Datacast: A Scalable and Efficient Group Data Delivery Service for Data Centers

Chuanxiong Guo, Guohan Lu, Yongqiang Xiong, Jiaxin Cao,
Yibo Zhu, Chen Chen, Yongguang Zhang
Microsoft Research Asia

ABSTRACT

Reliable group data delivery (RGDD) is a pervasive traffic pattern in data centers. In an RGDD group, a sender needs to reliably deliver a copy of data to all the receivers. Existing solutions either do not scale due to the large number of RGDD groups (e.g., IP multicast) or cannot efficiently use network bandwidth (e.g., end-host overlays).

We design *Datacast* for RGDD. Datacast turns group state management to in-network packet caching by leveraging the content centric networking (CCN) concept. It uses centrally calculated multiple edge-disjoint trees for efficient and fast data delivery. By introducing a robust rate-based congestion control, which interprets duplicated CCN interest packets as congestion signal, Datacast adapts well to varying network conditions.

We have implemented Datacast using the ServerSwitch [13] platform. Our implementation shows that all the features needed in Datacast, including flexible packet filtering, source routing, and in-network packet caching, can be implemented using commodity devices. Our implementation can offload packet forwarding to switching chip for 50% nodes in a typical group with 280 members and over 500 intermediate nodes. Our testbed experiments demonstrate that Datacast achieves 1.74Gb/s goodput using two 1GbE Steiner trees, it uses network bandwidth efficiently, adapts to changing traffic conditions and reacts to network failures quickly.

1. INTRODUCTION

Driven by technology advances and economic forces, data centers are being built around the world to provide various cloud computing services. These data centers may contain hundreds of thousands servers. The servers together with the network devices, e.g., Ethernet switches and IP routers, form huge networking systems to support various infrastructure services (e.g., GFS and MapReduce) and applications (e.g., social networking, Search, scientific computing).

Reliable group data delivery (RGDD) is a pervasive traffic pattern in these applications and services. In RGDD, we have a group which contains one data source and a set of receivers. We need to reliably deliver the same copy of data from the source to all the receivers. We describe several typical RGDD cases as follows.

Case 1: In data centers, servers are typically organized as physical clusters. During bootstrapping or OS upgrading, the same copy of OS image needs to be transferred to all the servers in the same cluster. A physical cluster is further divided into sub-clusters of different sizes. A sub-cluster then is allocated to a service. All the servers in the same sub-cluster may need to run the same set of applications. We need to distribute the same set of program binaries and configuration data to all the servers in the sub-cluster.

Case 2: In distributed file systems such as GFS [23], a chunk of data is replicated to several (typically three) servers to improve reliability. The senders and receivers form a small replication group. A distributed file system may contain tens of Peta bytes. Hence the number of replication groups is huge. In distributed execution engine (e.g., Dryad [25]), a copy of data may need to be distributed to many servers for Join operations.

Case 3: In Amazon EC2 or Windows AZure, a tenant may create a set of virtual machines. These virtual machines form an isolated computing environment dedicated to that tenant. When setting up the virtual machines, customized virtual machine OS and application images need be delivered to all the physical servers that host these virtual machines.

There are several common characters in the above RGDD cases. First, reliable data delivery is mandatory and all the receivers need to receive the exact copy of data. Second, though the group may be created statically (cases 1) or dynamically (cases 2 and 3), the group membership is static. Due to the lack of support for RGDD in the network, many current systems support RGDD at application layer using unicast TCP. This solution is undesirable due to its inefficiency and the fact that many applications need to duplicate the same effort.

One solution category for RGDD is reliable IP multicast.The design space of reliable IP multicast has been nicely described in [16]. IP multicast has scalability issues for maintaining a large number of group state in the network. And adding reliability to IP multicast is hard

due to the well known ACK implosion problem [17]. Furthermore, IP multicast uses a single multicast tree. Many data center networks (DCN, e.g., BCube [11] and CamCube [14]) have multiple edge-disjoint trees which cannot be utilized by IP multicast.

Another solution category is end-host based overlays (e.g. [4, 3, 5]). Overlays are scalable, since devices in the network do not maintain group state. Reliability is easily achieved by directly using TCP. Overlays, however, do not use network bandwidth efficiently. Both link stress and network stress can be very high. For example, the worst-case link stress of SplitStream can be tens [3], and the average and worst-case network stresses of ESM [4] are 1.9 and 9, respectively. See 8 for more details.

In this paper, we propose a new design called *Datacast*, which achieves both scalability and efficiency. To achieve scalability, Datacast turns one-to-many group communication to *in-network* packet caching by leveraging the content centric networking (CCN) [19] concept. In order to receive a data packet, a receiver needs to issue a pulling *interest* packet. The returned data packet is then cached along the return path. Later interest packets asking for the same data packet can be served by the cached packets. By so doing, no group hard state is maintained in intermediate network devices. To achieve efficiency, Datacast calculates data delivery trees for RGDD groups in a centralized way, and uses source routing to enforce routing path. It further uses multiple edge-disjoint trees for speedup when these trees are available. We further design a Datacast transport protocol (DTP) for reliable data delivery. DTP splits traffic among multiple trees, and uses a simple yet robust algorithm for rate-based congestion control. By interpreting duplicate interest packets as congestion signal, DTP reacts to varying network conditions quickly.

