Prince: an effective router mechanism for networks with selfish flows

Lazaros Tsavlidis, Pavlos S. Efraimidis, and Remous-Aris Koutsiamanis

***Abstract*— Starting from the premise that modern routers are not protected from aggressive and unresponsive flows, we define a new, almost stateless, active queue management scheme, called Prince. The basic idea is to protect the fair share of well-behaved flows. We adopt a game theoretic view, where incentive is given to the majority flow by dropping its packets at congestion. In order to find the majority flow, we focus on the queue of the router and detect the flow with the most packets in it. From a game-theoretic point of view, Prince manages to track and bound aggressive flows and favor socially responsible ones. Our results show that in this context Prince resembles MaxMin Fairness allocation. Finally, we also examine a streaming version of the algorithm that can be fine-tuned to any desired performance/accuracy trade-off point.**

***Index Terms*– Algorithmic Mechanism Design, Fair Resource Allocation, Active Queue Management, Algorithmic Game Theory**

I. Introduction

Network congestion is a major issue on the Internet. Under congestion, networks struggle to allocate resources efficiently and fairly. Congestion builds up easily when some of the flows try to gain a large share of the network capacity, either by excessively increasing their sending rate or by not cutting back despite their packet losses. This situation, in which multiple selfish players can ultimately overload a shared resource even when it is obvious that it is not in anyone's long term interest, is an instance of the "Tragedy of the Commons" problem. This behavior leads to heavy congestion and threatens the stability and efficiency of the Internet.

Beyond these consequences, our greatest concern is the unfairness that arises. During congestion, misbehaving flows may retain their sending rate while well-behaved ones cut back. The result is that the misbehaving flows receive an unfair proportion of the bandwidth at the expense of the well-behaved flows. In this paper, a game theoretic point of view is adopted. In these terms, network flows are selfish and independent players, the router's queueing algorithm is the game mechanism, the players' bandwidth requests constitute the set of possible strategies and the allocated bandwidth is the players' utility.

Achieving fairness and efficiency in the network can be translated to achieving a desirable Nash Equilibrium (NE) in the game theoretic model. In order to accomplish this goal we turn to mechanism design. We opt not to try to control the flows but to give them incentives to act responsibly [17], [18]. We use the core elements of a network, the routers, to warn or diminish selfish flows. In a previous work [6], we analyzed the Prince algorithm in an abstract network-game model and obtained interesting results. In this paper, we adapt Prince to a realistic Internet-centric model. Our basic principle is to ground the packet dropping decisions on the buffer contents. In particular, at every congestion, a packet from a flow with the largest number of packets in the buffer, i.e. a majority flow, is dropped. We present three versions of Prince: Prince-G precisely implements the basic principle, Prince-S is a more vindictive instance of the basic principle and Prince-A approximates Prince-G with a data stream algorithm.

The novelty of this work is the application of game theoretic incentives in a real network in order to accomplish fairness among the players, while at the same time employing a lightweight mechanism on the routers. The mechanism is a new active queue management scheme which resembles MaxMin fairness by protecting the fair share of well-behaved flows. We do not achieve this by trying to implement a strict instance of MaxMin by continuously controlling every flow. Rather, and this is our innovation, we apply either moderate (Prince-G) or strong (Prince-S) incentives to the aggressive player who stresses the router most during congestion. This will force any rational player to back off in order to avoid further detriment to his utility. We study Prince and provide theoretical arguments and extensive experimental results. For the latter, we experimented with TCP, UDP and mixed TCP and UDP flows of varying aggressiveness and we compared Prince against other popular queueing policies such as DropTail, RED, CHOKe and MaxMin. Additionally, we propose a low complexity approximation of Prince to allow for an almost stateless router implementation.

II. Related Work

Nagle [13] proposed a game-theoretic view of network congestion and suggested a market solution according to which the rules of the game should be set in such a way, so that the optimal strategy for the individual user results in an optimal situation for all users. Shenker [18] correlates the selfish behavior of the users with the design of the switch service disciplines and suggests a fair share scheme which guarantees efficient and fair operating points. Other researchers also tried to model the interaction between Internet users with various game definitions [1], [6], [17], [18] and emphasized the importance of mechanism design in this process.

Router queue algorithms can be classified according to their computational requirements. On the one hand, there are stateless algorithms, which are lightweight and simple. For similar games to ours, it has been proven that DropTail or RED routers lead to undesirable NE when modern TCP flows (e.g. SACK) participate [1], [5]. The handicap of DropTail is its indiscriminate packet dropping mechanism, which causes unfairness. RED [7] notifies more flows about congestion than DropTail by deploying a randomized dropping mechanism. RED also constrains the queue length between two thresholds in order to prevent overflow and high queueing delay. The drawback is that RED imposes the same loss rate for all flows, therefore a flow has no incentive to be socially responsible.

On the other hand, there are stateful queueing policies, like Fair Queueing [3], which are too computationally demanding

Manuscript received March 17, 2010. Revision submitted March 30, 2011.

