FCP: A Framework for an Evolvable Transport Protocol

Dongsu Han **Robert Grandl**[†] **Aditya Akella**[†]
Srinivasan Seshan

May 2012
CMU-CS-12-125

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†] University of Wisconsin-Madison, Madison, WI, USA

This research was funded in part by the National Science Foundation under award number CNS-1040801.

Keywords: transport protocols, transport evolution, explicit congestion control, proportional fairness, flexible resource allocation, quality of service, evolution of congestion control

Abstract

Transport protocols must accommodate diverse application needs as well as network requirements. As a result, TCP has evolved over time with new congestion control algorithms such as support for generalized AIMD, background flows, and multipath. On the other hand, explicit congestion control algorithms have shown to be more efficient. However, they are inherently more rigid because they rely on in-network components. Therefore, it is not clear whether they can evolve to support diverse application needs at least as much as TCP. This paper presents a novel framework for network resource allocation that supports evolution by accommodating diversity and exposing a flexible but simple abstraction for resource allocation. The core idea is to design a resource allocation scheme that allows aggregation and local control. To implement this idea, we leverage ideas from economics-based congestion control (but without actual congestion charging) with explicit virtual price feedback. We show that our design, FCP, allows evolution by accommodating diversity and ensuring coexistence, while being as efficient as existing explicit congestion control algorithms.

1 Introduction

Networked applications often require a wide range of communication features, such as reliability, flow control, and in-order delivery, in order to operate effectively. Since the Internet provides only a very simple, best-effort, datagram-based communication interface, we have relied on transport protocols to play the key role to implementing the application's desired functionality on top of the simple Internet packet service. As application needs and workloads have changed over time, transport protocols have evolved to meet their needs. In general, changes to transport protocols, such as adding better loss recovery mechanisms, have been relatively simple since they require only changes to the endpoints. However, among the functions that transport protocols implement, congestion control is unique since it concerns resource allocation, which requires coordination among all participants using the resource. As a result, two very different styles of congestion control cannot coexist, making evolution of congestion control difficult.

The need for evolution in congestion control becomes apparent when we look at the history of TCP. While TCP's congestion control was not designed with evolution in mind, the development of TCP-Friendliness principles [14] enabled the development of a wide range of congestion control techniques to meet different application requirements; these include support for: streaming applications that require bandwidth guarantees [14, 9, 5, 29], or low-latency recovery [26, 20], non-interactive applications that can leverage low-priority, background transfer [39], applications that require multi-path communication for robustness [41], and Bittorrent-like content transfers that involve in transferring chunks from multiple sources.

The ability for TCP's congestion control to evolve has been enabled by two key aspects of its design: 1) Purely end-point based nature allowed new algorithms to be easily deployed and 2) its AIMD-based congestion avoidance led to the notion of TCP-friendliness.

Unfortunately, recent efforts, such as RCP [13] and XCP [22], rely on explicit congestion feedback from the network. While these designs are far more efficient than TCP, they limit the range of different end-point application behaviors that the network can support. It is not understood whether such explicit congestion control algorithms can be made flexible to allow evolution. In this paper, we explore the design of FCP (flexible control protocol), a novel congestion control framework that is as efficient as explicit congestion control algorithms (e.g., RCP and XCP), but retains (or even expands) the flexibility of TCP-friendliness based solutions.

FCP leverages ideas from economics-based congestion control [24, 23] and explicit congestion control. In particular, to enable flexible resource allocation with in a host, we allow each domain to allocate resources (budget) to a host, and make networks *explicitly* signal the congestion price. The flexibility comes from the end-points being able to assign their own resources to their flows and the networks being able to aggregate flows and assign differential price to different classes of flows to provide extra functionality. The system maintains a key invariant that the amount of traffic a sender can generate per unit time is limited by its budget and the congestion price. Co-existence of different styles and strategies of rate control is ensured simply by maintaining this key invariant, allowing evolution.

Our primary contribution is showing that explicit congestion control algorithms can be made flexible enought to allow evolution just as end-point based algorithms. To this end, we design an explicit congestion control algorithm, called FCP, and demonstrate that FCP easily allows different

features to coexist within the same network; end-hosts can implement diverse styles of rate control and networks can leverage differential pricing and aggregate control to achieve different goals.

To make economics-based congestion control [24, 23, 25] practical, this paper extends past theoretical work in three critical ways:

- FCP uses "pricing" as a *pure abstraction* and the key form of *explicit feedback*. In contrast, others [24, 8, 25, 11] directly associate congestion control to the real-world pricing and modify existing congestion control algorithms to support weighted proportional fairness.
- We address practical system design and resource management issues, such as dealing with relative pricing, and show that FCP is efficient.
- Finally, unlike previous explicit rate feedback designs [13, 23], FCP accommodates high variability and rapid shifts in workload, which is critical for flexibility and performance. This property of FCP comes from a novel *preloading* feature that allows senders to commit the amount of resource it wants to spend ahead of time.

In the remaining sections, we present related work (§2) and motivate our design (§3). We then present our detailed design and how it allows evolution (§4), discuss practical issues in deployment (§5), evaluate FCP (§6), and conclude in §7.

2 Related Work

Our framework builds upon concepts from previous designs to provide a generalized framework for resource allocation.

Economics-driven resource allocation has been much explored in the early days of the commercial Internet [31, 34, 10]. MacKie-Mason and Varian [28] proposed a *smart market* approach for assigning bandwidth. Since then many have tried to combine economics-driven resource allocation with congestion control, which led to the development of economics-based congestion control.

Kelly [23] proved that when users choose the charge per unit time that maximizes their utility, the rate can be determined by the network so that the total utility of the system is maximized, and the rate assignment satisfies the weighted proportional fairness criterion. It also proved that two classes of globally stable algorithms can achieve such rate allocation: 1) The primal algorithm implicitly signals the congestion price, and the senders use additive increase/multiplicative decrease rules to control rate. 2) The dual algorithm uses explicit rate feedback based on shadow pricing.

The primal algorithms use end-point based congestion control [11, 8, 18]. MulTCP [11], for example, adjusts the aggressiveness of TCP proportional to the host's willingness to pay. Gibbens and Kelly [18] show how marking packets (with an ECN bit) and charging the user the amount proportional to the number of marked packets achieves the same goal, and allows evolution of end-point based congestion control algorithms. The dual algorithm uses rate as an explicit feedback, and was materialized by Kelly et al. [25] by extending RCP [13].

FCP also adopts proportional fairness, but is different from these approaches in that 1) it uses congestion pricing as a pure abstraction, decoupling from real-world pricing and billing, and 2) it is an explicit congestion control algorithm that is designed for supporting evolution using price as an explicit feedback. FCP also addresses practical issues of implementation and deployment that others do not address [23, 25], such as differential pricing and accommodating large fluctuations in workload.

Congestion control with various features: We categorize works that introduce new features into congestion control in §3.1. Our work enables such diverse features to coexist by using pricing as the core abstraction in the common congestion control framework and by allowing aggregation and local control, generalizing the idea first explored in CM [3].

Extensible transport: Others focus on designing an extensible [32, 7] or configurable [6] transport protocol. These works either focus on security aspects of installing mobile code at the end-host or take software engineering approaches to modularize transport functionality at compile time. However, the core mechanism for coexistence is TCP-friendliness.

Virtualization: Finally, virtualization [1, 35] partitions a physical network allowing completely different congestion control algorithms to operate within each virtualized network. Although this is good for creating a testbed for new networking technologies and protocols that run in isolation [35], it is not meant to support coexistence of diverse protocols in reality and has a number of practical limitations. Slicing bandwidth creates fragments and reduces the degree of statistical multiplexing, which we rely on to provision resources. Also, this increases the complexity of applications and end-hosts who want to use multiple protocols simultaneously as they have to participate in multiple slices and maintain multiple networking stacks.

3 Motivation

In this section, we first identify two key requirements for evolution and closely look at each of them (§3.1). We then describe the two key principles that the FCP design uses to satisfy these requirements (§3.2).

3.1 Requirements for evolution

To support evolution, the system must **accommodate diversity** such that different algorithms and policies coexist and be **flexible** enough that new algorithms can be implemented to accommodate (future) changes in communication patterns.

Accommodating diversity: To understand the nature of diversity, we categorize previous work on network resource allocation into four categories:

1. Allocating resources locally: Prior works, such as congestion manager [3], SCTP [30], and SST [16], have shown benefits of flexible bandwidth sharing across (sub) flows that share a common path.

2. Allocating resource within a network: Support for differential or weighted bandwidth allocation across different flows has been explored previously. Generalized AIMD [42, 4], weighted fair-sharing [11, 12], and support for background flows [39] are some examples.

3. Allocating resource in a network-wide fashion: Bandwidth is sometimes allocated on aggregate flows or sub-flows: Multipath TCP (MPTCP) [41] controls the total amount of bandwidth allocated to its sub-flows; distributed rate limiting [33] and assured forwarding in DiffServ control resource allocation to a group of flows. In such systems, flows that do not traverse the same path are allocated a resource that is shared among themselves (e.g., MPTCP is fair to regular TCP at shared bottlenecks). In such designs, one can be more aggressive in some parts of the network at the expense of being less aggressive in other parts [33].

4. Stability in resource allocation: Although strict bandwidth or latency guarantees require in-network support, many transport designs aim to provide some level of performance guarantees [14, 38]. For example, TFRC is designed such that the throughput variation over consecutive RTTs is bounded and OverQoS [38] provides similar but probabilistic guarantees.

Flexibility is the key to accommodating as yet unforeseen communication styles of the future. To achieve this, more control should be exposed to the end-host as well as to the network. Furthermore, a scalable mechanism is needed to exert control over aggregate flows—a scheme that requires per-flow state in the network for control would not work.

Note that apart from these requirements for evolution, there are also common, traditional goals such as high efficiency, fast convergence, and fair bandwidth allocation. Many works [19, 22, 13, 37, 15], such as TCP-cubic, XCP, and core-stateless fair-queuing, fall into this category. Our goal is to design a flexible congestion control framework that accommodates above-mentioned diversity and flexibility, while still achieving these traditional goals.

3.2 Principles of design

FCP employs two key principles for accommodating diversity and flexible resource allocation: *aggregation* and *local control*.