In-network packet caching previously can only be implemented using software routers [32, 19]. Datacast makes a new contribution by implementing in-network caching with the fully commodity ServerSwitch [13] platform. ServerSwitch integrates a commodity ASIC switching chip and a multi-core server system using a low latency, high throughput PCI-E interface (we note similar trend in recent products, e.g., [20, 2]). Using ServerSwitch, we can program the switching chip to perform hardware-based source routing, and to filter only specific Datacast packets to CPU for further processing and caching. Compared with pure software-based implementation, our ServerSwitch-based implementation can offload packet forwarding to hardware for 50% nodes in data deliver trees for a typical Datacast group with 280 members and 500+ intermediate nodes. Our experiments further demonstrated that Datacast achieves 1.74Gb/s goodput using two 1GbE trees. It efficiently

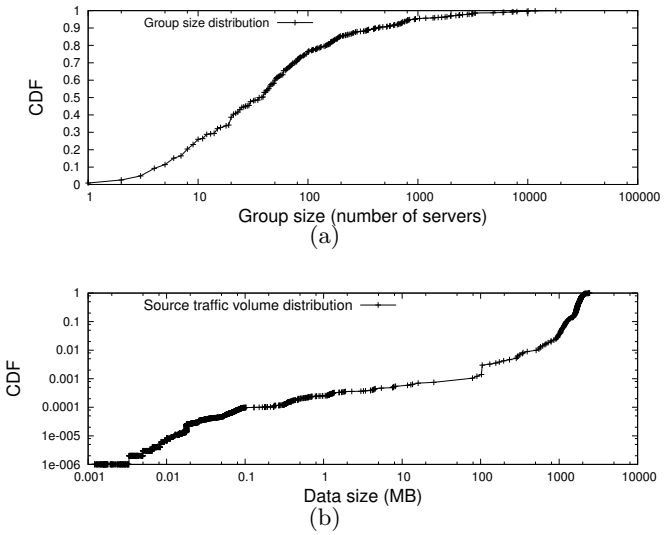


Figure 1: (a) The group size distribution in a large data center. (b) The source traffic volume distribution for a large distributed execution engine. Note the y-axis of (b) uses logarithmic scale.

uses network bandwidth, handles network failures and adapts to changing network conditions, as designed.

The rest of the paper is organized as follows. We discuss design goals in Section 2, and present Datacast architecture in Section 3. Section 4 describes how to calculate multiple edge-disjoint trees. Section 5 designs DTP. Sections 6 and 7 presents simulation, implementation, and experimental results. Section 8 discusses related work and Section 9 concludes.

2. DESIGN GOALS

In this section, we discuss Datacast design goals and the associated research challenges.

Scalable group state management. State management in one-to-many communication is a well known hard problem. It is even challenging for DCN for due to the following reasons:

First, Datacast needs to support a large number of groups. In Windows AZure or Amazon EC2, groups are created on demand for different tenants. A large data center needs to support hundreds of thousands or even more tenants. For distributed file system, a chunk of data may form its own Datacast group. For a distributed file system with hundreds PB storage, which is not uncommon now, we may have millions of small data delivery groups. These groups introduce huge amount of group state, which are hard to manage and maintain in the network.

Second, Datacast needs to support groups of various group sizes. The size of a Datacast group can vary from several servers to tens of thousands. Fig. 1(a)

shows the group size distribution for a data delivery service in a large production data center. There are many small groups (20% groups with less than 10 servers), and the majority group size is between 10 to 1000 with mean size 280. But there are several extremely large groups with tens of thousands servers. Fig. 1(b) shows the source traffic volume for group communications in a platform similar to MapReduce. Though there are only 8% groups transmitting more than 550MB in size, these groups contribute 99% source traffic volume. Hence in this paper, we focus on groups with large data size.

Efficient and reliable data delivery. Previous studies showed that there seems to exist a tradeoff between efficiency and reliability. IP multicast is very efficient in using network bandwidth. But due to the ACK/NAK implosion problem, achieving reliability in IP multicast is hard. Achieving reliability in end-host overlays is easy since end-hosts directly use TCP. But end-host overlays cannot use network bandwidth efficiently due to their high link stress and the suboptimal transmission topology. In Datacast, we try to achieve efficiency and reliability simultaneously.

Data center network topology is specially designed and many data center network structures [10, 11, 14] contain multiple edge-disjoint trees. It is desirable to accelerate data delivery by using these multiple Steiner trees. But even calculating one single efficient Steiner tree is NP-hard[22]. Furthermore, how to split data among the multiple trees and how to perform congestion control within every single tree are challenging tasks.

Commodity device based implementation. To take advantage of the economics of scale, data center networks use commodity devices and merchant ASIC switching chips. It is commonly believed that these ASICs are hard-coded and can hardly support new network functions. Hence new designs and prototypes are implemented using software routers, which suffer from low performance. In this paper, we show that by leveraging the (limited) programmability of switching ASIC and carefully splitting functions between hardware and software, we can achieve high performance and at the same time use only commodity devices.