L. Tsavlidis, P. S. Efraimidis and R.-A. Koutsiamanis are with the Department of Electrical and Computer Engineering at the Democritus University of Thrace in Xanthi, Greece (e-mails:{ltsavlid,pefraimi,akoutsia}@ee.duth.gr).

to be deployed at routers. Fair Queueing accomplishes the desired result (fairness) but at the cost of a separate queue for each flow and increased management complexity. In response, a variety of buffer management schemes were proposed that maintain a FIFO queue while trying to fairly allocate bandwidth. For example, CSFQ [19] does not need to maintain state on core routers but it has to on the edge routers. Its disadvantage is that the architecture of the Internet has to be modified to allow routers to exchange messages relaying the flows' rate estimations. Other queueing policies use the history of packet drops (e.g. RED-PD [12]) or the history of the incoming packets (e.g. AFD [15]) to detect the aggressive flows. While these policies do not keep separate queues for each flow, they still require complex computations and extra buffering operations.

CHOKe [16] is based on the assumption that the queue content during congestion constitutes a sufficient statistic about the incoming traffic and provides useful information about candidate flows for pruning. CHOKe penalizes flows that overcome their fair share by deploying a probabilistic algorithm. Every incoming packet is compared with an already queued packet and if they match they are both dropped. The performance of this algorithm is good when only one misbehaving flow is traversing the router but degrades when more than one flow is aggressive. Another approach was also based on the same queue management guidelines and a game theoretic model [8]. Despite that it also aims at the highest rate flow, it requires delicate refinement of the in-between queue thresholds. Additionally, its dropping policy does not shield the fair share when the queue usage is above the predefined high threshold.

Outline.

In Section III we describe the basic guidelines for the design of the Prince queueing policy and present three variants. Moreover, some theoretical arguments, the detailed description and the corresponding computational requirements of the Prince algorithm are provided. In Section IV, we discuss the game-theoretic idea that culminated in Prince, while in Section V we describe the experimental methodology and the results. Finally, in Section VI we survey both the theoretical claims and the experimental results, as well as our future work in this area.

III. The Prince Algorithms

As already stated, our goal was to design a game mechanism for the network which provides incentives to the flows to behave in a socially responsible manner. The design criteria we used for the mechanism should:

- Lead the game towards a desirable NE
- Provide a stateless and simple implementation.
- Not depend on being deployed on the whole network.

Based upon our criteria, we propose the Prince mechanism, which uses the router queue and focuses on the majority flow in it, i.e., the flow with the most packets.

Our algorithms work on a FIFO router queue and drop packets only during congestion. We define three implementations with different trade-offs:

- Prince-G (*G*entle) drops a packet from the majority flow whenever a packet drop is required.

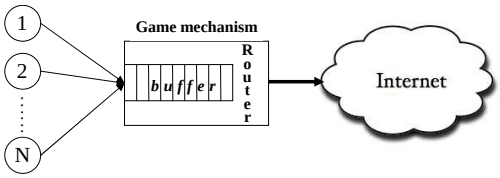

Fig. 1. The game model

- Prince-S (*S*evere) marks all the majority flow's packets and drops one of the marked packets whenever a packet drop is required.
- Prince-A (*A*daptive) emulates Prince-G with a data stream algorithm adapted from [10].

A. Theoretical Arguments

We will introduce a simple but concise game definition in order to specify the model under analysis. The game that represents the interaction between the flows and the Internet infrastructure (Figure 1) is the following:

- The n players of the game are the flows that compete for the common resource (link capacity).
- The moves available to each player are:
  - set the AIMD parameters (α,β) for TCP flows,
  - set the constant sending rate for UDP flows.
- The mechanism of the game is the router's packet dropping protocol.
- The goal of each player is to maximize their utility function (e.g. maximizing goodput).
- The solution concept of the game is the Nash Equilibrium.

A desirable NE for the above game is characterized by efficiency and fair bandwidth allocation. While fairness can be defined in multiple ways, we consider the MaxMin Fairness criterion [3] to be the most appropriate for our model. According to MaxMin, a set of rates is fair if no rate can be increased without simultaneously decreasing another, smaller, rate. MaxMin Fairness results in an equal share of the bottleneck link for each flow traversing it [11] unless a flow requests less than its fair share. In this case, the frugal flow receives the bandwidth it requested, and the remaining capacity is distributed equally to the more greedy flows.

The Prince algorithm attempts to protect the fair share of each player in the game. In essence, the Prince-G algorithm resembles the MaxMin Fairness bandwidth allocation by minimizing the majority flow's sending window and sharing the released bandwidth with the rest of the players. Every time a new packet arrives at the queue the Prince-G algorithm is triggered. If the queue is full, then a decision has to be made on which flow's packet to drop. As the following lemma shows, Prince and MaxMin both decide on a flow with the maximum number of packets.

Lemma 1: The Prince-G policy implements MaxMin Fairness for buffer sharing.

Proof: Assume a Prince-G router with queue size C and a set of n flows. Assume that the queue is full and that a new packet has just arrived at the router. Hence, a total number of $C+1$ packets are currently at the router. Let $w_1, w_2, \ldots, w_n$ be the number of packets that belong to flows $1, 2, \ldots, n$, respectively. Without loss of generality we can assume that

$$w_1 \leq w_2 \leq \cdots \leq w_n \ . \tag{1}$$

The Prince-G policy will drop a packet from the flow n with the largest number of packets in the queue (ties are solved randomly). This way Prince-G implements the MaxMin criterion. ■

Lemma 2: In both of the Prince-G and Prince-S policies, a flow that did not exceed its fair share in the queue buffer, does not lose any packet. Furthermore, in Prince-G, a flow is never forced to have a buffer share smaller than its fair share.

