IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Mitigating Congestion in High-Speed Interconnects for Computer Clusters

Vladimir Zdornov and Yitzhak Birk

CCIT Report #723
March 2009

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #723 March 2009

Mitigating Congestion in High-Speed Interconnects for Computer Clusters

Vladimir Zdornov and Yitzhak Birk

March 2009

Contents

Abstract

Congestion arises in cluster-based supercomputers due to contention for links, spreads due to oversubscription of communication resources, and reduces performance. We mitigate it using efficient, scalable adaptive routing and explicit rate calculation. We use virtual circuits for in-order packet delivery; path setup is performed by switches locally with no blocking or backtracking. For random permutations in a slightly enriched fat-tree topology, maximum contention is reduced by up to 50% relative to static routing, but only rate control can translate this into actual gain. Unfortunately, TCP's window-based rate control fails because of the low bandwidth-delay product, and small buffers moreover cause congestion spreading even with a single-packet window. InfiniBand's CCA employs multiple parameters, which must apparently be tuned per topology and traffic pattern. Instead, we present distributed explicit rate-assignment algorithms, which are designed under three cluster-specific performance goals. Also, we propose a packet injection scheme, and show that desired rates can be realized even with very small buffers. Simulation results suggest that adaptive routing alone offers little benefit, rate control's effectiveness varies with traffic pattern, but together they boost performance by tens of percents. Finally, our explicit algorithms are faster than current reactive schemes.

Chapter 1

Introduction

1.1 Modern Clusters and Interconnects

Computer clusters are presently the architecture of choice for most supercomputers for various applications. Such clusters usually comprise hundreds to tens of thousands of off-the-shelf, general purpose computing elements interconnected by a high-speed network [2]. In many cases, performance of these parallel machines highly depends on the properties of their interconnects. These can be proprietary (such as in IBM Blue Gene) or standard, with the latter being the common choice.

The interconnect standards currently dominating the supercomputer market are InfiniBand [17] and Gigabit Ethernet [16] (approximately 24% and 58% of top-500 supercomputers, respectively). InfiniBand, from its inception, was oriented to operate in high-performance, clustered environments. It introduced many important features, some of which are discussed below, that match the communication needs of clusters. Ethernet, on the other hand, entered the market from the world of general purpose LAN and WAN communication, in an attempt to unify different types of interconnects under the same standard. Interestingly, we observe cluster versions of Ethernet adopting more InfiniBand-like properties, bringing the two standards closer over time. For that reason, we chose InfiniBand as the platform for our work, while expecting the results to be relevant for cluster interconnects in general.

In this work we will concentrate only on several important characteristics of InfiniBand. First, InfiniBand networks use virtual cut-through switching, which

greatly reduces the bandwidth-delay product. Second, similarly to other domains, virtual-output queuing is used by switches, placing the buffers at the inputs of the switch. The size of the buffers is practically very small, designed to hold only several MTU-size packets. Third, the fabric is lossless, which means that no packets are dropped due to an attempt to transmit them into a full buffer. Instead, hop-by-hop flow control is used to prevent the transmission in this case. Finally, switches use oblivious, destination-based routing to forward packets. The combination of oblivious routing with lossless communication has a dramatic impact on the performance and implementation complexity, since together they guarantee in-order packet delivery. As a result, no retransmissions are required, unless a (rare) physical packet corruption occurs.

From the management point of view, InfiniBand networks have three fundamental traits. To begin with, these networks constitute a managed environment. This means that all network elements (switches, HCAs) can be configured to operate according to some chosen protocol, without being concerned about interoperability or malicious behavior. Reliable communication is the second cardinal property, since it allows us to formulate management algorithms under the assumption that every packet eventually reaches its destination, while passing through each link exactly once. Of course, physical failures may occur but, being very rare, they can be treated in an exceptional manner, without worrying about efficiency. Finally, small network diameter (both in terms of hops and actual distance), high data rates and low-latency switches result in a very low end-to-end packet delay. This low delay reduces the cost of global management operations such as broadcasts and barriers.

1.2 Network Model and Terminology

A network comprises switches, host channel adapters (HCAs), and bi-directional links. We use H and SW to denote the sets of HCAs and switches, respectively. Switches and HCAs are collectively referred to as *network elements*, denoted $E = SW \cup H$. A bi-directional link is a pair of uni-directional links,with *link* referring to the latter. The set of links is denoted by L. (Note the deviation from common

graph notation.) Although InfiniBand allows the use of multiple virtual lanes (VLs) for QoS segregation of traffic, unless stated otherwise, we will assume networks to have a single VL. Also, for clarity of presentation we assume all links in the network to have a fixed capacity $c_l = 1$.

A *connection* is a ("logical sender", "logical receiver") pair. (Any given logical sender or receiver resides in a single physical entity.) Packets sent over the same connection must be delivered in their transmission order. A connection may represent a reliable connection (the InfiniBand equivalent of TCP-like socket association) but is not limited to it. Every source can have multiple concurrent connections, each being uniquely identified by a combination of (physical) source address (SLID), destination address (DLID), and a source-unique connection identifier (CID) [1].

Data is transmitted over a connection as a contiguous sequence of *flows*, each flow comprising a contiguous sequence of packets belonging to said connection. Flow is a management abstraction; it may comprise anything between part of an InfiniBand message and a contiguous sequence of such messages.

Unless stated otherwise, we consider an interconnect that operates as follows. Only hop-by-hop flow control is used, whose only function is to prevent packet drops. Thus, sources and switches push packets onto links as long as free buffers are available at a link's remote end. The switches have buffers at input ports, and every output port is allowed to send at most one packet over its link in every time step. An output port serves pending packets from different inputs using a First-Come-First-Served (FCFS) policy, and there is no restriction on the number of packets that can leave an input port's buffer for transmission over different output links in a single time step. (This is known as infinite speedup.) Under this assumption, packets at a given input port's buffer that are destined to different outputs do not suffer from head-of-line blocking, yet they do share a common set of buffers.

[1] In InfiniBand *queue-pair number* (QPN) can be used when possible.

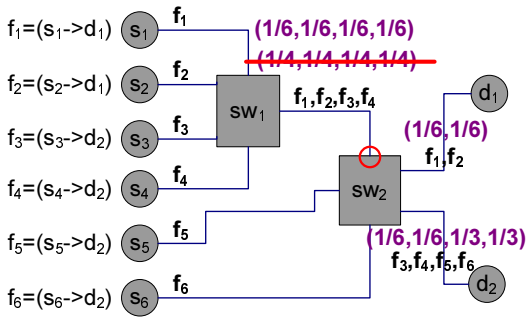


Figure 1.1: The damage of congestion

1.3 The Congestion Problem

In an ideal network, flows do not compete for link capacity, and every flow is transmitted at the line speed, leading to a situation in which no additional management is needed. In practice, however, flows share links, contend for capacity, and their transmission rates are directly affected by the contention. If k flows cross a link l, then at least one of the flows will have an average rate of at most $\frac{1}{k}$. Also, as some of the flows increase their rates above $\frac{1}{k}$, other flows are forced to reduce their rates towards zero. Consequently, reducing contention by means of adaptive routing is expected to improve overall network performance.

Even if optimal routing is used, link sharing may still lead to inefficiencies caused by poor management of shared buffer space. We demonstrate this phenomenon below, but first state our assumptions on the network model. Assume that only hop-by-hop flow control is used, whose only function is to prevent packet drops. Thus, sources and switches push packets to links as long as free buffers are available at a link's remote end. The switches have buffers at input ports, and every output port is allowed to send at most one packet over its link in every time step. Assume also that an output port serves pending packets from different inputs using a Round-Robin (RR) policy, and that there is no restriction on the number of packets that can leave an input-queue for transmission over different output links in a single time step. (This is known as infinite speedup.) Under this assumption, packets destined to different outputs do not suffer from head-of-line blocking, yet they do share a common set of buffers.

Consider the scenario in Figure 1.1. Six flows, $f_1, ..., f_6$, traverse a network that

includes two switches, sw_1 and sw_2. The sources start injecting packets into the network at time $t = 0$. For clarity of presentation, *here only*, we assume that output ports serve pending packets from different inputs using a Round-Robin (RR) policy. Initially, every flow $f_1, ..., f_4$ is transmitted at rate $\frac{1}{4}$ on link $sw_1 \rightarrow sw_2$. The output port $sw_2 \rightarrow d_2$ serves incoming packets in round robin (RR) order among input ports. Therefore, f_5, f_6 each get a rate of $\frac{1}{3}$, while f_3, f_4 each get $\frac{1}{6}$. Initially, the packet insertion rate of f_3, f_4 into sw_2 is thus higher than their service rate. Consequently, regardless of the size of buffers, the input buffer at the end of the link $sw_1 \rightarrow sw_2$ becomes full, which causes the link to reduce its transmission rate. (At moments when no free buffers are present no packet can be transmitted). A simple simulation shows that in the steady state, $sw_1 \rightarrow sw_2$ transmits only at rate $\frac{2}{3}$. Using the accepted terminology, $sw_2 \rightarrow d_2$ is the *root of congestion*, because it is oversubscribed yet transmits at full rate, while $sw_1 \rightarrow sw_2$ is a *congestion victim*, since it is not saturated despite the fact that it has more data to push. As long as the flows do not end, the rate assignment vector is $(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{3}, \frac{1}{3})$.

From the link capacity perspective, the above result is sub-optimal, because in this situation f_1, f_2 should apparently be able to increase their rates to $\frac{1}{3}$. The assignment is moreover unfair, since flows are assumed to be equally important, yet f_5's transmission rate is double that of f_4.

If we consider finite flows of the same size, we notice that the overall completion time (by which all flows end) suffers as well. Assume that all flows have the same large size of a unit of data, which takes a unit of time to transmit at line speed. After three units of time, flows f_5 and f_6 end. At this point $sw_1 \rightarrow sw_2$ has transmitted two units of data and its rate is increased to 1. It will take two more time units to transmit its remaining two data units, making the total completion time to be five units of time. This is 25% worse than the lower bound for a setup in which each link is required to pass at most four units of data.

The above example demonstrates *congestion spreading*, which is a form of high-order head-of-line blocking. It was first described in [33] as tree saturation. In larger networks, the congestion root can create a tree of full buffers. Every flow crossing one of the buffers in the tree, even if it avoids the root, can be adversely affected.

Note that the poor performance cannot be improved by means of changing the routing, since the core problem is not load-balancing, but rather the inefficient buffer sharing. The fundamental cause underlying the phenomenon is that sources attempt to push more traffic than the network is capable of accommodating. Therefore, an appropriate rate control mechanism that limits the injection rate of the sources is expected to remedy the matter and ensure fairness. For instance, in the above examples a suitable solution would be for all flows to be transmitted by the sources at rate $\frac{1}{4}$.

1.4 Performance Measures and Goals

In order to assess the quality of various solutions, we must first state the performance measures of interest. We derive those form three general scenarios considered in this work:

1. A single phase-based application – the cluster is used by a single application that alternates between computation and communication phases. Each communication phase is followed by a global barrier. At the beginning of a communication phase each source knows its destinations and the exact amount of data to be transferred.

2. Independent flows – different flows have no meaningful relations among each other. Sources not necessarily know the size of their flows and how the set of these flows is about to change in the future.

3. Multiple phase-based applications – the cluster is shared by multiple phase-based applications running concurrently. Different applications enter and leave their communication phase independently.

In addition, we assume flows in all the scenarios to have a large size (hundreds to thousands of packets). This assumption is motivated by relatively high overhead, for any control scheme, when (very) short flows are considered. It is assumed that flows have a bulk, rather than sporadic, nature: every time a packet is transmitted, a new one is ready for transmission until the flow ends.

The phase-based applications are typical of High-Performance Computing (HPC), and they usually exhibit regular traffic patterns. As mentioned above, the communication phase is concluded by a synchronization barrier. Therefore, the length of the phase is determined by the latest completion time of the participant flows (the total completion time). Whenever a single application is considered, our goal is to minimize the total completion time.

The independent flows scenario is the default scenario in the research of networking. It may arise in a cluster used for processing of large quantities of data, such as video or 3D graphics. In this scenario, the flows can be considered as consumers contending for shared, limited resources. We want to allocate the resources in a fair and efficient manner. Therefore, our goal is to maximize instantaneous rates of individual flows in a fair manner. We choose to stick to a broadly accepted max-min notion of fairness [4]. We intentionally avoid using completion time of the flows as the optimality criterion, since flows start at arbitrary times and are unrelated, so ensuring the instantaneous progress is a more natural requirement.

The scenario of multiple phase-based applications can be regarded as a combination of the previous two. As with independent flows, our goal is to maximize the instantaneous progress of different applications in a fair manner. At the same time, we use the instantaneous rate of an application's slowest flow as that application's progress indicator, since it is expected to determine the length of its communication phase. Here we will use max-min fairness again, this time at the inter-application level.

1.5 Contributions of this Work

Our discussion of the congestion problem mentioned that congestion can be mitigated by adaptive routing and rate control. In this work, we propose novel approaches for these two mechanisms and examine their combination. In the considered networks, a large number of flows may traverse a network element. As a result, storing and manipulating per-flow state at the elements is an impractical task. For this reason, all our solutions require switches and HCAs to hold state, whose size is

independent of number of flows traversing the network, or its topology.

Our first contribution is the adaptive routing scheme described in Chapter 3. The scheme relies on local heuristic decisions. Importantly, it guarantees in-order packet delivery by means of a kind of virtual circuit (VC) mechanism. However, we avoid the scalability issues typical of "classic" VC schemes. The heuristic nature of the routing is obviously not optimal, and is used similarly in the different characteristic scenarios.

Our second contribution is a set of rate control algorithms, presented in Chapter 4, that are capable of setting optimal rates according to different performance measures for any fixed choice of routing. (In principle, we first adaptively fix a routing and then compute the rates). All our algorithms rely on explicit rate calculation.

The rate calculation algorithms are initially developed and evaluated under an assumption of the so-called "fluid model"; i.e., any rate assignment is considered feasible if it does not violate the capacity constraints of any of the links. This approach ignores the discrete nature of packet networks, queuing issues, and limited buffers. As a result, it is not clear whether the theoretically feasible rates can be indeed implemented in practice.

To this end, we show through simulation that an adapted *Shaped Virtual Clock* (SVC) packet injection scheme [38] is capable of realizing calculated rates even with a very moderate size of buffers in switches. The injection scheme is designed to suppress bursts of individual flows and the aggregate traffic leaving a source. The matter of realization of calculated rates is discussed in Chapter 5.

Finally, we collect and present experimental results achieved through simulation. When adaptive routing or rate control are used individually, minor performance improvement is witnessed. (The only exception is the independent flows scenario, in which rate control alone has a major impact.) In fact, adaptive routing can even hurt. However, when used together, the results demonstrate a very significant performance boost according to the relevant performance measures.

The rest of the work is organized as follows. Chapter 2 describes the related work in adaptive routing, rate control, and packet injection areas. Our proposals on adaptive routing and rate control are presented in Chapter 3 and Chapter 4,

respectively. In Chapter 5, practical issues of rate control are addressed, with the realization of calculated rates among them.

Chapter 6 presents and discusses the empirical results collected through simulation. These results were acquired through simulation in OMNeT++ event-driven simulation framework [1]. We created special InfiniBand models for that purpose. Since our goal was to simulate large networks with thousands of nodes, our models operate at the functional, rather than cycle-accurate, level. Although the methods proposed in this work are topology agnostic, we used the k-ary n-tree [31], which is a variant of a practical fat tree [24], as the basis for all our experiments. This topology is popular in modern clusters. Finally, Chapter 7 offers concluding remarks.

Chapter 2

Related Work

2.1 Adaptive Routing

As defined by the standard, InfiniBand networks rely on oblivious routing. This choice has some key advantages: it greatly simplifies the routing procedure of individual packets, and guarantees in-order delivery. However, this type of routing cannot react dynamically to the applied traffic pattern. No matter how good the a-priori routing configuration is, it will always be possible to find a pattern for which the chosen routing will cause high contention, resulting in poor performance. The alternative is to use adaptive routing, namely to let the network route packets from the same source to the same destination along different paths, according to the existing conditions. Finding an optimal routing, under various optimality criteria, generally leads to one of the variants of the Multi-commodity Flow problem, which is NP-complete problem [13]. Consequently, in practice heuristic approaches are used.

Lossless networks are prone to deadlocks that arise whenever a group of packets can't advance due to a cyclic dependency among them. Therefore, deadlock avoidance is a crucial property of any routing scheme, either static or adaptive. In a fundamental work [10], authors formulate the necessary and sufficient condition for deadlock-free routing. This condition is used as a basis for several topology agnostic routing schemes [30, 37].

In some cases, the deadlock-freedom is inherently supported by the topology of

a network. This is particularly true for practical fat trees (see Section 6.1), in which no cyclic dependency can arise if minimal paths are used[1]. Topology-aware static routing schemes for fat trees were proposed in [15, 45], while adaptive routing for the same topology was examined in [15, 23]. The latter proposed to use packet-level adaptation, routing different packets of the same flow independently. This approach, taken also in [29], is fairly simple to implement, but it breaks the in-order delivery guarantee, thus violating one of the fundamental properties of InfiniBand.

An alternative approach that preserves in-order delivery was proposed in [25, 27, 42]. InfiniBand allows assigning more than one address (LID) to every HCA, so the same destination can be addressed using multiple LIDs. Since the routing for each LID is fixed independently, multiple paths can be defined for each source-destination pair at network configuration. The source is given the power to choose a specific path by choosing a LID from among those assigned to the destination. This approach enforces packet order, provided that the entire flow is routed using the same LID (otherwise additional care must be taken). Multiple-LID routing does not provide full routing flexibility, because the alternative paths are set once, at network configuration time. Moreover, a limited (64K) number of LIDs introduces a tradeoff between the cluster size and its routing capabilities.

Flexibility and in-order delivery are achieved if routing is incorporated into a virtual circuit (VC) mechanism, as proposed in [11, 40]. Generally, virtual circuits is a technique for resource reservation along the path of a connection in packet switched networks. The resources, which can include link capacity, buffers, routing entries etc., are reserved during the *setup* of a circuit, which occurs before the first packet of the connection is sent. When a connection ends, its resources are released in a *tear-down* procedure. Since the path is set up once, and all packets follow it, the in-order delivery is guaranteed even if the path is chosen adaptively.

When VCs are used, finite resources in switch routing tables limit the number of connections that can cross a switch. As a result, some circuits can fail to be set up. If this happens, the connection is said to be *blocked* and its setup has to be restarted

[1] In a practical fat tree, unlike an ideal one, multiple routing alternatives exist for the same source-destination pair.

later. In some schemes alternative paths are tried, but unfortunately those may also be blocked. So, the penalty of blocking may be very high. In Chapter 3, we propose a scheme that uses virtual circuits to route individual flows (rather than connections). Our circuits reserve only routing resources, and efficiently avoid blocking.

2.2 Rate Control

As we saw in Chapter 1, the basic hop-by-hop flow control provides neither efficient nor fair exploitation of link capacity. The undesired effects can be potentially removed if injection rates of flows are appropriately throttled. The goal of rate control is to avoid link oversubscription and the resulting clogging of buffers, while ensuring fair and efficient utilization. We classify existing rate control schemes into four main groups: *reactive window-based*, *reactive rate-based*, *precise explicit rate calculation* and *approximate explicit rate calculation*. To the best of our knowledge, all existing solutions were designed for the independent flows scenario.

In reactive schemes, a source increases the load of a flow optimistically, until it receives some kind of indication that the flow passes through a congested link. The source reacts by reducing the applied load until the flow is believed not to cross congested links again. Reactive schemes usually never reach a truly steady state; instead, they exhibit small oscillations around the desired solution.

Every reactive scheme can be characterized by three main criteria. First, the amount of load that a flow may apply on the network is determined either by the size of a congestion window, or using an explicit rate value. Second, the congestion indication can be implicit or explicit. Implicit indication is derived from long round-trips or (in lossy networks) from sudden packet loss. In the explicit case, switches play an active role in informing the source, usually by marking data packets crossing the congested link. Finally, various schemes differ in the increment/decrement policy, where the change in applied load is optionally dependent on the acquired feedback. With Additive-Increase, Multiplicative-Decrease (AIMD) [9], for example, the new load is a function of its current state only.

Probably the most commonly known congestion control mechanism is that of

TCP. It is a reactive, window-based scheme, which has spawned many variations over the years to suit different network environments [5, 14, 22, 43, 44]. The congestion window controls the number of unacknowledged packets that may be injected into the network. The typical network traversed by TCP traffic is characterized by relatively long round-trip delay (high bandwidth-delay product) and switching elements with large buffers, which makes the congestion window a convenient tool.

Cluster networks, in contrast, do not exhibit these properties. Virtual cut-through switching makes the bandwidth-delay product very low even for networks with large radii. In fact, in most practical scenarios, the maximum window size should be no more than several MTU packets per flow, leaving littel space for any substantial control. Moreover, with very small buffers, congestion spreading can occur even if the window size is fixed to one packet, as demonstrated in [35]. For these reasons, InfiniBand *Congestion Control Architecture* (CCA) uses *Inter-Packet Delay* (IPD), rather than window size, to set the desired load. Thus, CCA is a reactive, rate-based control scheme.

The components of CCA are presented in Figure 2.1. Switches monitor the number of packets awaiting transmission on each of the output ports. If an output port transmits at full line speed, yet the number of awaiting packets exceeds a predefined threshold, the port is said to be a *congestion root*. If the threshold is crossed, but the port does not operate at full speed (due to the back pressure), it is called a *congestion victim*. Basically, congestion roots generate the back pressure, which creates congestion spreading and congestion victims.

When a switch detects one of its ports to be a congestion root, it starts marking packets that use the port with *Forward Explicit Congestion Notification* (FECN) bit. The marking rate is a parameter of the scheme. A marked packet proceeds to its destination; upon its arrival, a special *Backward Explicit Congestion Notification* (BECN) packet is sent back to the source. Sources hold a table of increasing IPDs, known as the *Congestion Control Table* (CCT) and an *index* variable pointing to the table. When a source receives a BECN, the index is incremented and the rate is effectively reduced. A *timer* is used to decrease the index (increase rate) over time, as long as no BECNs are received.

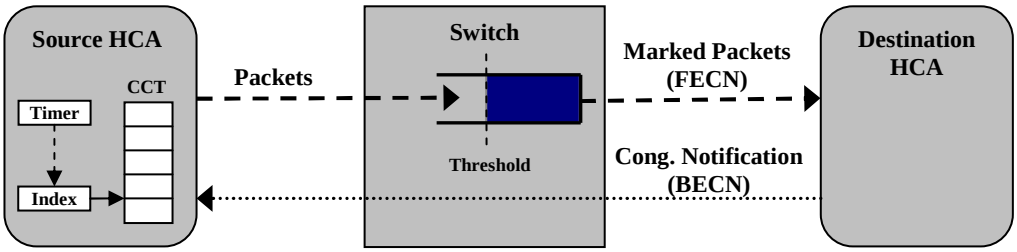


Figure 2.1: Congestion Control Architecture

The multitude of parameters (threshold, marking rate, CCT values, decrease time) is a serious obstacle for correct configuration of the scheme. The multitude of parameters (threshold, marking rate, CCT values, decrease time) is a serious obstacle for correct configuration of the scheme. Our attempts to choose CCA parameters lead to a grim conclusion whereby a setting (set of control parameters, not a specific transmission rate) that is optimized for one traffic pattern can have catastrophic results for another.

For example for "all-to-one" traffic pattern the link connected to the destination HCA is a congestion root, as it bears a very large number of flows. Consequently, the switch should mark every crossing packet in order to provide feedback to all flows in a reasonable time. However, marking every packet when permutation traffic, with much lower contention, is applied, causes switches to over-react and leads to rate oscillation without convergence. The reduced marking rate that fits the permutation traffic does not avoid buffer clogging and the congestion spreading under the all-to-one traffic.

Nevertheless, attempts to set CCA parameters were made in previous works. Analytic derivation such as those in [35] and [45] were never tested in networks of a realistic size with non-trivial traffic. At the same time the systematic simulation method chosen in [32] suffers from partial sample space coverage, typical for this approach. The apparent conclusion is that CCA requires careful tuning for each particular topology and traffic pattern in order to control the fabric efficiently.