3. DATACAST OVERVIEW

3.1 Datacast system

We introduce Datacast architecture and its key design choices in this section. Similar to [23, 6, 12], we assume there is a central controller, called Fabric Manager. Fabric Manager is responsible for maintaining the network topology. It can use an out-band or in-band channel for such purpose. An in-band channel example is the reliable spanning tree built in [12]. As long as the physical network is connected, Fabric Manager can use that reliable spanning tree as the signaling channel to collect the network topology update.

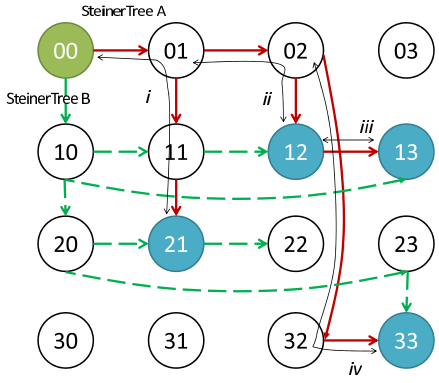


Figure 2: An example to show how Datacast works. The network we use is a two dimensional torus. Server 00 is the source, and 12, 13, 21, 33 are receivers. The bold red links and dashed green links form two edge-disjoint Steiner trees.

In Datacast, *Masters* are responsible for creating Datacast groups. They can be distributed file systems master, or MapReduce/Dryad job scheduler, or Fabric Manager itself. All the masters know the up-to-date network topology from Fabric Manager. When creating a Datacast group, a master calculates the data delivery trees for the group. Since the algorithm for tree calculation has low time complexity (as we will show in Sections 4), a single master can perform the job.

Datacast groups are created by the masters dynamically, though there may exist static groups such as the one-to-all broadcast group. Once a group is created, the group membership is static. This group model is based on scenarios we have described in Section 1. The master then distributes the data deliver trees together with the set of files to be fetched from the master to the source and all the receivers. When the group size is small, the master can use unicast to deliver the information to all the receivers. When the group size is large, the master can reliably broadcast the information across the whole network using the signaling channel.

Datacast is designed as a service. End users simply run a command *datacast* with a list of parameters to create a Datacast group at a master node. The parameters include the set of files to be delivered, the source, the set of the receivers, and the location to hold the received files, etc. Once a receiver receives the Datacast group related information from the master, it begins to send request packets to the source server and data transfer begins. In the next subsection, we use an example to illustrate how data transfer works in Datacast. We then explain the design choices after the example.

3.2 An example

We use an example to illustrate how data packets are

delivered in Datacast. Fig. 2 shows a two-dimensional torus network with 16 servers. Suppose server 00 is the source and servers 12, 13, 21, 33 are the receivers. The source and receivers form a Datacast group. The bold red lines show a *Steiner tree* with 00 as the root. Servers {01, 02, 11, 32} are the *Steiner nodes*. Steiner nodes are not Datacast group members, but they are needed to relay data for the receivers. In step i, server 21 sends a request packet (or using the CCN terminology, an interest packet), to the source server 00 through the path {21, 11, 01, 00}. Server 00 sends back the requested data packet along the reversed path. The data packet is cached at server 01. In step ii, server 12 also sends an interest packet along path {12, 02, 01, 00} asking for the same data packet. When the interest packet arrives at server 01, server 01 finds that it has already cached the packet, so it terminates the interest packet and sends back the requested data packet. The data packet is cached in 02 and 12. In step iii, server 13 sends its interest packet along path {13, 12, 02, 01, 00}. Since 12 already has the data, it sends back the cached packet to 13. In step iv, server 33 sends its interest packet along path {33, 32, 02, 01, 00}, and server 02 returns the data packet.

We note that the execution order of steps i, ii, iii, and iv is not important. They can be executed in arbitrary order while still achieving the same result. This is because, in the end, all the steps together cover the same Steiner tree by traversing every link of the tree exactly once.

In this example, servers 01 and 02 are called branching Steiner nodes since they have multiple children in the tree. Servers 11 and 32 are non-branching Steiner nodes which have only one child. For brevity, we use (non-)branching node for (non-)branching Steiner node in the rest of the paper. In Datacast, only branching nodes and receivers cache data and non-branching nodes do not cache data. This helps the whole system save memory, and as we will show later, helps the system to offload packet forwarding of non-branching nodes to hardware.

Fig. 2 also shows a dashed green Steiner tree, which has the same source server (00), and the same receivers (12, 13, 21, 33). The two Steiner trees are edge-disjoint in that no edge in one tree appears in the other tree. Hence we can simultaneously use the two edge-disjoint trees to speedup data delivery.

The example gives a high-level overview on how data transfer works in Datacast. Next, we discuss various design choices we made in Datacast.

3.3 Datacast design overview

Multiple trees. Traditional IP multicast can only use one tree. Data center networks, however, provide high network capacity by introducing advanced network structures including fat-tree [1, 9], BCube [11], DCell [10], and torus [14]. Many of these structures [11, 10, 14] inherently have multiple trees. Hence, if we use only one tree for data delivery, we cannot fully utilize the capacity of the network. But how to generate edge-disjoint trees is not easy. As we have shown in Section 2, Datacast groups have very different sizes. Hence, in order to use multiple trees, we need to address the following two problems: how to generate multiple edge-disjoint trees for groups with arbitrary sizes, and how to split data among these multiple trees, which may have very different available bandwidths. We will address these two problems in Section 4 and Section 5.