Proof: As in Lemma 1 assume a router with queue size C and a set of n flows. A new packet has just arrived while the queue is full. Let $w_1, w_2, \ldots, w_n$ be the number of packets that belong to flows $1, 2, \ldots, n$, respectively, and assume relation 1 holds.

By combining relation 1 with

$$\sum_{i=1}^{n} w_i = C + 1 \ , \tag{2}$$

we can show by contradiction that the number of packets of flow n is $w_n > C/n$. Clearly, if $w_n \leq C/n$ then $\sum_{i=1}^{n} w_i \leq n \cdot (C/n) = C < C + 1$, a contradiction.

We conclude that in both Prince-G and Prince-S, a flow that has not exceeded its fair share cannot experience packet drops. Furthermore, since Prince-G identifies a flow with the maximum number of packets each time a packet has to be dropped, a flow is never pushed strictly below its fair share. ■

We consider the above lemmas to be evidence that our algorithms and especially Prince-G lead the game to desirable NE. Further evidence is provided by the experimental results in Section V.

B. Algorithm Descriptions

We examine three algorithms that embody the basic principle of Prince, i.e., dropping packets from the majority flow. All three algorithms operate by dropping packets when the router experiences congestion, that is, when the router queue is full and another packet arrives for which there is no more space. The algorithms are differentiated by the way they select which packet to drop under such circumstances. We consider a router queue with C packets and n unique flows.

Prince-G : The Prince-G algorithm scans the queue and counts the packets of each flow whenever a packet needs to be dropped. Then it drops the first[1] packet in the buffer of the most frequent flow, making space for the new packet to enter the queue. If the new packet belongs to the most frequent flow in the queue, then only this packet is dropped immediately.

Complexity : Building the list of frequencies per flow can be achieved in $O(C)$ amortized time, by using a single pass over the queue and accumulating the counts in a hash-table-based dictionary (key:flowid, value:packet count). This time complexity can be improved to $O(1)$ worst case with high probability if one of the hashing algorithms of [4] or [2] is used. The most frequent flow can be identified within the same process. The required space is $\Theta(\min\{C, n\})$.

Prince-S : The Prince-S algorithm retains a list of marked packets which are candidates for being dropped. To create the list, we execute once what is essentially a two-pass Prince-G algorithm resulting in all of the majority flow's packets being marked (one pass to find the majority flow as in Prince-G, one pass to mark all the majority flow's packets). If the queue experiences congestion and there are no marked packets in the queue, the list is created on-demand and then the first marked packet is selected immediately for dropping. On the other hand, if the list already contains marked packets then the marking process is not executed and the next marked packet in the list is dropped.

Complexity : Building the list of frequencies per flow is achieved in the same $O(C)$ time as Prince-G. The marking of the most frequent flow's packets, w_{max} in number, can be stored in a linked list in time $O(C)$ and in space $\Theta(w_{max})$. Dropping a marked packet can be achieved in $O(1)$ time.

Prince-A : Prince-A is the window-based adaptation of the data stream algorithm of Karp et al. [10]. The data stream technique identifies the top-k heavy hitters in order to approximately spot the majority flow while being as lightweight as possible at the same time. Prince-A uses only a limited number of counters (k) which is significantly less than the queue capacity. The purpose is to implement the Prince algorithm with less computational resources.

When a new packet arrives, irrespective of congestion, the original algorithm is executed and the flow who sent the packet may or may not get a counter. More precisely, the router examines if the flow that the incoming packet belongs has already a counter. If it already has a counter then this counter is incremented by one. If it doesn't, first checks if there is an empty counter to correlate it with the current flow or else decrements all counters by one.

The adaptation consists of triggering when a packet is either served or dropped. In these cases, if the packet's flow had a counter associated with it, its value is decremented by one. This function allows fast adaptation to changing network conditions.

Complexity : The more complex implementation for storing the counts, also proposed in [10] is used, allowing for $O(1)$ worst case with high probability time complexity when a packet arrives. Space complexity in this implementation is $\Theta(1/\theta + c)$, where c is the largest frequency and $1/\theta$ is the maximum number of counters used. Both have a small upper bound: $c, 1/\theta \leq \min\{C, n\}$. When a packet is served or dropped, the time complexity is the same $O(1)$ as when one arrives.

Additionally, if one is willing to trade space complexity for time complexity, it is possible to substitute the approximate Prince-A algorithm with HL-HITTERS, a recently developed exact $O(1)$ worst case with high probability time and $O(C)$ space complexity algorithm for finding the heaviest-k hitters [22].