Aggregation: FCP assigns resources to a group of flows. Aggregation allows the network to control the amount of resources that the group is consuming in a distributed and scalable fashion while preserving the relative weight of the individual flows within the group. For example, we assign resources (a budget) to a host so that the host can generate traffic proportional to the amount of budget it has. Aggregation also simplifies control and enforcement; the network can enforce that a host is generating traffic within its allocated resources without having to keep per-flow state. For additional flexibility, we allow aggregation at various levels. As networks aggregate flows that

come from a host, ISPs can also aggregate traffic coming from another ISP and assign resources. For example, each domain can independently assign weights to the neighboring domains traffic or to any group of flows. As we show in §4, such aggregation provides a mechanism for the network to ensure fairness while allowing flexibility at the end-host, one of the key ingredients for evolution.

Local control: Local control gives freedom to distribute the resource to individual flows, once resources are allocated on aggregate flows. For example, when the network assigns resources to the aggregate flows from a single host, the host can control how to distribute the resource locally amongst its flows (diversity category 1 in §3.1). The network then respects the resource (or weight) given to the flow and assigns bandwidth proportional to its weight (category 2 support). A host can also spend more resources on some flows while spending less on others (diversity category 3 support). Various points in the network may also control how their own resources are shared between groups of flows. For example, networks can allocate bandwidth in a more stable manner to certain group of flows (category 4 support).

As such, both the end-host and the network have local control, ensuring flexibility; the network decides how its own resource is used within its domain, and end-hosts decide how to use their own resources given the network constraint.

Feedback design: Designing the form of network to end-point feedback that meets both requirements is challenging. Providing an absolute feedback as in XCP or RCP leaves no local control at the end-point because the feedback strictly defines the end-point's behavior. Using an abstract or implicit feedback, such as loss rate or latency, or loosely defining the semantics of feedback, on the other hand, increases end-point control, allowing a range of end-host behaviors. However, using such feedback typically involves guesswork. As a result, end-hosts end up probing for bandwidth, which in turn sacrifices performance and increases the convergence time (e.g., in a large bandwidth-delay product link). Providing differential feedback for differential bandwidth allocation is also hard in this context. As a result, differential bandwidth allocation typically is done through carefully controlling how aggressively end-points respond to feedback relative to each other [42]. This, in turn, makes enforcement and resource allocation on aggregate flows very hard. For example, for enforcement, the network has to independently measure the implicit feedback that an end-point is receiving and correlate it with the end-point's sending rate.

We take a different approach by leveraging ideas from Kelly [23, 18]. We use pricing as a form of explicit feedback and design an evolvable explicit congestion control algorithm by supporting flexible aggregation and local control. Contrary to the common belief, we demonstrate explicit congestion control can be made flexible to allow evolution.

4 Design

We now describe our design in steps.
(1) Each sender (host) is assigned a budget ($\$/sec$), the maximum amount it can spend per unit time. We first focus on how FCP works when the budget is assigned by a centralized entity. Our

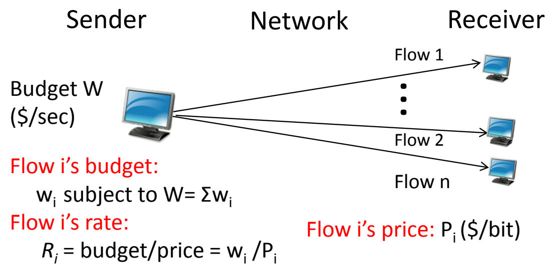


Figure 1: Design Overview

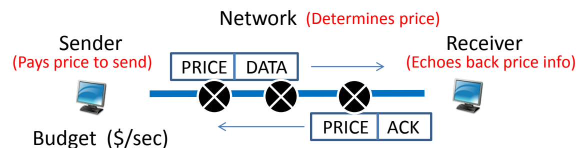


Figure 2: Network generates the price feedback.

budget defines the weight of a sender, but is different from the notion of "willingness to pay" in [23] in that it is not a real monetary value. Later, in §4.3, we extend FCP to support distributed budget management in which each domain can assign budgets completely independently and may not trust each other. The central challenge there is how to value a budget assigned by a different domain. We show that FCP allows flexible aggregate congestion control by dynamically translating the value of a budget assigned by different entities.

(2) At the start of a flow, a sender allocates part of its budget to the flow. This budget now determines the weight of the flow. End-point evolution is enabled at this stage because the sender can implement various strategies of resource allocation to achieve different goals (see §4.2).

Figure 1 illustrates this. The sender has a budget of W, and can distribute its budget to its flows at its will provided that the sum of the flows' budgets, $\sum w_i$, is less than or equal to W. The rate of flow i, R_i, is then defined as w_i/P_i where P_i is the price for the path flow i is traversing. This allows the end-host to control its own resource and achieve differential bandwidth allocation on a flow-by-flow basis, as we show in §4.2. FCP also allows this budget assignment to change at any time in the lifetime of a flow, providing additional flexibility. As such, we reintroduce the end-point's flexibility of end-point based designs to an explicit congestion control algorithm. Next, we show how the path price is generated.

(3) The network determines the congestion price ($\$/bit$) of each link. In FCP, the sender learns the path price in the form of explicit feedback. The price of path r, P_r, is defined as the sum of link prices: $P_r = \sum_{l \in r} p_l$, where p_l is the price of link l in path r. To ensure efficiency, the price adapts to the amount of budget (traffic times its price) flowing through the system and its capacity. In §4.1, we first show a uniform pricing scheme in which all packets see the same price. In §4.2, we show how networks can employ differential pricing and aggregate congestion control to support various features such as quality of service, aggregate resource allocation, and multicast-aware congestion control.

(4) The flow now ramps up to its fair-share, R_i, by spending its budget w_i. Two problems remain at this stage: At the start of the flow the sender does not know the path price, P_i. More importantly, the network does not know the budget amount (w_i) it should expect for accurate price generation. This leads to serious problems when the workload changes rapidly. For example, when a path is not congested, its congestion price will be an infinitesimal value, ϵ. Any $w_i >> \epsilon$ will set the sending rate to an infinite value, and overload the network significantly. To address this problem,

we introduce *preloading*, a distinct feature of our design that allows a host to rapidly increase or decrease a flow's budget. Preloading allows the sender to specify the amount of budget increase for the next round in multiples of the current price, allowing the network to adjust the expected input budget and generate prices based on this committed budget. The challenge here is to accurately calculate price using this information and for hosts and routers to update the budget w_i and the price P_i in a coordinated fashion so that w_i is only spent when the network is expecting it. This is especially challenging when the system is asynchronous and feedback is delayed. For this, in FCP, routers update the price on every packet reception, and hosts preload on a packet-by-packet basis when the budget changes (e.g., initial ramp up). In §4.1, we show how our pricing works with preloading and how the sender updates its budget, w_i.

The sender now ramps up in two steps using preloading:
(4-1) Along with the first packet, we preload how much we want to send in the next RTT. However, at this point we do not yet know the path price. Therefore, we preload a conservative amount. (See §4.1 for details.)
(4-2) After the price is discovered, the sender can preload the appropriate amount to spend the budget assigned to the flow. Preloading is also used when an active flow's budget assignment changes.

4.1 Flexible Control Framework

We now describe the details. In FCP, the system maintains the following *invariant*: For all hosts h,

$$\sum_{s \in Packets} price(s) \cdot size(s) \leq W_h \tag{1}$$

where *Packets* is the set of packets sent by host h during unit time, price(s) is the most recent price of the path that packet s is sent through, and W_h is host h's budget.

Explicit price feedback: To satisfy the invariant, senders have to know the path price. As Figure 2 illustrates, an FCP header contains a price value, which gets accumulated in the forward path and echoed back to the sender. Each router on the path updates the price in the header as: $price = price + p_l$, where p_l is the egress link price.

Pricing ensures network efficiency by dynamically adapting the price to the amount of incoming budget. We show how each router calculates the link price to achieve this. Each link's price must reflect the amount of incoming budget and the capacity of the link. Each router calculates the link price per bit $p(t)$ ($\$/bit$) at time t as:

$$p(t) = \frac{I(t)}{C - \alpha q(t)/d} \tag{2}$$

$$I(t) = \frac{\sum_{s \in (t-d,t]} p(t - rtt(s)) \cdot size(s)}{d} \tag{3}$$

Equation 2 sets the price as the incoming budget (amount of traffic times its price) over remaining link capacity. $I(t)$, the numerator, denotes the total incoming budget per unit time ($\$/sec$), which is the sum of all packets' prices ($) seen during the averaging window interval (t-d,t]. The denominator reflects the remaining link capacity, where C is the link-capacity, $q(t)$ is the instantaneous queue size, α is a constant, and d is the averaging window. $rtt(s)$ is the RTT of the packet s, and $p(t - rtt(s))$ is the past feedback price per bit. Unlike other explicit congestion control protocols [22, 13], the router calculates the link price at every packet reception, and keeps the time series $p(\cdot)$. We set d as multiples of average RTT (twice the RTT in our implementation).

Equation 2 deals with efficiency control by quickly adapting the price based on the incoming budget and the remaining link capacity. Fairness is achieved because everyone sees the same price, and therefore the bandwidth allocation is proportional to budget. In §6.1, we show simulations in various environments to demonstrate the stability of this algorithm, and perform local stability tests by introducing perturbations.

Equation 3 estimates the amount of incoming budget that a router is going to see in the future using recent history. When incoming budget, $I(t)$, is relatively constant over time the price is stable. However, when the input budget constantly changes by a large amount, the price will also fluctuate because the fair-share rate also fluctuates. This, coupled with the inherent delayed feedback, can leave the system in an undesirable state for an extended period. During convergence, the network may see high loss rate, under-utilization, or unfairness. Other state-of-the-art congestion control frameworks also have this problem. For example, when new flows arrive RCP results in periods of high loss, and XCP results in slow convergence to fair-share, as we show in §6.1.

This problem has been regarded as being acceptable in other systems because changes are viewed as either temporary or incremental. Theory of economics-based congestion control [23] shows that even for a theoretical proof of global stability the analysis requires such an assumption when users can dynamically adapt their willingness to pay. However, this is not the case in our system. One of the key enablers of evolution in FCP is the end-host's ability to arbitrarily assign its budget to individual flows, and we expect that rapid change in the input budget will be the norm. Increasing budget rapidly also allows senders to quickly ramp up their sending rates to their fair-share. However, this is especially problematic when the amount of budget can vary by large amounts between flows. For example, consider a scenario where a link has an incoming budget of $1\$/sec$. When, a new flow with a budget of 1000 $\$/sec$ arrives, this link now sees 1001x the current load. Preventing this behavior and forcing users to incrementally introduce budget will make convergence (to fairness) significantly slower and limit the flexibility of the end-hosts.