In-network packet caching. In Datacast, data packets are cached in intermediate devices when they traverse the network. When a receiver asks for the same piece of data packet, the device that has the cached data can serve the request. Datacast therefore turns group communication to in-network packet caching by adopting the content centric networking (CCN) [19] concept. The direct benefit of in-network caching is that the network devices do not need to maintain hard group state. The intermediate devices do not even know that they are part of a group communication session. Hence Datacast is inherently scalable. When a cached packet gets dropped, e.g., due to cache replacement, Datacast can still work (but less efficient). Back to the example we use in Fig. 2, when server 13 sends an interest packet along path {13, 12, 02, 01, 00}, even if the cached packet was dropped in the caching servers 12 and 02, server 13 can still get the packet from server 01. In the worst case, server 13 can get the packet from the source server 00.

In Datacast, we easily achieve reliability by using a receiver-driven pull model based on CCN. A receiver always sends an interest packet to ask for a data packet. If the receiver does not receive the requested data packet in a time interval, it simply re-sends the interest packet. The first node that has the packet along the path will serve the re-sent interest packet. Hence the ACK/NAK implosion problem does not exist in Datacast.

Due to its soft-state nature, Datacast works well when only part of devices support in-network caching and can be deployed incrementally.

Source routing. As we have shown in Fig. 2, a receiver needs to send an interest packet from itself to the source along a specific path in the spanning tree. For example, when server 12 sends an interest packet to the source server 00 in the red spanning tree, the interest packet should take path {12, 02, 01, 00}. When source 00 sends back a data packet to server 12, the data packet needs to follow the reverse path {00, 01, 02, 12}.

One way to pin routing path is to use virtual circuit technologies, e.g., MPLS (multi-protocol label switching). But these approaches need a signaling protocol (e.g., LDP, label distribution protocol for MPLS)

to setup and release the virtual circuits, and all the switches need to maintain hard state. Hence they face similar scalability issues as IP multicast.

In Datacast, we use source routing to address the problem. With source routing, the routing path is encoded into packets, and all the intermediate nodes along the path do not need to maintain routing state. The cost is that we need to carry the whole routing path in every packet. Since the depths of the trees are small values (less than 10 hops), it is an affordable overhead. We further demonstrate that source routing can be implemented using current commodity network devices (Section 6).

Branching and non-branching nodes. As shown in Fig. 2, nodes 01 and 02 are branching nodes and nodes 11 and 32 are non-branching ones in Steiner tree A. In Datacast, non-branching nodes do not cache data packets, since no receivers will ask for the data from them. For Datacast groups that have many non-branching nodes (which is the case as we will show in Section 4.3), our design saves caching memory. Furthermore, we will show in Section 6 that we can offload packet forwarding of non-branching nodes to hardware by implementing Datacast using the ServerSwitch [13] platform, hence save both CPU cycles, memory, and I/O bandwidth.

Rate-based congestion control. Within a single Datacast tree, the sending rate of the source needs to adapt to the receiving rate of the slowest receiver. The source needs to slow down when congestions happen, and speedup when no congestions. But both the slowest receiver and its receiving rate may change due to varying network condition. Previous designs such as pgmcc [28] need to track and select the slowest receiver, which increases the complexity of the design.

In Datacast, we find a natural congestion signal: duplicate interest packets for the same data packet. Receiving a duplicate interest at the source indicates that the original data packet has disappeared, and the current sending rate is larger than the receiving rate of the receiver that sends the duplicate interest. In this case, the source should slow down so the slowest receiver can catch up. When there is no duplicate interests received during a period of time, the source infers that there is no congestion, and it should increase the sending rate. Datacast therefore naturally introduces a simple rate-based congestion control. See Section 5.1 for the details.

In the subsequent sections, we will present technical details on multiple edge-disjoint Steiner trees calculation, and the design of our reliable Datacast transport protocol (DTP), which splits data among multiple Steiner trees, and performs congestion control for every single tree.

4. MULTIPLE EDGE-DISJOINT STEINER TREES

4.1 The problem

The problem is, given a network $G(V, E)$, where V is the set of nodes and E is the set of edges, and a Datacast group D, in which D has one source src and a set of receivers $\{r_0, r_1, \cdots, r_{m-1}\}$, we need to get k edge-disjoint trees for D, where k is the maximum number of edge-disjoint trees. The sum of the cost of the trees should be minimized. For simplicity, the cost of a tree is its number of links. This is the well known multiple edge-disjoint Steiner trees problem. Even calculating a single min-cost Steiner tree in a general graph is NP-hard [22]. In fact, calculating Steiner tree for specific networks e.g., hypercube [24], BCube, and multi-dimensional torus is still NP-hard.

We therefore turn our attention to find efficient heuristics. There are efficient algorithms for calculating multiple edge-disjoint spanning trees. Specifically, Edmonds [8] showed that there exist k' edge-disjoint spanning trees rooted at src, where k' is the minimum number of edges which can be deleted from G in order to make at least one node unreachable from src. We can first find the k' edge-disjoint spanning trees, then prune the unneeded edges and nodes to get the Steiner trees.