C. Effects of the Packet Size Assumption

In this work, we have focused on scenarios with packets of equal size and showed that Prince handles them very well. Indeed, the case of packets with different packet sizes is very important for network routers. In brief, the Prince-G and Prince-S algorithms can be adapted to count the total size of the packets of each flow and then drop one or more packets from the majority (in bytes) flow. For Prince-A this approach does not apply. However, we can still handle packets of various sizes by exploiting the fact that the size of IP packets does not vary more than a constant factor. Thus, for Prince-A we can consider a minimum packet size (mps) and handle any larger packet as being k minimum packets for some appropriate integer k. The data structure of [22] mentioned earlier can

[1] to quickly alert the flow about congestion

continuously monitor the majority flow in a router queue with a time complexity of $O(1)$ worst case with high probability per packet. This data structure can also be adapted to packets of variable size with the same trick as above in Prince-A.

IV. Discussion

The Prince mechanism embodies the following fundamental game theoretic principle. At the moment of congestion, we drop packets from the player who contributes the most to the congestion. As a result, his utility diminishes if he continues to be aggressive. This is a strong incentive for a selfish but rational player to back off, when he wants to maximize his utility function, even if packet loss has a minor cost for him. At the same time, Prince ensures that well-behaved players receive appropriate service. The power of this technique lies in that we need only target the most aggressive player to motivate *all* the players to behave well. Even though all players desire the largest possible proportion of the link capacity, no one will want to have the maximum share because of the penalty. Since it is not possible for a player to find out the shares of the other players, he will have to be careful not to request too much bandwidth in order not to become the most aggressive one. The result is that the players restrain themselves to avoid the penalty, until no congestion is present.

We should note that both Prince-G and Prince-S are motivated by the same principle, i.e. punishing the most aggressive player, but they accomplish this using different means and have slightly different results. Arguably, Prince-S is the most "vengeful" of the two. It will invariably provide the strongest incentive to moderate aggressiveness, at the expense of being less sensitive to majority flow fluctuations due to the lag between majority flow re-evaluations. It will also be more computationally lightweight, on average, than Prince-G.

The Prince algorithms presented in this work implement work-preserving queue disciplines that drop packets from the router queue only in case of overflows and, even then, the minimum possible number of packets is dropped. When there is no overflow, every flow is granted the buffer capacity it requests. During overflows, the Prince algorithms implement (Prince-G) or approximate (Prince-S, Prince-A) MaxMin fairness for queue buffer sharing. MaxMin fairness is considered, in general, one of the most effective ways to handle resource sharing for heterogeneous (and homogeneous) demands.

Under the above perspective, Prince is a queueing mechanism that can either enforce socially responsible behavior on a misbehaving player or cooperate with a player who has the following desirable features:

- Adoption of end-to-end congestion control, that is, being responsive to packet losses by throttling down upon congestion and throttling up to discover the fair share.
- Self-optimization by taking into account the packet losses in the utility function.

It should be noted that the buffer size plays an important role in the Prince algorithm. On the one hand, using a large buffer provides us with a good approximation of the players' sending windows. The more packets the buffer contains at congestion, the better our queue snapshot captures each flow's contribution. On the other hand, a large buffer creates more queueing delay for all the flows traversing the router and extra computational cost to the router's overall job. However, in our experiments we obtained fair bandwidth allocations even with small buffer sizes.

V. Experiments

A. Experimental setup

We carried out a large set of experiments on the established ns2 network simulator [14]. As a first step, we verified that Prince manages to shield the fair share of the well-behaved flows by reducing the bandwidth of the aggressive players. We also examined the efficiency of our algorithm by monitoring its achieved goodput, loss rate and fairness. Finally, we used the heuristic methodology of [1] to find symmetric NE for our game and then evaluated its efficiency.

This methodology is executed in iterations. In the first iteration, we set $\alpha^1 = 1$ for flows $1, \ldots, n-1$ and search for the best response of flow n. Let $\alpha^{1,best}$ be the value α, with which n achieves the best goodput. In the next iteration, flows $1, \ldots, n-1$ play with $\alpha^2 = \alpha^{1,best}$ and we search for the best α_n in this profile. If at iteration k, $\alpha^{k,best} = \alpha^k$ then this value, denoted by α_E, is the SNE of the game.

Furthermore, we defined the Normalized Fairness Index (NFI) which is the Fairness Index normalized to the MaxMin Fairness bandwidth allocation, in order to measure the distance between the bandwidth allocation of Prince and MaxMin. The NFI is given by:

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \frac{(\sum_{i=1}^{n} \frac{x_i}{y_i})^2}{n \sum_{i=1}^{n} (\frac{x_i}{y_i})^2}$$

where x_i is the goodput of the i-th flow using the under examination algorithm and y_i is the goodput of the same flow achieved with MaxMin (DRR implementation).

We selected a simple dumbbell topology with two set of parameters. The first set (Topology 1) defines a topology with a bottleneck connection of 10Mbps/10ms (Bandwidth/Delay) and source/sink connections of 10Mbps/1ms. The queue size of the congested router is set to the Bandwidth$\times$Delay product (BWxD), which is 25 packets. The second set (Topology 2) uses a topology with bigger capacity; 100Mbps connections. For this set, the queue size is 100 packets, which is significantly less than BWxD packets (250), in order to examine the effectiveness of Prince under limited information.