Preloading: To address this problem, we introduce *preloading*, a distinct feature of our design that allows a host to rapidly increase or decrease the budget amount per flow. Preloading allows the sender to specify the amount of budget willing to introduce in the next round, allowing the network to adjust the expected input budget and generate prices based on this committed budget.

However, the sender cannot just specify the absolute budget amount that it wants to spend on the path because the path price is a sum of link prices. Given only the total amount, each individual link cannot estimate how much budget is spent traversing it. Instead, we let senders preload in multiples of the current price. For example, if the sender wants to introduce 10 times

more budget in the next round, it specifies the preload value of 10. For routers to take this preload into account, we update Equation 3 as follows:

$$I(t) = \frac{\sum_{s \in (t-d,t]} p(\cdot)\; size(s)\; (1 + preload(s) \cdot d/rtt(s))}{d} \tag{4}$$

The additional preload term takes account the expected increase in the incoming budget, and $rtt(s)$ accounts for the difference between the flow's RTT and the averaging window. Preloading provides a hint for routers to accurately account for rapid changes in the input budget. Preloading significantly speeds up convergence and reduces estimation errors as shown in §6.

Header: We now describe the full header format. An FCP data packet contains the following congestion header:

RTT	price	preload	balance

When the sender initializes the header, RTT field is set to the current RTT of the flow, price is set to 0, and preload to the desired value (refer to sender behavior below). The balance field is set as the last feedback price—i.e., the price that the sender is paying to send the packet. This is used for congestion control enforcement (§5). Price and preload value are echoed back by an acknowledgement.

Sender behavior with preloading: Senders can adjust the allocation of budget to its flows at any time. Let x_i be the new target budget of flow i, and w_i the current budget whose unit is $\$/sec$. When sending a packet, it preloads by $(x_i - w_i)/w_i$, the relative difference in budget. When an ACK for data packet s is received, both the current budget and the sending rate are updated according to the feedback. When preload value is non-zero, the current budget is updated as:

$$w_i = w_i + paid \cdot size(s) \cdot preload/rtt$$

where paid is the price of packet p in the previous RTT. The sending rate r_i is updated as: $r_i = w_i/p_i$, where p_i is the price feedback in the ACK packet. Note that preloading occurs on a packet by packet basis and the sender only updates the budget after it receives the new price that accounts for its new budget commitment. Also note that the preload can be negative. For example, a flow preloads -1 when it is terminating. Negative preloading allows the network to quickly respond to decreasing budget influx, and is useful when there are many flows that arrive and leave. Without negative preload, it takes the average window, d, amount of time to completely decay the budget of the flow that has departed.

Start-up behavior: We now describe how FCP works from the start of a flow. We assume a TCP-like 3-way handshake. Along with a SYN packet, we preload how much we want to send in the next RTT. However, because we do not yet know the path price, we do not aggressively preload. In our implementation of FCP, we adjust the preload so that we can send 10 packets

per RTT after receiving a SYN/ACK. The SYN/ACK contains the price. Upon receiving it, we initialize the budget assigned to this flow w_i as: $price \cdot size \cdot preload/rtt$. After this, the sender adjusts its flows' budget assignments as described earlier.

4.2 Evolution

FCP's common framework enables evolution by providing greater flexibility with local control and aggregation. We show a number of examples that demonstrate end-point and network evolution. Coexistence is guaranteed as long as end-hosts stick to the invariant of Equation 1, which defines the host-level fairness. Therefore, it is much easier to design various resource allocation schemes.

End-point evolution: End-hosts can assign budget to individual flows at will. This enables intelligent and flexible policies and algorithms to be implemented. Below we outline several strategies that end-hosts can employ.

- Equal budget: Flow-level fairness between flows within a single host can also be achieved by equally partitioning the budget between flows. For example, when there are n flows, each flow gets $budget/n$. Then the throughput of each flow will be purely determined by the path's congestion level.

- Equal throughput: End-points may want to send equal rates to all parties with which it is communicating (e.g., a conference call). This is achieved by carefully assigning budget (i.e., assigning more budget to expensive flows).

- Max throughput: The goal is to maximize throughput. To maximize throughput, end-points use relatively more budget towards inexpensive paths (less congested) and use little budget for more congested paths. This generalizes the approach used in multipath TCP [41] even to subflows sent to different destinations as we show in §6.

- Background flows, similar to those in TCP Nice [39], are also supported by FCP. A background flow is a flow that only utilizes "spare capacity". When foreground flows are able to take up all the capacity, background flows should yield and transmit at a minimal rate. This can be achieved by assigning a minimal budget to background flows. When links are not fully utilized, the price goes down to "zero" and path price becomes marginal. Therefore, with only a marginal budget, background flows can fill up the capacity.

- Statistical bandwidth stability: Some flows require a relatively stable throughput [14]. This can be achieved by reallocating budget between flows; if a bandwidth stability flow's price increases (decreases), we increase (decrease) its budget. When the budget needs to increase, the flow steals budget from other flows. This is slightly different from smooth bandwidth allocation given by TFRC in that temporary variation is allowed, but the average throughput over a few RTTs is probabilistically stable. The probability depends on how much budget can be stolen and the degree of path price variation. Such a statistical guarantee is similar to that of OverQoS [38]. Later, we show how we can achieve guaranteed bandwidth stability with network support.

Note that these algorithms are not mutually exclusive; they can coexisit as long as the invariant (Eq.1) is satisfied. Different hosts can use different strategies. Even within a host, different groups of flows can use different strategies.

Network and end-point evolution: End-points and networks can also simultaneously evolve to achieve a common goal. FCP's local control and aggregation allow routers to give differential pricing to different groups of flows. We show how this may support various features in congestion control. In our examples, algorithms change from the original design including the path price generation, link price calculation, and preloading behavior. However, the invariant of Equation 1 remains unchanged.

- Bandwidth stability: We implement a version of slowly changing rate with end-point cooperation and differential pricing: 1) For stability flows, end-points limit the maximum preload to 1 which limits the speed at which flow budget and rates can ramp up. 2) Routers have two virtual queues: stability and normal[1]. The stability queue's price variation during a time window of twice the average RTT is bounded by a factor of two. When the price has to go up more than that amount, it steals bandwidth from the normal queue to bound the price increase of the stability queue. As a result, the normal queue's price increase even more, but the bandwidth stability queue's price is bounded. When the price might go down by a factor of two during a window, it assigns less bandwidth to the stability queue. However, because we allow the stability queue's price to change, at steady state the prices of both queues converge to the same value. Below shows the algorithm in pseudocode.

```
function UPDATEPRICE(Packet p)
    // update average RTT
    // update total input budget, deadline queue's and best effort queue's input budget.

    // Calculate price according to Eq.2 using the total input budget
    price = calculatePrice(totalInputBudget, linkCapacity)
    if isOutofRange(price) then
        stabilityPrice = priceLimit(price) // Price increase/decrease is bounded
        // calculate BW needed to bound the price
        requiredBW = stabilityInputBudget/stabilityPrice/avgRTT/d
    else
        // calculate normal queue's price
        normalPrice = calculatePrice(nomalInputBudget, linkCapacity - requiredBW)
    end if
end function

function ISOUTOFRANGE(price)
    return true if the price is too high or too low.
    otherwise return false
```

[1]We assume that routers can easily classify packets into these queues

end function

- Multicast-aware congestion control: FCP allows the network to evolve to support a different service model. For example, FCP can support multicast-aware congestion control by changing how the path price is generated from link prices. We sum up the link price in a multicast tree, by aggregating the price information at the router. The price of a multicast packet is the sum of the price of links that it traverses. To calculate the price, we sum up the price of a packet in a recursive manner in a multicast tree. When a router receives a multicast packet, it remembers its upstream link price, resets the price feedback to zero, and sends out a copy of the packet to each of its outgoing interfaces along the multicast tree (See pseudocode for ReceiveMulticast below). This allows us to account for the link price only once. Receivers echo back the price information. Upon receiving an ACK, each router computes the price of its subtree and sends a new ACK containing the new price. The price of the subtree is computed by summing up the price of the link from its parent to itself that it remembered previously and all the price feedback from its children in the multicast tree (See pseudocode for ReceiveACK below). Through a recursive process, the sender at the root receives the total price of sending a packet down the multicast tree.

```
function ReceiveMulticast(Packet p, SourceAddress sa, MulticastAddress ma)
    upstremPrice[(sa, ma)] = p.price  // remember upstream link price
    p.price = 0  // reset price
    for all c in children[(sa,ma)] do
        send(p)
    end for
end function

function ReceiveACK(Packet p, SourceAddress sa, MulticastAddress ma)
    price = 0
    price += upstremPrice[(sa, ma)]
    subTreePrice[previousHop(p)] = p.price
    for all c in children[(sa,ma)] do
        price += subTreePrice[c]
    end for
    p.price = price  // SubTree's price
    sendDelayed(p)  // Limit the number of ACKs to prevent ACK implosion
end function
```

- FCP can also support Paris Metro Pricing (PMP) sytle [31] differential pricing. The network can assign higher price to a priority service in multiples of the standard price. For example, the priority queue's price can be 10 times the standard queue. The end-host will only use the priority queue when it is worth it and necessary making the priority queue see much less traffic that the standard one.

