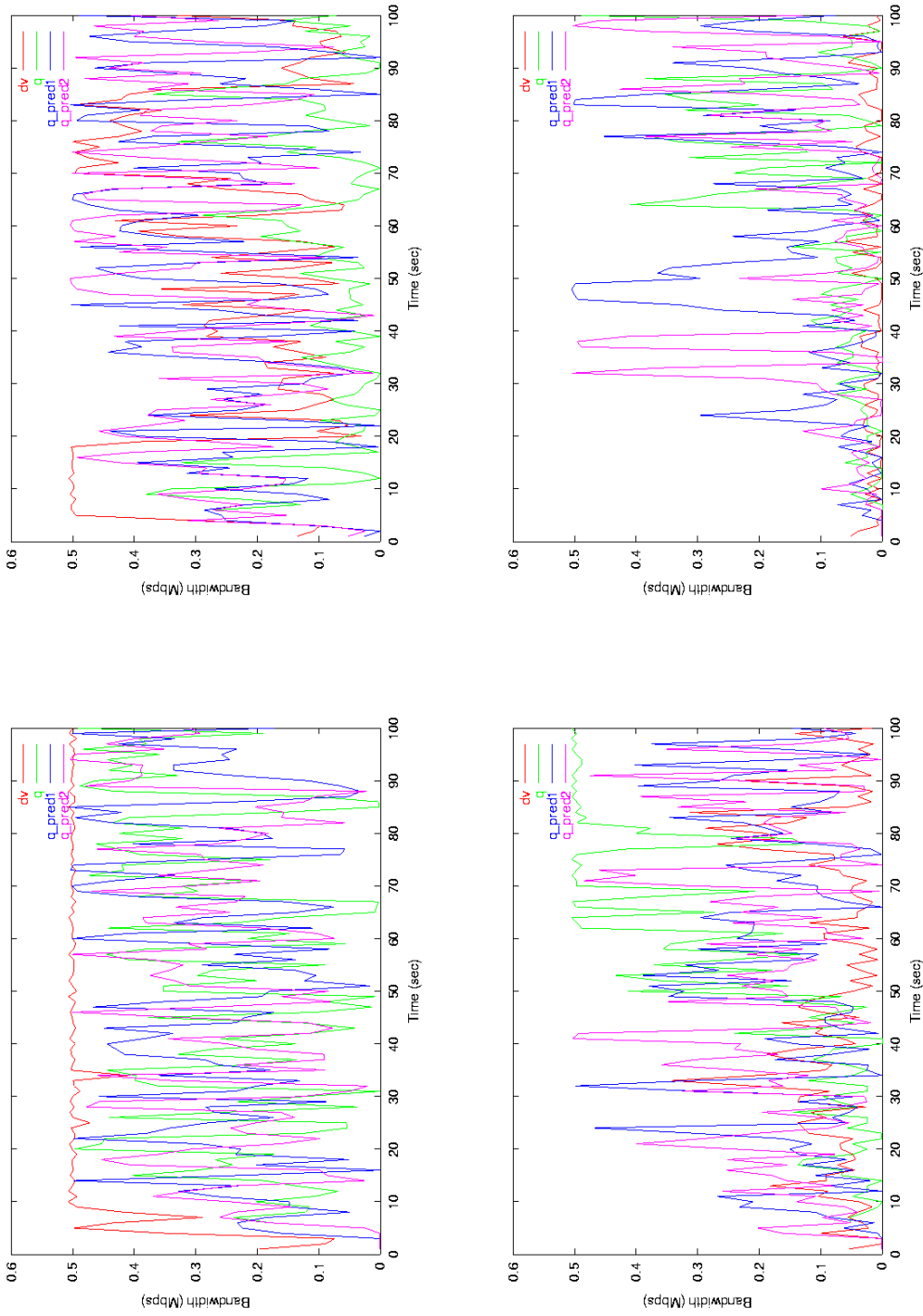


Figure 4 – Periodic Web Traffic Traces on Representative links, 2x3 grid  
(Graphs 1-4, clockwise from bottom left)



**Figure 5 – Self-Similar Traffic Traces on Representative links, dual 3x3 grid**  
 (Graphs 1-4, clockwise from bottom left)

On the dual 3x3 grid topology, we only used the 10-session self-similar web traffic generator, simulated for 100 seconds. We did not evaluate the periodic traffic generator for this topology, since periodic traffic (with period = 100 seconds) required at least 300-400 seconds of simulation time, which proved to be impossible due to simulator memory issues. The package proved entirely unstable with larger topologies and longer simulation times, and there's no telling what issues might come up when interactively using SVM along with ns2. Each run for the dual 3x3 grid topology required approximate 45 minutes to complete.

One key point is that while the algorithms actually use *latency* as the performance metric, we analyzed and evaluated the results of each simulation by computing instantaneous *bandwidth* across each link. Bandwidth was easy to extract from generated trace data, and it also allowed us to more easily observe the load-balancing characteristics of each algorithm (in which latency would be irrelevant).

## Results and Analysis

Figure 3 shows the results for our 2x3 grid simulation with self-similar traffic. The four graphs show bandwidth usage on four links versus time for distance vector, standard Q-Routing, Q-Routing with the low pass predictor (q\_pred1), and Q-Routing with the bin predictor (q\_pred2). We graphed these links because they are representative of bandwidth usage across all links in the topology.