The number of the players in the game was 10 for the first topology and in the range $10 \ldots 100$ for the second one. The players were TCP, UDP or mixed TCP and UDP flows. The TCP flows could define their strategy by selecting the value for the additive increase parameter α from 1 (standard TCP value) to 20. We have chosen the TCP SACK version for the implementation of the loss recovery mechanism because it is widespread and tolerant to packet losses. UDP flows can define their strategy by selecting their constant sending rate from the fair share value to the bottleneck's bandwidth value.

We evaluated the performance of Prince and compared it to MaxMin, DropTail, RED and CHOKe. For MaxMin and RED we used the default implementations of ns2 while for CHOKe (which was not available) we used the implementation from [20]. Each experiment starts with a 10sec period for stabilization and continues with 100sec for measurements. The flows start randomly between $0 \ldots 1$ sec and use a constant packet size of 1Kbyte. The minimum and maximum thresholds for RED and CHOKe were set automatically, depending on the link bandwidth and delay. The ideal MaxMin Fairness policy was represented by DRR (Deficit Round Robin) [9] with the number of queues equal to the number of players. The number of counters for Prince-A was set according to the queue size and number of flows of each experiment; in the

following figure legends, the number of counters used appears parenthesized.

B. Results

1) Synthesis of TCP flows : This synthesis was examined with both topologies and various aggressive players. Using Topology 1 we ran experiments with nine standard TCP players and an aggressive one that changes his additive increase parameter α from 1 to 20 in a series of identical games. The results showed that the aggressive player gains at most 15% more than his fair share under Prince-G and Prince-A, and at most 25% under Prince-S (Figure 2). Note the inability of DropTail to restrict the aggressive flow. RED has similar performance to DropTail and is omitted from the figure for clarity.

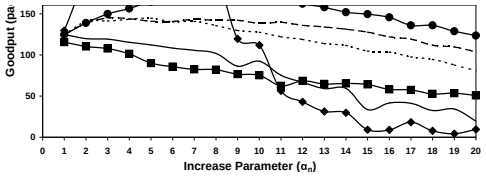

Fig. 2. Goodput of the aggressive TCP player

With MaxMin or CHOKe the aggressive player has goodput below his fair share for all α values except $\alpha = 1$, but the loss rate for CHOKe is higher (over 10%) than Prince-G (max 5%) and the total goodput is lower (1150 versus 1250 packets/sec). Prince-G sets an upper bound to the goodput of each player and a lower bound which is close to the fair share. Therefore the Fairness Index of Prince-G is close to 1 regardless of the aggressiveness of the player (Figure 3). DropTail has similar performance to RED and is omitted from the figure for clarity.

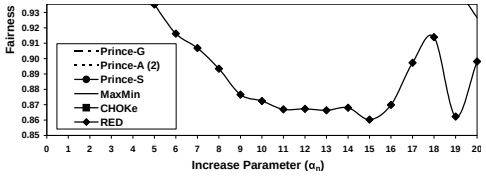

Fig. 3. Fairness Index

For Topology 2 with 99 standard TCP players and an aggressive one, all variants of Prince manage to track and restrict the selfish player (Figure 4), having similar loss rate and goodput with MaxMin and CHOKe. A direct comparison of Prince-G to MaxMin (Figure 5) showed that the difference between the goodput of the standard and the aggressive player is lower under Prince-G, achieving a better Fairness Index.

Furthermore, we performed additional experiments with larger numbers of aggressive players and found that Prince-G's performance advantage increases. For the same topology with 90 standard and 10 aggressive TCP players, Prince-A achieves to moderate the aggressive players despite using only 10 counters. Prince-G and Prince-S can easily detect

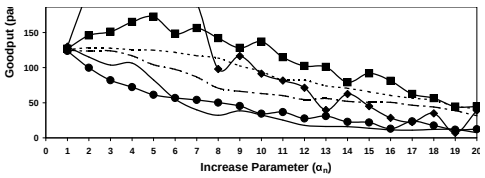

Fig. 4. Goodput of the aggressive TCP player

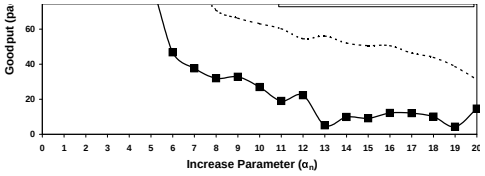

Fig. 5. Prince-G Vs. MaxMin

the aggressive flows. This is due to the fact that standard players are more rarely the majority players when many greedy players participate, so their fair share is guaranteed. Moreover, the more aggressive a player is, the easier it is for Prince to protect the standard players. On the contrary CHOKe fails when many selfish flows participate and the deficiency of RED and Droptail is also obvious on Figure 6.

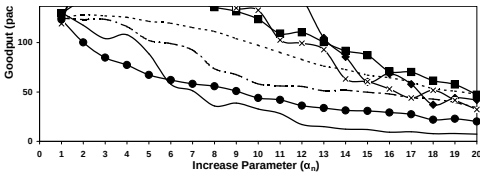

Fig. 6. Average Goodput of the aggressive TCP players

On Figure 7 a direct comparison of Prince-G and CHOKe is depicted. Prince-G shields the fair share of the standard players no matter how aggressive the players are. As the aggressive players increase their parameter α the difference between them and the standard players becomes more pronounced and thus Prince-G can more easily safeguard the latter. CHOKe seizes the selfish players only when they choose high values for parameter α ($\alpha > 10$).