- D^3[40]-style deadline support can be implemented using differential pricing with two virtual queues: deadline queue and best effort queue. Assume the best effort queue is assigned a small

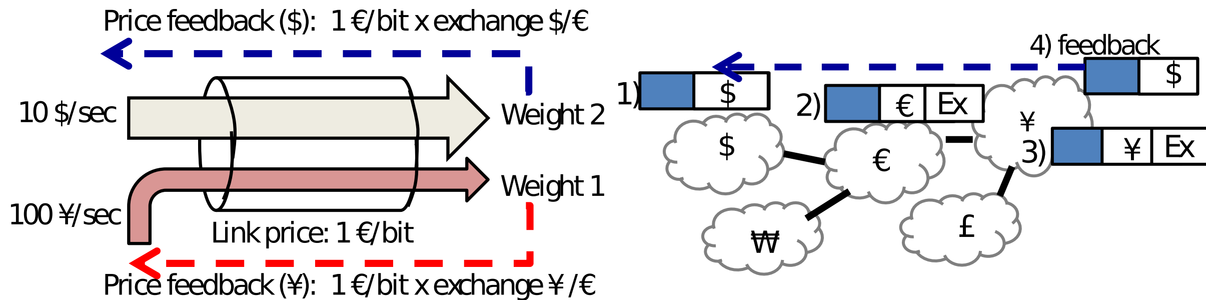


Figure 3: Budget management: (a) Aggregate congestion control using dynamic value translation. (b) Value translation between domains.

fraction of bandwidth at least. The deadline queue always gives a small agreed-upon fixed price. Flows indicate their desired rate by preloading. The router gives the fixed price only when the desired rate can be satisfied. Otherwise, it puts the packet into the best-effort queue and gives a normal price to control the aggregate rate to the bandwidth allocated to the normal queue. The host first tries deadline support, but falls back to best effort service when it receives a price different from the fixed value after the preloading. Below is the algorithm in pseudocode.

```
function UpdatePrice(Packet p)
    (omitted)... // update average RTT

    if p in Deadline flow then
        updateDeadlineInputBudget(p)
    else
        updateBestEffortInputBudget(p)
    end if

    requiredBW = stabilityInputBudget/fixedDeadlinePrice/avgRTT/d
    if requiredBW > linkCapacity then
        subtractDeadlineInputBudget(p)  // Demote the flow to best-effort flow
        updateBestEffortInputBudget(p)
    end if

     // calculate best-effort queue's price
    bestEffortPrice = calculatePrice(bestEffortInputBudget, linkCapacity - requiredBW)
end function
```

4.3 Budget Management

Budget management is another form of flexibility that FCP provides. This allows networks to dynamically translate the value of a budget belonging to different flow groups or assigned by

different domains. The former allows aggregate congestion control between groups of flows, and the latter enables distributed budget assignment between mutually untrusted domains.

Aggregate congestion control: FCP allows the network to perform aggregate congestion control over an arbitrary set of flows so that each group in aggregate attains bandwidth proportional to some predefined weight. This, for example, can be used to dynamically allocate bandwidth between multiple tenants in a data-center [36].

To achieve this, FCP views each group as having its own currency unit whose value is proportional to the weight of the group, and inversely proportional to the aggregate input budget of the group. When giving feedback, a router translates its price into the group's current currency value (Figure 3 (a)). For example, if flow group A has a weight of 2 and B has a weight of 1, and their current input budgets are, respectively, 10 $\$/sec$ and 100 $¥/sec$, A's currency has more value. To estimate the input budget for each group of flows, we use the `balance` field. Each router keeps a separate input budget for each group. It also keeps the aggregate input budget using its own link's price history and updates the price as in the original scheme using Equation 2. Thus, we calculate the normalized exchange rate E_G for flow group G as:

$$E_G(t) = \frac{w_G}{\sum_{H \in Group} w_H} / \frac{I_G(t)}{\sum_{H \in Group} I_H(t)}$$

And adjust the price feedback for packet s as:

$$price(s) = price(s) + p(t) \cdot E_G(t)$$

where $p(\cdot)$ is defined in Equation 2.

The implementation can be efficient, requiring O(1) computation for every packet arrival. This can be achieved using a timer, where a timer decays the input budget and updates the exchange rate for the queue when a packet expires out of the averaging time window. At every packet reception, we also update the input budget and exchange rate of the queue that the incoming packet belongs to.

Inter-domain budget management: In an Internet scale deployment, budget assignment must be made in a distributed fashion. One solution is to let each ISP or domain assign budget to its customers without any coordination, and rely on dynamic translation of the value. Here, we outline this approach, and leave the exact mechanism and demonstration as future work. When a packet enters another domain, the value the packet is holding (balance field in the header) can be translated to the ingress network's price unit (Figure 3 (b)). One can use a relatively fixed exchange rate that changes slowly or a dynamic exchange rate similar to the previous design. This allows the provider (peering) network to assign bandwidth to its customers (peers) proportional to their weight. Only one thing needs to be changed in the feedback. For the price feedback to be represented in the unit of the sender's price, the data packet also carries an exchange rate field, `Ex`. This mechanism also protects the provider's network from malicious customer networks who intentionally assign a large budget in an attempt to get more share because their budget's actual value will be discounted.

5 Practical Issues

§4 addressed the first order design issues. We now address remaining issues in implementation and deployment.

Real number to floating point: The first thing that has to be addressed in implementing Equation 2 is that price cannot be arbitrarily small as it must be represented as a floating point. However, in our relative pricing, the price can be an infinitesimal value. To address this problem, we define a globally agreed upon minimum price. We treat this value as zero when summing up the price of each link, i.e., for any price P, $P = P + MININUM$. Now, an uncongested path has the minimum price. When a new flow joins, the sender preloads to its target budget. If the target budget is larger than the bottleneck capacity over *MINIMUM*, the flow can saturate the link. For a host with a unit budget to be able to saturate any link, this minimum price has to be sufficiently lower than unit budget over any link's capacity. We use the minimum price of $10^{-18}\$/byte$ or $\$1/Exabyte$ in our implementation.

Computational complexity: To calculate price, routers need to keep a moving window of input budget and update the average RTT. When packets arrive at the router, the router updates these statistics and assigns a timer for the packet so that its value can expire when it goes out of the window. It requires roughly 20 floating point operations for each packet.[2] We believe high-speed implementation is possible even with software implementations on modern hardware.[3]

Security and Enforcement: FCP easily lends itself to enforcement of fair-share with only per-user state at the edge router. The network has to enforce two things: 1) the sender is paying the right amount, and 2) the sender is operating within its budget. To enforce 1), we use Re-feedback's [8] enforcement mechanism. The sender set the `balance` field to the amount it is paying, and every router along the path subtracts its link's price for this packet, $p(t - rtt)$. If the amount reaches a negative value, this means that the sender did not pay enough. The egress edge router maintains an average of the balance value. If the value is consistently below zero, the egress router computes the average balance for each sender and identifies who is consistently paying less, and drops its packet. This information is then propagated upstream to drop packets near the source. Details on statistical methods for efficient detection are described in [8]. To enforce 2), the ingress edge router enforces a modified version of invariant that accounts for preloading:

$$\sum_{\text{packet in (t-1,t]}}^{\text{for each}} paid(packet) \cdot size \cdot preload \leq budget$$

This requires a classifier and per-user state at the ingress edge, and ensures that a host does not increase its budget arbitrarily. Ingress edge routers can drop packets once a host uses more budget than it is assigned.

[2]It requires 6 operations to update intermediate values when packets arrive and 2 operations when packets leave the window. An additional 7 operations are required to update the final price, average RTT, and calculate the feedback.

[3]As a rough estimate, Intel Core i7 series have advertised performance of ~100 GFLOPS. At 100 Gbps, this gives a budget of ~500 FLOPS/packet.

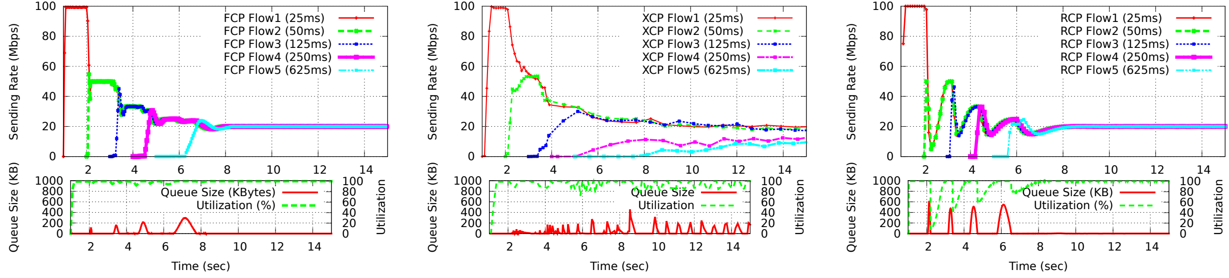


Figure 4: Convergence dynamics of a) FCP, b) XCP, and c) RCP: FCP achieves fast convergence and accurate rate control. When new flows arrive, XCP's convergence is slow, and RCP's rate overshoot is significant.

Implementation details: In our experiments in §6, we set the average window size, d in Equation 2, to twice the average RTT. The queue parameter α is set to 2, but to prevent the remaining link capacity (denominator of Equation 2) from becoming negative,[4] we bound its minimum value to 1/2 the capacity. Under this setting, we verified the steady-state local stability of the algorithm using simulations in §6.1.

Finally, we also make our implementation robust to RTT measurement errors. Equation 4 uses $p(t-rtt(s))$, however, when price variation is large around time $t-rtt(s)$, small measurement error can have adverse effect on a router's estimate of the feedback price. Or even worse, routers may not store the full price history. We take the minimum of $p(\cdot)$ and $balance$ field in the congestion header. This bounds the error between the paid price of the sender and the router's estimation of it, even when routers do not fully remember its past pricing.

6 Evaluation

We answer three questions in this section:

1. How does FCP perform compared to other schemes and what are the unique properties of FCP?

2. Does the flexibility allow evolution in the end-point?

3. Does the flexibility allow evolution in the network?

6.1 Performance

We implement FCP's algorithm described in §4 using ns-2. Using simulations, we study the performance of FCP. First, we compare the performance of FCP with other schemes (TCP, RCP, and XCP). We, then, look at unique characteristics of FCP, including fairness, preloading effectiveness, and its stability. Finally, we look at FCP's performance under a wide range of scenarios. The

[4]This can happen when the queue capacity is relatively large compared to the current bandwidth delay product.

Protocol	AFCT	Loss rate	Avg. queuing delay	Avg. utilization
XCP	0.52 sec	0%	0.13 ms	64%
TCP	0.60 sec	1.6%	12 ms	68%
RCP	0.21 sec	0%	3.2 ms	66%
FCP	0.33 sec	0%	0.027 ms	65%

Figure 5: Short flows: Comparison of various statistics. AFCT stands for average flow completion time.