Unlike their reactive counterparts, explicit rate calculation schemes actively compute rate values to be assigned to flows. The rate assignment is chosen to utilize links efficiently and fairly, while preserving the feasibility. Explicit rate calculation

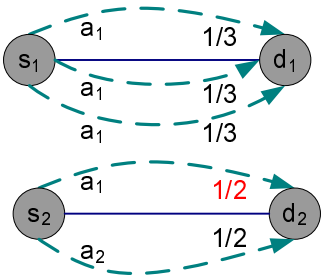


Figure 2.2: Failure of existing rate control

makes it easier to define clear and flexible design goals that provide firm guarantees on performance. The calculation itself is usually performed in a distributed manner. *Precise* algorithms [7, 8, 20] do not depend on specific network characteristics and are guaranteed to converge to a steady solution under very weak assumptions. *Approximate* algorithms [3, 12, 19, 21, 41] are expected to operate faster, but they rely on parameter tuning and may exhibit oscillatory behavior.

More details on various rate control techniques can be found in [18, 28]. We note that even when operating ideally, the existing approaches to rate control have inherent limitations. First, most of the existing schemes do not include flow weights, as an instrument of prioritization, which will be used in Chapter 4. Second, none of them can operate correctly in the multiple application scenario (as defined in Section 1.4). We demonstrate this using the example in Figure 2.2.

The example features five flows. Three of them traverse the link $s_1 \rightarrow d_1$ and belong to application a_1. The other two traverse the link $s_2 \rightarrow d_2$, such that one of the flows belongs to $a_1$and the other to a_2. Since existing rate control schemes have no information on the association between flows and applications, the only reasonable result for them would be assigning a rate of $\frac{1}{3}$ to the flows on $s_1 \rightarrow d_1$ and $\frac{1}{2}$ to those on $s_2 \rightarrow d_2$. Recall that the progress of each application is defined by the progress of its slowest flow. Therefore, a_1 unnecessarily uses too much capacity on $s_2 \rightarrow d_2$, which could improve the progress of a_2.

The algorithm that uses precise, explicit rate calculation to assign correct rates for the example is presented in Section 4.3. Additional algorithms for rate assignment in single application and independent flows scenarios are described in Section 4.1 and Section 4.2, respectively.

2.3 Packet Injection

When explicit rate calculation is used for rate control, an additional effort has to be made to realize the acquired rates. In this context, the packet injection scheme employed by a source has a key role. Such a scheme has to enforce the calculated rates and to multiplex different flows onto a common outgoing link. We assume that flows have packets of a fixed size (to be partly relaxed later). Since InfiniBand switches typically have very small buffers, it is highly desired that bursts be suppressed in the aggregate outgoing traffic. Because and long flows are delay insensitive, so doing does not hurt them.

Although the injection problem appears related to the domain of real-time scheduling (E.g., rate-monotonic scheduling [26]), the fundamental goals in the two cases differ. As in real-time scheduling, we are interested to co-schedule periodic processes (flows originating packets at the computed rate). However, unlike there, we are also interested to reduce burstiness of the traffic.

Leaky bucket [39] is a well known technique for traffic shaping of an individual flow. It is designed to enforce average transmission rate, while allowing bursts of a bounded size (fixed bound). When multiple flows are considered, packets exiting different buckets (each associated with a different flow) are inserted into a FIFO queue, from which they are sent out at line speed [6, 34]. Assessing compatibility of this mechanism for our needs, we draw the following conclusions: 1) the allowed burst size of each flow would have to be set to one packet, and 2) the FIFO at the output could still potentially create bursts when the aggregate traffic is considered.

An alternative approach to packet injection is *Shaped Virtual Clock* (SVC) [38]. Its basic operation can be described as follows. When transmission of some packet is finished, the flow f to provide the next packet is selected such that: 1) the immediate transmission of f's packet would not exceed f's calculated rate, and 2) f has so far transmitted the smallest amount of data relative to its rate. SVC inherently prevents bursts of any single flow, but bursts of the aggregate traffic can still occur.

In Chapter 5, we adapt the SVC approach to our needs by forcing the source to transmit packets at most at its aggregate (over all its flows) rate, thereby completely eliminating bursts. We empirically verify the realizability of calculated rates under

realistic sizes of input buffers in switches.

Chapter 3

Adaptive Flow Routing

In Chapter 2, the existing approaches to adaptive routing were surveyed. There, we concluded that using virtual circuits for routing is the only approach that combines in-order packet delivery with routing flexibility. In this chapter, we propose a generic routing scheme based on flow-level VC-like routing. Our main goal is to obviate the need in storing routing information for every flow crossing a switch, hence effectively avoiding scalability limitations that are inherent to "classical" VCs. The application of the generic scheme in the fat-tree topology and the associated empirical results are discussed in Chapter 6.

3.1 Routing Scheme

As defined in Section 1.3, a connection is a sequence of flows, each of which is a sequence of packets. In-order delivery is required among all packets of a connection. Each packet carries a combination of source address (SLID), destination address (DLID), and a source-unique connection identifier (CID), which jointly constitute a *globally unique connection identifier* (GCID). Our approach is to adapt the routing at flow boundaries, with all packets of any given flow following the same path. Furthermore, transmission of packets of given flow does not commence until the source ensures that all packets of the previous flow of the same connection have arrived.

Data Structures

In large clusters using high-radix switches, the number of flows traversing any given switch may be very large. When "classical" VCs are used, this gives rise to a scalability issue (since per-flow routing information has to be stored) and to a possible performance problem (associative lookup in a large table). We avoid this problem through intelligent use and extension of the standard InfiniBand routing table (InfiniBand RT). This table has a constant size equal to the maximum allowed number of destinations (defined by the standard), and is addressed (as a regular table, not associatively) by DLID. Therefore, InfiniBand RT is accessed very quickly. Each entry in InfiniBand RT holds a port number.

We assume each switch to have a single *extended routing table* (ERT), which is an extension of the regular InfiniBand RT. Similarly to the InfiniBand RT, it has an entry for each destination in the network, and is directly addressed by the DLID. However, ERT entries have two fields. The first, as before, holds a port number, which we refer to as the *default port* (DP) for the DLID. The second field is used to store a list of several *alternate ports* (APs) for the destination.

We refer to the table of VC routing entries as *route cache* (RC). It stores port numbers, and is associatively addressed by GCID. Since we aim to provide a scalable solution, the size of this table is irrelevant for correctness (but of course has an impact on performance). For the same reason, unlike in "classical" VC, not all flows crossing a switch have an entry in RC. Packets belonging to flows that do not have an associated entry in RC are routed by the switch to a DP according to their DLID.

We find it preferable not to hold a centralized RC, but rather to distribute it in the form of a *port routing cache* (PRC) for each input port. In this manner only information regarding flows crossing a specific input port are stored in any given table, and different PRCs can be accessed in parallel. The PRC still has to be associatively addressed by GCID. For that purpose, it can be implemented as a *Content Addressable Memory* (CAM) or as a hash table.

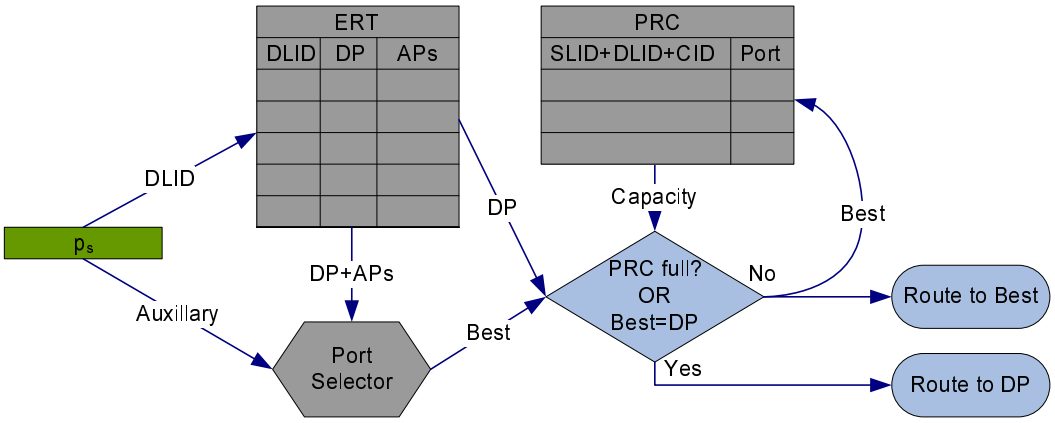


Figure 3.1: Route setup

Route Setup

When a flow f starts, a control packet p_s is sent to set up the route. Each switch on p_s's path routes the packet, basing its heuristic decision on local information. The routing procedure is presented in detail in Figure 3.1. Upon arrival of p_s to an input port, both the ERT and the PRC are accessed. The ERT is accessed by DLID, and it passes a list of all the matching output ports to a *port selector unit* (PSU). The PSU retrieves the heuristic measure for each port and selects the best available port, based on the heuristic and other optional information (to be discussed below). At the same time, the PRC returns its free capacity. If the PRC appears to be full, or the DP is found to be the best output port, then p_s is routed to the DP, and no entry is added to the PRC. Otherwise, p_s is routed through a chosen alternate port. In the latter case, the chosen output port is stored in the PRC along with p_s's GCID. Note that after being routed on a DP at some switch, p_s can still be adaptively routed at later switches on its path.

Since p_s advances through switches regardless of the state of RC, it should eventually reach its destination. Once this happens, the packet is sent back to the source on a *default path* (defined by default ports). On its way back, p_s does not affect the state of the switches.

Upon p_s's return, the route setup procedure is complete, and the source starts sending data packets. These packets are routed by switches according to PRC entries that match their GCID. If at some switch no matching entry is found, the packet is

sent to an appropriate DP based on its DLID. As shown below, data packets of any given flow are guaranteed to traverse the network on the same path and to arrive at the destination in order.

Route Tear-Down

After a flow's last data packet is sent, the source waits until the receipt of this packet is acknowledged by the destination. This is the only *data* packet that requires an acknowledgment (control packets are returned to the source as well though). When the acknowledgment arrives, a control packet p_t is sent to perform the tear-down of flow's path. This packet is routed like a regular data packet, and it erases matching PRC entries on its way.

The source will not set up a route for a new flow within the same connection until p_t returns. This behavior guarantees that a new flow is routed only after the source learns that all packets of the previous flow have arrived and the path has been torn down. (Tear-down completion enables the reuse of the GCID without a risk of confusion between new and old routes, which could occur if a p_s sets up a faster route that overtakes the p_t of the previous flow along an alternate sub-path, and subsequently rejoins its route, now ahead of it.) Optionally, p_t can be piggy-backed on the last data packet of the flow.

Correctness

The above generic scheme does not state how the path of p_s is chosen among the alternatives; i.e., how the ERT is configured and the ports are selected by the PSU. However, the correct operation requires this path to include no cycles. Note that this is a necessary condition for deadlock-freedom as well. The correctness of our approach is summarized in Theorem 3.1.1.

Theorem 3.1.1. *If for every flow $f \in F$ the path of p_s is acyclic, and in every switch, packets advance between any given (input port, output port) pair in the order of their arrival, then the in-order delivery within every connection is preserved.*

Proof. Consider an arbitrary connection with I being its GCID. Let f_1 be the first flow belonging to the connection. Since GCIDs are globally unique and f_1 is the

first flow, when f_1 starts, no RC in the network contains a matching entry for I. Let π_1 be the path (a sequence of network elements) of p_s.

Since inside switches the order of the packets is preserved, it is sufficient to show that all f_1's packets follow π_1, to prove the in-order delivery within f_1. Assume by contradiction, that there exists some packet of f_1 (including p_t among the considered) that doesn't follow π_1. Let p be such a packet, whose path has a shortest common prefix with π_1. Denote by e the last element on the common prefix. Note e must be a switch.

First assume p to be a data packet. Let t be the arrival time of p to e. If at t the last data packet of f, denoted as p', is already sent by the source, then it still follows p on π_1. To see this, recall that p' shares the path with p at least until e, and that packets on the same path are advancing in their original order.

As a result, at t, p_t is not sent yet $\Rightarrow$ no other flows with GCID of I have started. In addition, we know that p_s visits e only once (π_1 is acyclic). Therefore, at t, the state of e's RC in regards to I is similar to its state just after the routing of p_s (by e). Note that this conclusion is also true if $p = p_t$, because a new flow cannot start before p_t returns to the source. Therefore, in any case, p follows p_s for one more hop after $e \Rightarrow$ a contradiction.

To this end, we showed that all packet of f_1, including p_t, follow the path of p_s. Particularly, this means that when p_t returns to the source, no RC in the network contains a matching entry for I. Therefore, the above proof is trivially repeated for a new flow f_2 within the connection I.

More generally, using induction on the serial number of a flow within the observed connection, we show that: 1) all packets of the same flow follow the same path, 2) all packets of a flow reach its destination before a new flow starts. These two conditions are sufficient for the correctness of the theorem. □

Reducing the ERT Size

The regular InfiniBand RT stores an entry per every possible destination in the network (currently 64K entries according to the standard), which holds a single port number. According to the description above, every ERT entry is required to

hold multiple port numbers (up to the radix of the switch). As a result, the relatively large size of the InfiniBand RT is multiplied, which may make the implementation of the ERT impractical. It would, therefore, be useful to try and reduce its size.

To this end, we observe that oftentimes there is a large overlap among the sets of ports that can be used to reach any given destination. Accordingly, we construct *groups of alternate ports* (GAPs). (Any given port can be in multiple groups.) An ERT entry holds a port number of a DP and a group number (no more than 5-6 bits) of one (or more) GAP. A separate *port group table* (PGT), indexed by group number, provides the set of output ports belonging to the group. Consequently, the size of the ERT is only slightly increased compared to the InfiniBand RT, while the size of the additional PGT is expected to be very small (32-64 entries).

Importantly, the PGT is accessed *only* during route setup because for data packets just the DP is used. Therefore, the access time to the ERT in the common case is unaffected.

3.2 Adaptation Policy

3.2.1 The Choice of Heuristic

Our adaptation scheme, similarly to other local adaptation approaches, relies on the ability of an individual switch to asses the quality of its output ports using some local heuristic. Obviously, successful adaptation depends on the correct choice of the applied heuristic (along with some good luck).

Packet level adaptation approaches [15, 23, 29] typically rely on buffer occupancy at the far end of port's outgoing link as the quality metric for that port. Such information is short-lived and it only reflects the instantaneous state of the link. We find it unsuitable for routing whole flows because we don't want long-term decisions to be taken based on a short-term measure.

The number of flows crossing a link, or in the weighted case their aggregate weight, is a simple and more appropriate alternative. To keep track of the number of flows, we need switches to hold a counter per output port. This counter is incremented when a first packet of a flow p_s passes through the port and decremented in

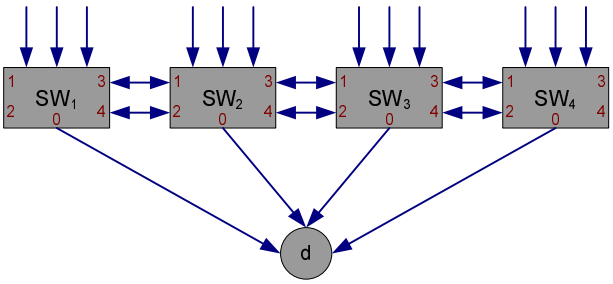


Figure 3.2: Dynamic use of alternate ports

response to the last packet p_t. Other more sophisticated heuristics can be considered in future work.

3.2.2 Adaptation Flexibility

In order to allow for maximal adaptation flexibility, the adaptation policy may require that the same switch, for the same state of the ports, route two flows heading to the same destination differently.

For example, consider the topology depicted in Figure 3.2. Flows arrive to the switches, and need to be routed to the destination node d. In order to provide routing flexibility, we want flows to be allowed to move horizontally before descending to d, as long as the heuristic measure for horizontal ports is strictly better than that of the descending one.

The resulting paths must not include cycles, so we restrict any given flow to follow (at most) a single horizontal direction (right/left). Since we want to maximize the number of available horizontal hops, flows initially arriving to switch SW_2 are routed to the right (same with SW_1; flows initially arriving to SW_3, SW_4 are routed to the left). However, flows arriving to SW_2 from SW_3 must be routed left. Therefore, aside from the heuristic, additional information has to be used in order to achieve a desired result.

Now, we will discuss a possible solution for the above example. Actually, we are interested in this scenario because we will use a similar adaptation policy in Chapter 6 for routing in a modified fat-tree topology. Although we concentrate on a particular case, the presented concepts can be considered as a part of a larger, general

adaptation framework. Such a framework should allow switches to be configured in a simple manner to implement different adaptation policies in various topologies. This framework is outside the scope of the current work.

Returning to our example, we begin by selecting the descending port (output port 0, OP-0) at every switch as the DP, such that when the PRCs are full, the switches route flows directly to d. The definition of the problem requires that the PSU prefer the DP over every alternative port if the heuristic measure for both is equal. Therefore, OP-0 is configured to have a higher *static priority*, which serves as a tie-breaker.

Additionally, we can conclude that the choice of horizontal direction when routing a flow *depends on the identity of the input port* from which that flow arrives to the switch. Consider SW_2 again. It has five outputs that can be potentially used for routing an arriving flow. All these outputs appear in the ERT and are passed to the PSU for the adaptive selection. If a flow arrives to SW_2 from the upward direction or from SW_1, the PSU removes OP-1 and OP-2 from consideration before proceeding to the best port selection. Otherwise (the flow arrives from SW_3), OP-3 and OP-4 are ignored. The remaining switches are configured in a similar manner. Note that such configuration of the PSU only requires a small table that holds a list (binary vector) of the "allowed" outputs per every input.

If the number of switches in the example of Figure 3.2 were high (rather than four), we might choose to limit the number of horizontal hops a flow can take before descending, to prevent the creation of extremely long paths. For this purpose, the route setup packet p_s should carry a hop-counter field. The counter is initialized to the allowed number of horizontal hops when the first horizontal hop is taken, and is decremented after every advance in the horizontal direction. Once the counter reaches zero, the PSU sends the packet to the DP.

Implementation of sophisticated adaptation schemes, such as the one presented above, is expected to increase the routing delay of p_s compared to the existing static routing. However, the penalty is incurred only once per flow, during the path setup. The routing of data packets is performed by just accessing the ERT and the PRC tables. Therefore, for long flows, the overhead of the setup is expected to be

effectively amortized.

Chapter 4

Explicit Rate Calculation

In Chapter 1, we defined our performance goals, and introduced the need for the combination of adaptive routing and rate control to mitigate congestion. The existing work on both subject was discussed in Chapter 2. In this chapter, we propose a rate control method based on precise explicit rate calculation. For each of the three representative scenarios and associated performance goals, we propose one or more optimal algorithms for rate calculation under the assumption of a "fluid model", and present its distributed version. In addition, we formally compare the different algorithms. The discussion of implementation of the calculated rates is postponed to Chapter 5. The simulation results associated with rate control are presented in Chapter 6.

4.1 A Single Phase-Based Application

As mentioned in Chapter 1, the goal is minimization of total completion time of the communication phase. This means that we do not care about the behavior of individual flows, as long as their completion time does not exceed the total completion. Below, we present two distributed algorithms that achieve the optimal total completion time. Importantly, the algorithms require network elements to store a small amount of state information, independent of number of flows or network topology. Moreover, the size of control packets that are used for calculations is fixed as well.

4.1.1 The Optimal Completion Time

Let us begin by presenting a precise mathematical model. We assume the routing to be fixed for the duration of the calculation. Let all links $l \in L$ have the same capacity of 1[1]. Denote by F_l the set of flows crossing link l, and by L_f the set of links on the path of flow f. Let every $f \in F$ have an associated weight w_f being equal to its size d_f. Later we will regard w_f in units of time, i.e., the time it takes to transmit f at the line speed[2]. Let W_l be the aggregate weight of flows crossing $l \in L$. Denote by W_f the maximum W_l among links in L_f. Finally, let W be the maximum link weight in the network. The precise relations between different values are given by:

$$W_l = \sum_{f \in F_l} w_f,\ W_f = \max_{l \in L_f} \{W_l\},\ W = \max_{l \in L} \{W_l\} = \max_{f \in F} \{W_f\}. \tag{4.1}$$

Since we consider long flows, we ignore small disparities among the starting times of different flows, and assume all flows belonging to the communication phase to start at $t = 0$ (the beginning of the phase). Particularly, we forbid flows to start in the middle of the phase. The control of the network is reduced to assigning each flow $f \in F$, at any given time τ, an instantaneous rate $r(f, \tau)$, without violating the capacity of the links. The amount of data transmitted for f, by time t is given by:

$$D(f, t) = \int_0^t r(f, \tau) d\tau. \tag{4.2}$$

The total completion time is defined to be the smallest time by which the data of all flows has been transmitted[3]. Our goal is to find a feasible rate assignment vector r that is constant in time, i.e. $r(f, \tau) = r(f)$, and guarantees the shortest total completion time (later we prove that the optimum can be achieved by constant rate assignment).

First, consider a scenario in which all flows have the same size and weight $w_f = 1$.

[1] This assumption is used for facility of exposition; all our methods can be trivially extended for the general case.

[2] We assume potential disparities in units to be solved by appropriate constant coefficients.

[3] Long flow length allows us to neglect the difference between transmission and arrival time.

Thus, the maximum link weight W is, in fact, an integer. Giving $r(f) = \frac{1}{W}$ to all flows minimizes completion time. It is furthermore the most resource conserving, as it assigns to each flow the minimum rate required for minimizing completion time. Unfortunately, however, it requires some kind of global coordination mechanism (a type of broadcast)[4].

We next make two important observations: 1) giving flows higher rates than the above minimum does not delay the completion time, and 2) so doing does not increase the amount of transmitted data, as a completed flow will cease to consume bandwidth. With this in mind, we next propose a simpler rate assignment.

Setting $\forall f \in F : r(f) = \frac{1}{W_f}$ results in a feasible rate assignment. Moreover, since $\frac{1}{W_f} \geq \frac{1}{W}$, the optimum total completion time is still guaranteed. When flows of different size are considered, we define the normalized rate to be $\bar{r}(f) = \frac{r(f)}{w_f}$. Theorem 4.1.1 shows that setting $\bar{r}(f) = \frac{1}{W_f}$ results in an optimal assignment, which we refer to as *Single Application Assignment* (SAA).

Theorem 4.1.1. *Given a set of flows with respective sizes and a route for each flow, assigning to every $f \in F$ a rate $r(f) = \frac{w_f}{W_f}$ (equivalently $\bar{r}(f) = \frac{1}{W_f}$) is feasible and achieves a completion time W, which is the minimum.*

Proof. $\forall l \in L, \forall f \in F_l : r(f) \leq \frac{w_f}{W_l}$. Therefore, $\forall l \in L : \sum_{F_l} r(f) \leq 1$, so the assignment is feasible.

The link bearing W needs at least W units of time to transmit all of the applied data, so the lower bound on the completion time is W.

Finally, $\forall f \in F$, the completion time is given by $\frac{w_f}{r(f)} = W_f \leq W$, so the lower bound is achieved. □

Corollary 4.1.2. *A routing that minimizes W, combined with SAA, achieves the globally optimal completion time*

Although our adaptive routing, described in Chapter 3, does not achieve the minimum W, as required in Corollary 4.1.2, it is designed to reduce the maximum

[4]This can be the best approach from the practical point of view when the problem is extended to include *multicast* flows.

contention heuristically which leads to a direct improvement of the performance as well.

Now, let us make some additional definitions. We say that a rate assignment satisfies *SAA requirement* if $\forall f \in F : \bar{r}(f) \geq \frac{1}{W_f}$. An algorithm that satisfies SAA-requirement is said to *implement SAA*. It is important to note that SAA does not maximally exploit link capacity; i.e., some flows may be capable to increase their rates without affecting others.

An algorithm that satisfies both the SAA-requirement and maximality is said to *implement SAA-M*. In the same network, with the same flows and routing, different SAA-M assignments can be found with potentially different $\sum_F r(f)$. All of them are, however, maximal from the flow perspective. Note that SAA-M does not affect the total completion time, but does allow some flows to end earlier. We will find this type of algorithms useful in Section 4.2 for the independent flows scenario.