Prince-A is highly effective when many selfish TCP flows are traversing the same bottleneck. The convergence of the goodput between the standard and the aggressive flow is depicted on Figure 8. For $\alpha < 8$, Prince-A allocates equally the bandwidth between standard and aggressive flows, while RED encourages players to behave greedily.

2) Synthesis of UDP flows : For Topology 1, we use nine UDP players with sending rate equal to their fair share (1Mbps) and one aggressive player that chooses his rate in the range $1 \ldots 10$ Mbps for each game. It is evident that only Prince-G and MaxMin can minimize the greedy player, while

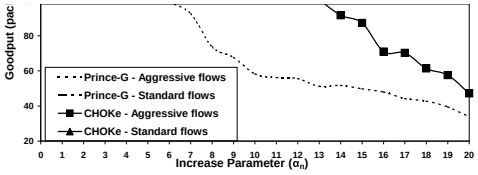

Fig. 7. Prince-G Vs. CHOKe

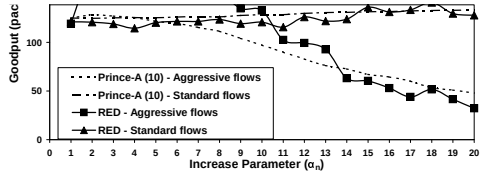

Fig. 8. Prince-A Vs. RED

DropTail and RED fail (Figure 9).

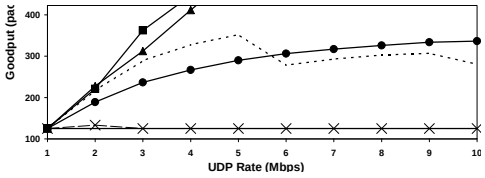

Fig. 9. Goodput of the aggressive UDP player

Prince-S has identical performance to Prince-G and is omitted. CHOKe does not effectively minimize the selfish player, therefore standard players suffer losses. A UDP flow sending at the fair share cannot be the majority player in the buffer. Therefore, Prince-G shields its fair share and achieves a Fairness Index equal to 1 (> 0.99), just like MaxMin (Figure 10).

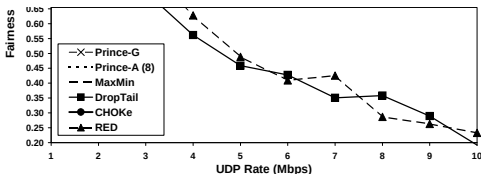

Fig. 10. Fairness Index

For Topology 2, we used 90 standard UDP players and 10 aggressive players that choose their rate in the range $1 \ldots 100$ Mbps for each game. The effectiveness of Prince-G is depicted in Figure 11, where the fair share of the standard UDP players is shielded even better than by MaxMin. For MaxMin, the goodput of the standard players is less than the fair share because the queue capacity is less than the BWxD product.

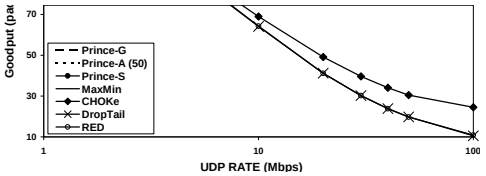

Fig. 11. Goodput of the standard UDP player

3) Mixed synthesis of TCP and UDP flows : It is important to examine the efficiency of our queueing mechanism with diverse player sets. Therefore, in Topology 1, we use four standard TCP players ($\alpha = 1$) and four standard UDP players (1Mbps) as well as one aggressive TCP player with $\alpha = 2$ and one aggressive UDP player with a 10Mbps sending rate. In Figure 12, we see that Prince resembles MaxMin Fairness for the aggressive UDP player, unlike DropTail, RED and CHOKe.

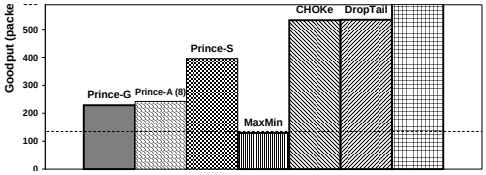

Fig. 12. Goodput of the aggressive UDP player

Moreover, the aggressive TCP flow is also limited to the fair share (Figure 13). With RED and CHOKe all the TCP players are deprived of their fair share (equal to 125 packets/sec), while with DropTail the aggressive TCP player obtains 30% more than his fair share.

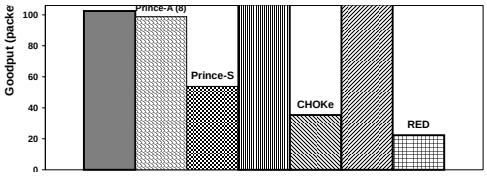

Fig. 13. Goodput of the aggressive TCP player

The convergence of Prince-G to MaxMin is more clear by using the Normalized Fairness Index, shown in Figure 14. Moreover, Prince-G ensures a fair allocation of bandwidth to all the players and as a consequence achieves a high Fairness Index.