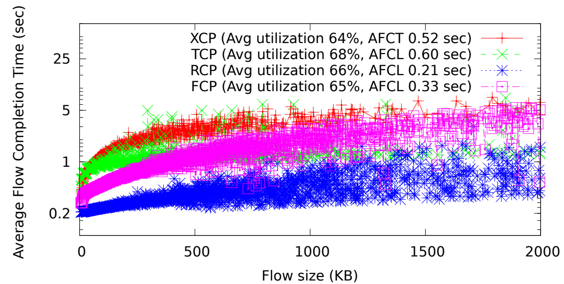


Figure 6: Short flows: Average flow completion time versus flow size. Mean flow size is 30 KB.

results show that FCP provides fast convergence while providing more accurate feedback during convergence and is as efficient as other explicit congestion control algorithms in many cases.

6.1.1 Performance comparison

First, we compare the performance of FCP with other schemes (TCP, RCP, and XCP) in various environments.

Long Running Flows: We first compare the convergence dynamics of long-running flows. We generate flows with different round-trip propagational delays ranging from 25 ms to 625 ms. Each flow starts one second after another traversing the same 100 Mbps bottleneck in a dumbbell topology. Each flow belongs to a different sender, and all senders were assigned a budget of 1 $/sec. For RCP and XCP, we used the default parameter setting. Figure 4 shows the sending rate of each flow, queue size, and utilization for a) FCP, b) XCP, and c) RCP over time.[5] FCP's convergence is faster than RCP's and XCP's. XCP flows do not converge to the fair-share even at t=10 sec. In RCP, all flows get the same rate as soon as they start because the router gives the same rate to all flows. However, large fluctuation occurs during convergence because the total rate overshoots the capacity and packets accumulate in the queue when new flows start. Average bottleneck link utilization was 99% (FCP), 93% (XCP), and 97% (RCP).

Short flows: To see how FCP works with short flows, we generate a large number of short flows. We generate a Pareto distributed flows size with mean of 30 KB and shape of 1.2, flows arriving as a Poisson process with a mean arrival rate 438 flows/sec (offered load of 0.7). The bottleneck bandwidth and round trip delay is set to 150Mbps and 100 ms.

All FCP senders have the same budget of 1 $/sec. However, because the flow is short, using the entire budget is unnecessary. Thus, we only preload to achieve at most $\frac{flowsize}{RTT}$ Bytes/sec. Table 5 and Figure 6 show the statistics and average flow completion time by flow size. For flow size less than 500KB, FCP performs better than TCP and XCP. However, RCP is faster

[5] Sending rate is averaged over a 100 ms. For XCP ($t \geq 3$), we use a 500 ms averaging window.

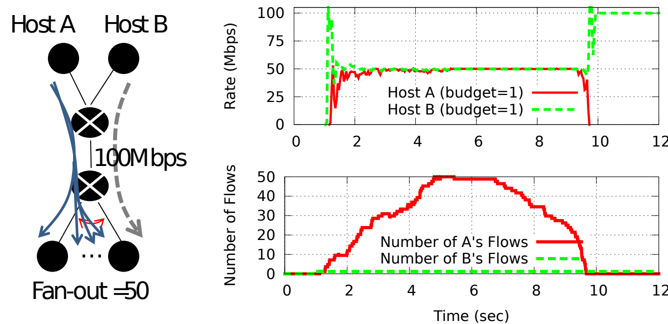


Figure 7: FCP's fair-share is determined by the budget.

because it jump-starts with the current rate after the SYN-ACK exchange, whereas FCP has to conservatively preload in the first round. (We assume every flow is going to a different destination.) For larger flows, FCP performance is in between that of XCP and TCP, but RCP still performs better. However, we saw that RCP makes an undesirable trade-off to achieve this. We further study RCP's limitation in §6.1.2.

The large variance for FCP and RCP is due to the fair-share rate changing as the load changes. On the other hand, XCP and TCP's rate is largely a function of how long the flow has been active because they slowly reach the fair-share rate. Finally, FCP shows the same zero loss rate as other explicit congestion control schemes, but has a much lower queueing delay. This is because FCP uses preloading to accounts for variations in the workload.

6.1.2 Fairness, stability, and preloading

To better understand FCP, we look at behaviors specific to FCP. In particular, we look at the fairness, stability, and the preloading behavior of FCP. We show that 1) the fair-share in FCP is determined by the budget, 2) FCP is locally stable under random perturbations, and 3) preloading allows fast convergence and accurate feedback.

Fairness: Fairness in FCP is very different from traditional flow-level fairness. In FCP, two hosts that have the same budget achieve the same throughput if their traffic goes through the same bottleneck. To show this, we create two hosts A and B of equal budget. Host A generates 50 flows, but B sends only one, which go through a common bottleneck link. Host A's flow size is 1 MB each and they arrive randomly between [1,3]. Host B sends a long-running flow starting at t=1. Figure 7 shows the topology (left), the sending rate of each host, and the number of active flows (right). Host A's traffic, when present, gets 50% share regardless of the number of flows.

Local stability: A common method to prove the local stability is to show the stability of a linearized equation of the congestion controller near its equilibrium point [22]. However, it is shown that explicit congestion control algorithms whose equilibrium is at zero queue length is discontinuous at equilibrium and that the above traditional method produces incorrect results [2]. As a

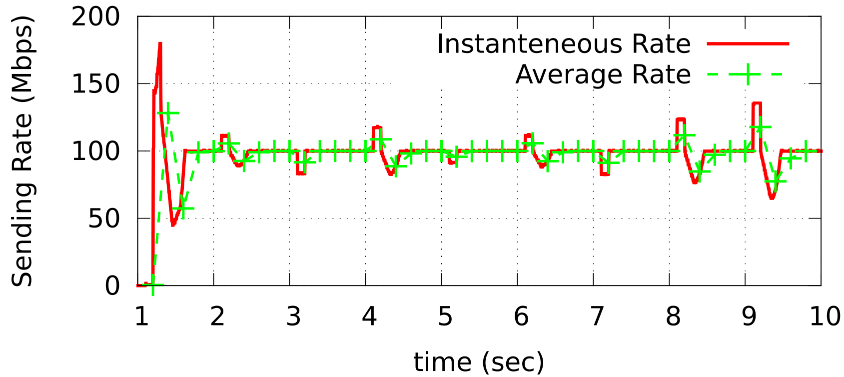


Figure 8: The algorithm is stable under random perturbation.

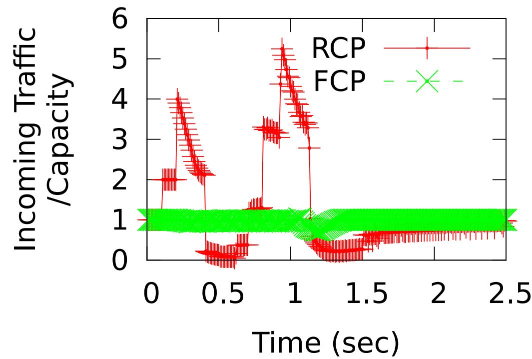


Figure 9: Preloading allows fast convergence and accurate rate estimation.

result, several studies have used much more complicated methods, such as Lyapunov methods, and verified the result with simulations [27, 2].

In this work, we demonstrate the stability using packet-level simulations. In particular, in this experiment, we perform a local stability test by introducing perturbations. The system's stability depends on the stability of the price feedback equation. We, therefore, intentionally introduce errors in the price calculation and observe whether the system is able to restore itself on its own. If the system is stable, it will return to the equilibrium state shortly after the perturbation is introduced. We use a single link with a round trip delay of 100 ms with 100 Mbps of capacity. A long running flow starts at t=1.1 sec with a unit budget. At every second from t=2, we introduce random perturbation of [-30%, 30%] in the router's price calculation for a duration of 100 ms. For example, during the interval, t=[2,2.1] sec, the feedback price off from the correct price by a random fraction between 1 to 30%. Figure 8 shows the instantaneous and average rate of the flow. We observe after the perturbation is introduced the system either overshoots or undershoots the capacity. However, the system recovers the equilibrium shortly (i.e., the sending rate stabilizes at 100 Mbps).

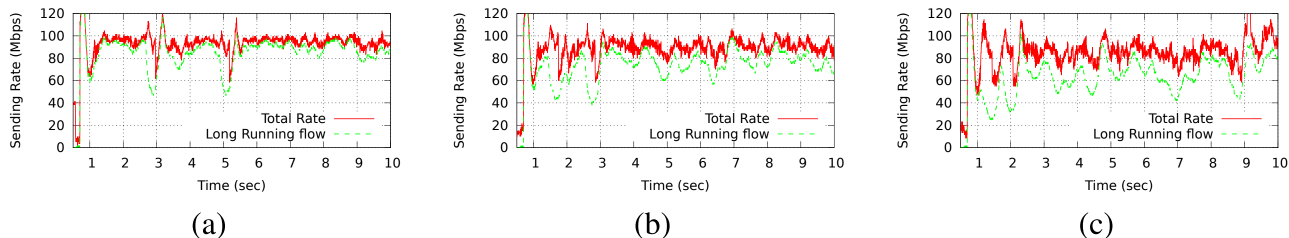


Figure 10: Performance with long-lived flows and short-lived flows.

Case	new flows/sec	utilization (%)
(a)	41.6	98.2
(b)	83.2	95.7
(c)	125	81.9

Figure 11: Performance statistics with long-lived flows and short-lived flows.

How effective is preloading? Preloading is one of the distinguishing features of FCP. So far, we have seen the end-to-end performance of FCP with preloading. Here, we look at the benefit of preloading in isolation.

We demonstrate the benefit of FCP by comparing it with RCP that does not have preloading. We compare a FCP flow that continuously doubles its budget every 100 ms (one RTT) with RCP flows that double in flow count for a duration of 1 second. In both cases, the load (input budget for FCP and flow count for RCP) doubles every RTT.

Figure 9 shows the normalized incoming traffic at the bottleneck link while the load is ramping up from 1 to 1000 during the first second. We see that preloading allows fast convergence and accurate rate control; FCP's normalized rate is close to 1 at all times. On the other hand, RCP overestimates the sending rate by up to 5 times the capacity because RCP allocates bandwidth in a very aggressive manner to mimic processor sharing. With preloading, however, end-hosts can rapidly increase or decrease a flow's budget without causing undesirable behaviors in the network.