But the generic multiple spanning tree algorithms do not work well for us. First, the time complexity for calculating the spanning trees is high. The best algorithm we know is Po's algorithm [30]. Its time complexity is $O((k')^2|V||E|)$. We have run Po's algorithm on a powerful server. The time for calculating the 4 spanning trees for a BCube network ($n = 8$ and $k = 4$) with 4096 servers is 40+ seconds. With 1Gb/s network, we can transmit more than 4GB data during that period of time. Hence we cannot afford high complexity for Steiner tree calculation. Second, the depths of the spanning trees generated by the generic algorithm can be very large. For example, in the previous example for BCube, the average and worst-case depths of the trees can be 1000+ and 2000+ hops, whereas the network diameter is only 8. Large depth increases network latency and the number of Steiner nodes, hence wastes network bandwidth. We note that these are not the faults of these algorithms: They simply were not designed for our purpose.

Fortunately, data center networks, e.g., fat-tree [1], BCube [11], multi-dimensional torus [14], are well structured, and we know how to design fast algorithms for calculating edge-disjoint spanning trees for these structures. These algorithms are not only fast, but also create spanning trees with optimal depth. A disadvantage of using structure specific algorithms is of course these algorithms work only for specific structures. In practice, we believe it is not an issue since data center network structures are well studied. Next, we describe our algorithm for multiple edge-disjoint Steiner trees calculation.

```
/**
 * G is the DCN network, D is the datacast group.
 */

CalcSteinerTrees(G, D):
    SPTSet =G.CalcSpanningTrees(D.src);
    foreach (SPT_i in SPTSet)
        SteinerTree_i = Prune(SPT_i, D);
        SteinerTreeSet.add(SteinerTree_i);
repairing:
    foreach (SteinerTree_i in SteinerTreeSet)
        if (SteinerTree_i has broken links)
            if(Repair(SteinerTree_i, G)==false);
                Release(SteinerTree_i);
    return SteinerTreeSet;
```

Figure 3: The algorithm for multiple Steiner trees calculation.

4.2 The algorithm

Fig. 3 shows the algorithm. We first use network specific algorithm to calculate a set of edge-disjoint spanning trees. We then prune each spanning tree to get a set of Steiner trees. For each Steiner tree, we see if it is affected by network failures, i.e., broken links. We try to repair the Steiner tree if it is affected. If we fail to repair the tree, we release it. One key step in Fig. 3 is to calculate the set of edge-disjoint spanning trees. See Appendix A. for the detailed algorithms for calculating the spanning trees for fat-tree, BCube, and torus. Fig. 4 shows a BCube network and its spanning tree.

Steiner tree repairing. In order to minimize interruption and maximize the usage of the already cached data, it is desirable that the repairing algorithm is of low time complexity, and the repaired tree is recovered from the original one with low depth. We can achieve the above requirements with simple heuristic: When a Steiner tree is broken, instead of trying to replace the broken links, we first *release* the whole tree. We then try to construct a new Steiner tree from the updated residual network using breadth first search (BFS). When we perform BFS, we give search priority to the links that are in the original tree. This trick maximizes the similarity of the repaired and original trees.

If the repairing succeeds, we know the tree depth is optimal due to the use of BFS. If it fails, we know that repairing is not possible and we go on to repair the next broken tree. It is easy to see that the repairing heuristic can at least guarantee one tree as long as the source and the receivers are connected.

Note that in the algorithm, for both spanning tree calculation and Steiner tree repairing, we do not explicitly try to reduce the number of Steiner nodes. Instead, we minimize tree depth. Since if the tree depth is small, the number of Steiner nodes should be small as well.

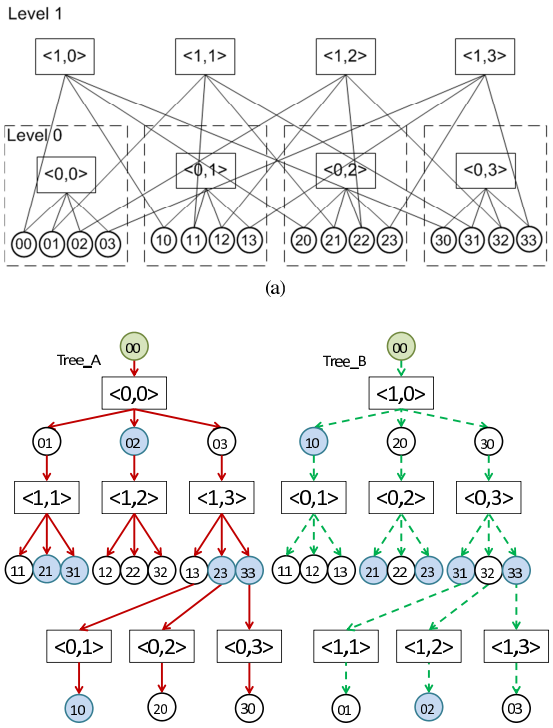


Figure 4: (a) A BCube(4,2) network with 16 servers and two layers of switches. Each layer has 4 switches. (b) The two spanning trees of the BCube(4,2) network with server 00 as the root. In the two spanning trees, we show a Datacast group with 00 as the source and servers {02, 10, 21, 23, 31, 33} as the receivers.

The repairing time is the time to run BFS. Suppose we have k' Steiner trees to repair, the worst-case repairing time is $O(k'|E|)$, where $O(|E|)$ is the time to run BFS once.

4.3 Performance

We use simulations to study the performance of the algorithm. All the details can be found in Appendix A.4. We summarize the results as follows.