4) NE results : We used the aforementioned methodology to heuristically find a symmetric NE of the game, with either only TCP or only UDP flows. For the TCP game, a part of the results can be deduced directly from Figure 2. If the

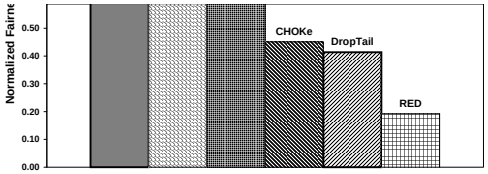

Fig. 14. Normalized Fairness Index

mechanism of the game is Prince-G, Prince-A or MaxMin, then the player has nothing to gain by increasing his additive increase parameter α beyond the standard TCP value. For Prince-S, CHOKe, RED and DropTail, the derived NE are less desirable due to the high loss rate and the slightly reduced goodput (Figure 15).

Queue Policy	α_E	goodput packets/sec	loss rate (%)
Prince-G	1	124,9	5,62
Prince-S	1	123,0	12,12
Prince-A	1	124,9	5,65
MaxMin	1	124,9	5,71
DropTail	2	122,5	8,26
RED	2	122,8	8,19
CHOKe	2	121,7	9,26

Fig. 15. Efficiency of NE with TCP players

For the UDP game, MaxMin, Prince-G and Prince-S lead to efficient and fair NE because a player has equivalent performance for almost every available sending rate (Figure 9). The other queueing policies fail to control the aggressive players, so the game results in an unfair and inefficient NE (Figure 16).

Queue Policy	sending rate (Mbps)	goodput packets/sec	loss rate (%)
Prince-G	1	125,0	0,0
Prince-S	1	125,0	0,0
Prince-A	5	125,0	79,51
MaxMin	1	125,0	0,0
DropTail	10	125,0	89,12
RED	10	125,0	89,89
CHOKe	10	125,0	89,99

Fig. 16. Efficiency of NE with UDP players

5) Comparison: The three variants of Prince express the same game theoretic idea but do not always achieve equivalent results. Prince-G adopts a moderate treatment to limit aggressive flows, so it leads the game to a desirable NE. Prince-A can achieve similar performance to Prince-G, despite its stateless implementation, in certain problem classes. It allows us fine grained control over the complexity/performance trade-off, by selecting the desired number of counters. When the number of counters reaches the upper limit, i.e., the maximum queue size, then we obtain a streaming version of Prince-G. Prince-S features lower computational complexity than Prince-G at the expense of increased loss rate at the NE due to the aggressive penalization of the majority flow.

We ran experiments to evaluate whether Prince-S is computationally less intensive than Prince-G. At the same time we examined the severity of Prince-S, namely, how often Prince-S drops a packet from the last majority flow even though the majority flow has in the meantime changed to another flow. The following Figures (17, 18 and 19) show how many packets were dropped with Prince-S in relation to the aggressiveness of the greedy flow(s). In particular, the third column shows how many already marked packets were dropped and originated from the current majority flow. The fourth column shows the same, except that these packets were dropped from a flow that is no longer the majority flow. Finally, the last column shows how many times Prince-G ran on behalf of Prince-S, i.e. no marked packets existed in the queue.

For the TCP synthesis on Topology 1 (Figure 2) we can discern that althought Prince-S restricts the aggressive player less than Prince-G, it also needs to compute the majority flow 60% less often than Prince-G. (Figure 17).

Increase parameter α_n	Loss rate	Dropped packets		
		Prince-S (hit on the majority flow)	Prince-S (hit on a non majority flow)	Prince-G (deployed by Prince-S due to lack of marked packets)
1	5,5%	1863	3008	3218
2	5,9%	2132	3316	3308
3	6,4%	2371	3504	3334
4	6,7%	2479	3779	3374
5	6,9%	2717	4012	3354
6	7,2%	2834	4233	3289
7	7,5%	3017	4309	3307
8	7,7%	3163	4504	3305
9	7,7%	3243	4467	3311
10	8,0%	3373	4513	3266
11	8,0%	3463	4599	3248
12	8,1%	3759	4543	3217
13	8,3%	3892	4683	3212
14	8,2%	3778	4455	3209
15	8,1%	3858	4559	3255
16	8,2%	3920	4583	3205
17	8,3%	3821	4438	3145
18	8,2%	3867	4395	3195
19	8,1%	3997	4384	3164
20	8,1%	3874	4325	3152

Fig. 17. Prince-S with TCP synthesis

For the UDP synthesis, Prince-S has the same efficiency as Prince-G on limiting the aggressive flow. The results (Figure 18) for this corner case show that Prince-S periodically deploys Prince-G (from 33% to 6% in inverse proportion to the aggressiveness of the UDP flow) while the effect is the same. The column which shows the hits on a non majority flow is replaced by the Prince-G deployment percentage column, because a standard CBR flow cannot be marked as a majority flow (never exceeds its fair share). Finally, in the

UDP rate (Mbps)	Loss rate	Dropped packets		
		Prince-S (hit on the majority flow)	Prince-G (deployed by Prince-S due to lack of marked packets)	Prince-G deployment percentage
1	0 %	0	0	0%
2	9,1%	9075	4538	33,3%
3	16,6%	20489	6830	25,0%
4	23,0%	34110	6822	16,6%
5	28,5%	47756	6828	12,5%
6	33,3%	61436	6835	10,0%
7	37,5%	73735	8201	10,0%
8	41,1%	86318	9087	9,5%
9	44,4%	102216	7205	6,6%
10	47,4%	112771	10254	8,3%

Fig. 18. Prince-S with UDP synthesis

mixed synthesis the deployment of Prince-S succeeds in the restriction of the aggressive TCP flow but fails to diminish the fairly greedy UDP flow (Figure 12). However, its effectiveness is quite qood considering that it executes Prince-G for only 10% of the dropped packets (Figure 19).