6.1.3 In-depth study of FCP's behavior

Finally, we look at detailed behaviors of FCP with mixed flow sizes, the convergence dynamics with multiple links, and the impact of misbehaving users.

Mixed flow sizes: We now study how FCP performs with long-lived flows and short-lived flows. All flows go through a common bottleneck of 100 Mbps. Long running flows start at t=0.5 sec and introduce a unit budget to the bottleneck link. They run for the duration of the simulation. Short flows arrive as a Poisson process and their size is Pareto distributed as earlier with a mean size of 30 KB. We vary the fraction of bandwidth that short flows occupy from 10% to 30%. Figure 10 shows

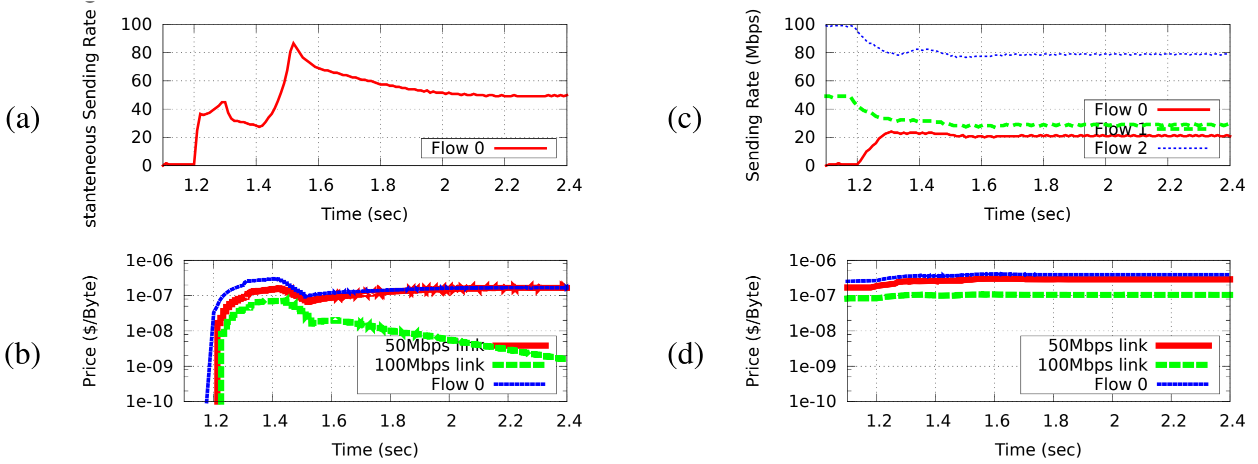
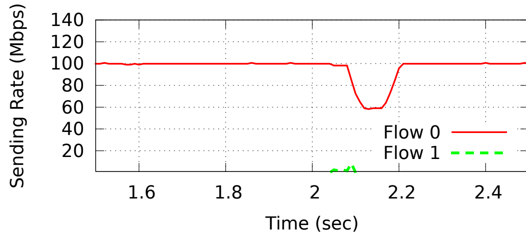


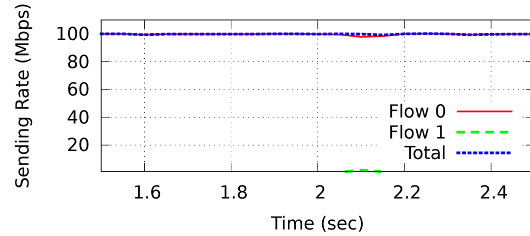
Figure 12: FCP's convergence dynamics with multiple links. Flow 0's round trip delay is 100 ms. Each column presents a different scenario: 1) (a) and (b) shows the dynamics of the sending rate and price of the two links when Flow 0 starts up first. 2) (c) and (d) shows the dynamics when Flow 0 starts up after Flow 1 and Flow 2.

the link utilization and the throughput of the long-lived flows. Figures 10 (a), (b), (c) respectively illustrate the case when the short flows account for 10%, 20%, and 30% of the bottleneck. Table 11 show the average number of new short flows per second and the average link utilization from t=0.5 sec to t=30 sec for each experiment. We observe that the link utilization becomes lower as the number of new flows increase. This is because when flows terminate, even though they preload a negative value, it takes some time for the price to reflect due to the averaging window. During this period, it results in a slight underutilization. The utilization has a negative correlation with the amount of input budget variance per unit time. In other words, the utilization is high when a small fraction of input budget changes because of flow arrival and departure. For example, when the average flow size of the short flows is made larger (300 KB) with the fraction of short flows being 30%, the utilization slightly goes up to 90% from 82%. Also, when the long running flows' total budget is high, the utilization also goes up. When the long running flow's budget is set to $10\$/sec$, the utilization goes up from 82% to 99% This shows that when the fraction of stable flows' input budget is relatively high, the system is highly efficient.

Convergence dynamic with multiple links: FCP adopts proportional fairness. FCP's convergence dynamics with multiple links best illustrates how link prices are determined with proportional fairness. To demonstrate this, we use the topology of Figure 19 (b) with a larger link latency of 12.5 ms to better observe the dynamics. First, we start a single flow (Flow 0) at t=1.1 sec with a unit budget. Figure 12 (a) shows the instantaneous sending rate of the flow. Figure 12 (b) shows the how the price of the two links and the price of Flow 0 (the sum of the two prices) change over time. At the start of the flow, the price is at its minimum, $10^{-18}\$/byte$ (not visible in the figure). Because the sum of two minimum prices is the minimum price, the path price is also min-



(a) A misbehaving flow



(b) A well-behaving short-lived flow

Figure 13: A misbehaving flow negatively impacts the link utilization.

imum. Both links' prices go up when Flow 0 preloads to use its entire budget. However, because the path price was at minimum, each link thinks that the flow's budget is entirely used toward its own link, which results in a temporary over-estimation of the path price. However, soon the price goes down because there is spare capacity. During this process, the system results in a temporary under-estimation of path price because the two links adjust its price independently. However, the path price and the sending rate stabilizes soon after. In particular, only the bottleneck link's price remain above the minimum price; the 100 Mbps link's price eventually falls down to the minimum price as its link is never fully utilized. On the other hand, the 50 Mbps bottleneck's price stabilizes above the minimum price level and dominates the path price. As a result, Flow 0's sending rate stabilizes at 50 Mbps. Our result shows that there is an interaction between links within a path in adjusting the link's price. However, it converges to the correct equilibrium point.

Above, we observed the cold startup behavior of a flow with multiple links. The behavior is slightly different when both links' prices do not start from the minimum price. Two show this we introduce a flow (Flow 0 of Figure 19) after Flow 1 and Flow 2 of Figure 19 have started. Old flows starts at t=0 sec and the new flow (Flow 0) starts at t=1.1 sec. All flows have a unit budget. Now, Flow 0 traverses two bottleneck links (i.e., the price of both links are not minimum). Figure 12 (c) and (d) respectively shows the sending rate of each flow and price of each link. We observe that the convergence is faster and overestimation/underestimation is much smaller. This is because the relative difference in the input budget after the introduction of a new flow is much smaller in this scenario than the cold startup. We also observe that the throughput of Flow 0 is lower than that of flow 1 even though they are sharing the 50 Mbps bottleneck link. This is because Flow 0's price is higher than Flow 1's price; Flow 1 only pays for the 50 Mbps link bottleneck where as Flow 0 pays for both bottlenecks.

In summary, we saw that the system converges to proportional fairness relatively quickly even though the link prices are calculated independently and the feedback only contains the total price. We also observed that only the bottleneck link's price stay above the minimum price level.

Misbehaving users: Misbehaving users may commit to use its own budget, but not live up to its promise. For example, a user may preload to use a budget of 1 $\$/sec$, but actually use nothing. This results in an underutilization because when a user preloads the network update its price reflecting the expected incoming budget. We illustrate this scenario here. We generate two flows, a normal

flow and a misbehaving flow, on a common 100 Mbps bottleneck link. The round-trip delay is set to 40 ms. Figure 13 (a) shows the sending rate for both flows at every 10 ms period. For comparison, Figure 13 (b) shows the correct behavior when Flow 1 is a short-lived flow. In this case, the utilization is close to 100%. This is because a well-behaving flow does not over-preload. It also performs negative preloading at the end of the flow, as explained earlier.

Figure 13 (a) shows the misbehavior. A long-running flow (Flow 0) starts up at t=1 sec and starts to saturate the bottleneck link. At time t=2 sec, a user who has a unit budget starts up a misbehaving flow (Flow 1). It preloads its entire budget (1 $\$/sec$) but does not send any traffic after the preload. As a result, the path price goes up and Flow 0 nearly halves its sending rate. The period underutilization ends after the duration of the averaging window because the routers lower the price to achieve full utilization. However, in general, if the misbehaving users preload constantly without actually using the budget in the next round, the network is going to be underutilized constantly. We believe that this is not any worse than sending junk traffic. Even if a user preloads and does not live up to its promise, it can never harm other traffic more than what would have been its fair-share had it generated real traffic. However, the network can easily detect misbehaving users by comparing the expected incoming budget calculated in the previous averaging window to the actual incoming budget of the current averring window. If the former is consistently greater than the latter, the network classify the sender as an attacker and take appropriate measures.

Another type of misbehavior is to perform negative preload, but not decreasing the budget spent. This results in an overload. Similar to the underutilization problem, if a user constantly performs negative preloading, the network will be overloaded constantly. Here, the solution is to perform enforcement at the network similar to the method mentioned above. When the expected incoming budget for a host is consistently less than the actual incoming budget, the network should classify the host as an attacker and block its traffic.

6.2 End-point Evolution

In FCP, an end-point can freely distribute its budget to its own flows. This allows evolution in the end-point's resource allocation schemes. Here, we evaluate end-point based algorithms outlined in Section 4.2. For easy comparison, we use the topology of Figure 14. Sender 0 (S0) has a flow to receiver 0. S1 and S2 each has a flow to receiver 0 and receiver 1. S0 and S2 have a budget of 1 $\$/sec$, and S1 has twice as much.

Equal-budget (baseline) splits the budget equally among flows within a host. For example, S1 splits its budget in half; the top flow (to receiver 0), and the bottom flow (to receiver 1), each gets 1 $\$/sec$. Figure 15 shows the instantaneous sending rate of each sender with S1's rate broken down. It shows FCP achieves weighted bandwidth allocation. The top 250 Mbps bottleneck link's total input budget is 2.5 $\$/sec$. Because S0 and S1's top flow use 1 $\$/sec$, their throughput is 100 Mbps.