As seen from the graphs, DV balanced load poorly across the links due to its static nature. DV shows a high bandwidth usage in Graphs 1 and 4, while Graphs 2 and 3 show low usage. For standard Q-Routing, we see better load balancing across the links than with DV. However, Q-Routing's result on Graphs 2 and 4 show that there is still room for improvement. One thing to note is that Graph 2 shows bandwidth for Q-Routing is continuously increasing, and then remains high. This is potentially due to the hysteresis problem inherent to Q-Routing. Q-Routing fails to notice alternate superior paths after the initial congestion has taken place. Therefore, the primary route remains in a congested state when an alternate route could provide improvement.

For our predictive routing algorithms, we see a relatively balanced amount of bandwidth usage across the representative links (a relatively stable 0.4 Mbps on each link). This shows that our algorithms seem sufficient to balance load across the links and can potentially prevent hot-spots. Note that there is little difference between the low pass and bin predictor results. This seems reasonable since both predictors did equally well in initial our predictor evaluation for the self-similar traffic load test.

Figure 4 shows the results for our 2x3 grid with a periodic traffic load. Again, the four graphs show representative link bandwidth usage across the topology. Here we see better load balancing for our predictive routing algorithm in comparison to DV and standard Q-Routing. As with the self-similar generator, the difference between low-pass and bin predictor is also

insignificant. However, this is contrary to the initial predictor evaluation result, which gives a 30% edge in absolute error for the low-pass predictor over the bin predictor. This potentially shows the limitations in our initial evaluation where there is no interactive traffic generation, only static predictions which give rise to rough estimations. This may also show that our corrective fine-tuning mechanism for mispredictions is proving effective.

Finally, Figure 5 shows the results for the dual 3x3 grid, driven by the self-similar traffic generator for 100 seconds. Again, due to simulator memory issues, larger topologies and longer simulations could not be performed. Graphs 1 and 2 show two of the *bridge* links (the ones connecting the two 3x3 grids) going the same direction from one 3x3 grid to the other. The other two graphs represent illustrative links in the remaining topology.

Intuitively, the bridge links should become a bottleneck for traffic in the topology. It is clear that DV, which only chooses the shortest path, failed to balance the load (you can see that one link has high load, while the other does not). Q-Routing failed to balance across all four graphs as well but did a better job than DV. And finally, the results show that our predictive routing algorithms seemed to balance the load well across the two bridge links.

## Conclusions and Future Work

From our three experiments, we conclude that predictive routing algorithms seem to out perform DV and Q-Routing to balance load and eliminate hotspots.

Despite these relatively good results, there are still some things that need to be considered. For example, in "real world" internet topologies, there is a tendency for routers to aggregate nodes together using the longest-prefix matching of internet addresses. For example, an ISP running network address translation may hide a large group of nodes behind a firewall, and therefore Q-Routing's assumption that every destination has a unique address may become invalid. A potential solution is to simply aggregate these hidden nodes into a "supernode," in which at least part of our algorithm should be effective. Whether or not these algorithms are deployable in real-world situations is left for further analysis.

In addition, we did not analyze the overall improvement in end-to-end bandwidth or delay as experienced by users. Simply integrating our algorithms into ns2, evaluating the predictors, and setting up the simulations required the majority of the semester, and further analysis of the results is therefore left to future work.

Finally, testing our algorithms with real-world traces and also implementing them on real network nodes seems necessary to confirm the simulation results. Testing SVM further as a predictor is an additional goal, as we believe that this complex prediction scheme could potentially give us the best results for our algorithm.

## References

- [BGV92] B.E. Boser, I.M. Guyon, V.N. Vapnik. *A training algorithm for optimal margin classifier*, In Proc. 5<sup>th</sup> ACM Workshop on Computational Learning Theory, pages 144-152, Pittsburgh, PA, July 1992
- [BL93] J.A. Boyan, M. Littman, *A distributed reinforcement learning scheme for network routing*, In Proceedings of the First International Workshop on Applications of Neural Networks to Telecommunications, p45-51, Hilside, NJ, 1993. Erlbaum.
- [CV95] C. Cortes and V. Vapnik. *Support vector networks*, Machine Learning 20:1-25, 1995
- [KM97] S. Kumar, R. Miikkulainen, *Dual reinforcement Q-Routing: An on-line adaptive routing algorithm*, In Proceedings of the Artificial Neural Networks in Engineering Conference, 1997
- [KM98] S. Kumar, R. Miikkulainen, *Confidence-based Q-Routing: An on-line adaptive network routing algorithm*, In Proceedings of the Artificial Neural Networks in Engineering Conference, 1998
- [LMJ97] C. Labovitz, G. R. Malan, and F. Jahanian, *Internet Routing Instability*, Proceedings of SIGCOMM'97, September 1997
- [Pax96] V. Paxson, *End-to-End Routing Behavior in the Internet*. ACM SIGCOMM '96, August 1996, Stanford, CA.
- [Ryu97] B. Ryu, *Fractal Network Traffic Modeling: Past, Present, and Future*. 35th Allerton Conference on Communication, Control, and Computing, Sep. 1997.
- [S+99] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson, *The End-to-End Effects of Internet Path Selection*, 1999
- [Vap95] V. Vapnik. *The Nature of Statistical Learning Theory*. Srpinge, New York, 1995