Time complexity. The running time of the algorithm is small. For BCube(8,3) (with 4096 servers), torus(16,3) (with 4096 servers), and fat-tree(48,3) (with 27k servers), when the group size is 280 and the link failure rate is 5%, the worst-case running times are only 4.4ms, 4.1ms, and 3.3ms, respectively. The running time is less than 20ms when the group size is thousands or more.

Steiner tree quality. Simulation results show that the algorithm can always find the maximum number of Steiner trees. We also have studied the number of Steiner nodes in the generated Steiner trees. Our simulations show that the number of Steiner nodes generated from our algorithm is very close to that of Breadth

First Search (BFS). But Our algorithm generates multiple Steiner trees, whereas BFS can only generate one tree when group size is larger than 30.

Branching versus non-branching nodes. We define non-branching ratio for a Datacast group as the number of non-branching nodes divided by the number of Steiner nodes. The simulation shows that when the group size is small to average, the number of non-branching nodes contributes a significant portion of the total Steiner nodes. For example, for a BCube(8,4) with 1% link failure rate, the non-branching ratio is 75% (or 407 non-branching nodes over 543 Steiner nodes) for group size 280. (Note that 280 is the average group size we got from real data centers, Fig. 1(a)). In the above example, 50% of the total nodes are non-branching ones (or 407 out of 823). In Section 6, we will show that by using ServerSwitch, we can offload packet forwarding for non-branching nodes to hardware. This is a significant optimization due to the large number of non-branching nodes.

5. DATACAST TRANSPORT PROTOCOL

We design a Datacast transport protocol (DTP). The protocol provides reliable data delivery, handles congestion control for data transmission in one single Steiner tree, and distributes data among multiple, edge-disjoint Steiner trees. In Datacast, reliability can be achieved easily and efficiently, due to its pull-based design and in-network packet caching. A receiver always sends an interest packet asking for a data packet. When the requested data packet is not received in a time interval, the interest is re-sent. Since data packets are cached along the intermediate nodes in the tree, a re-sent interest packet can be served in the nearest intermediate node. The source may not even see the re-sent interest packet. In what follows, we address the rest two issues sequentially.

5.1 Congestion control for single tree

For every single tree in a Datacast group, every receiver is given w credits, which means a receiver can send at most w interests without getting back a data packet. w is setup when the master creates the Datacast group. When a receiver sends out an interest packet, its credit is decremented by one; when it receives a data packet, its credit is incremented by one. When there is no data packet received during a period of time RTO, the receiver considers that either the interest or the data packet is lost, it then regenerates one credit and re-sends the interest packet. Suppose the data packet size is pkt_size bytes and the round trip time is rtt, the maximum achievable data delivery rate for a single tree is then $r_{\max} = \frac{w \times pkt\_size}{rtt}$. The round trip time in data center is small, typically several hundred microseconds. Hence small w can generate high throughput. When $rtt = 300$ us, $w = 16$, and $pkt\_size = 1.5$ kB, the maximum data delivery rate is 640Mb/s. When 9kB Jambo frame is used, for the same w and rtt, we can achieve 3.84Gb/s. Hence with w and pkt_size, we can control the maximum bandwidth assigned to a Datacast group.

The source performs congestion control for every tree independently. Datacast uses rate-based congestion control, and it uses an AIMD sending rate control scheme. For each tree, the source maintains a token bucket for rate control. The token bucket has two parameters, a rate r and a depth σ. σ controls the burstiness and r controls sending rate. In the beginning, r is set to a rate that maps well to w.

The source considers that there is congestion when the following conditions are all met: it receives a duplicate interest for a same data packet; the duplicate interest is from the same incoming interface of the original interest packet; the sequence gap dup_δ between this duplicate interest and the previous duplicate interest is at least w ($dup\_\delta \geq w$). When the source detects a congestion, it reduces its sending rate by half. The sequence gap dup_δ is to guarantee that there is only one rate reduction during one round trip time.

When the source does not receive duplicate interest packets in a small time interval T, it increases its sending rate. In our design, the source increases the rate by a constant value δ. The sending rate adjustment algorithm is therefore as follows.

$$r = \begin{cases} \frac{r}{2}, & \text{when a congestion signal is detected;} \\ r + \delta, & \text{when no congestion signal in T.} \end{cases}$$

We have used extensive simulations to decide the parameters T, δ, RTO, and the cache size. See B for more details. We use T=1ms, δ=50Mb/s, RTO=20ms in the paper. We will show the performance of the congestion control protocol using real implementation in Section 7.

5.2 Data distribution among multiple trees

Suppose we have k edge-disjoint Steiner trees $\{T_0, T_1, \cdots, T_{k-1}\}$, the data we need to distribute has L bytes. We divide the data into a set of blocks each of size B. Hence the number of chunks is $n = \lceil \frac{L}{B} \rceil$. We number the blocks $\{B_0, B_1, \cdots, B_{n-1}\}$. The source uses the algorithm in Fig. 5 to distribute data blocks to multiple Steiner trees.

Though Fig. 5 looks deceptively simple, there is several tricky issues to be handled carefully: how to decide if a tree becomes idle and how to decide the size of B.