Equal throughput: Now, S1 changes its budget assignment to equal-throughput where it tries to achieve equal throughput on its flows, while others still use equal-budget assignment. For this, we start with the equal-budget assignment, and reassign every 2 average RTTs, and increase the budget of a flow whose rate is less than the average rate. Figure 16 shows the result. At steady-

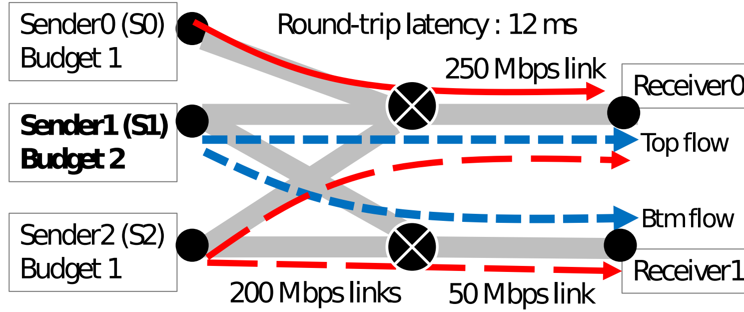


Figure 14: Topology

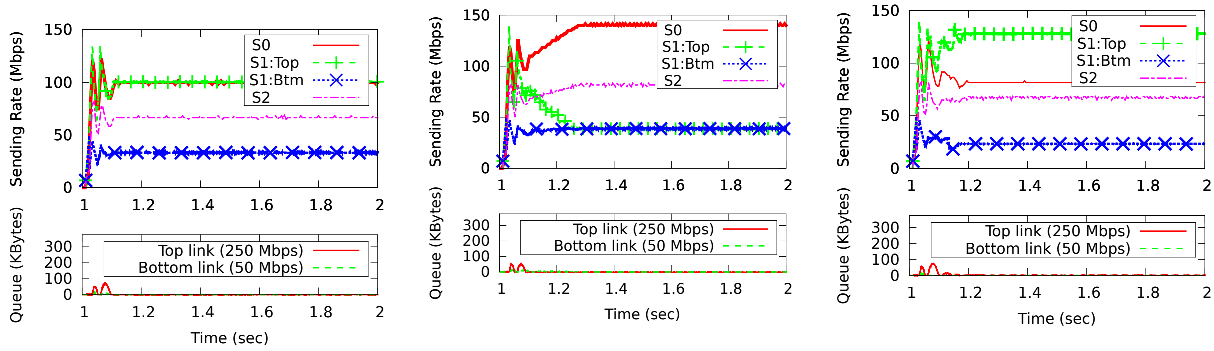


Figure 15: Equal-budget flows

Figure 16: Equal-throughput flows

Figure 17: Max-throughput flows

state, S1's two flows achieve equal throughput of 38.7 Mbps. A budget of 0.27 $\$/sec$ is assigned to the top flow, and 1.73 $\$/sec$ to the bottom.

Max-throughput: S1 now uses max-throughput assignment, where it tries to maximize the its total throughput. Others still use the base-line assignment. We implement this using gradient ascent. S1's flows start with equal budget, and at every 2 average RTT, it performs an experiment to change the budget assignment. It chooses a flow in round robin and increases its budget by 10% while decreasing others uniformly to maintain the total budget assignment. After 2 average RTTs, it compares the current throughput averaged over an RTT with the previous result, and move towards the gradient direction. The algorithm terminates when the throughput difference is less than 0.5%. The algorithm restarts when it observes a change in the price of a flow.

Figure 17 shows the result. S1's total throughput converges at 150.6 Mbps, and the assigned budget for the top flow (X) converges at 1.56 $\$/sec$. Figure 18 a) shows the theoretical throughput versus the budget assignment, X. The theoretical maximum throughput is 151.6 Mbps at $X = 1.68$ When more (less) budget is spent on X than this, the top (bottom) bottleneck link's price goes up (down), and the marginal utility becomes negative. Figure 18 b) shows such non-linear utility (rate per unit budget) curve for the two flows.

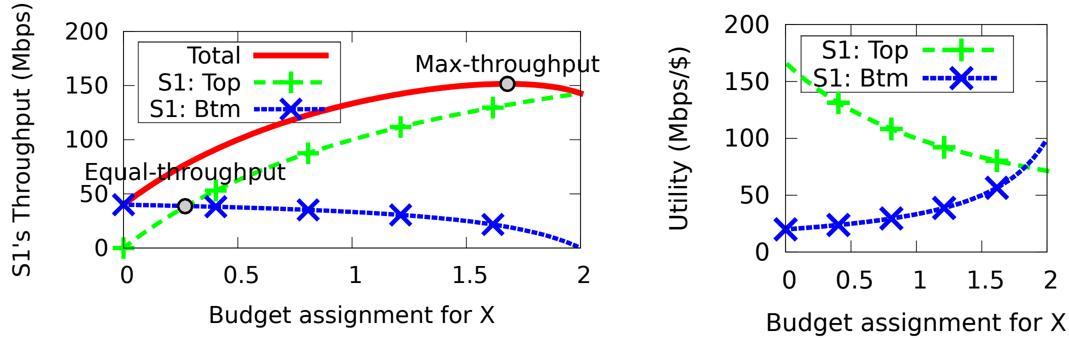


Figure 18: Theoretical (a) throughput and utility (b) versus budget assignment X for flow S1's top flow.

Background flows: FCP can also support background flows. A background flow by definition is a long-running flow that only occupies bandwidth if there's no other flow competing for bandwidth. This can be achieved with a flow having a very small assigned budget compared to other long running flow. For this, each host uses 1/10000 of its budget towards background flows. Using the topology in Figure 19, we ran a background flow with three other flows with a unit budget. Flow 0 starts at t=1 and immediately occupies the 50 Mbps bottleneck link (Figure 19). Flow 1 arrives at t=2 and shares the bottleneck with Flow 0. At t=5, the background flow (Flow 2) starts and occupies the remaining 75 Mbps of the 100 Mbps link. Note this did not influence Flow 0's rate. Flow 3 arrives at t=10 with unit budget and drives out the background flow. Note that now the 100 Mbps link also became a bottleneck, and Flow 0 is getting a smaller throughput than Flow 1. This is because Flow 0 is now paying the price of the two bottlenecks combined.

6.3 Network Evolution

We now evaluate the network evolution scenarios in §4.2.

Bandwidth stability: We the implemented bandwidth stability feature described in §4.2. Figure 20 shows the sending rate of a stability flow (`Flow 2`) compared to a normal flow (`Flow 1`) under changing network conditions. Flow 2 starts at t=2 and slowly ramps up doubling its assigned budget at every RTT (100 ms). Cross traffic (Flow 3, dashed blue line) that has 50 times the budget of Flow 1 repeats a periodic on/off pattern starting from t=3. Flow 1's sending rate (solid black line) changes abruptly when the cross traffic join and leave, but the stability flows react slowly because its price does not change by more than twice in any direction during a window of 200 ms (twice the average RTT).

Multicast congestion control: FCP allows the network to evolve to support a different service model. For example, FCP can support multicast-aware congestion control as described in §4.2. We use the topology of Figure 24 (b) and generate a multicast flow from the top node to the bottom

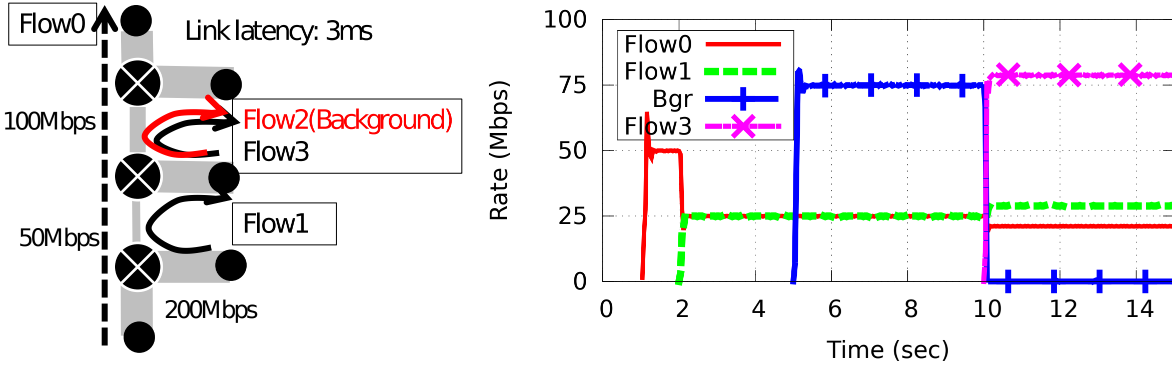


Figure 19: Background flows only take up the spare capacity. The sending rate is averaged over 40 ms.

four receiver nodes. The link capacity varies by link from 10 to 100 Mbps. We also introduce unicast cross traffic to vary the load, and show that the multicast flow dynamically adapts its rate.

Figure 21 shows the result. A multicast flow starts at t=1 and saturates the bottleneck link capacity of 10 Mbps. Three unicast flows then start sequentially at t=2, 2.5, 3. When `Unicast 1` arrives, it shares equally shares the bottleneck bandwidth. As other unicast flows arrive, other links also become a bottleneck and their price goes up. As a result, the multicast flow's price (the sum of all link price) goes up and its sending rate goes down. At steady state, the multicast flow's throughput is around 4 Mbps. The unicast flows take up the remaining capacity (e.g., unicast 3's rate is 36 Mbps).

Deadline support: As we described in §4.2, FCP can offer D^3-style deadline support using two virtual queues (best effort and deadline) and differential pricing. Deadline flows are guaranteed a fixed price when admitted. We use the minimum price (§5) as the fixed price. A deadline flow at the beginning of the flow preloads the amount required to meet the desired rate R at once. If the routers on the path can accommodate the new deadline flow, they return the fixed price. The deadline flow is then able to send at desired rate and meet the deadline. Otherwise, routers give the best effort pricing and treat the flow as a best effort flow. We run an experiment to show both cases.