In the following subsections, we present algorithms that implement SAA and SAA-M. The algorithms are easily adapted to operate in a network with links of varying capacity by redefining link weights to be:

$$W_l = \frac{\sum_{f \in F_l} w_f}{c_l}. \tag{4.3}$$

4.1.2 SAA Algorithm

In SAA, the rate of an individual flow depends on a single parameter, W_f. Our goal is to enable every flow to derive its W_f value. In order to do so, links must track their weights (similarly to the proposed for adaptive routing in Chapter 3), and flows to periodically check link weights along their paths. Note, that a single sample at the starting time of a flow is not sufficient, since flows are allowed to start at slightly different times at the beginning of a communication phase.

The detailed behavior of flows and links is summarized in Algorithm 4.1. The subroutines of the main algorithm are presented in Algorithm 4.2. The subroutines are used for communication between a flow f and links in L_f. In fact, it is the source of f that communicates with network elements on f's path, but we find it convenient to use the flow-link abstraction and terminology.

Algorithm 4.1 SAA

Initialization (at network setup):
$\forall l \in L : W_l \leftarrow 0$

Upon the start of flow f:
1. $W_f \leftarrow$ `AnnounceStart(`w_f`)`
2. $r(f) \leftarrow \frac{w_f}{W_f}$

Periodically:
1. $W_f \leftarrow$ `ProbeLinks()`
2. $r(f) \leftarrow \frac{w_f}{W_f}$

Upon the end of flow f:
1. `AnnounceEnd(`w_f`)`

Algorithm 4.2 SAA subroutines

`real` **AnnounceStart**(w_f)

1. `send` $p : p.w_f \leftarrow w_f, p.W_f \leftarrow 0$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l + p.w_f$ /*Update weight*/
   2.2 $p.W_f \leftarrow \max\{p.W_f, W_l\}$ /*Collect $\frac{1}{W_f}$*/
3. `return` $p.W_f$ `when` p `arrives back to the source`

`void` **AnnounceEnd**(w_f)

1. `send` $p : p.w_f \leftarrow w_f$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l - p.w_f$ /*Update weight*/

`real` **ProbeLinks**()

1. `send` $p : p.W_f \leftarrow 0$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $p.W_f \leftarrow \max\{p.W_f, W_l\}$ /*Collect $\frac{1}{W_f}$*/
3. `return` $p.W_f$ `when` p `arrives back to the source`

Each subroutine involves sending a control packet p to the destination of f, which carries information, from f to links in L_f. In addition, at each link, p's arrival triggers a certain action that affects the state of the link, and p potentially collects some data from the link. Whenever p reaches the destination, it is sent back to the source with the collected data (on the returning path p is simply forwarded without affecting link state). The collected data is returned to f at the end of the subroutine.

We make the following remarks. First, we make no assumptions on the return path of p; in particular, it can differ from the forward path. In addition, during the invocation of a subroutine a single control packet is sent, and the subroutine ends only after its return. Therefore, only a single control packet per flow can be in-flight at any given time. We will make similar use of subroutines when describing other algorithms later.

At the beginning of a communication phase, each flow (independently) executes `AnnounceStart()`, which updates the weight of links in L_f and collects an initial W_f. In practice, this procedure uses the first packet of the flow, and can thus be combined with establishing the route of f adaptively. When f ends, it updates the links again by executing `AnnounceEnd()`. In between, the flow samples the relevant weights using `ProbeLinks()`, which can potentially be piggy-backed on data packets. In this case, the need to use ACKs to return the collected weight can be obviated in the following manner.

Let each data packet carry two weight fields $p.W_f^c$ and $p.W_f^u$. The first holds W_f as it was known to f at the time when p left the source, and the second collects the updated maximum encountered link weight. The source has to be notified only if the two weights do not agree when p reaches its destination.

It is not hard to see that after all flows enter the network and set correct weights on the links, every flow is guaranteed to eventually acquire a correct W_f. From then on, for each $f \in F$, the known W_f stays unchanged until f ends. To see this, note that flows crossing $l \in L_f$ for which $W_l = W_f$ cannot end before f. As a result, f is assigned the correct rate throughout its transmission. Finally, we note in passing that if the traffic pattern remains the same for several communication phases, rate assignments can be reused.

Calculation Time

After all flows announce start, it takes each flow a single probing to acquire the correct rate. The frequency of sampling can be tuned (statically or dynamically), and as mentioned above, it can be piggy-backed on data packets. The probing itself is completed within a single round-trip delay.

The frequency of sampling can be tuned (statically or dynamically), and as mentioned above, it can be piggy-backed on data packets. The probing itself is completed within a single round-trip delay. This delay comprises propagation and queueing components. (Even if control packets are given absolute priority, they still wait for each other.) The queueing delay, however, is greatly reduced by the fact that each flow has in flight at most a single control packet at any time.

The propagation delay may vary with topology and routing. Generally, we expect it to be bounded from above by around $10\mu s$. The queueing delay highly depends on the traffic pattern. In presence of higher contention, the queueing delay and the calculation time grow. However, if we consider calculation overhead to be the product of calculation time and a flow's transmission rate, the queueing delay is not directly translated into overhead, as lower transmission rates are expected. Nevertheless, the calculation time of SAA is by far lower than of any existing scheme (which requires multiple round-trips to converge). Finally, we note in passing that if the traffic pattern remains the same for several communication phases, rate assignments can be reused.

4.1.3 SAA-M Algorithm

By definition, an algorithm that implements SAA-M has to guarantee each flow a normalized rate of $\frac{1}{W_f}$ and to distribute any excess claimable capacity left (some of this capacity can't be claimed simply because all flows crossing a link are bottlenecked by other links). Importantly when SAA-M is employed, as opposed to DFFA, changes of$F$ in the steady state, trigger only local recalculations, as will be shown later.

In order to simplify the algorithm and make it work faster, we choose to let every flow attempt to claim as much excess capacity as it can in a greedy manner. The greedy capacity claiming can lead to a situation in which several flows contend for the same free capacity. The unnecessary oversubscription is avoided by forcing the flows to *reserve* the intended capacity before actually *claiming* it. If the reservation fails, the flow backs off for a random amount of time before retrying. Nevertheless, as flows come and go, some links can become oversubscribed (i.e., the sum of claimed rates

of flows in F_l is greater than the total capacity). In this case, an oversubscribed link forces its flows to reduce their normalized rates to $\frac{1}{W_l}$, by marking control packets that cross it. Unlike in InfiniBand CCA, marking rate is not a parameter as all crossing packets are marked (which has no negative impact on the convergence).

In order to support the desired behavior, we require each link to store a small, fixed amount of state in three variables:

- W_l – link weight
- c_l – free link capacity, initially (at network setup) 1, can be negative when link is oversubscribed
- c_l^r – reserved, not yet claimed, capacity

Detailed Algorithm

The detailed description of the algorithm is given in Algorithm 4.3; its subroutines are found in Algorithm 4.4. Similarly to SAA, a new flow f uses `AnnounceStart()` to update the weights of links in L_f and to collect the initial W_f, which is used to set the rate $r(f) \leftarrow \frac{w_f}{W_f}$. Unlike in SAA, the links in L_f need to be updated about the rate increase, so that they can track c_l correctly. For this purpose, `UpdateLinks()` is called with $r(f)$ as its first argument Δ. The execution of `UpdateLinks()`reduces c_l by Δ for all links in L_f. The second argument is used to cause the links to reduce c_l^r when reserved capacity becomes claimed.

Again like in SAA, after the start of f is completed, the flow periodically samples the state of its path with `ProbeLinks()`. However, now, `ProbeLinks()` collects three types of data from the links along the probing flow's path: 1) the up-to-date W_f value, 2) free potentially claimable capacity c_u (which is the minimum free capacity on the path), and 3) an indication whether an oversubscription occurs on one of the links, in form of a weight value W_o. When no oversubscription is present, the returned value is $W_o = 0$. Otherwise, W_o equals the maximum W_l among the oversubscribed links in L_f. Note that a link l is guaranteed not be oversubscribed if $\forall f \in F_l : r(f) \leq \frac{w_f}{W_l}$.

Algorithm 4.3 SAA-M

Initialization (at network setup):
$\forall l \in L: c_l \leftarrow 1, W_l \leftarrow 0, c_l^r \leftarrow 0$

Upon the start of flow f:
1. $W_f \leftarrow$ `AnnounceStart(`w_f`)`
2. $r(f) \leftarrow \frac{w_f}{W_f}$ `/*`*Start by assigning SAA rate*`*/`
3. `UpdateLinks(`$r(f)$, 0`)`

Periodically:
1. $[W_f, W_o, c_u] \leftarrow$ `ProbeLinks()`
2. `if` $r(f) < \frac{w_f}{W_f}$ `then /*`*Current rate below SAA rate*`*/`
   2.1 $\Delta \leftarrow \frac{w_f}{W_f} - r(f)$
   2.2 $r(f) \leftarrow \frac{w_f}{W_f}$ `/*`*Increase rate to SAA rate*`*/`
   2.3 `UpdateLinks(`Δ, 0`)`
3. `elseif` $W_o > 0$ `then /*`*Oversubscribed link encountered*`*/`
   3.1 $\Delta \leftarrow \min\left\{\frac{w_f}{W_o}, r(f)\right\} - r(f)$ `/*`*Reduce rate if necessary*`*/`
   3.2 $r(f) \leftarrow \min\left\{\frac{w_f}{W_o}, r(f)\right\}$
   3.3 `UpdateLinks(`Δ, 0`)`
4. `elseif` $c_u > 0$ `then /*`*Claimable capacity found*`*/`
   4.1 $res \leftarrow$ `MakeReservation(`c_u`)`
   4.2 `if` $res = true$ `then /*`*Reservation successful*`*/`
      4.2.1 $r(f) \leftarrow r(f) + c_u$
      4.2.2 `UpdateLinks(`c_u, c_u`)`
   4.3 `else /*`*Reservation failed*`*/`
      4.3.1 `CancelReservation(`c_u`)`
      4.3.2 `back-off for random amount of time`

Upon the end of flow f:
1. `AnnounceEnd(`$w_f, r(f)$`)` `/*`*Update link weights and capacity*`*/`

The reaction of f after `ProbeLinks()` differs depending on the returned values. If $r(f)$ is found to be lower than $\frac{w_f}{W_f}$ (not satisfying the SAA requirement), f increases its rate to $\frac{w_f}{W_f}$ and runs `UpdateLinks()` with $\Delta = \frac{w_f}{W_f} - r(f)$. If one of the links in L_f is oversubscribed, then the returned $W_o > 0$. In this case, the flow reduces its rate to $\min\left\{\frac{w_f}{W_o}, r(f)\right\}$. Finally, if neither of the previous two conditions hold, the possibility to increase rate greedily is checked ($c_u > 0$).

If c_u is found to be positive, f attempts to reserve *all* of the capacity before claiming it by calling `MakeReservation()` with c_u as its argument. This subroutine increases c_l^r of links in L_f. The reservation is deemed successful only if at all links

Algorithm 4.4 SAA-M subroutines

`real` **AnnounceStart**(w_f)

1. `send` p : $p.w_f \leftarrow w_f$, $p.W_f \leftarrow 0$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l + p.w_f$
   2.2 $p.W_f \leftarrow \max\{p.W_f, W_l\}$ /*Collect $\frac{1}{W_f}$*/
3. `return` $p.W_f$ `when` p `arrives back to the source`

`void` **AnnounceEnd**(w_f, Δ)

1. `send` p : $p.w_f \leftarrow w_f$, $p.\Delta \leftarrow \Delta$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l - p.w_f$
   2.2 $c_l \leftarrow c_l + p.\Delta$

`[real,real,real]` **ProbeLinks**()

1. `send` p : $p.W_f \leftarrow 0$, $p.W_0 \leftarrow 0$, $p.c \leftarrow 1$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $p.W_f \leftarrow \max\{p.W_f, W_l\}$ /*Collect $\frac{1}{W_f}$*/
   2.2 $p.c \leftarrow \min\{p.c, c_l - c_l^r\}$ /*Collect claimable capacity*/
   2.3 `if` $c_l < 0$ `then`
      2.3.1 $p.W_o \leftarrow \max\{p.W_o, W_l\}$ /*Oversubscribed link*/
3. `return` $[p.W_f, p.W_o, p.c]$ `when` p `arrives back to the source`

`void` **UpdateLinks**(Δ, c^r)

1. `send` p : $p.\Delta \leftarrow \Delta$, $p.c^r \leftarrow c^r$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $c_l \leftarrow c_l - p.\Delta$
   2.2 $c_l^r \leftarrow c_l^r - p.c^r$

`bool` **MakeReservation**(c^r)

1. `send` p : $p.c^r \leftarrow c^r$, $p.s \leftarrow true$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $c_l^r \leftarrow c_l^r + p.c^r$ /*Update reservation*/
   2.2 `if` $c_l - c_l^r < 0$ `then`
      2.2.1 $p.s \leftarrow false$ /*Reservation fails*/
3. `return` $p.s$ `when` p `arrives back to the source`

`void` **CancelReservation**(c^r)

1. `send` p : $p.c^r \leftarrow c^r$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $c_l^r \leftarrow c_l^r - p.c^r$ /*Update reservation*/

$c_l - c_l^r \geq 0$ after the update. The latter indicates that no oversubscription will arise when the reserved capacity is actually claimed. If the reservation succeeds, f increases its rate by c_u and runs `UpdateLinks()` to notify the relevant links about

the change. Here, the second argument is used to cause the links to reduce c_l^r by c_u, as they reduce c_l by the same amount.

If the reservation fails, the reserved capacity is released using `CancelReservation()`, which reduces back the c_l^r values. In order to avoid livelock in some pathological scenarios (to be discussed below), the flow is required to back off for a random amount of time.

It is possible that the result of `ProbeLinks()` triggers no special activity. In fact, if the set of flows F stops changing and the algorithm converges (yet to be shown), it is expected to happen most of the time. Finally, whenever a flow ends it executes `AnnounceEnd()`, reducing the weights of the links in L_f by w_f and increasing the free capacity by $r(f)$.

Discussion of the Reservation Policy

The proposed reservation policy can be summarized as follows. Upon discovering free, claimable capacity c_u, a flow attempts to reserve all of it. If this attempt fails the reservation is completely canceled, and the flow backs off for a random amount of time. Now, we turn to discuss the motivation for each component of this policy.

To begin with, reserving c_u (maximum available capacity) is chosen for its simplicity. Other more sophisticated alternatives may also be considered. For example, a flow can reserve only a fraction of c_u if it is higher than some predefined, small value. This approach is expected to provide a more "fair" result when several flows contend for the same excess capacity. In the case of a single contender, the capacity would be claimed in several iterations of the reserve-claim procedure.

We forbid control packets to carry per-hop information. This choice is made to keep the algorithm simple and because in some cases a flow's path be relatively long. As a result, each packet carries out the same operation on all links on its path. For this reason, if the desired capacity is not available for reservation on one of the links, the reservation fails even if that link has a smaller amount of free capacity. Therefore, we apply the "all or nothing" reservation policy.

Finally, let us demonstrate the need for randomized back-off when a reservation fails. Consider the scenario depicted in Figure 4.1. Two flows f_1, f_2 originate at

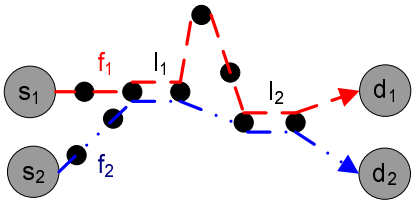


Figure 4.1: The possibility of livelock

sources s_1, s_2 and are destined to d_1, d_2, respectively. The flows share two links on their paths, l_1 and l_2. At some point in time, both flows attempt to reserve available free capacity on their paths. Assume that, due to the network topology, f_1 is always the first to make reservation on l_1, while f_2 is the first to reserve capacity on l_2. As a result, both flows fail to complete the reservation and engage in a never-ending cycle of reservation and cancellation. The randomized back-off gives a positive probability for one of the flows to succeed in the reservation and claim the capacity.

Correctness

In order to show correctness of the proposed algorithm, we assume that at some point in time the set of flows F stops changing. It is not hard to see that as a result, link weights eventually become steady as well. Ideally, we would like to prove that our algorithm is guaranteed to converge to a state in which flow rates are constant and comply with feasibility, SAA, and maximality requirements. In practice, due to the use of randomized back-off we prove a slightly weaker Theorem 4.1.5, which guarantees that eventually, rates satisfy feasibility, SAA requirement, and are monotonically non-decreasing. The success of reaching maximality depends on a correct choice of the random back-off times.

Next, we assume that a steady state rate assignment (which satisfies all requirements, including maximality) is reached and a single flow starts/ends. We prove that as a result only a local recalculation takes place.

Definitions:

- t_0 – a time after which $\forall l \in L : W_l$ is fixed.
- $W_f(\infty)$ – steady state value of W_f, $W_f(\infty) = \max_{l \in L_f} \{W_l(t_0)\}$.

- $\xi_l^s(t)$ – a "shadow" variable, used for proof purposes only, that holds the total capacity reserved on link l at time t by packets that succeed in reservation. This value grows when capacity is reserved by a packet that succeeds to make the reservation on all links on its path (this knowledge actually violates causality). It is reduced when the reserved capacity is claimed.

- $\xi_l^f(t)$ – a "shadow" variable, used for proof purposes only, that holds the total capacity reserved on link l at time t by packets that eventually fail in reservation ($c_l^r(t) = \xi_l^s(t) + \xi_l^f(t)$).

- $r(f,t)$ – the local rate variable of f.

- $r_l(f,t)$ – a "shadow" variable used for proof purposes only, it holds the rate of f as known by l. The value is assumed to be updated with c_l in `UpdateLinks()`. The following equality holds $c_l(t) = 1 - \sum_{f \in F_l}(r_l(f,t))$.

Lemma 4.1.3. *There exists a time $t_1 > t_0$ such that for every $t > t_1$, $\forall f \in F$: $r(f,t) \geq \frac{w_f}{W_f(\infty)}$, and `UpdateLinks()` is called only from lines 3.3 and 4.2.2 after the periodic probing.*

Proof. Consider a flow f at time $t > t_0$ just after an invocation of `ProbeLinks()`. If $r(f,t) < \frac{w_f}{W_f(\infty)}$, then it is immediately set to $r(f,t) = \frac{w_f}{W_f(\infty)}$. After t, $r(f)$ is reduced only if a later `ProbeLinks()` returns $W_o > 0$. However, after t_0 $\forall l \in L : W_l$ is stable, so a collected $W_o \leq \max_{l \in L_f}\{W_l(t_0)\} = W_f(\infty)$. Therefore, $r(f)$ is never reduced below $\frac{w_f}{W_f(\infty)}$. As a result, after later invocations of `ProbeLinks()` the condition of line 2 is never true. □

Lemma 4.1.4. *There exists a time $t_2 > t_1$ such that $\forall t > t_2, \forall l \in L : c_l(t) \geq 0$*

Proof. Consider a link l at time $t > t_1$. We divide the proof into two cases:

1. $c_l(t) - \xi_l^s(t) \geq 0$. We need to show that $\forall t' > t$ the inequality holds and thus $c_l(t') \geq 0$. Following Lemma 4.1.3, $c_l(t)$ and $\xi_l^s(t)$ can change only as a result of lines 3.3, 4.1, 4.2.2 executed by a flow $f \in F_l$. Line 3.3 involves a negative Δ, which increases c_l. Line 4.2.2 decreases c_l and ξ_l^s by the same amount. Therefore, these two lines preserve the non-negative $c_l(t) - \xi_l^s(t)$. Line

4.1 invokes `MakeReservation()`, which can potentially increase ξ_l^s. However, such an increase cannot turn $c_l(t) - \xi_l^s(t)$ to be negative, because in this case the reservation would fail (causing an increase in $\xi_l^f(t)$ instead).

2. $c_l(t) - \xi_l^s(t) < 0$. Here, all we need to show is that at some $t' > t$ $c_l(t') - \xi_l^s(t') \geq 0$, because then we return to case 1. Assume by contradiction that the given situation persists. From t on, every $f \in F_l$ will get $W_o \geq W_l$ after invoking `ProbeLinks()`. In addition, eventually all successful reservations will be translated into decreases of c_l and no new reservations are made. Thus, at some $\tilde{t} > t$ we have $c_l(\tilde{t}) < 0$, $\xi_l^s(\tilde{t}) = 0$, this state is expected to stay true from $\tilde{t}$ on. However, as ordered by $W_o \geq W_l$, flows $f \in F_l$ reduce their rates to $r(f) \leq \frac{w_f}{W_l}$. The $r_l(f)$ values are updated in the same manner, which causes c_l to become non-negative, a contradiction.

□

Theorem 4.1.5. *The SAA-M algorithm eventually guarantees the following:*

1. $\forall f \in F : r(f) \geq \frac{w_f}{W_f(\infty)}$
2. $\forall f \in F : r(f)$ *is monotonically non-decreasing with time*
3. $\forall l \in L : \sum_{f \in F_l} (r(f,t)) \leq 1$ *(feasibility)*
4. $\forall f \in F : \min_{l \in L_f} \{c_l\} = 0$ *(maximality)*

Proof. Let's prove the guarantees one by one

1. Immediate from Lemma 4.1.3
2. Lemma 4.1.3 states that eventually the condition of line 2 is constantly false. Lemma 4.1.4 indicates the same about the condition of line 3. As a results eventually all flows will only attempt to increase their rates or do nothing.
3. Lemma 4.1.4 states that there exists time t_1, such that $\forall t > t_1 : 1 - c_l(t) = \sum_{f \in F_l} (r_l(f,t)) \leq 1$. Assume by contradiction that at some $t_2 > t_1$ $\sum_{f \in F_l} (r(f,t_2)) > 1$ (capacity violation). Since $r_l(f)$ is updated (with delay) to $r(f)$, and $r(f)$ is monotonically non-decreasing, eventually at $t_3 > t_1$ it would be true that $\sum_{f \in F_l} (r_l(f,t_3)) > 1$, a contradiction.

4. Consider a flow $f \in F$ at time t when the first three guarantees already hold (and are not violated later). Assume that $\min_{l \in L_f} \{c_l\} > 0$. Because flows continue the periodic probing indefinitely, and randomized backoff probabilistically resolves reservation conflicts, eventually this capacity will be claimed by f and/or flows that share links with it.

□

Locality

Now, we show the locality of change when a single flow starts/ends after a steady state rate assignment is reached. Consider a flow $f \in F$, denote $F_0(f) = \{f\}$. Next, we make the following recursive definition: $F_i(f)$ is a set of flows not belonging to any $F_j(f)$ for $j < i$ that share a link with a flow in $F_{i-1}(f)$. Propositions 4.1.6, 4.1.7 state the locality properties.

Proposition 4.1.6. *Let flow f start after a steady state rate assignment is reached. As a result, only rates of flows in $F_1(f)$ and $F_2(f)$ may change.*

Proof. The start of f increases the claimed capacity on links in L_f, some of which may become oversubscribed. As a result, some flows in $F_1(f)$ may be ordered to reduce rates. Thus, some flows in $F_2(f)$ may *greedily* claim the freed capacity. The rates of flows in $F_3(f)$ cannot change because for each $f' \in F_3(f)$ there was a saturated link before f started and flows in $F_2(f)$ only increase rates in a manner that doesn't force other flows to reduce theirs. Consequently, $\forall i \geq 3$ the rates of $F_i(f)$ stay unchanged. □

Proposition 4.1.7. *Let flow f end after a steady state rate assignment is reached. As a result, only rates of flows in $F_1(f)$, $F_2(f)$ and $F_3(f)$ may change.*

Proof. The end of f frees capacity and decreases weights on links in L_f. As a result, some flows in $F_1(f)$ increase their rates due to the change in their W_f. This potentially causes rate reduction of some flows in $F_2(f)$ (greedy capacity claiming by flows in $F_1(f)$ on its own does not). Like in the proof of Proposition 4.1.6 above, flows in $F_3(f)$ may increase their weights, but $\forall i \geq 4$ the rates of $F_i(f)$ stay unchanged. □

Calculation Time

Here, like in SAA, after all flows announce start (assuming that a period of changes is followed by a steady period), it takes each flow f a single probing to set its rate to $\frac{w_f}{W_f}$. It is natural to increase the frequency of sampling during the active stage of calculation and to reduce it when a flow assumes that a steady state is reached. During the latter, one can piggy-back the sampling on data packets. Here, it suffices to let data packets somehow detect events of oversubscription, existence of claimable capacity, and changes in W_f. Later, control packets can be used to collect the exact values and to perform the calculation.