One may consider that receivers should decide if a tree is idle by themselves. For example, when a receiver finds that it has received all the packets of a block using T_i, it can use T_i for the next data block. This approach cannot work well for our multiple Steiner trees case. Suppose we have two trees T_1 and T_2, and two receivers R_a and R_b. In the beginning, the receivers use T_1 for B_1 and T_2 for B_2. It is possible that R_a first finishes

```
Data splitting algorithm:
    for (i = 0; i < n; i++)
        BlockList.append(B_i);
    while(BlockList not empty)
        if(T_i becomes idle)
            B_j = BlockList.front();
            use T_i to distribute B_j;
            piggyback the decision info to all receivers;
            BlockList.pop_front();
```

Figure 5: The algorithm for the source to distribute data blocks among multiple trees.

receiving B_1 and uses T_1 for the next block B_3, and R_b first finishes receiving B_2 and uses T_2 for B_3. So R_a uses T_1 for B_3 and T_2 for B_4, and R_b uses T_2 for B_3 and T_1 for B_4. In more general case, receivers get de-synchronized. Datacast cannot work well in de-synchronized situation since cached packets in the intermediate nodes cannot be fully utilized by the rest receivers.

In order to make synchronized decision, one may suggest that receivers report back to the source when they finish receiving the previous block. But this may cause report packets implosion when the group size is large. Further, how to provide reliable report is difficult.

To address this issue, in Datacast, source is the one to make decision. It considers a tree becomes idle after the last packet of the previous block has been sent. The next issue is how to send the decision to all the receivers. In our design, this piece of information is *piggybacked* in the last packet of the previous block. Hence, all the receivers can receive the source's decision reliably, and no additional notification mechanism is needed.

The block size B needs careful consideration too. When the previous block finishes transmitting, the receivers need to wait for the piggybacked decision information. To mitigate the effect of this stop-and-wait, we should use a reasonably large block size. But it cannot be too large, or we may not have enough number of blocks to fully utilize all the Steiner trees. In this paper, we set B = 100MB. Data sources with size less than 100MB therefore can use only one tree. In practice, this is not an issue, since data sources with size larger than 550MB contribute to more than 99% group communication traffic as we have shown in Fig. 1(b).

6. SERVERSWITCH BASED IMPLEMENTATION

We have implemented both Datacast control and data plane functions. In its control plane, we have prototyped Fabric Manager and Master. We rely on standard replicated state machine (RSM) to provide the needed reliability for these control plane functions. We focus on data plane in this paper.

We use our ServerSwitch [13] platform to implement Datacast. ServerSwitch hardware is composed of an ASIC switching chip and a commodity server. The switching chip is connected to the server CPU and memory using PCI-E. ServerSwitch software includes a set of APIs to program the switching chip and a set of kernel modules for controlling the switching chip and processing data and control packets. ServerSwitch is built from all commodity components. See [13] for details.

ServerSwitch is a desirable platform for implementing Datacast for two reasons. First, with programmable switching chip, we filter only Datacast packets at branching node to CPU for processing and caching. For all the rest packets, we can offload packet forwarding to hardware. Since data centers have non-Datacast traffic, and, as we have shown in 4.3, Datacast groups have many non-branching nodes, our implementation saves CPU cycles and memory for these nodes as compared with pure software based approaches. Second, the large volume server memory and high computing ability of server CPU can be used to implement packet caching and processing.

In Datacast, we have interest and data packets. The two packet types share similar packet format. Every packet starts with a fixed length (8 bytes) source routing path, followed by the name of the packet. We use the BCube source routing packet format to implement source routing (see Fig.5 of [13]). We use one byte to identify one next hop, and further use the most significant bit in that byte to denote if the corresponding node is a non-branching node or not. We introduce two new "BCube protocol" values to indicate if a packet is a Datacast interest or data packet. The name part has one byte flag (which is used to indicate if a data packet is a retransmit packet) and one byte to indicate the name length, followed by the name. Next, we describe our ServerSwitch based Datacast implementation.

6.1 Key components

Our ServerSwitch based implementation has three components: a Datacast daemon at user space, a kernel driver module at OS kernel, and a ServerSwitch hardware. Next, we describe these components one by one.

Datacast daemon. In every server, we run a Datacast daemon at user space. The daemon uses a Datacast agent to communicate with its masters for Datacast configuration information. The configuration information includes the source and receivers, the Steiner trees, the set of files to distribute, the credit for the receivers, etc. The daemon has a sending module and a receiving module. The sending and receiving modules are for the server to act as source and receiver, respectively.

In Datacast, a receiver needs to send an interest packet to explicitly ask for a data packet. The pull based design makes it easy to achieve reliable data delivery in Datacast. Every receiver can send out a number of interest

packets constrained by its credits. After that, it has to wait for data packets. For one received data packet, one more interest packet can be generated. After sending an interest packet without receiving the asked data packet in a period of time (default to 20ms), the receiver will re-send the interest packet.

The sending module is responsible for sending out data packets when it receives interest packets from the kernel. When acting as the data source, the source runs the DTP protocol. The sending module implements the data splitting algorithm in Fig. 5 at user space, and the Datacast kernel driver implements the rate-based congestion control for each single tree at kernel space.