Figure 22 shows the instantaneous rates of deadline and best effort flows going through a 100 Mbps bottleneck link with a round-trip delay of 2 ms. The queue is set to admit up to 80 Mbps of deadline flows and assigns at least 20 Mbps to best effort flows in a work conserving manner. A best effort (BE) flow starts at the beginning and saturates the link. At t=0.1 sec, two deadline flows (D1 and D2) requiring 30 Mbps of throughput arrive to meet a flow completion time of $\sim$200 ms. Because the link can accommodate the deadline flows, they both get admitted and complete within the deadline. At t=0.6 sec, four deadline flows (D3 to D6) arrive with the same requirement. However, the network can only accommodate two deadline flows (D4 and D5). The other two (D5 and D6) receive the best effort pricing and become best effort flows. We then additionally preload

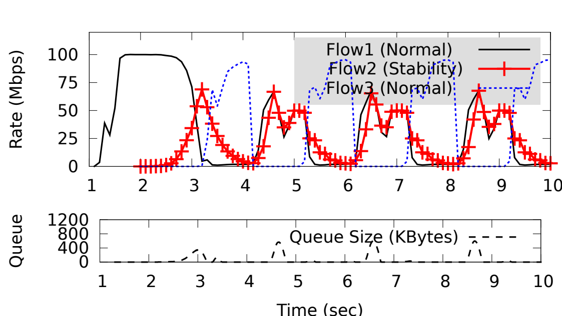


Figure 20: Support for bandwidth stability

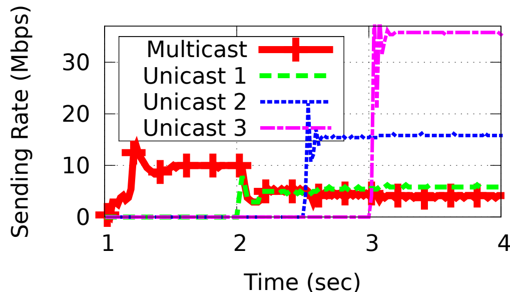


Figure 21: Multicast flow (red line) adapts to congestion.

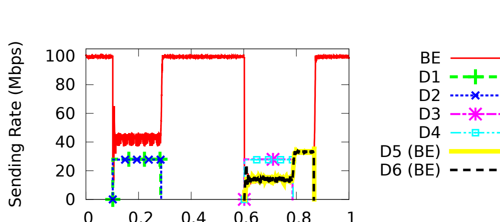


Figure 22: Deadline flows

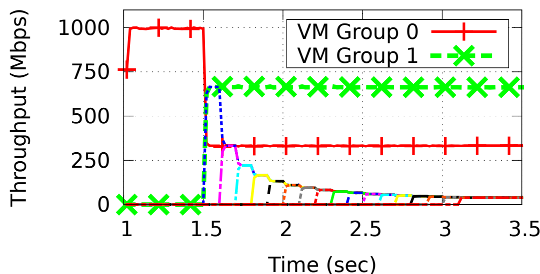


Figure 23: Aggregate control

to make it achieve its fair-share in the best effort queue. As a result, all the best effort flows achieve the same throughput.

Budget management and aggregate congestion control: We implement aggregate congestion control using the algorithm described in §4.3. To show how aggregate congestion control may work in a data-center to allocate resources between tenants, we use a flow pattern of Figure 24 (a), similar to the example shown in [36]. Within a rack, there are 20 servers and each have two virtual machines (VMs). Each server is connected to the ToR switch at 1 Gbps. Odd numbered VMs (VM group 1) belong to one tenant and even ones (VM group 0) to the other. Each tenant allocates a budget to each of its own VM independently. We assume all VMs have an equal budget. The data-center provider allocated a weight of 1 to group 0, and weight of 2 to group 1. In group 0, there is only one flow. In group 1, there are multiple flows starting one after another, all originating from a different VM (see Figure 24 (a)). Figure 23 shows the throughput of each group as well as the individual flows. At t=1, a flow starts in the VM group 0. From t=1.5 sec, VM group 1's flows start to arrive, and get twice as much share as group 0. Dotted lines show that the throughput of group 1's individual flows who have equal budget achieve the same throughput. The result highlights two benefits of aggregate congestion control: 1) regardless of the number of flows in the group, the aggregate flows get bandwidth proportional to their weights (group 1 gets twice as much as group 0), and 2) the weight of flows within each group is preserved (individual flows in group 1 receive

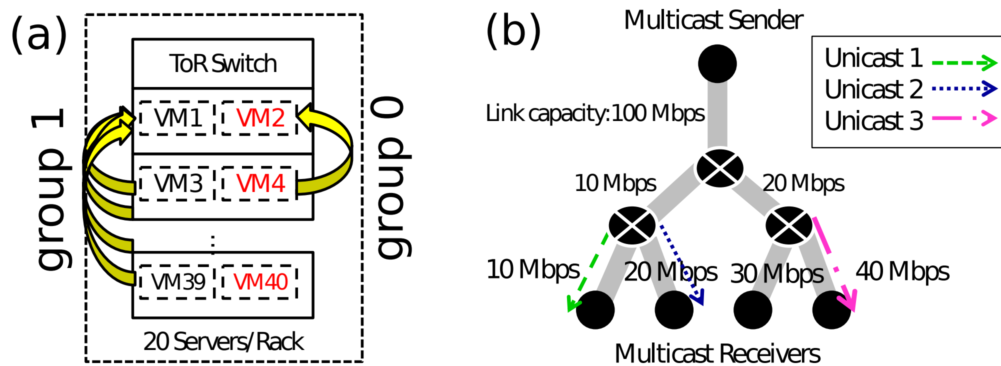


Figure 24: Topologies and flow patterns of experiments

the same throughput).

7 Conclusion

This paper explores an important open problem of designing networks for evolution. While there have been a number of initial studies on evolvable Internet architectures [17, 21], little research has been done in designing an evolvable transport layer. In this paper, we argue existing approaches such as virtualization and TCP-friendly designs have limitations and call for a universal framework for congestion control in which various strategies and algorithms can coexist. We propose a design based on a pricing abstraction and a simple invariant. We present an explicit feedback-based algorithm that realizes this idea and demonstrate that it is flexible enough to allow evolution. Finally, we address practical concerns such as enforcement of fair-share and implementation issues.

References

[1] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38, April 2005.

[2] H. Balakrishnan, N. Dukkipati, N. McKeown, and C.J. Tomlin. Stability analysis of explicit congestion control protocols. *Communications Letters, IEEE*, 11(10):823 –825, october 2007.

[3] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM*, Cambridge, MA, September 1999.

[4] Deepak Bansal and Hari Balakrishnan. Binomial Congestion Control Algorithms. In *IEEE Infocom*, Anchorage, AK, April 2001.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), December 1998.

[6] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: a highly configurable aspect-oriented IP stack. In *Proc. ACM MobiSys*, June 2012.

[7] Patrick G. Bridges, Gary T. Wong, Matti Hiltunen, Richard D. Schlichting, and Matthew J. Barrick. A configurable and extensible transport protocol. *IEEE ToN*, 15(6), December 2007.

[8] Bob Briscoe, Arnaud Jacquet, Carla Di Cairano-Gilfedder, Alessandro Salvatori, Andrea Soppera, and Martin Koyabe. Policing congestion response in an internetwork using re-feedback. In *Proc. ACM SIGCOMM*, 2005.

[9] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proc. ACM SIGCOMM*, Baltimore, MD, August 1992.

[10] Costas Courcoubetis, Vasilios A. Siris, and George D. Stamoulis. Integration of pricing and flow control for available bit rate services in ATM networks. In *Proc. IEEE GLOBECOM*, 1996.

[11] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM CCR*, 28, 1998.

[12] Dragana Damjanovic and Michael Welzl. MulTFRC: providing weighted fairness for multimedia applications (and others too!). *ACM SIGCOMM CCR*, 39, June 2009.

[13] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-shen, and Nick Mckeown. Processor sharing flows in the Internet. In *Proc. IWQoS*, 2005.

[14] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proc. ACM SIGCOMM*, 2000.

[15] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1, August 1993.

[16] Bryan Ford. Structured streams: a new transport abstraction. In *Proc. ACM SIGCOMM*, 2007.

[17] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. Intelligent design enables architectural evolution. In *Proc. ACM HotNets*, 2011.

[18] Richard J. Gibbens and Frank P. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, pages 1969–1985, 1999.

[19] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42, July 2008.

[20] Dongsu Han, Ashok Anand, Aditya Akella, and Srinivasan Seshan. RPT: Re-architecting loss protection for content-aware networks. In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012.

[21] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012.

[22] Dina Katabi, Mark Handley, and Chalrie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, August 2002.

[23] F P Kelly, A K Maulloo, and D K H Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3), 1998.

[24] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 1997.

[25] Frank Kelly, Gaurav Raina, and Thomas Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM CCR*, 2008.

[26] A. Li. RTP Payload Format for Generic Forward Error Correction. RFC 5109 (Proposed Standard), December 2007.

[27] Zongtao Lu and Shijie Zhang. Stability analysis of xcp congestion control systems. In *Proc. IEEE Wireless Communications and Networking Conference (WCNC)*, April 2009.

[28] Jeffrey K. MacKie-Mason and Hal R. Varian. Pricing the internet. Computational Economics 9401002, EconWPA, January 1994.

[29] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application performance pitfalls and TCP's nagle algorithm. *SIGMETRICS PER*, 27, March 2000.

[30] Preethi Natarajan, Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. SCTP: an innovative transport layer protocol for the web. In *Proc. World Wide Web*, 2006.

[31] Andrew Odlyzko. Paris metro pricing: The minimalist differentiated services solution. In *Proc. IWQoS*, 1999.

[32] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading transport protocols using untrusted mobile code. In *Proc. ACM SOSP*, 2003.

[33] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proc. ACM SIGCOMM*, 2007.

[34] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: reshaping the research agenda. *ACM SIGCOMM CCR*, 26(2), April 1996.

[35] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010.

[36] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *Proc. 8th USENIX NSDI*, 2011.

[37] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. In *Proc. ACM SIGCOMM*, 1998.

[38] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. OverQoS: An overlay based architecture for enhancing Internet QoS. In *Proc. 1st USENIX NSDI*, San Francisco, CA, March 2004.

[39] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36, December 2002.

[40] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[41] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proc. 8th USENIX NSDI*, Boston, MA, April 2011.

[42] Y.R. Yang and S.S. Lam. General aimd congestion control. In *Proc. ICNP*, 2000.