After getting the guaranteed rate, a flow may attempt to reserve and claim excess capacity. The time required for that, until no further excess capacity is available, highly depends on the reservation policy and particularly on the choice of the backoff distribution. In principle, we can decide to make flow give up claiming additional capacity after several failed reservations attempts. The tuning of the reservation policy is an interesting topic for further research. We note again that SAA-M bounds the set of affected flows when changes in F occur.

4.2 Independent Flows

In the independent flows scenario, there is no meaningful relation between different flows and they can potentially start and end in an unpredictable manner. As a result, our goal is to find an efficient and fair assignment of instantaneous rates for a given set of flows. As before, we assume the routing to be fixed for the duration of the calculation. In addition, we seek a solution in form of a distributed algorithm, which makes uses control packets of a fixed size, and stores no per-flow state at the network elements.

4.2.1 Weighted Max-Min Fairness

Fairness can be defined in various ways; we use max-min fairness for our purposes.

Definition 4.2.1. Let r be a feasible rate assignment. We say that r is max-min fair

if for every other feasible assignment r' it holds that, $r'(f) > r(f)$ implies existence of f_{victim}, for which $r(f_{victim}) \leq r(f)$ and $r'(f_{victim}) < r(f_{victim})$.

In other words, a rate assignment r is max-min fair if for each flow f, $r(f)$ cannot be increased without decreasing $r(f_{victim})$ for some flow f_{victim} for which $r(f_{victim}) \leq r(f)$. Alternatively, we can say that max-min fair assignment recursively maximizes the minimum rates; i.e., it maximizes the minimum rate, then the second smallest rate, then the third one, etc (hence the name max-min).

Note that the max-min fairness incorporates the efficiency requirement. An inefficient rate assignment, one that allows the rate of some flow to be increased without affecting others, clearly violates the definition. In addition, max-min fair assignment can be shown to be unique for given network and flow set.

As before, flows may have associated weights w_f. Here, however, the weight can be chosen arbitrarily to represent flow priority rather than size. If we replace the rate $r(f)$ with a normalized rate $\bar{r}(f) = \frac{r(f)}{w_f}$ in the above definition, we end up with the *weighted* max-min fairness. We will use the latter as a general optimality criterion in the following discussion.

4.2.2 Centralized Algorithm

The centralized algorithm for calculation of the max-min fair rate assignment under a *fixed* set of flows is well known, and can be found in [4]. We present its weighted version in Algorithm 4.5, and refer it as a *flow-fair algorithm* (FFA). The basic operation of FFA can be described as follows. Start with zero normalized rates. Increase the normalized rates of all flows continuously at the same speed, until one of the links in the network becomes saturated. Remove saturated links and flows crossing them from the network. Continue in the residual network with remaining link capacities.

More precisely, Algorithm 4.5 works in iterations. At the beginning of every iteration, the set of unsaturated links is L and the set of active, non-bottlenecked, flows is F. The normalized rate of all active flows is increased by $\bar{\Delta} = \min_{l \in L} \left\{ \frac{c_l}{W_l} \right\}$, where c_l is the residual capacity of a link l and W_l is aggregate weight of active flows crossing it. It can be shown that the increase by $\bar{\Delta}$ *feasibly* reduces c_l, and is

Algorithm 4.5 FFA

Initialization:
$\forall l \in L : c_l \leftarrow 1$
$\forall f \in F : \bar{r}(f) \leftarrow 0$

Loop:
1. $\bar{\Delta} \leftarrow \min_{l \in L} \left\{ \frac{c_l}{W_l} \right\}$ /*Find the allowed increment*/
2. $\bar{r}(f) \leftarrow \begin{cases} \bar{r}(f) + \bar{\Delta} & f \in F \\ \bar{r}(f) & f \notin F \end{cases}$ /*Update active flows*/
3. $L \leftarrow \{l | c_l - \sum_{F_l} \bar{r}(f) \cdot w_f > 0\}$ /*Remove saturated links*/
4. $F \leftarrow \{f | L_f \subseteq L\}$ /*Remove bottlenecked flows*/
5. if $|F| = 0$ terminate, else go to 1

guaranteed to saturate some of the links. As a result, some of the flows are removed from F, which (implicitly) changes the weights of the links. After a flow is removed from the network, its actual rate is given by $r(f) = \bar{r}(f) \cdot w_f$.

The correctness of Algorithm 4.5 is based on a trivial generalization of the correctness proof for the basic algorithm in [4]. Intuitively, the algorithm allows every flow to claim as high a normalized rate as it can, in a fair manner. Once the flow becomes bottlenecked, the rest of the capacity of links in L_f is further fairly distributed. As a result, when the algorithm ends, increasing rate of any flow f necessarily violates fairness at some saturated link on which f has the highest normalized rate.

4.2.3 Distributed Algorithm

Previous Work

A popular distributed implementation of non-weighted FFA was proposed in [7]. The algorithm is designed to operate in an asynchronous environment and guarantees convergence to the max-min fair rate assignment, given that the network reaches a steady state (constant flow set and routing). Unfortunately, however, this algorithm requires every link to keep state information for every flow that traverses it. Also, it requries $O(|F_l|)$ computations at each link in L_f for every crossing control packet.

The variation proposed in [20] reduces the computational complexity to $O(1)$ but leaves memory requirements unchanged. Keeping per-flow state may reduce convergence time when dynamically changing network state is considered, but it comes at a price. Unlike in Chapter 3, where a limited size of routing cache only affected performance, in [7, 20] the lack of flow state buffers violates the correctness.

The desire to reduce memory complexity lead to a synchronous algorithm [8], however, it has three major drawbacks: 1) it relies on setting correct network-dependent time parameters, 2) periodic transmission of control packets is required for each flow, and moreover cannot be piggy-backed on data packets, and 3) a finite and limited set of rate values is used.

Below, we propose an alternative algorithm, named *distributed FFA* (DFFA). DFFA stores no per-flow state, is designed to operate in asynchronous environment, uses continuous rate levels, and quiesces. Moreover, similarly to SAA and SAA-M in Section 4.1, control packets used for calculation carry no per-hop information. Finally, our algorithm is capable of dealing with weighted flows and achieves weighted max-min fairness.

DFFA*

As a first step, we formulate a distributed algorithm that is designed to perform a single calculation for a given set of flows, *after* initialization of state of links and flows. We call this algorithm DFFA*; it will be used as a building block in a broader scheme later, so for now we do not care how the initialization is performed. Intuitively, a distributed implementation of FFA should allow flows to claim "excess" capacity (above normalized rate of $\frac{1}{W_f}$) only after it is explicitly given up by other bottlenecked flows, when they become inactive. For that purpose, each link holds a *fair share* value s_l. Flows in F_l are allowed to set their normalized rate as high as s_l, which grows only when one of the flows turns inactive without fully exploiting its share.

In DFFA*, each link is required to hold the following variables, that constitute an $O(1)$ size state:

- W_l – link weight, the aggregate weight of crossing flows: $\sum_{F_l}(w_f)$. Decreases

Algorithm 4.6 DFFA*

Initialization:
$\forall l \in L: c_l \leftarrow 1, W_l \leftarrow \sum_{f \in F_l}(w_f), s_l \leftarrow \frac{c_l}{W_l}$
$\forall f \in F: \bar{r}(f) \leftarrow 0$

In parallel by each flow for itself:
1. $\bar{\Delta} \leftarrow$`CollectMinShare()`$-\bar{r}(f)$ /*Collect the allowed increment*/
2. $\bar{r}(f) \leftarrow \bar{r}(f) + \bar{\Delta}$
3. *active* $\leftarrow$ `UpdateLinks(`$\bar{\Delta} \cdot w_f$`)` /*Distribute the increment*/
4. `if (`*active* $=$ *true*`) goto 1` /*Check if flow is bottlenecked*/
5. `else`
   5.1 `DeclareInactive(`$\bar{r}(f), w_f$`)` /*Notify links*/
   5.2 `terminate`

Algorithm 4.7 DFFA* subroutines

`real` **CollectMinShare()**
{
1. `send` p `with` $p.share = \infty$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $p.share \leftarrow \min(p.share, s_l)$
3. `return` $p.share$ `when` p `arrives back to the source`
}

`boolean` **UpdateLinks(**Δ**)**
{
1. `send` p `with` $p.\Delta = \Delta$, $p.active = true$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $c_l \leftarrow c_l - p.\Delta$ /*Reduce free capacity*/
   2.2 `if` $c_l = 0$ /*Check if saturated*/
      2.2.1 $p.active \leftarrow false$
3. `return` $p.active$ `when` p `arrives back to the source`
}

`void` **DeclareInactive(**$\bar{r}(f), w_f$**)**
{
1. `send` p `with` $p.\bar{r}(f) = \bar{r}(f)$, $p.w_f = w_f$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l - p.w_f$ /*Reduce link weight*/
   2.2 `if` $W_l > 0$
      2.2.1 $s_l \leftarrow s_l + \frac{(s_l - p.\bar{r}(f)) \cdot p.w_f}{W_l}$ /*Increase share*/
}

as flows become inactive.

- c_l – free capacity, non-negative
- s_l – the fair share

The algorithm itself is presented in Algorithm 4.6. It starts with an initialization *after* which the main loop is executed independently and concurrently by each flow for itself. The main loop uses three subroutines that are summarized in Algorithm 4.7. For every flow, each iteration consists of the following actions.

First, a minimum s_l among the links in L_f is collected using `CollectMinShare()`. This value is used to update the current normalized rate $\bar{r}(f)$. As a result, $\bar{r}(f)$ can grow or stay unchanged (yet to be shown). The change in absolute (not normalized) rate is distributed using `UpdateLinks()`, which reduces c_l of links in L_f accordingly. In addition, if one of the links becomes saturated ($c_l = 0$), the flow is notified by *active* being set to $false$ (after a link becomes saturated it might take another iteration for a flow to discover this fact). In this case, f announces becoming inactive by executing `DeclareInactive()`. This subroutine reduces the weights of links in L_f by w_f and updates the fair shares for links where $\bar{r}(f) < s_l$.

The fair share is, in fact, the rate per unit of weight that the active flows can claim, and saturate the link. Therefore, when one of the flows becomes inactive it leaves $(s_l - \bar{r}(f)) \cdot w_f$ of unclaimed capacity. This capacity is divided by the remaining weight W_l to acquire the increase in s_l.

DFFA* Correctness

We prove that DFFA* terminates for all flows and leads to a weighted max-min fair rate assignment. Our only assumption on the order of events is that the processing of control packets is atomic for individual links (lines 2.1-2.2 in the subroutines). We use the following definitions:

- $\bar{r}_l(f,t)$ – a "shadow" variable used for proof purposes only. It holds the normalized rate of f as known by l. The value is assumed to be updated with c_l in `UpdateLinks()` by $\frac{p.\Delta}{p.w_f}$ (for that purpose, assume that p carries w_f as well). The following equality holds $c_l(t) = 1 - \sum_{f \in F_l} (\bar{r}_l(f,t) \cdot w_f)$.
- F_l – all flows crossing l be they active, inactive, or terminated.

- $\widehat{F}_l(t)$ – subset of F_l consisting of flows that didn't execute `DeclareInactive()` on l yet, i.e. flows known as active to l.

- $x(t^-)$, $x(t^+)$ – values of variable x just before, and immediately after, a change at time t.

Lemma 4.2.2. *If a flow f enters `DeclareInactive()` at time t_1, then $\forall t_2 > t_1$, $\forall l \in L_f : \bar{r}(f,t_2) = \bar{r}_l(f,t_2) = \bar{r}(f,t_1)$*

Proof. Every increase in $\bar{r}(f,t)$ is followed by an execution of `UpdateLinks()`, which distributes the new Δ to links in L_f. Therefore, at a moment when `UpdateLinks()` returns, $\forall l \in L_f : \bar{r}(f,t) = \bar{r}_l(f,t)$. At t_1 the call to `DeclareInactive()` immediately follows the return from `UpdateLinks()`, thus $\forall l \in L_f : \bar{r}(f,t_1) = \bar{r}_l(f,t_1)$. After the call, the different rate variables stay unchanged. □

Lemma 4.2.3. *During a run of DFFA*, for every link l the following conditions hold: 1) $s_l(t)$ is monotonically non-decreasing, 2) $\forall l \in L_f$ at any time t, $\bar{r}(f,t) \leq s_l(t)$ (where $\bar{r}(f,t)$ is the local value stored at the source).*

Proof. Both conditions hold at $t = 0$. Assume by contradiction that t_1 is the smallest time when one of the conditions is violated.

1. $\exists l \in L : s_l(t_1^+) < s_l(t_1^-)$. The decrease can occur only as a result of some flow f executing line 2.2 of `DeclareInactive()` on l with $p.\bar{r}(f) > s_l(t_1^-)$ ($W_l(t_1^-) > 0$). However, since $s_l(t)$ is known to have been monotonically non-decreasing until t_1, we must conclude that at the time $t_0 < t_1$, when p left the source, we had $s_l(t_0) < \bar{r}(f,t_0)$. This is a contradiction to minimality of t_1 because claim 2 is violated at t_0.

2. $\exists f \in F, l \in L_f : \bar{r}(f,t_1^+) > s_l(t_1^+)$. The increase in $\bar{r}(f,t)$ follows the execution of `CollectMinShare()`. The control packet p, that collected the fair share, visited l at $t_0 < t_1$. As a result, the return value of `CollectMinShare()` is bounded from above by $s_l(t_0)$. Since $s_l(t)$ does not decrease until t_1, we have $\bar{r}(f,t_1) \leq s_l(t_0) \leq s_l(t_1^+)$, a contradiction to $\bar{r}(f,t_1^+) > s_l(t_1^+)$.

□

Corollary 4.2.4. *The following conditions hold: 1) $\bar{r}(f,t)$ is monotonically non-decreasing, 2) $c_l(t)$ is monotonically non-increasing, 3) $\forall l \in L_f$ at any time t, $\bar{r}_l(f,t) \leq \bar{r}(f,t) \leq s_l(t)$.*

Proof. Shortly explaining every claim:

1. $\bar{r}(f,t)$ is periodically updated to the minimum s_l on links in L_f. This minimum is monotonically non-decreasing.

2. $c_l(t)$ is updated when for some flow in F_l, $\bar{r}(f)$ changes. Claim 1 states that all rates are monotonically non-decreasing, so $c_l(t)$ cannot grow.

3. $\bar{r}_l(f,t)$ is updated to $\bar{r}(f,t)$ by `UpdateLinks()`. Since $\bar{r}(f,t)$ is non-decreasing $\bar{r}_l(f,t) \leq \bar{r}(f,t)$.

□

Lemma 4.2.5. *At any given moment t, if $|\widehat{F}_l(t)| > 0$, then the following equation holds:*

$$\sum_{f \in \widehat{F}_l(t)} w_f \cdot (s_l(t) - \bar{r}_l(f,t)) = c_l(t). \tag{4.4}$$

Proof. Let l be a link with $|F_l^k(0)| > 0$. At $t = 0$, the right side of the equation is $c_l(0) = 1$, and the left side is $W_l(0) \cdot \frac{1}{W_l(0)} = 1$. From then on, the equation can become invalid only when `UpdateLinks()` or `DeclareInactive()` subroutines are executed for some flow f' on l at time t_1.

1. When `UpdateLinks()`is executed, both sides of the equation are reduced by the same amount $p.\Delta$ ($\bar{r}_l(f', t_1)$ increases by $\frac{p.\Delta}{p.w_{f'}}$).

2. When `DeclareInactive()`is executed, yet there are other active flows that still traverse l, the right side of the equation stays unchanged. Let p be the control packet sent by the source. Following Lemma 4.2.2, $p.\bar{r}(f') = \bar{r}_l(f', t_1)$. The left side of the equation is decreased by $w_{f'} \cdot (s_l(t_1^-) - \bar{r}_l(f', t_1))$ since the element of f' is removed from the sum. On the other hand, however, this side is increased, due to the increase in $s_l(t_1^+)$, by:

$$\sum_{f \in \widehat{F}_l(t_1^-) \setminus \{f'\}} \left(w_f \cdot \frac{(s_l(t_1^-) - p.\bar{r}(f')) \cdot p.w_{f'}}{W_l(t_1^-) - p.w_{f'}} \right) = w_{f'} \cdot (s_l(t_1^-) - \bar{r}_l(f', t_1)). \quad (4.5)$$

Hence, the left hand side of the equation stays unchanged.

□

Lemma 4.2.6. *At any given moment t, $\forall l \in L : c_l(t) = 1 - \sum_{f \in F_l} (\bar{r}_l(f,t) \cdot w_f) \geq 0$*

Proof. If $F_l = \phi$, the condition trivially holds for any moment. Otherwise, we have the following two cases at time t_1:

1. If $|\widehat{F}_l(t_1)| > 0$, then the condition of Lemma 4.2.5 holds. Applying Corollary 4.2.4, we conclude that $c_l(t_1) \geq 0$.

2. If $|\widehat{F}_l(t_1)| = 0$, consider a moment t_0 when the last flow in F_l updates l about becoming inactive (executing `DeclareInactive()` on l). At t_0^-, according to case 1, $c_l(t_0^-) \geq 0$. After t_0 the capacity stays unchanged.

□

Lemma 4.2.7. *Eventually all flows exit the main loop, and DFFA* terminates.*

Proof. Assume by contradiction that at time t_1 there are active flows in the network, and none of them executes `DeclareInactive()` at or after t_1. Since $s_l(t)$ values change only when flows become inactive, we conclude that $\forall t > t_1$, $\forall l \in L : s_l(t) = s_l(t_1)$. Let l' be a link with the smallest $s_l(t_1)$ among links for which $|\widehat{F}_l(t_1)| > 0$. Let $t_2 > t_1$ be the time by which every $f \in \widehat{F}_{l'}(t_1)$ starts and completes at least one iteration of the main loop after t_1. As a result, both $\bar{r}(f, t_2)$ and $\bar{r}_{l'}(f, t_2)$ are equal to $s_{l'}(t_1)$. According to Lemma 4.2.5, this implies that $c_l(t_2) = 0$, which must eventually cause flows in $\widehat{F}_{l'}(t_1)$ to become inactive, a contradiction. □

Theorem 4.2.8. *The rate assignment calculated by DFFA* is max-min fair in regards to $\bar{r}(f)$.*

Proof. Following Lemma 4.2.7, all flows eventually become inactive and terminate. A flow $f' \in F$ terminates only after it finds, at time t_1, that one of the links l in $L_{f'}$ has $c_l(t_1) = 0$. Since $c_l(t)$ is known to be monotonically non-increasing and, according to Lemma 4.2.6, non-negative, $\forall t \geq t_1 : c_l(t_1) = 0$.

Consider a moment $t_2 > t_1$ when f' notifies l about becoming inactive. According to Lemma 4.2.5, since $c_l(t_2) = 0$, $\bar{r}_l(f', t_2) = s_l(t_2^-)$. According to Lemma 4.2.2, we also have $\bar{r}(f', t_2) = s_l(t_2^-)$, which is the final rate of f'. In fact, $\forall f \in \widehat{F}_l(t_2) : \bar{r}(f, t_2) = s_l(t_2^-)$. Therefore, s_l can't be increased anymore, i.e., $\forall t \geq t_2 : s_l(t) = s_l(t_2^-)$.

Following Lemma 4.2.3, by the end of the run $\forall f \in F_l : \bar{r}(f, \infty) \leq s_l(t_2) = \bar{r}(f', \infty)$. As a result, any attempt to increase $\bar{r}(f')$, while preserving feasibility, will reduce a normalized rate of a weaker flow. This means that r is max-min fair in regards to $\bar{r}(f)$, or (equivalently) weighted max-min fair in regards to $r(f)$. □

Although the correctness is summarized in Theorem 4.2.8, there are several more properties we choose to present.

Eliminating Centralized Initialization and Startup

Since, aside from the initialization, the algorithm is capable of operating in an asynchronous environment, different flows can begin the execution of the DFFA* main loop at different times after the initialization. Moreover, Lemma 4.2.9 states that there is no necessity in a centralized initialization of the state of links.

Lemma 4.2.9. *If $\forall f \in F$, $\forall l \in L_f$ the parameters W_l, c_l, s_l are initialized correctly before f starts the execution of DFFA* loop, then the algorithm preserves correctness.*

Proof. Consider a run in which initialization happens not in a centralized manner. Since no flow is allowed to see an uninitialized link's state, and an asynchronous environment is considered, we can arbitrarily move the initialization events back in time to happen at the same moment, without violating the behavioral correctness of the run. As a result, we achieve a correct run with a centralized initialization, whose output is identical to the initial run. □

Reducing Calculation Time

It can readily be seen that the main loop can be shortened by allowing `UpdateLinks()` to also to collect the minimum fair share, in addition to its other activities. This

Algorithm 4.8 Optimized DFFA*

Initialization:
$\forall l \in L : c_l \leftarrow 1, W_l \leftarrow \sum_{f \in F_l} \{w_f\}, s_l \leftarrow \frac{c_l}{W_l}$
$\forall f \in F : \bar{r}(f) \leftarrow 0$

In parallel by each flow for itself:
1. $\bar{\Delta} \leftarrow 0$
2. $[active, s] \leftarrow$ `UpdateLinks(`$\bar{\Delta} \cdot w_f$`)`
3. $\bar{\Delta} \leftarrow s - \bar{r}(f)$
4. $\bar{r}(f) \leftarrow \bar{r}(f) + \bar{\Delta}$
5. `if (`$active = true$`) goto 2` */* Check if flow is bottlenecked */*
6. `else`
   6.1 `DeclareInactive(`$\bar{r}(f), w_f$`)` */* Notify links */*
   6.2 `terminate`

Algorithm 4.9 Optimized DFFA* subroutines

`[boolean,real]` **UpdateLinks**(Δ)
{

1. `send` p `with` $p.\Delta = \Delta$, $p.active = true$, $p.share = \infty$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $c_l \leftarrow c_l - p.\Delta$ */* Reduce free capacity */*
   2.2 `if` $c_l = 0$ */* Check if saturated */*
      2.2.1 $p.active \leftarrow false$
   2.3 $p.share \leftarrow \min(p.share, s_l)$
3. `return` $p.active$, $p.share$ `when` p `arrives back to the source`

}
`void` **DeclareInactive**($\bar{r}(f), w_f$)
{

1. `send` p `with` $p.\bar{r}(f) = \bar{r}(f)$, $p.w_f = w_f$
2. `foreach` $l \in L_f$ `upon receipt of` p
   2.1 $W_l \leftarrow W_l - p.w_f$ */* Reduce link weight */*
   2.2 `if` $W_l > 0$
      2.2.1 $s_l \leftarrow s_l + \frac{(s_l - p.\bar{r}(f)) \cdot p.w_f}{W_l}$ */* Increase share */*

}

effectively removes the call to `CollectMinShare()`, without compromising the correctness of the algorithm. The optimized algorithm is found in Algorithm 4.8; its subroutines are presented in Algorithm 4.9.

Calculation Time Analysis

We prove an upper bound on the convergence time of the optimized algorithm, using the following definitions:

- T_{rt} – an upper bound on the total round-trip delay (including queuing and computation at links) of control packets. The computation time at the sources is assumed to be negligible.
- R – the set of *distinct* rate values as computed by the centralized FFA (and consequently by the DFFA*). $|R|$ equals the number of iterations of FFA. Let L' be the set of links with $|F_l| > 0$. Since in each iteration at least one link and one flow are removed from the network, $|R| \leq \min\{|F|, |L'|\}$. In practice, R can have a small size (we witnessed $|R| = 40$ in a network with thousands of flows and links). However, in the worst case $|R| = \min\{|F|, |L'|\}$.
- $\widetilde{F}_l(t)$ – subset of F_l consisting of flows that didn't terminate the main loop yet, including those that are known to l as inactive .
- $\widetilde{L}(t)$ – set of links for which $|\widetilde{F}_l(t)| > 0$.
- $l_m(t)$ – link with the minimum fair share, among links in $\widetilde{L}(t)$.
- $s_m(t)$ – the minimum fair share, among links in $\widetilde{L}(t)$, $s_m(t) = s_{l_m(t)}(t)$. If $\widetilde{L}(t) = \phi$, then $s_m(t) = \infty$.