Loss rate	Dropped packets		
	Prince-S (hit on the majority flow)	Prince-S (hit on a non majority flow)	Prince-G (deployed by Prince-S due to lack of marked packets)
42,2%	78392	11927	7715

Fig. 19. Prince-S with mixed synthesis

C. Multiple Flows

In the experiments we implicitly assumed that every network flow is considered to be a selfish player that seeks to optimize its utility function. One can consider all packets originating from the same IP address or the same subnet address to belong to the same selfish player. This would ensure that a user/player that can launch multiple flows concurrently (for any reason) will not be able to obtain an unfair part of the network bandwidth in total. Moreover, the impact of multiple flows per user on the fairness of the network is a general issue discussed for example in [21]. In general, it should be possible to apply any other successful approach to handle this issue (beyond the simplistic grouping of flows) to the Prince algorithms.

VI. Conclusion

Based on our theoretical and experimental results, the following features of Prince emerge:

- It allocates bandwidth to each player close to his fair share.
- It leads to efficient NE with high goodput and low loss rates.
- It sustains its high performance even in the presence of multiple aggressive TCP or unresponsive high-rate UDP flows.
- It exhibits the positive side-effect of avoiding both the synchronization and the starvation of flows.

The previous features make us confident that Prince, besides being simple, is highly effective.

The basic game-theoretic idea of Prince, targeting and restricting the majority flow, yielded interesting results. A secondary outcome is that fair *buffer* sharing can result in fair *bandwidth* sharing.

Our future endeavors include examining hybrid variants of Prince in order to further optimize its computational performance. Additionally, we need to examine the behavior of Prince in complex network topologies and heterogeneous router compositions. Finally, we are working on an optimized version of Prince-G for packet streams.

References

[1] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. Selfish behavior and stability of the internet: a game-theoretic analysis of tcp. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130, New York, NY, USA, 2002. ACM Press.

[2] Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable Worst-Case performance and experimental results. In *ICALP*, volume 5555 of *LNCS*, pages 107–118. 2009.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM.

[4] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In Paterson M., editor, *ICALP*, volume 443 of *LNCS*, pages 6–19. Springer Berlin / Heidelberg, 1990.

[5] D. Dutta. Oblivious aqm and nash equilibria. In *INFOCOM 03*, 2003.

[6] P.S. Efraimidis and L. Tsavlidis. Window-games between tcp flows. In *Lecture Notes in Computer Science (SAGT 2008)*, volume 4997, pages 95–108, Springer-Verlag Berlin Heidelberg, 2008.

[7] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[8] X. Gao, K. Jain, and L. J. Schulman. Fair and efficient router congestion control. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1050–1059, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[9] E. L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal of Selected Areas in Communications*, 9(7):1024–1039, 1991.

[10] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.

[11] F. P. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *Journal of Operation Research*, 49(3):237–252, 1998.

[12] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. *Network Protocols, 2001. Ninth International Conference on*, pages 192–201, Nov. 2001.

[13] J. B. Nagle. *On packet switches with infinite storage*. Artech House, Inc., Norwood, MA, USA, 1988.

[14] NS-2. The network simulator. http://www.isi.edu/nsnam/ns/.

[15] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *SIGCOMM Comput. Commun. Rev.*, 33(2):23–39, 2003.

[16] R. Pan, B. Prabhakar, and K. Psounis. Choke, a stateless active queue management scheme for approximating fair bandwidth allocation. In *INFOCOM*, pages 942–951, 2000.

[17] C. Papadimitriou. Algorithms, games, and the internet. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 749–753, New York, NY, USA, 2001. ACM Press.

[18] S. J. Shenker. Making greed work in networks: a game-theoretic analysis of switch service disciplines. *IEEE/ACM Trans. Netw.*, 3(6):819–831, 1995.

[19] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. *SIGCOMM Comput. Commun. Rev.*, 28(4):118–130, 1998.

[20] T. Wang. Choke source code for ns2. http://cs-people.bu.edu/wtwang/paper/.

[21] J. Arnaud, B. Bob and M. Toby. Policing freedom to use the internet resource pool. Proceedings of the 2008 ACM CoNEXT Conference, CoNEXT '08.

[22] R.-A. Koutsiamanis, and P.S. Efraimidis. An exact and O(1) time heaviest and lightest hitters algorithm for sliding-window data streams. In *MUE '11: Proceedings of the 2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*, pages 89–94, Washington, DC, USA, 2011. ACM.