Lemma 4.2.10. *If $s_m(t_1) < \infty$, then at $t_2 = t_1 + 6T_{rt}$, $s_m(t_2) > s_m(t_1)$*

Proof. $s_m(t)$ only changes as a result of either of the following two events: 1) a change in the fair share of some $l \in \widetilde{L}(t)$ (according to Lemma 4.2.3, it can only increase), and 2) the removal of a link from $\widetilde{L}(t)$ when all its flows terminate. In both cases, $s_m(t)$ increases, so $s_m(t_2) \geq s_m(t_1)$.

Assume by contradiction, that $s_m(t_2) = s_m(t_1)$. Therefore, there exists a link $l' = l_m(t_1) = l_m(t_2)$. By $t_1 + 2T_{rt}$, every active flow in $F_{l'}$ is guaranteed to collect $s_m(t_1)$ as the allowed rate. By $t_1 + 3T_{rt}$, l' is saturated, after being updated by the flows. No later than $t_1 + 5T_{rt}$, all flows crossing l' discover its saturation and execute

`DeclareInactive()`. As a result, by t_2, we have $|\widetilde{F}_{l'}(t_2)| = 0$, a contradiction to $l' = l_m(t_2)$. □

Lemma 4.2.11. *If $s_m(t)$ changes at t_1, then $s_m(t_1^-) \in R$*

Proof. Let $l' = l_m(t_1^-)$. We want to show that $l' \notin \widetilde{L}(t_1^+)$. Assume by contradiction that this is not true. Therefore, $s_{l'}(t_1^+) > s_{l'}(t_1^-)$ and $|\widetilde{F}_{l'}(t_1^+)| > 0$. The increase in the fair share of l' implies that one of the flows $f \in F_{l'}$ executes `DeclareInactive()` on l' at t_1, yet $\bar{r}(f, t_1) < s_{l'}(t_1^-)$. This, in turn, implies (proof of Theorem 4.2.8) that there exists another link $l'' \in \widetilde{L}(t_1^-)$, such that $\bar{r}(f, t_1) = s_{l''}(t_1^-) < s_{l'}(t_1^-)$, which is of course a contradiction to the definition of $l_m(t)$.

Therefore, $s_m(t_1^-)$ changes when the last flow $f \in F_{l'}$ terminates. From the proof of Theorem 4.2.8, we conclude that at the end of the run $\bar{r}(f, \infty) = s_m(t_1^-)$. □

Theorem 4.2.12. *After all flows start the main loop, the execution time of DFFA* is bounded by $6|R|T_{rt}$.*

Proof. Following Lemma 4.2.10, until $t = 6|R|T_{rt}$ the $s_m(t)$ changes at least $|R|$ times. Lemma 4.2.11 states that each change is associated with a distinct rate in R. Therefore, no more changes can possibly occur, i.e. $\forall l \in L : |\widetilde{F}_l(6|R|T_{rt})| = 0$. □

DFFA

DFFA* allows us to calculate the weighted max-min fair rate assignment for a fixed flow set F, if for every flow f the state of the links in L_f is initialized before f starts execution of the main loop. Now, our goal is to use DFFA* as a tool when $F(t)$ is allowed to change dynamically before reaching the steady state $F(\infty)$. We seek to acquire a rate assignment that is optimal for $F(\infty)$.

Ideally, we could wait until $F(t)$ stabilizes, then initialize link state and execute DFFA*. However, we don't know when the steady state is reached. Therefore we restart the calculation every time a change in $F(t)$ occurs, i.e. a flow starts or ends. In order to detect these changes, we assign every network element (switch or HCA) a *round* variable rnd_e. It serves for triggering flows to restart the DFFA* loop, and for distinguishing between packets of older and newer instances of the loop of the same flow.

Algorithm 4.10 DFFA

Initialization (at network setup):
$\forall e \in E : rnd_e \leftarrow 0$
$\forall l \in L : \widetilde{W}_l \leftarrow 0$

HCA h_s **upon start/end of flow** f**:**
1. $rnd_{h_s} \leftarrow rnd_{h_s} + 1$
2. `Restart DFFA*()`
3. `send an announcement packet` p_a `to destination` h_d
   3.1 $p_a.rnd \leftarrow rnd_{h_s}$
4. `for each` $sw \in SW_f$ `upon receipt of` p_a
   4.1 $rnd_{sw} \leftarrow \max\{rnd_{sw} + 1, p_a.rnd\}$
   4.2 `update` $\widetilde{W}_l$ `of the relevant link`
   4.3 $p_a.rnd \leftarrow rnd_{sw}$
   4.4 `ResetLinks()` */*Initialize according to current flows*/*
5. `HCA` h_d `upon receipt of` p_a
   5.1 $rnd_{h_d} \leftarrow \max\{rnd_{h_d} + 1, p.rnd\}$
   5.2 `send broadcast packet` p_b `to all adjacent switches`
      5.2.1 $p_b.rnd \leftarrow rnd_{h_d}$
   5.3 `RestartDFFA*()`

HCA h **upon receiving broadcast packet** p_b: */*All HCAs*/*
1. `if` $p_b.rnd > rnd_h$
   1.1 $rnd_h \leftarrow p_b.rnd$
   1.2 `RestartDFFA*()`

Switch sw **upon receiving broadcast packet** p_b **from port** I**:**
1. `if` p_b `is the first received packet with` $p_b.rnd = rnd_{sw}$
   1.1 `forward` p_b `on all ports except` I
2. `if` $p_b.rnd > rnd_{sw}$
   2.1 $rnd_{sw} \leftarrow p_b.rnd$
   2.2 `ResetLinks()` */*Initialize according to current flows*/*
   2.3 `forward` p_b `on all ports except` I

Because the calculation can be restarted several times, each link has to hold two weight variables. The first, W_l is the one used in DFFA* and it decreases throughout the execution. The second, $\widetilde{W}_l$ is the "true" link weight which changes only when flows crossing link l join or leave the network.

The details of round usage are given in Algorithm 4.10 and Algorithm 4.11. We use SW_f to denote the set of switches crossed by a flow f on its path. At network setup, all rounds are set to zero. When a flow f starts/ends, the source HCA h_s increments rnd_{h_s} and sends a control packet p, with $p.rnd = rnd_{h_s}$, to the

Algorithm 4.11 DFFA subroutines

```
void RestartDFFA*()
{
    1. foreach originating flow f
       1.1 Stop ongoing:  /*If any*/
           1.1.1 Waiting for ∀sw ∈ SW_f : rnd_sw = rnd_h
           1.1.2 Execution of DFFA* loop
       1.2 Wait/probe until ∀sw ∈ SW_f : rnd_sw = rnd_h
       1.3 r̄(f) ← 0
       1.4 Restart execution of DFFA* loop
}
void ResetLinks()
{
    1. foreach link l
       1.1 W_l ← W̃_l
       1.2 c_l ← 1
       1.3 s_l ← c_l / W_l
}
```

destination h_d. This packet updates $\widetilde{W}_l$ of links in L_f, and performs the following round related activities.

As p traverses the network, it increments the rounds of switches in SW_f and the destination HCA h_d. In addition, p collects the maximum resulting round along its path. After h_d is reached, the collected result is broadcast to all network elements. A broadcast packet p_b is forwarded by a switch sw on all its ports except the one through which it was received if $p_b.rnd > rnd_{sw}$, or $p_b.rnd = rnd_{sw}$ and p_b is the first broadcast packet received with this round value. The broadcast round value is adopted by an element e only if $p_b.rnd > rnd_e$.

When round variable is increased in a switch for some reason (after receiving an announcement or broadcast packet), the state of all its links is reset through re-initialization of W_l, c_l and s_l variables according to the current state of F_l. When rnd_h is increased in HCA h, the execution of DFFA* loop is restarted for each of its flows, in the following manner.

First, all activities related to the previous execution (if any) are stopped. These include the actual execution of DFFA* loop and waiting for $\forall sw \in SW_f : rnd_{sw} = rnd_h$. Then, the HCA waits until $\forall sw \in SW_f : rnd_{sw} = rnd_h$; it discovers this

by repeatedly probing the rounds with control packets. Finally, $\bar{r}(f)$ is reset and DFFA* loop restarted. Not surprisingly, this order of operation ensures that the state of all links in L_f is correctly initialized before the execution begins.

During the execution of DFFA* loop, all control packets sent by f are marked with rnd_h of its HCA. Switches, in turn, ignore DFFA* packet if its $rnd_h < rnd_{sw}$. This prevents earlier DFFA* loop instances from affecting link state after a round change. The correctness of DFFA is summarized in Theorem 4.2.14.

Lemma 4.2.13. *Following the cessation of changes in $F(t)$, all network elements eventually have the same round value.*

Proof. Every change in $F(t)$ can cause at most two increases of rnd_e at some $e \in E$: 1) when an announcement packet p_a crosses e, and 2) when a broadcast packet p_b triggered by p_a is received. Since the number of changes in $F(t)$ is finite, round values of all elements eventually reach stability.

Let rnd_{max} be the maximum round in the network in the steady state. This value initially appears when an announcement packet p_a crosses some network element. When p_a reaches its destination h_d, a broadcast packet p_b with $p_b.rnd = rnd_{max}$ is sent. It is not hard to see that a broadcast packet with rnd_{max} is guaranteed to reach every network element, and is and consequently adopted.

To show this, assume by contradiction that some element e never receives a broadcast packet with rnd_{max}. Consider the minimum hop path between h_d and e in the network. Let sw be the closest switch to h_d on this path that never forwards a broadcast packet with rnd_{max} (there must be such a switch). The sw is guaranteed to receive a broadcast packet with rnd_{max} at some time t. We know that at t, $rnd_{sw} \leq rnd_{max}$ and that sw have not forwarded a broadcast packet with rnd_{max} before t. Therefore, sw forwards the broadcast packet at t, a contradiction. □

Theorem 4.2.14. *The DFFA eventually terminates and calculates a max-min rate assignment in regards to $\bar{r}(f)$.*

Proof. For every $f \in F(\infty)$ originating at HCA h_s, according to Lemma 4.2.3, there exists the earliest t_1 such that $\forall sw \in SW_f : rnd_{sw}(t_1) = rnd_h(t_1) = rnd_{max}$. Consider a switch $sw \in SW_f$; let $t_0 \leq t_1$ be the earliest moment when $rnd_{sw}(t_0) =$

rnd_{max}. We know that at t_0, sw resets W_l, c_l and s_l of all its links, according to $F_l(\infty)$ (the arrival of new flows would trigger more round changes). In addition, the latest execution of DFFA* loop by f starts after $t_1 \geq t_0$ (when rnd_h becomes rnd_{max}, f waits until $\forall sw \in SW_f : rnd_{sw} = rnd_{max}$ and restarts DFFA* loop). Therefore, the conditions of Lemma 4.2.9 hold for the latest execution of DFFA* loop by every flow and $F(\infty)$. □

The Lack of Steady State

In practical networks, $F(t)$ never truly reaches a steady state. The strongest practical assumption we can make is that short periods of changes in $F(t)$ are interleaved with long periods when no such changes occur. In principle, DFFA is suitable for this case too, as it is expected to calculate the correct rate assignment for the long steady periods of time. However, as changes in $F(t)$ become more frequent, the overhead of recalculations can potentially grow to be intolerable. Moreover, if the frequency is too high, the algorithm can actually be in a constant state of calculation.

The main drawback of DFFA is that it potentially performs a from-scratch, global recalculation per every change in $F(t)$. An alternative is to invoke DFFA periodically, using time or number of changes as the trigger. In between the invocations, we seek to use some simpler mechanism for distribution of capacity. This mechanism has to provide some guarantee on the minimum rates of flows, including the newly starting ones. At the same time, it should operate quickly when distributing the excess capacity. A suitable candidate having similar properties is the SAA-M, presented in Subsection 4.1.3. The exact combination of DFFA and SAA-M is beyond the scope of this work and is left for the future work.

4.3 Multiple Phase-Based Applications

4.3.1 Application Rates

Goals

The third characteristic scenario, in which several phase-based applications run concurrently in the same cluster, can be considered as a combination of the previous two. Since different applications enter and leave the communication phase independently, we want to maximize the progress of each application fairly, as for the independent flows. At the same time, at the level of an individual application, we are interested to minimize the total completion time of the communication phase. Therefore, similarly to the single application scenario, we choose its slowest flow to define the progress of the application.

Recall that in the single application case, setting the same rate to all flows of an application was one of the alternatives, though increasing rates of some flows above the common minimum created no problem. Here, in contrast, exploiting such excess capacity may come at expense of other applications. Therefore, we stick to the same-rate solution because it preserves the progress of an application, and doesn't unnecessarily use excess bandwidth. We use the following definitions:

- $A(t)$ – the set of applications that are in the midst of their communication phase at time t
- a – application, set of flows, $a \in A(t)$
- β_a – application priority factor
- w_f – flow weight of $f \in a$, $w_f = \beta_a \cdot d_f$, where d_f is the *initial* size of f
- $\bar{r}(a,t)$ – the progress/rate of application a, $\bar{r}(a,t) = min_{f \in a} \{\bar{r}(f,t)\}$

Our goal is to find a rate assignment that is max-min fair with regards to $\bar{r}(a,t)$. As mentioned above, we set $\forall f \in a : \bar{r}(f,t) = \bar{r}(a,t)$. In this context, the weight of flow w_f has a dual purpose, as expressed by its two factors. The size of flows d_f is used to give higher absolute rates to longer flows of the same application,

so that all flows progress equally, relatively to their size. The application priority factor β_a prioritizes some applications over others. Among other purposes, it can be used to prevent applications with long flows from getting priority in weights (and consequently higher transmission rates).

Below, we propose centralized, parallel and distributed algorithms for finding the desired rate assignment under a *fixed* sets of applications and flows. Then, we further enhance the distributed algorithm to operate under changing conditions as applications enter and exit the communication phase. Here, unlike in the previous two scenarios, in the distributed algorithm we let network elements to store $O(|A|)$ amount of state (instead of $O(1)$). Yet, we assume that $|A| \ll |F|$ and that it can be fixed (limited) a-priori. Similarly, control packets used in the distributed calculation include $|A|$ bits of information.

Dynamic Environment Effects

Although setting $\forall f \in a : \bar{r}(f,t) = \bar{r}(a,t)$ reduces the interference between applications, it potentially causes the assignment to be non-maximal. Exploiting the residual capacity will not improve immediate progress of applications, but as other applications join and leave the network it can potentially shorten the completion time. An application benefits from the excess capacity only if all its flows succeed to increase their average (over the communication phase time) normalized rate. This observation implies that the potential advantage of maximizing link utilization may be small, as it is still dependent on the worst case among flows.

In general, the residual capacity left after ensuring max-min fairness of application rates can be further distributed using some kind of flow-level rate assignment algorithm. However, given the questionable contribution to the overall performance, we leave this matter outside the scope of this work.

4.3.2 Centralized Algorithm

A centralized, one-time *application-fair algorithm* (AFA) that finds a max-min fair assignment of application rates can be implemented as a slightly modified version of FFA. In fact, it is sufficient to change line 4 in Algorithm 4.5 to:

$$A' \leftarrow \{a \in A | \forall f' \in a : L_{f'} \subseteq L\}$$

$$F \leftarrow \bigcup_{a \in A'} \{f | f \in a\}.$$

AFA, similarly to FFA, increases the normalized rate of active flows at an equal speed until one of the links becomes saturated. This time, however, all flows of an application are removed from the network once one of them traverses a saturated link (even if others do not). After the flows and saturated links are removed from the network, the operation is restarted in the residual network. We next prove the correctness of AFA in Theorem 4.3.1.

Theorem 4.3.1. *AFA terminates. Its rate assignment is feasible and max-min fair with regards to* $\bar{r}(a)$.

Proof. AFA inherits two key properties from FFA. First, it doesn't violate link capacity at any iteration. Second, at least one link becomes saturated during every iteration. As a result, AFA terminates with a feasible rate assignment.

Next, consider an application $a \in A$, whose flows are removed from the network at the end of iteration i. This means that for some $f \in a$, a link $l \in L_f$ becomes saturated at the end of that iteration. If no $a' \neq a$ uses l, then an increase in $\bar{r}(a)$ requires increasing rates of all flows in F_l, which clearly violates feasibility.

Otherwise, we know that every flow $f' \in a'$ is removed from the network by the end of iteration i (possibly at earlier iterations). Therefore, in view of the monotonic increase in rates over time, $\bar{r}(f') \leq \bar{r}(f)$. The algorithm assigns the same normalized rate to all flows of an application, thus $\bar{r}(a') = \bar{r}(f') \leq \bar{r}(f) = \bar{r}(a)$. As a result, $\bar{r}(a)$ cannot be increased in a feasible manner without harming weaker applications. □

Algorithm 4.12 provides an alternative implementation of AFA, which we will use as a basis for a distributed algorithm later. This implementation does not deal with individual flows. Instead, the following variables are used:

- W_l – the aggregate weight of active flows crossing l

Algorithm 4.12 AFA

Initialization:
$\rho \leftarrow 0$
$\forall l \in L: c_l \leftarrow 1, W_l^a \leftarrow \sum_{f \in F_l \cap a}(w_f), W_l \leftarrow \sum_{a \in A}(W_l^a), A_l \leftarrow \{a | \exists f \in a : f \in F_l\}$

Loop:
1. $\bar{\Delta} \leftarrow \infty$
2. `foreach` $l \in L$ /*Calculate new increment*/
   2.1 $\delta_l \leftarrow \frac{c_l}{W_l}$ /*If $W_l = 0, \delta_l \leftarrow \infty$*/
   2.2 $\bar{\Delta} \leftarrow \min\{\bar{\Delta}, \delta_l\}$
3. `if` $(\bar{\Delta} = \infty)$ `terminate` /*Check termination condition*/
4. `else` $\rho \leftarrow \rho + \bar{\Delta}$
5. $B \leftarrow \phi$ /*Reset newly bottlenecked apps set*/
6. `foreach` $l \in L$
   6.1 $c_l \leftarrow c_l - \bar{\Delta} \cdot w_l$ /*Update residual capacity*/
   6.2 `if` $(c_l = 0)$ $B \leftarrow B \cup A_l$ /*Newly bottlenecked apps*/
7. `foreach` $l \in L, a \in B$ /*Update weights*/
   7.1 $W_l \leftarrow W_l - W_l^a$
   7.2 $A_l \leftarrow A_l \setminus \{a\}$
8. $\forall a \in B : \bar{r}(a) \leftarrow \rho$ /*Assign rate*/

- W_l^a – the aggregate weight of active flows belonging to a crossing l
- A_l – set of active applications crossing l
- δ_l – rate increment allowed by link l
- ρ – cumulative rate variable
- B – set of applications that are active at the beginning of an iteration, and become inactive by its end

The operation of Algorithm 4.12 is almost identical, and completely equivalent to the operation of the modified FFA. At the beginning of an iteration, a new allowed normalized rate increment $\bar{\Delta}$ for active applications is calculated. If $\bar{\Delta} = \infty$ there are no active applications left in the network, and the algorithm terminates. Otherwise, ρ is increased by $\bar{\Delta}$. Later, the residual capacity of links is computed, and active applications whose flows traverse newly saturated links are added to the set B (which is emptied at the beginning of the iteration). Applications in B are

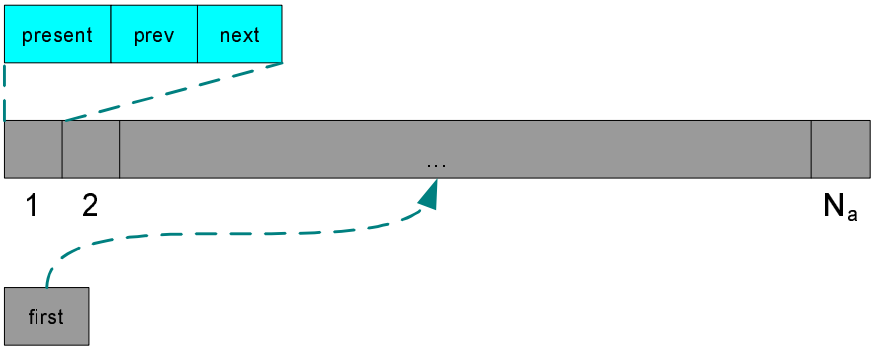


Figure 4.2: Doubly-linked list

removed from all links, as W_l and A_l are updated, accordingly. Finally, the newly removed applications are assigned the final rate of ρ.

For complexity analysis we assume that A_l and B are doubly-linked lists implemented in an array of size $|A|$. The data structure is presented in Figure 4.2. The position of an element corresponding to every application in both arrays is fixed by the index of an application. Every element has the following fields: 1) *present* field states whether the application is in the list, 2) *prev* (array index) points to a predecessor in the list, 3) *next* (array index) points to a successor. The *first* variable (array index) points to the head of the list. The list holds the actually present applications in the structure.

The complexity of operations on the proposed linked list is as follows: initialization – $O(|A|)$, inserting an element to the head of the list – $O(1)$, removing element by index – $O(1)$, list traversal – $O(1)$ per step. The complexity of Algorithm 4.12, assuming that the linked lists are used, is given in Theorem 4.3.2.

Theorem 4.3.2. *The time complexity of Algorithm 4.12 is* $O(|A| \cdot |L|)$.

Proof. Let B^k be the content of the list B at the end of iteration k. The complexity of iteration k can be analyzed in the following manner. The complexity of the loop in line 2 is $O(|L|)$. The complexity of line 5 is $O(|B^{k-1}|)$, because $|B^{k-1}|$ elements are removed from the list. Since the size of every A_l is bounded from above by $|B^k|$, the complexity of the loop in line 6 is $O(|L| \cdot |B^k|)$. The loop in line 7 has a similar complexity, as lines 7.1-7.2 are executed in $O(1)$. Line 8 is executed in $O(|B^k|)$. The total complexity of iteration k is $O(|L| \cdot |B^k| + |B^{k-1}|)$. We know that $\Sigma_k |B^k| = |A|$,

so the cumulative complexity of all iterations is $O(|A| \cdot |L|)$. □

4.3.3 Distributed Algorithm

We next develop a distributed version of Algorithm 4.12. Initially, we present PAFA, a parallel algorithm that relies on synchronized information transfer primitives. Then, we present distributed mechanisms that implement the required primitives, culminating in the distributed algorithm, dubbed DAFA.

Parallel AFA (Algorithm 4.13)

The algorithm uses a single main thread, called *the leader*, and one worker thread per link. The leader uses the following two variables (of fixed size):

- ρ – cumulative rate variable
- B – set of newly bottlenecked applications. Collected from scratch during the current iteration, to be removed at the beginning of the next iteration.

Each worker holds the following state, whose size is $O(|A|)$:

- c_l – residual link capacity
- W_l – the aggregate weight of active flows crossing l
- W_l^a – the aggregate weight of active flows belonging to a crossing l
- A_l – set of active applications crossing l
- B_l – set of newly bottlenecked applications *as known* to the worker of link l
- δ_l – rate increment allowed by link l

The computation proceeds as the main thread distributes and collects results from the worker threads, in a synchronized manner. For instance, in the shared memory model, distribution involves workers reading a global variable, while collection is performed by the leader reading local worker variables.

The main loop can be logically decomposed into two phases. During the *rate collection phase* (RCP) – lines 1-6, B is distributed to workers. Each worker removes

Algorithm 4.13 Parallel AFA

Initialization:
$B \leftarrow \phi$, $\rho \leftarrow 0$
$\forall l \in L : c_l \leftarrow 1, W_l^a \leftarrow \sum_{f \in F_l \cap a}(w_f), A_l \leftarrow \{a | \exists f \in a : f \in F_l\}$

Loop:
1. **distribute** B `to links`
2. `In parallel foreach` $l \in L$
   2.1 `foreach` $a \in B$ */*Remove newly bottlenecked apps*/*
      2.1.1 $W_l^a \leftarrow 0$
      2.1.2 $A_l \setminus \{a\}$
   2.2 $W_l \leftarrow \sum_{a \in A}(W_l^a)$ */*Calculate new aggregate weight*/*
   2.3 $\delta_l \leftarrow \frac{c_l}{W_l}$ */*If $W_l = 0, \delta_l \leftarrow \infty$*/*
3. **collect** δ_l `from links`
4. $\bar{\Delta} \leftarrow \min_{l \in L}\{\delta_l\}$ */*Aggregate allowed normalized rate*/*
5. `if` $(\bar{\Delta} = \infty)$ `terminate` */*Check termination condition*/*
6. `else` $\rho \leftarrow \rho + \Delta$
7. **distribute** $\bar{\Delta}$ `to links`
8. `In parallel foreach` $l \in L$
   8.1 $B_l \leftarrow \phi$ */*Reset newly bottlenecked app set*/*
   8.2 $c_l \leftarrow c_l - \bar{\Delta} \cdot W_l$ */*Update residual capacity*/*
   8.3 `if` $(c_l = 0)$ $B_l \leftarrow A_l$ */*Bottlenecked apps*/*
9. **collect** B_l `from links`
10. $B \leftarrow \bigcup_{l \in L}\{B_l\}$ */*Aggregate bottlenecked apps*/*
11. $\forall a \in B : \bar{r}(a) \leftarrow \rho$ */*Assign rate*/*

inactive applications from A_l, updates W_l, and computes its local δ_l. When all workers finish, δ_l results are collected by the leader. The leader aggregates the collected information by finding minimum and storing it in $\bar{\Delta}$. If the new $\bar{\Delta}$ value happens to be ∞, the algorithm terminates. Otherwise, the new ρ value is computed and the execution moves to the *rate distribution phase* (RDP) – lines 7-11.

The RDP begins by distribution of $\bar{\Delta}$ to workers. Each worker updates its c_l; if this results in the link becoming saturated, its B_l is set to be A_l. Once all the workers finish, the leader collects B_l sets and aggregates them into their union B. These are the applications that are going to be removed from the network in the next iteration; their rates are set to ρ.

DAFA*

Algorithm 4.13 is suitable for any parallel environment that supports synchronized information distribution and collection semantics. In fact, it can be used in a distributed network environment as well. For this purpose, each network element (switch, HCA) is required to store the state of the worker thread for each of its links. Here, for practical reasons, the number of applications must be small and constant (fixed in the hardware)[5]. In this context, A_l and B_l sets can be implemented as fixed size binary vectors. Similarly, the set of W_l^a values is a vector of real numbers.

Every network element has to be capable of executing lines 2.1-2.3, 8.1-8.3 for each of its outgoing links. These computations can be performed by dedicated hardware for each link, or by a centralized controller of the network element. One of the elements is designated as the leader. Now, we only need to define how the information is distributed, collected, and aggregated.

Since aggregation involves associative reduction operations of finding minimum and set union, it is convenient to use a technique similar to "map-reduce" (or "scatter-gather") on a spanning tree of network elements rooted at the leader. The RCP and RDP are executed on the spanning tree as follows. The relevant information is disseminated over the tree from the root towards the leaves. After receiving the distributed information, every node computes δ_l/B_l for its "outbound" links and waits for additional results from its sons. Upon arrival of the additional inputs, the node finds the minimum/union of all values. The output is sent to the parent of the node. In this manner, when information converges to the root, the leader gets the aggregated result and can begin a new phase.

The spanning tree can be explicitly constructed and maintained over time. Alternatively, we can use Segall's PIF algorithm [36] to build the tree implicitly during every execution of RCP or RDP. The operation of PIF itself is logically divided into two phases. During the *forward phase*, PIF distributes a message sent by the leader to all network elements. In addition, during this phase a logical spanning tree is

[5]We believe that in reality 32 to 128 applications should be sufficient for the needs of utility clusters.

defined by the "first-heard-from" relation. During the *feedback phase*, just as we need, each network element (except for the leaves) waits to receive feedback messages from its sons. When this happens, it sends a feedback message to its parent and finishes the execution of PIF.

The use of Algorithm 4.13 in conjunction with PIF for information distribution and collection provides us with a distributed algorithm for a one-time rate calculation, given that the links' state is updated *a-priori*. The behavior of this algorithm, which we call DAFA*, is analogous to the behavior of DFFA* from Section 4.2.

DAFA

As with DFFA in Section 4.2, we want DAFA* to be executed anew every time a change in $F(t)$ occurs. However, in the multiple applications scenario, $F(t)$ is expected to change only when some application enters or leaves its communication phase. Therefore, in *distributed AFA* (DAFA) the leader should restart DAFA* at these occasions. In order to let network elements know that the execution is restarted, rounds are used again.

Similarly to DFFA, each link must hold both W_l^a, A_l and $\widetilde{W}_l^a$, $\widetilde{A}_l$ variables. The former change during every DAFA* calculation, while the latter constitute the "true" values that are affected only when a flow crossing l starts or ends.

DAFA is summarized in Algorithm 4.14. It states the behavior of three entity types: the leader, network elements, and processing nodes of applications. All round values are initialized to zero. The leader waits for an external trigger to restart the execution of DAFA*. When the trigger is received, rnd_{lead} is incremented to distinguish new control packets from the old ones. Every network element $e \in E$ has its own round rnd_e. Network elements ignore control packets marked with $rnd_{lead} < rnd_e$. Upon receipt of a packet with $rnd_{lead} > rnd_e$, the state of all links together with the PIF state are reset, and the element joins the new execution of DAFA*.

When an application starts/ends a communication phase, and before it sends a trigger to the leader, links have to be updated about flows joining/leaving the network. For this purpose, the relevant sources send control packets to make the

Algorithm 4.14 DAFA

Initialization (at network setup):
$rnd_{lead} \leftarrow 0$
$\forall e \in E : rnd_e \leftarrow 0$

Leader upon receipt of a trigger:
1. $rnd_{lead} \leftarrow rnd_{lead} + 1$
2. $\rho \leftarrow 0$
3. `Restart DAFA*`

Leader upon completion of DAFA*:
1. `Distribute` $\bar{r}(a)$ `to` $\forall e \in E$

Network element e upon receipt of a packet with $rnd_e < rnd_{lead}$:
1. $rnd_e \leftarrow rnd_{lead}$
2. `foreach` l `of` e
   2.1 $c_l \leftarrow 1$
   2.2 $W_l^a \leftarrow \widetilde{W}_l^a$, $A_l \leftarrow \widetilde{A}_l$
   2.3 `Join the new execution of DAFA*`

Processing nodes of application a upon start/end of communication phase:
1. `Send control packets to update` $\widetilde{W}_l^a$ `and` $\widetilde{A}_l$ `on the path`
2. `Wait for packets to return`
3. `Perform synchronization barrier`
4. `Let one of the nodes send a trigger to the leader`

necessary announcements. The announcement packets carry the weight of a flow and the index of the application. As a result, $\widetilde{W}_l^a$ and $\widetilde{A}_l$ values are updated ($a \in \widetilde{A}_l$ iff $\widetilde{W}_l^a > 0$). Once the packets return, a synchronization barrier is executed, to make sure that when the trigger is sent, all links have a correct, updated state. When the barrier ends, the trigger is finally sent to the leader. After the new execution of DAFA* ends, the leader distributes $\bar{r}(a)$ values to all network elements.

4.4 Theoretical Comparison

The various algorithms, each of which was introduced in the context of a particular scenario with the corresponding performance goal in mind, all end up prescribing a transmission rate for every flow. Thus, some of them can be used for multiple scenarios, albeit possibly yielding sub-optimal rate assignments and/or being overly

complex. This section addresses this issue by formally examining the properties of the algorithms. The discussion is arranged by scenario. The formal nature of the examination leads to worst case analysis. The empirical counterpart of the comparison appears in Chapter 6.

4.4.1 Single Application

Three algorithms that achieve optimal rate assignments for the single application scenario were presented in Section 4.1. Those are: SAA, SAA-M, and assigning $\frac{w_f}{W}$ to all flows. In fact, the optimality condition for a feasible rate assignment was shown to be $\forall f \in F : \bar{r}(f) \geq \frac{1}{W}$. It is not hard to see that FFA and also AFA satisfy this condition. As a result, these algorithms yield optimal results for the single application scenario as well. However, the algorithms differ significantly in their efficiency in terms of running time, number of messages and required state information. It is therefore not recommended to use multi-application algorithms for single-application systems.

4.4.2 Independent Flows

FFA was shown in Section 4.2 to provide the optimal rate assignment for independent flows. The problem of finding weighted max-min fair rate assignment for independent flows can be reduced to finding weighted max-min fair rate assignment for single-flow applications. Therefore, theoretically, AFA can be used for this calculation. One has to remember though, that in its distributed implementation, network elements are required to store $O(|A|)$ size state, which may become impractical as it becomes $O(|F|)$.

Now, let's examine the behavior of SAA. As was mentioned in Section 4.1, this algorithm does not provide maximality. As a result, it obviously cannot be optimal for independent flows. For a fixed flow set and routing, Proposition 4.4.1 bounds the slowdown that an individual flow may experience due to use of an assignment calculated by SAA instead of one calculated by FFA.

Proposition 4.4.1. *If maximum link weight is W and $\forall f \in F : w_f \geq 1$, then the*

following is the tight upper bound on the relative normalized rates assigned to a flow by FFA and SAA:

$$\forall f \in F : \frac{\bar{r}_{FFA}(f)}{\bar{r}_{SAA}(f)} \leq \frac{(W+1)^2}{4W}. \tag{4.6}$$

Proof. First, we need to show that the proposed expression indeed constitutes an upper bound. Consider a flow f'. Let l be the link that determines $\bar{r}_{SAA}(f') = \frac{1}{W_l}$, and let $F_l^- = F_l \setminus \{f'\}$. Maximum link weight is W, so $\forall f \in F : r_{FFA}(f) \geq \frac{w_f}{W}$ ($r_{FFA}(f)$ is an absolute, rather than normalized, rate). As a result:

$$\bar{r}_{FFA}(f') \leq \frac{1 - \sum_{f \in F_l^-} \left(\frac{w_f}{W}\right)}{w_{f'}}, \tag{4.7}$$

$$\frac{\bar{r}_{FFA}(f')}{\bar{r}_{SAA}(f')} \leq \frac{W_l}{w_{f'}} \left(1 - \frac{1}{W} \sum_{f \in F_l^-} w_f\right) = \left(1 + \frac{1}{w_{f'}} \sum_{f \in F_l^-} w_f\right) \cdot \left(1 - \frac{1}{W} \sum_{f \in F_l^-} w_f\right). \tag{4.8}$$

The second equation is attributed to the fact that $W_l = w_{f'} + \sum_{f \in F_l^-} (w_f)$. If we denote $\omega = \sum_{f \in F_l^-} (w_f)$ and recall that $w_f \geq 1$, we achieve the following inequalities:

$$\frac{\bar{r}_{FFA}(f')}{\bar{r}_{SAA}(f')} \leq \left(1 + \frac{1}{w_f} \cdot \omega\right) \cdot \left(1 - \frac{1}{W} \cdot \omega\right) \leq (1 + \omega) \cdot \left(1 - \frac{1}{W} \cdot \omega\right). \tag{4.9}$$

Applying basic calculus, we find the maximum of the rightmost expression. The maximum is achieved for $\omega = \frac{W-1}{2}$ and its value is $\frac{(W+1)^2}{4W}$, as required.

Now, we present an example in which the upper bound is achieved. Consider the scenario in Figure 4.3. Three flows, represented by dashed arrows, cross a switch sw_1. Respective weights are $w_{f_1} = 1$, $w_{f_2} = \frac{W-1}{2}$, $w_{f_3} = \frac{W+1}{2}$. In SAA, the normalized rate of f_1 is determined by link $sw_1 \rightarrow d_1$ to be $\bar{r}_S(f_1) = \frac{2}{W+1}$. In FFA, the normalized rate of f_2 is determined by link $s_2 \rightarrow sw_1$ to be $\bar{r}_F(f_2) = \frac{1}{W}$. The normalized rate of f_1 in this case is determined by the residual capacity on $sw_1 \rightarrow d_1$. As a result, $\bar{r}_F(f_1) = \frac{W+1}{2W}$. The desired ratio immediately follows. □

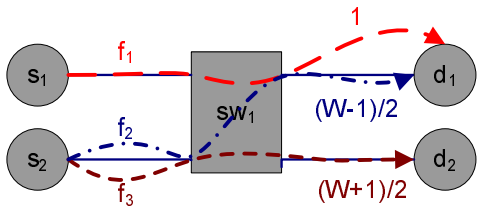


Figure 4.3: SAA vs. FFA bound

Unlike SAA, SAA-M calculates a maximal rate assignment. It also guarantees $\forall f \in F : \bar{r}(f) \geq \max_{l \in L_f} \left\{ \frac{1}{W_l} \right\}$. As a result, $\frac{(W+1)^2}{4W}$ can be shown to constitute an upper bound for SAA-M, as well. Since SAA-M distributes the excess capacity greedily, it is possible that some flows do not benefit from it. Therefore, we conjecture without proof that a more complex example can be constructed to demonstrate the tightness of the bound, or at least to show that for large W the tight bound is very close to $\frac{(W+1)^2}{4W}$.

4.4.3 Multiple Applications

AFA is fundamentally different from FFA and SAA, as it accounts for flows' belonging to different applications. As a result, AFA is capable of setting the same normalized rate to all flows of an application, while leaving more free capacity to other applications. For a fixed routing and sets of flows and applications, Proposition 4.4.2 bounds the slowdown that an individual application can experience due to the use of FFA or SAA in lieu of AFA. Surprisingly, the bound is equal to the one presented in Proposition 4.4.1.

Proposition 4.4.2. *If maximum link weight is W and $\forall f \in F : w_f \geq 1$, then the following tight upper bounds hold:*

$$\forall f \in F : \frac{\bar{r}_{AFA}(a)}{\bar{r}_{SAA}(a)}, \frac{\bar{r}_{AFA}(a)}{\bar{r}_{FFA}(a)} \leq \frac{(W+1)^2}{4W}. \tag{4.10}$$

Proof. We show the proof for FFA, and state that it is trivially repeated for SAA. Consider an application a. Let $f' = \arg\min_{f \in a} \{\bar{r}_{FFA}(f)\}$; according to our definitions $\bar{r}_{FFA}(a) = \bar{r}_{FFA}(f')$. The behavior of FFA dictates that at the end of its run,

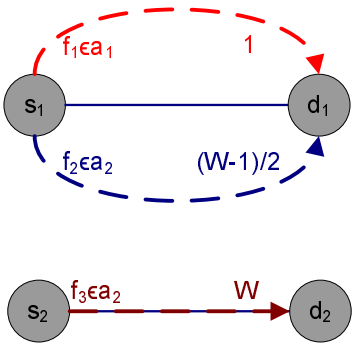


Figure 4.4: SAA, FFA vs. AFA

there exists a saturated link $l \in L_{f'}$ for which $\forall f \in F_l : \bar{r}_{FFA}(f) \leq \bar{r}_{FFA}(f')$. Since l is saturated and f' has the highest normalized rate, we get:

$$\bar{r}_{FFA}(f') = \bar{r}_{FFA}(a) \geq \frac{1}{W_l}. \tag{4.11}$$

Let $F_l^- = F_l \backslash \{f'\}$; maximum link weight is W, so $\forall f \in F : r_{AFA}(f) \geq \frac{w_f}{W}$, thus:

$$\bar{r}_{AFA}(a) = \bar{r}_{AFA}(f') \leq \frac{1 - \sum_{f \in F_l^-} \left(\frac{w_f}{W}\right)}{w_{f'}}. \tag{4.12}$$

From here, we repeat the proof of Proposition 4.4.1 and achieve $\frac{\bar{r}_{AFA}(a)}{\bar{r}_{FFA}(a)} \leq \frac{(W+1)^2}{4W}$.

The example in Figure 4.4 shows that the bound is tight. Flows f_1, f_2, f_3 belong to applications a_1, a_2, a_2, respectively. The weights are given by: $w_{f_1} = 1$, $w_{f_2} = \frac{W-1}{2}$, $w_{f_3} = W$.

When AFA is applied we have:

$$\bar{r}_{AFA}(a_2) = \frac{1}{W} \Rightarrow \bar{r}_{AFA}(a_1) = \frac{1 - \frac{w_{f_2}}{W}}{w_{f_1}} = \frac{W+1}{2W}. \tag{4.13}$$

When FFA is applied we have:

$$\bar{r}_{FFA}(a_1) = \bar{r}_{FFA}(f_1) = \frac{1}{\frac{W+1}{2}} = \frac{2}{W+1}. \tag{4.14}$$

□

Due to their formal nature, the above bounds describe the worst-case behavior. An empirical comparison of different algorithms is found in Chapter 6.

Chapter 5

Practical Issues of Rate Control

In the discussion in Chapter 4, two simplifying assumptions were deliberately made: 1) we assumed that all traffic in the network consists of long, steady flows, and 2) the calculated rates were believed to be realizable in practice. In this chapter, we consider implications of these practical issues and offer solutions that allow a correct and beneficial application of our method.

5.1 Mixed Traffic

In practical networks, the long bulk transfer flows may at times be accompanied by additional, low-volume sporadic communication. The unstable, swiftly changing nature of the latter makes its control a doubtful task. Instead of trying to control the behavior of these different types of traffic in a unified manner, we propose to deal separately with each one of them.

The use of *virtual lanes* (VLs) in InfiniBand enables the partitioning of a physical network into two or more logical networks with private resources. The only common resource shared by all VLs is the capacity of physical links, access to which is governed by a priority mechanism.

We can use one dedicated logical network for long flows and another for sporadic traffic. Because long flows are controlled and well-behaving, we can limit the maximum capacity they use on every link, and leave the rest to the sporadic traffic. This is easily achieved by declaring only $1 - \alpha_l$ of a link's capacity as available to

the rate calculation algorithms. The value of α_l may vary for different links, and is expected to be the capacity allocated to the sporadic traffic on link l.

How the chosen capacity division is implemented requires further investigation. On the one hand, it seems reasonable to give long flows an absolute priority in access to the physical link since, unlike the sporadic traffic, long flows can't claim more than their allowed capacity $(1 - \alpha_l)$. (In addition, as will be shown below, long flows may exhibit a periodic pattern of link utilization, which prevents temporary starvation of the sporadic traffic packets.) On the other hand, the sporadic traffic may be more latency sensitive which suggests that it should be prioritized.

In general, an overestimation of α_l might leave link capacity underutilized, while its underestimation would starve the sporadic communication. Perhaps the values should be adjusted from time to time. The choice of α_l values is beyond the scope of this work and this may be an interesting topic for future research.

5.2 Packet Injection Scheme

The discrete nature of packet networks was abstracted away from the discussion in Chapter 4. There, we considered the traffic to be fluid and governed only by the capacity of the links. While this approach facilitates the formal reasoning, it does not accurately reflect reality. Three major factors affect the gap between the fluid model and practical networks. First, the packet injection policy at sources determines the size and frequency of bursts of the injected traffic. We concentrate on a policy for individual sources, since it is impractical to coordinate injection of multiple sources. Second, buffering can smooth quantization-related phenomena. Unfortunately, as mentioned in Chapter 1, InfiniBand switches typically have relatively small buffers. Finally, the effect of scheduling of packets contending for the same output port may affect the ability of the fabric to keep up with the injected traffic.

Our goal in choosing an injection scheme is to restrain the burstiness of the traffic, in order to reduce the dependency on buffer smoothing. As a result, we propose to let sources transmit packets periodically, while applying a selection (among flows) mechanism for every transmission.

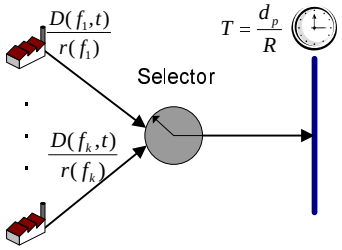


Figure 5.1: Periodic Selection injection scheme

In Section 6.6, we evaluate the success of the realization of calculated rates by means of simulation with varying buffer size and the following assumptions on the behavior of switches: 1) output ports apply a first come – first served (FCFS) service policy ; i.e., an output port sends packets from different input ports in the order of their arrival to the switch, and 2) an input port can send a packet to every output port during the same time step (infinite speedup).

5.2.1 The Scheme

Assume that a single flow f with packets of a fixed size d_p originates at the source. The source can set the inter-packet delay (IPD) to $\frac{d_p}{r(f)}$ and transmit a single packet every IPD. This approach can be generalized to deal with several flows having the same rate. Now, the injection is performed in cycles of the same IPD, but more than one flow transmits a packet during the cycle, as the period is divided between the flows in a simple time-division multiplexing manner. When several flows with different rates, yet the same packet size d_p, are considered, the multiplexing of multiple flows onto a single output link is a challenge.

We propose the *periodic selection* (PS) injection scheme that is an adapted version of SVC [38]. Let $D(f,t)$ stands for the amount of data sent by the source for a flow f until time t. In the SVC, every time a source ends transmission of a packet, the next packet is chosen from a flow f that satisfies the following conditions: 1) f is "eligible" to send a packet; i.e., the time that passed from the last transmission of f's packet is at least $\frac{d_p}{r(f)}$ (the transmission of the new packet would not violate f's rate), and 2) $f = \arg\min\left\{\frac{D(f,t)}{r(f)}\right\}$ among eligible flows. If no eligible flow is found, the transmission of the next packet is delayed until one of the flows becomes eligible.

Note that while the SVC prevents bursts of individual flows, it allows bursts of aggregate traffic leaving the source. In PS, which is illustrated in Figure 5.1, unlike in SVC, the source transmits a single packet every $\frac{d_p}{R}$. Yet again, the transmitted packet is (logically) selected just before the transmission. The flow to provide a new packet at time t is $f = \arg\min\left\{\frac{D(f,t)}{r(f)}\right\}$. In this manner the aggregate traffic leaving the source has a periodic nature as well.

If a packet cannot be transmitted due to back-pressure, its transmission is delayed until free buffer space on the receiving side is available again. The waiting time is considered lost, so once possible, the source will resume its periodic operation without attempting to compensate for the lost work. Therefore, such waiting can be seen as added to the time until completion of all participating flows.

The PS scheme can be implemented in two different practical ways. The simple implementation performs selection of the next packet by comparing all flows when a transmission of a current packet begins. If dedicated hardware is used for that purpose, the comparison can be performed in a logarithmic time on a comparison tree. Otherwise, the flows must be compared serially, which might increase the delay considerably. Another alternative is to hold flows in a sorted list structure. The head of the list provides the next packet, and when a new packet is sent to transmission the head is moved to a new place as its $\frac{D(f,t)}{r(f)}$ changes. (This ratio remains unchanged for other flows.)

Now let us prove a fundamental property of the PS injection scheme: if the source succeeds to send a packet every $\frac{d_p}{R}$, then in the steady state ($t \to \infty$) the average rates of all flows converge to the calculated ones (see Theorem 5.2.2 below).

Lemma 5.2.1. *Assume that flows have an infinite length, and let $n_p(t)$ be the total number of packets actually transmitted by the source until time t. When $t \to \infty$, if $n_p(t) \to \infty$, then $\forall f_1,\ f_2 :\ \frac{D(f_1,t)}{D(f_2,t)} \to \frac{r(f_1)}{r(f_2)}$.*

Proof. Consider some moment t_1. Let $f' = \arg\max\left\{\frac{D(f,t_1)}{r(f)}\right\}$, and let $t_0 < t_1$ be the time when f' sent its last packet. It follows that $f' = \arg\min\left\{\frac{D(f,t_0)}{r(f)}\right\}$. Obviously, $\forall f \in F :\ \frac{D(f,t)}{r(f)}$ is monotonically non-decreasing. It follows that

$$\frac{D(f',t_0)}{r(f')} \leq \min_{f\in F}\left\{\frac{D(f,t_1)}{r(f)}\right\}. \tag{5.1}$$

f' sends no packets after t_0, so

$$\frac{D(f',t_1)}{r(f')} = \frac{D(f',t_0)}{r(f')} + \frac{d_p}{r(f')}. \tag{5.2}$$

As a result, we conclude that

$$\max_{f\in F}\left\{\frac{D(f,t_1)}{r(f)}\right\} - \min_{f\in F}\left\{\frac{D(f,t_1)}{r(f)}\right\} \leq \frac{d_p}{r(f')} \leq \max_{f\in F}\left\{\frac{d_p}{r(f)}\right\}. \tag{5.3}$$

Therefore, as $t \to \infty$, given that $n_p(t) \to \infty$, it is not hard to show that $\forall f$: $D(f,t) \to \infty$. Therefore:

$$\frac{D(f_1,t)}{r(f_1)} \simeq \frac{D(f_2,t)}{r(f_2)}$$

□

Theorem 5.2.2. *If a packet is injected to network every $\frac{d_p}{R}$ (without back-pressure), then $\frac{D(f,t)}{t} \to r(f)$ as $t \to \infty$.*

Proof. First, if no back-pressure is applied on the source, then the condition of Lemma 5.2.1 is true. Let $D(t)$ be the total amount of data sent by the source by time t. Because a packet of length d_p is transmitted by the source every $\frac{d_p}{R}$, when $t \to \infty$ we have:

$$\sum r(f_i) = R \simeq \frac{D(t)}{t} = \sum \frac{D(f_i,t)}{t}$$

Assume by contradiction, that for some i, $\frac{D(f_i,t)}{t} < r(f_i)$. If so, from the above equality we must conclude that for some j, $\frac{D(f_j,t)}{t} > r(f_j)$. However, this leads to a contradiction of Lemma 5.2.1, since:

$$\frac{r(f_j)}{r(f_i)} < \frac{D(f_j,t)}{D(f_i,t)}$$

□

As a final step, we extend the PS injection scheme to operate correctly when flows use varying packet size. We restrict the length of packets to be an integer

multiple of some basic size d_p. The extended scheme still operates in cycles of $\frac{d_p}{R}$. However now, when the transmission of packet of size $k \cdot d_p$ starts, no new packet is injected for $k \cdot \frac{d_p}{R}$ time, i.e. the next $k-1$ "clock ticks" are dedicated to the same packet. As in the simpler case, it is not hard to see that a d_p amount of data is sent every $\frac{d_p}{R}$ amount of time on average. Therefore, the overall transmission rate is R. In addition, given that the packet size is bounded from above, Theorem 5.2.2 can be proved to be right for the general case.

The PS injection scheme on its own cannot guarantee successful realization of calculated rates. As stated above, buffers' size and service policy in switches play a crucial role as well. In Section 6.6 we test empirically (through simulation) the behavior of the proposed scheme under FCFS service policy and varying size of input buffers. The results presented there indicate that the proposed injection scheme implements calculated rates with practical buffer sizes. Therefore, in all later experiments the calculated rates are regarded as the actual behavior of the flows.

Chapter 6

Empirical Results

In this chapter, we present and discuss empirical results for the proposed adaptive routing (Chapter 3) and rate calculation (Chapter 4) schemes. The results were acquired through simulation in a large scale fat tree topology. This topology was chosen due to its high popularity in clustered computing.

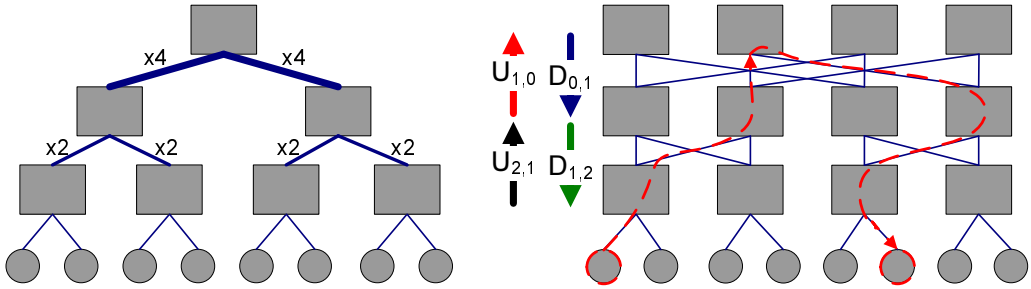
Following the conclusions of Chapter 5, in tests that apply rate control, we rely on calculated results, without actually simulating the injection of the traffic and its propagation through the network. In addition, although the algorithms in Chapter 3 and Chapter 4 support weighted flows, our experiments assume all flows to have the same length/weight.

6.1 Fat Trees

Topology

The *ideal fat tree* topology is almost identical to the regular tree. The two topologies differ only in the capacity of their links. While in the regular tree links have an equal capacity, in ideal fat tree the capacity grows (is multiplied by radix) for every ascending step. The increased capacity is intended to prevent creation of a severe bottleneck at the higher switches of the tree, in particular the root.

For instance, consider the ideal binary fat tree of height three (only switch levels are counted in height) presented in Figure 6.1-a. The end nodes are connected to switches by links of unit capacity. The capacity of links is doubled as the root



(a) Ideal binary fat tree (b) 2-ary 3-tree

Figure 6.1: Fat Trees

is approached, such that the capacity of links connected to the root is four times higher.

In practice, the increased link capacity is implemented by placing several parallel links of a constant, unit capacity. This leads to an unreasonable requirement on the number of ports at the higher levels of larger ideal trees. As a result, the "ideal" topology cannot be used in reality.

A practical implementation of the ideal fat tree is the *k-ary n-tree* [31]. Such binary tree with height of three is depicted in Figure 6.1-b. The k-ary n-tree replaces every logical node of the ideal tree with a number of physical switches. This number is multiplied with every ascending step. In the example, the root is realized as four switches and its children by two switches each. Importantly, the set of end-nodes that can be reached by descending from any given switch is identical to the one that can be reached from the associated logical node in the ideal tree.

Routing

In an ideal fat tree, just like in a regular tree, routing is simple because a single minimal path exists for every source-destination pair. Such a path includes ascending to the closest common ancestor of the two, and then descending to the destination.

The minimal routing in k-ary n-tree implies the same procedure (see example in Figure 6.1-b). Here, however, multiple minimal paths connect each source with every destination. In fact, the ascending can be performed arbitrarily until a switch belonging to a closest common ancestor in the ideal tree is (guaranteed to be) reached. Once the ascent is complete, the descending path is unambiguously determined.

As a static baseline in the following experiments, we used the oblivious routing algorithm proposed in [15]. This algorithm has two interesting properties. First, it guarantees complete contention avoidance for *shift permutation* traffic. In addition, for *any permutation*, flows do not collide on the descending part of the path.

Note that the k-ary n-tree topology is known to be *rearrangeably non-blocking*, i.e., for every given permutation, contention-free routing can be found (valid for *that* permutation). However, even if only permutation traffic is considered, achieving this optimum result requires globally re-computing the routing every time the pattern changes.

6.2 Adaptive Routing

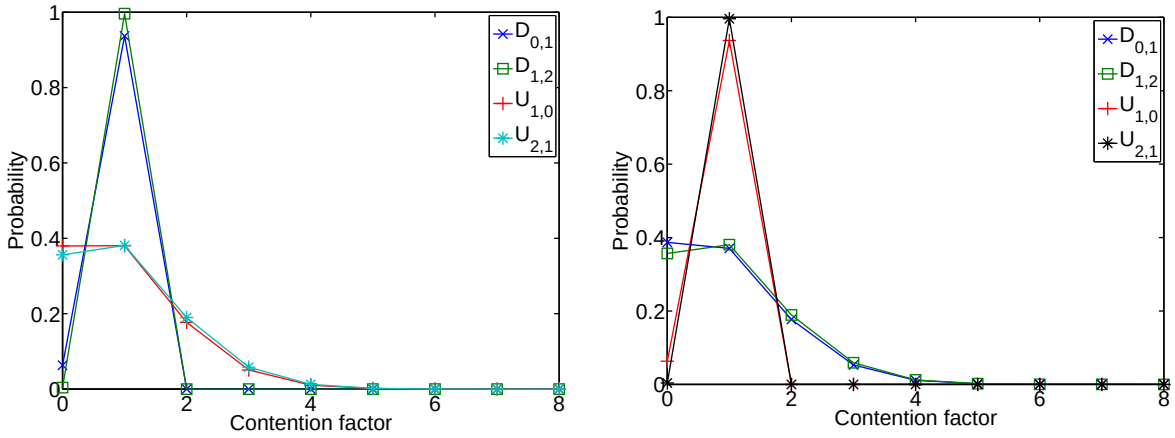
As mentioned in Chapter 1, the purpose of adaptive routing is to reduce the contention on links through load balancing. Note that when multiple flows target the same destination, no load-balancing can relieve the contention on its incoming link. For our initial capabilities test and parameter setting, we therefore chose a random permutation traffic pattern, for which every end-node has a single incoming and outgoing flow. Later, our adaptive routing combined with rate control were tested under additional traffic patterns.

Adaptive Ascent

As a first step, we examined the possibility of letting flows to be routed adaptively during the ascent, i.e., at each ascending step the port to be used is chosen adaptively. As mentioned above, no adaptation is possible on the descent. We collected simulation results for 1000 random permutations in 16-ary 3-tree (4096 end-nodes). Here and later, we used the number of flows crossing a link as a heuristic measure of dynamic link quality, and the size of the RC table was assumed to be unlimited[1].

For each permutation, we measured the number of flows crossing every link in the network under oblivious [15] (variants of this static routing are popular in real-

[1]The traffic patterns used in our simulations did not result in large numbers of flows crossing switches. Nevertheless, performance dependency on size of the RC table has yet to be examined.



(a) Oblivious (b) Adaptive ascent

Figure 6.2: Distribution of congestion factors in k-ary n-tree

life clusters) and adaptive ascending routing. We concentrated on four sets of links: $D_{0,1}$, $D_{1,2}$, $U_{1,0}$, and $U_{2,1}$, which are illustrated in Figure 6.1-b. D/U stand for down and up, respectively; the subscript describes the connected switch levels. Figure 6.2 depicts the probability of different levels of contention in each of these sets.

The baseline oblivious routing (Figure 6.2-a) avoids contention on the descending links under permutation traffic. For the ascending links, however, the tail of the distribution reaches as far as seven flows. The tail is important when total completion time is considered. When adaptive ascending was applied (Figure 6.2-b), the contention on the ascending links was effectively relieved, but local adaptation on the ascent failed to guarantee successful descent.

We conclude that the suggested adaptive ascending routing has no positive effect on the measured contention. Instead of trying to improve the heuristic by information exchange between switches, we propose to enrich the k-ary n-tree so as to enable adaptation for descending flows. We call the enriched topology *modified k-ary n-tree*.

Modified k-ary n-tree

In modified k-ary n-tree, switches belonging to the same logical node in the ideal tree are connected by horizontal links, as depicted in Figure 6.3. Due to the aforementioned properties of the k-ary n-tree topology, horizontal movement between such switches preserves reachability of a destination through descent.

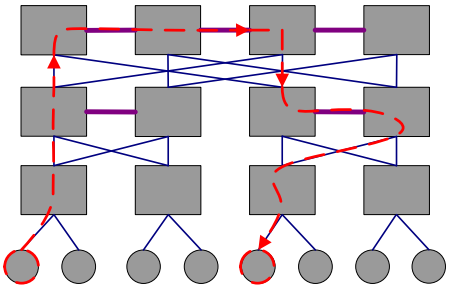


Figure 6.3: Modified k-ary n-tree

The routing in the enriched topology is still composed of the ascending and descending phases. It is depicted in Figure 6.3. The ascent is performed adaptively, exactly as in a regular k-ary n-tree. During the descending phase the routing is identical to that presented in Section 3.2, and is shortly described below.

A flow is allowed to take multiple horizontal hops, at every level of switches, before proceeding to a lower level. The horizontal direction (right/left) is chosen once, before the first horizontal hop, and is preserved until further descent. The direction is chosen at each level so as to maximize the permissible number of horizontal hops before reaching the boundary of a logical node. Thus, flows initially arriving to a switch in the left half of a logical node are sent to the right, and otherwise, to the left.

It is obvious that the proposed routing in the modified k-ary n-tree avoids cyclical dependency between links, and therefore is deadlock-free. In addition, the resulting flow paths do not contain cycles. The conditions of Theorem 3.1.1 are thus satisfied, and correct operation of our generic adaptive routing scheme is ensured.

We tested the modified topology and the proposed routing by repeating the random permutations experiment. The capacity of the horizontal links, i.e., the number of parallel links with unit capacity, referred to as *horizontal width*, was used as a parameter. This time, in every run, we measured the contention experienced by individual flows (maximum among links in L_f). The maximum and the average (over flows) results are summarized in Figure 6.4, which presents these measures for the oblivious and adaptive routing cases. Here and later, we averaged results over the (1000) runs.

We can conclude that with a horizontal width of two or more, the maximum con-

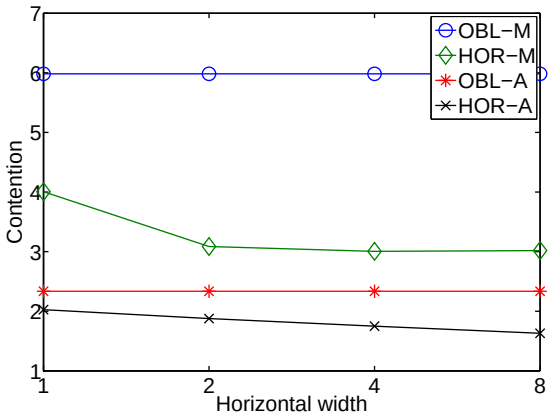


Figure 6.4: Number of horizontal links versus maximum congestion

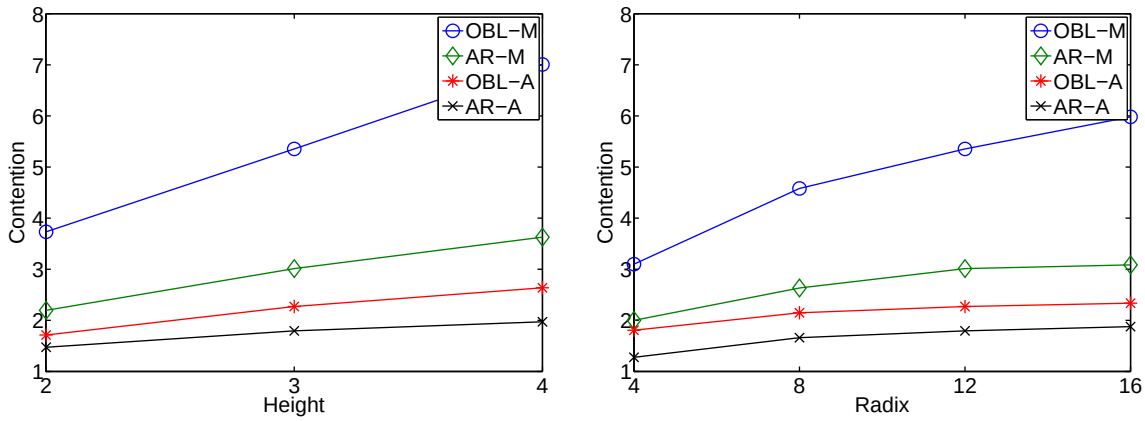
tention is reduced by approximately 50%, while the average contention is reduced by more than 20%. We stuck to horizontal width of two in the following experiments since it provides the best tradeoff between performance and overhead (to be discussed below).

Remark: Here and later, we compare the coupling of additional bandwidth with adaptive routing (which we commonly refer to as "adaptive routing") to static routing in non-modified tree. Apparently, in presence of unknown traffic pattern, no static routing can make efficient use of the horizontal links. We found it inconvenient to divide the additional bandwidth among the links in non-modified tree when static routing is applied, because this enhancement cannot be practically implemented.

Radix and Height

We tested the behavior of our adaptive routing under varying tree parameters in two experiments. In both, we used 1000 random permutations and measured the same quantities as in the previous setup. For the first experiment, we fixed the tree height to three and left radix as a parameter. In the second experiment, the radix was fixed to twelve, and the height was changed between two and four. Figure 6.5 summarizes the results.

The general observed trend is that the improvement achieved by our adaptive routing is higher for larger trees. This is especially true for maximum contention since in larger trees higher maximum contention is witnessed, as expected. The



(a) Radix = 12 (b) Height = 3

Figure 6.5: Adaptive routing under changing tree parameters

results suggest that the adaptation successfully balances the load.

Horizontal Movement Policies

So far, we fixed the horizontal direction based on the position of a switch within a logical node. Moreover, a flow was allowed to make an unlimited number of horizontal hops until the boundary of the logical node is reached. Now, we want to examine these adaptation policy decisions. For this purpose, we define two types of policies, *FreeDir(i)* and *ForcedDir(i)*, where i stands for the maximum allowed number of horizontal hops.

In FreeDir, the direction of the first horizontal movement at any given tree level is greedily chosen by a switch based on the heuristic quality of its ports, and the chosen direction remains fixed while at this level. By allowing additional freedom of choice, we apparently strengthen the adaptation mechanism. However, if a deciding switch is close to the boundary of its logical node, a greedy decision can severely limit the number of horizontal hops a flow can take before being forced to descend. ForcedDir is the policy we applied so far, in which the horizontal direction is unambiguously determined by the physical position of a switch within the logical node of which it is part.

This time, we measured the maximum contention for 1000 random permutations in modified 16-ary 3-tree. Figure 6.6 presents results for changing number of allowed

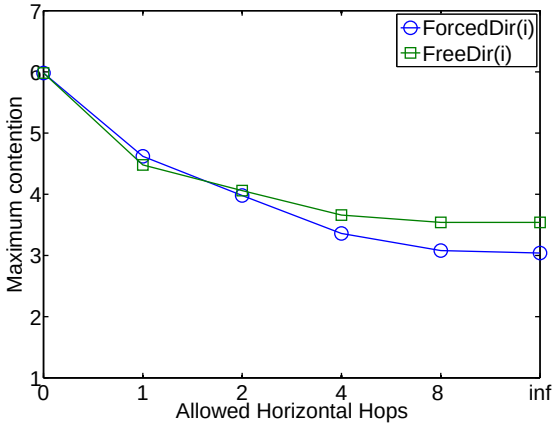


Figure 6.6: Effects of horizontal routing policies

horizontal hops between $i = 0$ (no horizontal movement) to $i = \infty$ (no limit on the number of allowed horizontal hops). Notice the unusual structure of the X-Axis. As we see, ForcedDir is generally better than FreeDir. The only exception is $i = 1$, when FreeDir provides more freedom of choice for the single allowed horizontal step. In addition, we notice that there is virtually no difference between the results for $i = 8$ and $i = \infty$.

From the practical perspective, setting $i = 8$ is preferable since it prevents creation of extremely long paths (the number of root switches in 16-ary 3-tree is 256). We stick to ForcedDir(8) in the rest of the experiments, in this chapter.

Overhead

The transition from a regular k-ary n-tree to the modified one entails adding ports and links to the topology. Table 6.1 summarizes the ratio between the number of added and existing switch ports, for varying radix and height parameters under horizontal width of two. We notice that for large clusters (height 3-4, radix 12-16) the overhead falls into the range of 10%-14%. As a reminder, applying adaptive routing in clusters of the same size leads to 40%-50% reduction in the maximum congestion factor. In light of these findings, the contention reduction more than justifies the investment in additional ports.

So far we only discussed adaptive routing. It is important to understand that the above results deal with *contention*, i.e., how many flows share a link, and not

h \ r	4	8	12	16
2	0.33	0.17	0.11	0.08
3	0.40	0.20	0.13	0.10
4	0.43	0.21	0.14	0.11

Table 6.1: Additional ports overhead of modified k-ary n-tree (horizontal width=2)

with the performance measures defined in Section 1.4. Now we add rate control and consider it both alone and in conjunction with adaptive routing. The discussion is organized by the three characteristic scenarios.

6.3 A Single Phase-Based Application

We tested the effect of SAA and adaptive routing on the length of the communication phase in a modified 16-ary 3-tree (4096 end-nodes). The application was assumed to send flows of a fixed size, that constitute a superposition of random permutations. Under this traffic pattern, every end node sends and receives a constant number of flows (equal to the number of permutations). Switches were configured to have input buffers with size of eight MTU packets. We demonstrate this size to be sufficient to implement calculated rates in Chapter 5 above.

The total completion time (in normalized units, averaged over 50 runs) for different control schemes is presented in Figure 6.7. We witness an interesting phenomenon. SAA alone (without adaptive routing) provides a slight improvement, of up to 13%, compared to the non-controlled baseline NC (no rate control, no adaptive routing). Adaptive routing alone (AR) increases the completion time when three or more permutations are used. However, when the two are combined (SAA+AR) they yield a major improvement of up to 50%.

We explain this apparently counter-intuitive behavior as follows. The adaptive routing reduces the maximum contention in the network, but, as a side effect, the path length of some flows increases significantly. As a result, those flows have an increased probability of being affected by congestion spreading. However, when the rate control (SAA) is applied, congestion spreading is avoided, and the reduced contention causes a major improvement in the total completion time. Thus, for a single

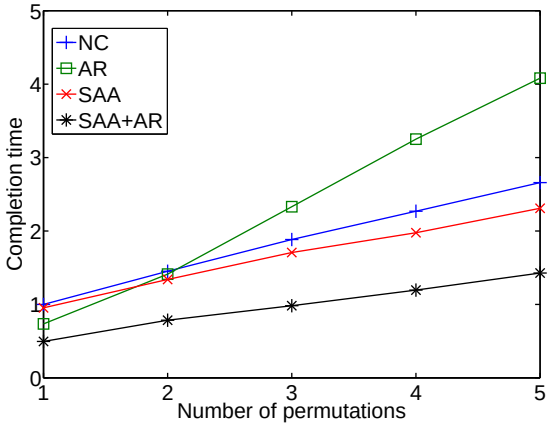


Figure 6.7: SAA performance

permutation, SAA+AR reduces the completion time twofold compared to SAA, as dictated by the twofold reduction of the maximum contention by the adaptive routing.

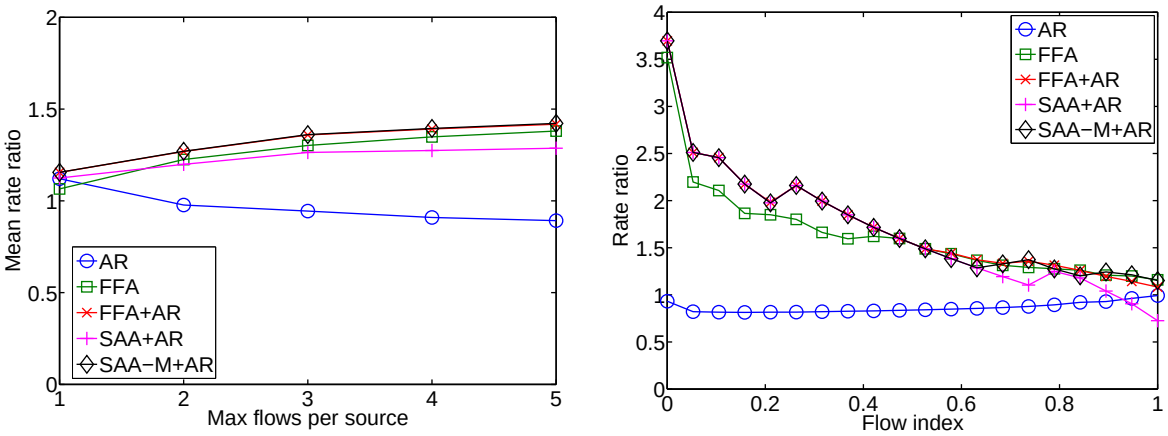
Note that the results in Figure 6.7 stay unchanged when SAA is replaced by SAA-M because both algorithms guarantee the same optimal completion time.

6.4 Independent Flows

Performance Analysis

In the independent flows scenario, we assessed the performance of adaptive routing and compared the performance of three rate control algorithms: SAA, SAA-M, and FFA. The FFA provides the optimal outcome according to the max-min fairness criterion. Also, recall that the SAA is obviously sub-optimal since it doesn't even guarantee maximality of capacity usage, i.e., some flows could increase their rates in a feasible manner, without affecting others.

The empirical data on the three algorithms was collected in the following setup. A modified 16-ary 3-tree topology was used. Yet again, switches were configured to have input buffers with size of eight MTU packets. A single parameter *max_flows* determined the actual traffic pattern. Before the beginning of a run, every source randomly (uniformly) chose an integer n_f in the interval [1,max_flows]. Then, the source sent n_f concurrent flows, each to a random destination. All flows had the



(a) Mean speedup (b) Rate vectors ratio (max_flows=5)

Figure 6.8: Rate control for independent flows

same weight.

In every run, the rates of all flows were collected for six control schemes: 1) no control (NC), 2), adaptive routing (AR) 3) FFA, 4) SAA with adaptive routing (SAA+AR), 5) SAA-M with adaptive routing (SAA-M+AR), and 6) FFA with adaptive routing (FFA+AR). For each control scheme, we derived the mean (among flows) rate and a sorted (by rate) vector of rates.

Figure 6.8-a presents, for each of the control schemes 2-6, the ratio between the mean rate with that scheme and with NC (averaged over 50 runs). We draw several important conclusions. To begin with, adaptive routing alone still has a potentially negative impact. However, now, if we compare FFA to FFA+AR, we notice that the contribution of rate control on its own is much more significant than in the single application scenario test. This result is attributed to two factors: 1) in independent flows performance does not depend on the worst case flow, and 2) under the chosen traffic pattern some flows suffer from congestion at destination, which cannot be relieved by the adaptive routing. Interestingly, the performance boost of SAA-M+AR is identical to that of FFA+AR. Overall, FFA+AR provides up to 43% increase in the mean rate.

Additional insight into the behavior of the different algorithms is gained if the element-wise ratio of the sorted rate vectors, instead of mean rates, is examined. This ratio is a natural measure in the context of the max-min fairness criterion,

as it expresses the improvement in the lowest rate, the second lowest rate, etc., for different control schemes (relative to NC). Because the number of flows slightly differed in different runs, in each run, ratio vector indices were (linearly) mapped to the [0,1] interval (the lowest rate at 0, the highest at 1). Then, ratio vector values at indices corresponding to a predefined set of points in [0,1], were collected. The results for max_flows=5 are presented in Figure 6.8-b and constitute an average over 50 runs.

If we compare SAA+AR, SAA-M+AR and FFA+AR, we observe that all algorithms provide a dramatic improvement for low-rate flows. Apparently, ensuring normalized rate of $\frac{1}{W_f}$ is sufficient to boost low rates. For higher rates, SAA+AR is worse than SAA-M+AR and FFA+AR. This is an expected result because SAA is known to leave underutilized capacity, even if some flows can exploit it. Unlike SAA+AR, the rates of SAA-M+AR are very close to those of FFA+AR. The only slight difference is that SAA-M+AR boosts high rates at the expense of medium ones. This behavior is a result of greedy capacity claiming applied by SAA-M.

Estimating Calculation Time of DFFA

Unlike distributed implementations of SAA and SAA-M, DFFA relies on a relatively large amount of computations and communication. (For SAA-M, the number of control packets that flows send after all flows enter the network and before acquiring a final rate in the above experiments, was no more than five on average. The maximum number of the sent packets was 18.) Recall that DFFA uses two special types of packets: control packets and round-broadcast packets. We estimate the calculation time of DFFA through simulation using the following timing model.

We assume that the special packets have an absolute priority in transmission over data packets, including preemption. Table 6.2 presents the set of parameters used in simulations. The set assumes that dedicated hardware is used to process the special packets in switches (and that the processing time in HCAs is negligible). The processing of control packets is performed *sequentially* at every output port, but in parallel among the ports. The round-broadcast packets are processed similarly at inputs. Refinements of the estimation model can be considered in future work.

Parameter	Value
Routing delay	100ns
Propagation delay	15ns
Link bandwidth	10Gb/s

(a) Network

Parameter	Control	Broadcast
Size	40 byte	8 byte
Processing Time	60 ns	30 ns

(b) Special packets

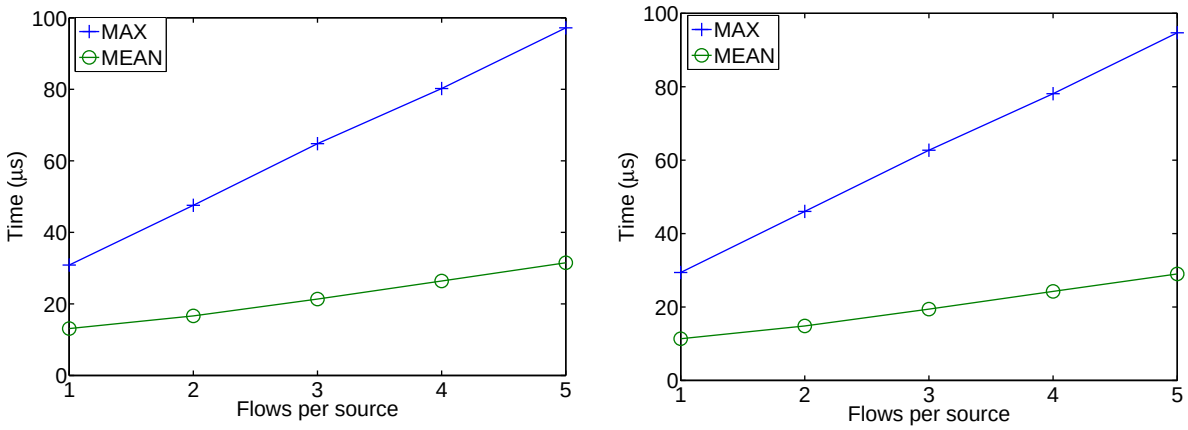
Table 6.2: DFFA overhead estimation parameters

We performed two experiments to evaluate the calculation time. In both experiments, sources sent a *constant* number (which is a parameter) of fixed weight, infinite size flows to random destinations. In the first experiment, all flows started approximately at time $t = 0$. For each flow, we measured the time at which its DFFA* loop is terminated for the last time (i.e., after the acquisition of the final result by that flow). In the second experiment, all flows except one started at $t = 0$. The remaining flow started after the initial rate calculation is over for all other flows. Its start triggers a re-calculation, the length of which we measured for each flow. In both cases, we collected the mean and maximum calculation time among flows (and averaged them over 50 runs).

The results are depicted in Figure 6.9-a and Figure 6.9-b, for the first and the second experiment, respectively. As a matter of fact, the two figures are almost identical. We observe that the mean calculation time in our experiments is bounded from above by $30\mu s$. The maximum time is bounded by $100\mu s$. For comparison, note that the transmission time of a 2KB packet is approximately $1.5\mu s$.

Although the measured times are not high, we must recall that in the worst case the overhead is incurred per *every* change in the set of flows $F(t)$. As a result, if the overhead becomes unbearable, we can invoke the DFFA periodically and use SAA-M, which is characterized by local effects of changes in $F(t)$ and fast convergence, between the invocations as proposed in Section 4.2.3.

Alternatively, we can rely solely on SAA-M. The latter approach is supported by the performance results presented earlier. The applicability of SAA-M as a general rate control algorithm is an interesting topic for further research.



(a) Common start (b) Triggered recalculation

Figure 6.9: DFFA overhead estimation

6.5 Multiple Phase-Based Applications

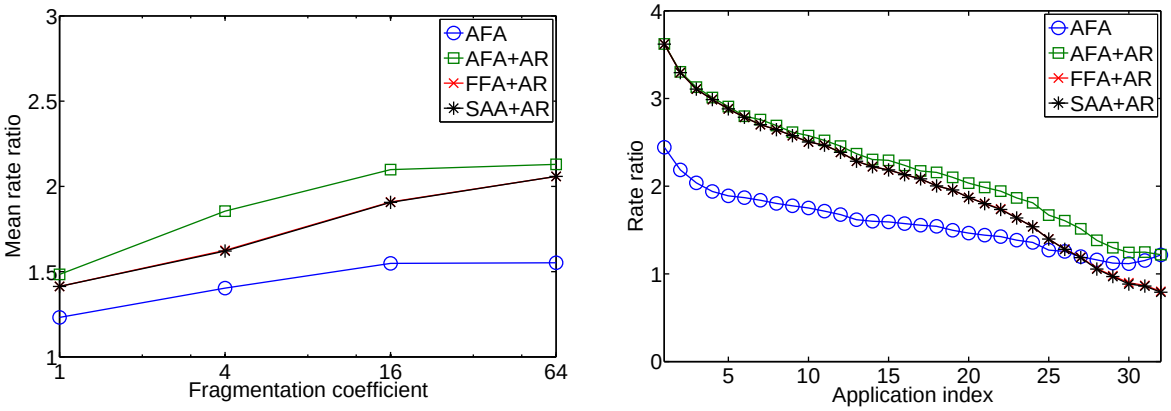
Performance Analysis

In order to test the behavior of our rate control algorithms in the multiple applications scenario, we decided to simulate the setup of a fragmented utility cluster. In such a cluster, applications with different numbers of required computing nodes come and leave. As a result, at any given time the unoccupied computing nodes are spread over the cluster in fragments. Therefore, a new application is not necessarily mapped onto physically neighboring nodes[2].

We let 32 applications use a modified 16-ary 3-tree. The mapping of nodes to applications was performed as follows. The 4096 nodes were divided into groups of neighbors of size $128/frag\_coeff$, where $frag\_coeff$ is a parameter. Each group was randomly mapped to one of the 32 applications, such that each application was ultimately assigned 128 dedicated nodes. The traffic pattern within the same application was a superposition of random permutations, whose number is chosen uniformly from the interval [1,5]. All participant flows had equal weights.

Five control schemes were tested: 1) no control (NC), 2) AFA, 3) SAA with adaptive routing (SAA+AR), 4) FFA with adaptive routing (FFA+AR), and 5) AFA with adaptive routing (AFA+AR). For every control scheme, in each run, the

[2]A similar fragmentation phenomenon is found in disks and memory.



(a) Mean speedup

(b) Rate vectors ratio (frag_coeff=4)

Figure 6.10: Rate control for multiple applications

rates of all applications were measured. Note that in general, higher rate means lower completion time. In the observed special case, when all applications enter their communication phase at $t = 0$ and flows are of equal size, the rate in inversely proportional to the completion time.

Here, similarly to the experiments for independent flows, we derived the mean rate among applications, and collected the ratio of mean rates between control schemes (2-5) and the baseline NC. In addition, the element-wise ratio of sorted rate vectors was gathered as well. Figure 6.10-a presents the speedup of mean rate. Figure 6.10-b depicts the ratio of sorted rate vectors for frag_coeff=4. All results were averaged over 50 runs.

Comparing the results achieved by AFA and AFA+AR, we conclude that the adaptive routing plays a key role in the current experiment. Generally, all rate control algorithms, combined with adaptive routing, greatly increase the mean rate. The best result is achieved by AFA+AR, which leads to up to 110% improvement (53% improvement of the average completion time).

When the sorted rate vectors are considered for SAA+AR, FFA+AR, and AFA+AR, we witness that all algorithms provide a dramatic improvement for weak applications. AFA+AR succeeds to help strong applications as well. This success is attributed to the ability of AFA to leave more free capacity to the strong applications without hurting the weak ones.

Interestingly, the results for SAA+AR and FFA+AR completely coincide. This evidence suggests that FFA+AR and SAA+AR assign identical rates to the flows that determine the rates of their respective applications. This conclusion is not surprising considering that SAA and FFA provide similar rates to slow flows in general, as shown by previous experiments.

Estimating Calculation Time of DAFA

As described in Section 4.3, the DAFA executes the AFA loop in a distributed manner after the leader receives a trigger. Each iteration of AFA loop includes performing two "map-reduce" operations originated at the leader. We want to estimate latency from the receipt of the trigger until the leader possesses the final application rates (i.e., the delay of DAFA* execution). For this purpose, we make the following definitions:

- I – number of required AFA iterations (bounded by $|A|$)
- h_{max} – maximum number of hops between the leader and any other network element
- T_d – single-hop information distribution delay (on a spanning tree) during the RDP (the "mapping" phase). Includes receiving, processing and further sending of data.
- T_c – single-hop information aggregation delay (on a spanning tree) during the RCP (the "reduction" phase). Includes receiving, processing and further sending of data.

Note that if we use PIF for the dynamic spanning tree construction, the expected height of the tree (number of hops between the root and the leaves) is exactly h_{max}. Given that, $h_{max} \cdot T_d$ is the upper bound on the time of information distribution from the leader to network elements. For the same reason, $h_{max} \cdot T_c$ is an upper bound on the aggregation time. Since each AFA iteration requires two map-reduce operations, the upper bound on the whole calculation time is:

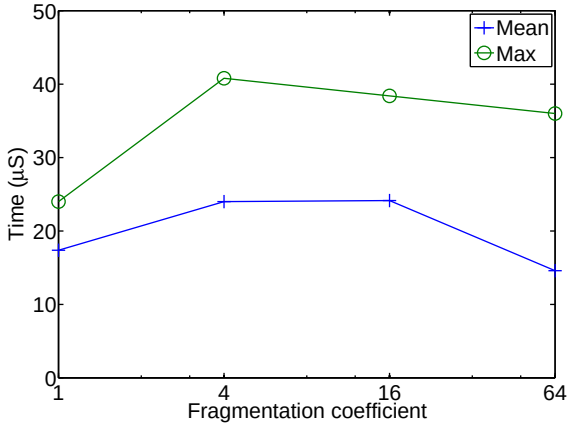


Figure 6.11: DAFA overhead estimation

$$T = 2I \cdot h_{max} \cdot (T_d + T_c)$$

In a 16-ary 3-tree, with leader placed at one of the root switches, it can be shown that $h_{max} = 4$. We assume T_c to be substantially higher than T_d, as it includes additional computation of δ_l/B_l and their aggregation (though, computation at different levels in the spanning tree occurs in parallel). The control packets used for calculation are assumed to have an absolute priority and be processed by dedicated hardware. Consequently, we set $T_d = 200ns$, which includes link propagation delay and link propagation delay. This estimation is based on specifications of real-life products and systems. We *conservatively* fix $T_c = 400ns$ (additional $200ns$ of processing time during aggregation).

The only unknown variable is I, which is theoretically bounded from above by $|A|$. We measure its actual value in the performance experiments that are described above. Figure 6.11 presents the average and maximum, *over 50 runs*, estimated calculation time.

The results show that even in the worst examined runs the calculation time does not exceed $40\mu s$. Note that this is a sheer calculation time, which does not include the overhead of synchronization at the beginning and the end of communication phase. The synchronization at one of these points can be coalesced with the global barrier performed by the application for every pair of computation and communication phases. The other, if executed on an overlay tree with reasonable radix (tree

height of 3-4), should be possible to perform within several end-to-end delays. We estimate the bound of the synchronization time to be $10 - 20\mu s$. The calculation overhead can be reduced if processing nodes initiate the calculation process ahead of time, before the actual beginning of the communication phase. Finally, it is important to note that in DAFA, as opposed to DFFA, the recalculations are expected to have a relatively low frequency, because they occur only when an application joins or leaves the communication phase.

6.6 Realization of Calculated Rates

We tested our injection scheme from Chapter 5 with both a single application and independent flows as described in the experimental parts of Section 6.3 and Section 6.4, respectively. In both cases, all flows consist of 2KB packets. Output ports of switches are assumed to employ FCFS policy on the packets arriving from different input ports. In addition, we assume that any number of packets can move from input buffer to *different* output ports ("infinite speedup"), yet an output port can only transmit a single packet at any given time. The number of flows per source n, and the size of the input buffers in switches (in units of 2KB) b, are used as test parameters. In every test, rates are calculated using one of our algorithms, the injection scheme is applied to implement the calculated rates, and the actual outcome is measured. All results are averaged over 50 runs. Most of the metrics exhibit very low variance.

For the single application scenario we use rates calculated for the SAA+AR control combination. As mentioned above, the completion time is dictated by the lowest rate flow. Therefore, in every run, we collect the ratio between the minimum measured and calculated rates (ratio of the minima). (Note that in any given run the two minimum results may correspond to different flows.) The collected results (averaged over 50 runs) are presented in Table 6.3. We see that with eight or more buffers per input port, the ratio is constantly 1, implying that the achieved completion time equals the optimal one.

For independent flows, we use the rates calculated with FFA+AR. Here, we are

n \ b	2	4	8	16
1	0.38	1.00	1.00	1.00
2	0.19	0.64	1.00	1.00
3	0.18	0.31	1.00	1.00
4	0.18	0.25	1.00	1.00
5	0.2	0.39	1.00	1.00

Table 6.3: Ratio of minimum measured and calculated rates

	MIN				AVG			
n \ b	2	4	8	16	2	4	8	16
1	0.19	1.00	1.00	1.00	0.91	1.00	1.00	1.00
2	0.20	0.58	1.00	1.00	0.85	1.00	1.00	1.00
3	0.24	0.48	1.00	1.00	0.83	0.99	1.00	1.00
4	0.31	0.47	1.00	1.00	0.80	0.99	1.00	1.00
5	0.40	0.53	1.00	1.00	0.78	0.98	1.00	1.00

Table 6.4: Average and minimum ratio of measured and calculated rates

interested in individual flows. Therefore, in every run we examine the ratio of the minima and that of the mean (over flows) measured and calculated rates (Table 6.4). Yet again, eight buffers per input port are sufficient to bring the minimum ratio to one, implying that the measured rate equals the calculated one for all flows. Even with only four buffers, we see that on average the vast majority of the flows achieve the desired rates.

Chapter 7

Conclusions

Congestion in high-speed computer-cluster interconnects cannot be overcome effectively with current schemes. In this work, we examined the application of novel adaptive routing and rate control schemes. InfiniBand was used as the platform for the work. The proposed adaptive routing relies on local heuristic adaptation and guarantees in-order delivery of packets and supports full scalability. We defined three major performance goals for clustered computing, and developed several explicit rate calculation algorithms (SAA, SAA-M, FFA, AFA) of varying complexity. Each algorithm is designed to achieve some of the goals.

Adaptive routing alone was shown to be effective in mitigating the "topological" congestion. However, due to the possibility of oversubscription to communication resources and the resulting congestion spreading, it was shown to be ineffective in regards to the defined performance metrics. Explicit rate calculation was shows to have some benefit in application-oriented scenarios and of greater benefit for independent flows. The combination of two was found to be very effective in both cases, boosting performance by tens of percents.

The proposed schemes are also practical. We showed that the calculated injection rates can be closely approximated even with a very limited number of buffers in switch ports. Also, the 16-ary 3-tree topology used with 4096 nodes used in the simulations is highly representative of current computer clusters, and the slight topological extension proposed for effective adaptive routing only entails a 10% increase in the number of switch ports. Finally, InfiniBand, whose architecture was

used as the base model, is the leading technology in current clusters.

The applicability of explicit rate calculation depends on the convergence time of the algorithms relative to flow durations. Our estimates and simulations suggest that these algorithms actually converge faster than reactive schemes, which moreover don't achieve optimal results. We moreover point out that whenever control messages are slowed down by contention among themselves for a shared link, the same will be true for the flows. Thus, the range of flow size for which the overhead is sufficiently low is insensitive to congestion and perhaps even benefits from it.

The schemes proposed in this work appear promising and practical. Topics for further research include: 1) a deeper examination of the implementation details, 2) tests with real-life benchmarks, 3) application of the adaptive routing in other topologies, 4) considering the role of SAA-M as a lightweight general purpose rate control algorithm, 5) testing the realizability of calculated rates under finite speedup and arbitration restrictions and 6) investigating schemes that permit "pipelining" of the algorithms: flow continue at their current rates while new rates are being calculated, possibly even in advance of the need for them, similar to prefetching.

Bibliography

[1] http://www.omnetpp.org.

[2] http://www.top500.org.

[3] Y. Afek, Y. Mansour, and Z. Ostfeld. Phantom: a simple and effective flow control scheme. In *Proc. of the ACM SIGCOMM*, pages 169–182, 1996.

[4] D. Bertsekas and R. Gallager. *Data Networks*, pages 448–453. Prentice Hall, 1987.

[5] L. S. Brakmo and L. L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[6] H. J. Chao and J. S. Hong. Design of an atm shaping multiplexer with guaranteed output burstiness. *Computer Systems Science and Engineering*, 12:131–141, 1997.

[7] A. Charny, D. Clark, and R. Jain. Congestion control with explicit rate indication. In *Proc. IEEE International Conference on Communications (ICC)*, pages 1954–1963, 1995.

[8] A. Charny, K. K. Ramakrishnan, and A. Lauck. Time scale analysis scalability issues for explicit rate allocation in atm networks. *IEEE/ACM Trans. Netw.*, 4(4):569–581, 1996.

[9] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, 1989.

[10] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987.

[11] B. V. Dao, S. Yalamanchili, and J. Duato. Architectural support for reducing communication overhead in multiprocessor interconnection networks. In *Proc. of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, page 343, 1997.

[12] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, 2006.

[13] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multicommodity flow problems. In *Proc. of the 16th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 184–193, 1975.

[14] S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, 1999.

[15] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, and J. Duato. Deterministic versus adaptive routing in fat-trees. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 26–30 March 2007.

[16] IEEE Computer Society. IEEE Std 802.3^{TM}-2005, December 2005.

[17] InfiniBand^{TM} Trade Association. Infiniband^{TM} architecture specification, release 1.2, October 2004.

[18] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and A survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.

[19] R. Jain, S. Kalyanaraman, and R. Viswanathan. The OSU scheme for congestion avoidance in ATM networks using explicit rate indication. In *Proc. WATM'95 First Workshop on ATM Traffic Management*, 1995.

[20] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. An efficient rate allocation algorithm for ATM networks providing max-min fairness. In *HPN*, pages 143–154, 1995.

[21] S. Kalyanaraman, R. Jain, S. Fahmy, R. Goyal, and B. Vandalore. The erica switch algorithm for abr traffic management in atm networks. *IEEE/ACM Trans. Netw.*, 8(1):87–98, 2000.

[22] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proc. of the ACM SIGCOMM*, pages 89–102, 2002.

[23] J. Kim, W. J. Dally, and D. Abts. Adaptive routing in high-radix clos network. In *Proc. of the 2006 ACM/IEEE conference on Supercomputing (SC)*, page 92, 2006.

[24] C. E. Leiserson. Fat-trees: Universl networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, c-34(10), Oct. 1985.

[25] X.-Y. Lin, Y.-C. Chung, and T.-Y. Huang. A multiple lid routing scheme for fat-tree-based infiniband networks. In *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[26] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[27] J. Liu, A. Vishnu, and D. K. Panda. Building multirail infiniband clusters: Mpi-level design and performance evaluation. In *Proc. of the 2004 ACM/IEEE conference on Supercomputing (SC)*, pages 33–33, 2004.

[28] L. Mamatas, T. Harks, and V. Tsaoussidis. Approaches to congestion control in packet networks. *Journal of Internet Engineering*, 1(1):22–33, 2007.

[29] J. C. Martínez, J. Flich, A. Robles, P. López, and J. Duato. Supporting fully adaptive routing in infiniband networks. In *Proc. of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, page 44.1, 2003.

[30] A. Mejia, J. Flich, J. Duato, S. A. Reinemo, and T. Skeie. Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In *Proc. 20th International Parallel and Distributed Processing Symposium IPDPS 2006*, page 10pp., 25–29 April 2006.

[31] F. Petrini and M. Vanneschi. k-ary, n-trees: High performance networks for massively parallel architectures. In *Proc. of the 11th International Parallel Processing Symposium (IPPS)*, pages 87–, 1997.

[32] G. Pfister, M. Gusat, and D. Craddock. Solving hot spot contention using infiniband architecture congestion control. In *Proc. High Performance Interconnects for Distributed Computing Workshop*, 2005.

[33] G. Pfister and V. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, 1985.

[34] J. Rexford, F. Bonomi, A. Greenberg, and A. Wong. A scalable architecture for fair leaky-bucket shaping. In *Proc. of IEEE INFOCOM*, pages 1056–1064, 1997.

[35] J. R. Santos, Y. Turner, and G. J. Janakiraman. End-to-end congestion control for infiniband. In *Proc. of IEEE INFOCOM*, volume 2, pages 1123–1133, 2003.

[36] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29:23–35, 1983.

[37] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (lash) routing in irregular system area networks. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 162–169, 2002.

[38] D. Stiliadis and A. Varma. A general methodology for designing efficient traffic scheduling and shaping algorithms. In *Proc. IEEE INFOCOM*, pages 326–335, 1997.

[39] J. Turner. New directions in communications (or which way to the information age?). 24(10):8–15, Oct 1986.

[40] Y. Turner and Y. Tamir. Connection-based adaptive routing using dynamic virtual circuits. In *Proc. International Conference on Parallel and Distributed Computing and Systems*, pages 379–384, 1998.

[41] H. Tzeng and K. Siu. Adaptive proportional rate control for abr service in atm networks. In *Proc. IEEE Computers and Communications*, pages 529–535, 1995.

[42] A. Vishnu, M. Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda. Hot-spot avoidance with multi-pathing over infiniband: An mpi perspective. In *Proc. Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 479–486, 2007.

[43] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One more bit is enough. *SIGCOMM Comput. Commun. Rev.*, 35(4):37–48, 2005.

[44] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast long-distance networks. In *Proc. IEEE INFOCOM*, volume 4, pages 2514–2524, 2004.

[45] S. Yan, G. Min, and I. Awan. An enhanced congestion control mechanism in infiniband networks for high performance computing systems. In *Proc. 20th Advanced Information Networking and Applications (AINA)*, volume 1, pages 845–850, 2006.