Datacast kernel driver. The Datacast kernel driver maintains two key data structures. The first is a pending interest table (PIT) which maintains the name of the requested data packet and the reversed paths from which the interest packets are received. The second is a content store (CStore), which stores all the cached data packets. The data structure for both PIT and CStore is a trie (or prefix tree) due to the hierarchical nature of packet naming. In CStore, we use LRU (least recently used) as the cache replacement algorithm. We maintain two separate caches, one for retransmit packets and one for original packets. The retransmit cache size is 10% of the total cache size. Later we will use experiment to show the interaction between cache size and the congestion control algorithm.

When acting as source, the kernel implements the rate-based congestion control. For every data delivery tree, the kernel maintains a token bucket. When a duplicated interest is received, the rate of the token bucket is halved. When no duplicated interest is received in interval T (default to 1ms), the rate is increased by δ. The two components of DTP: data splitting among multiple trees and rate-based congestion control for single tree are therefore implemented in Datacast daemon and kernel driver, respectively.

ServerSwitch hardware. The key data structure in the hardware is a TCAM table in the switching chip. By using the user defined lookup keys (UDLK), we can configure the programmable parser of the switching chip only filter Datacast data packets at branching node for caching and processing. We use the TCAM table to implement our source routing. As we have described, every packet includes a source routing path in its header. There is a next-hop-index to indicate the location of the next hop. Hence normal forwarding procedure is a two-steps approach: we first get the value of the index, then use that value to get the next-hop value. Using ServerSwitch, we can pre-build the TCAM table, and use the whole source routing path together with the next-hop-index as the key, and mark the index and its pointed location as the care bits. By so doing, we can simplify the two-steps approach into one table lookup operation. The tradeoff is we now need a pre-built the TCAM lookup table. In our implementation, the source routing path length is 8 hops and each hop needs 1 bytes. Hence we need 2048 entries in TCAM to hold the lookup table, which is easy to put into current merchandize switch chips.

6.2 Packet processing

Packet sending. After receiving an interest packet, the sending module of the source sends the corresponding data packet to the Datacast driver. The driver caches the packet in CStore and delivers the packet to the ServerSwitch hardware after the packet goes through the token bucket. The switching chip then delivers the data packet using the next hop contained in the packet header. Similarly when the receiving module generates an interest packet, it also sends the packet to the Datacast driver. The driver then updates the PIT table and delivers the packet to the ServerSwitch hardware, which in turn forwards the packet to its next hop.

Packet receiving and forwarding. When an interest packet is received by the ServerSwitch switching chip, the switching chip checks if the current node is the receiver or branching node. If it is not, the switching chip directly forwards the packet. Otherwise, the interest packet is forwarded to server CPU. When the Datacast kernel receives the interest packet, if the node is the destination of the packet, the interest is delivered to the sending module. Otherwise, the kernel uses the name of the interest packet to lookup its CStore. If there is a matching, the cached data packet is sent back to the receiver, and the interest packet is released. If there is no matching in CStore, the kernel then looks up its PIT. If there is a matching, the kernel updates the entry by adding the reverse routing path of the interest packet and terminates the packet. If there is no matching, the kernel creates a new entry for the interest packet and forwards the interest packet to the next hop.

When a data packet is received by the ServerSwitch switching chip, similarly, it uses the next hop information carried in the source routing path to decide its action. If the current node is a non-branching node, the switching chip directly forwards the data packet to the next hop. Otherwise, the data packet is forwarded to Datacast kernel driver. The driver then uses the name of the data packet to lookup the PIT table. If there is a matching, the data packet will be forwarded to all the receivers recorded in the entry. We note that the receivers may include the current node itself. If the node itself is the destination, the data packet is also delivered to the receiving module. For every received data packet, CStore is updated accordingly. We use LRU for cache management and we differentiate if a data packet is a retransmit packet or not. When the receiving module of Datacast daemon receives the data packet, it deletes the

Cache size	Total		Tree_A		Tree_B	
(#pkt)	rate(Mb/s)	dup(%)	rate	dup	rate	dup
128	1093	1.48	197	2.97	870	0.0002
512	1091	1.16	196	2.33	869	0.0004
2048	1100	1.18	196	2.36	877	0.0000
8192	1095	1.23	196	2.45	873	0.0000
16384	1084	1.15	196	2.30	862	0.0002

Table 1: Datacast performance with different cache sizes.

corresponding interest packet, generates a new credit, and stores the data packet accordingly.

Our implementation includes 18k lines of C/C++ code for Fabric Manager, 3k for Master, 2k for Datacast daemon, and 45k for ServerSwitch and Datacast drivers.

7. EXPERIMENT

We have built a ServerSwitch based BCube(4,2) network for our Datacast experiments. The network is shown in Fig. 4(a). It has 16 servers and 8 switches. All the 24 devices are Dell PowerEdge R610 servers. Every R610 has two E5520 Intel Xeon 2.26GHz CPU and 32GB RAM, and one ServerSwitch card. Every ServerSwitch card has four 1GbE ports. When an R610 acts as a switch, all the 4 ports of its ServerSwitch card are used; when it acts as a server, only 2 ServerSwitch ports are in use. We run both Fabric Manager and a Datacast master at server 00.

In the experiment, we create a Datacast group with server 00 as the source, and servers {02, 10, 21, 23, 31, 33} as the receivers. The credit assigned to receivers $w = 16$ packets. The congestion control parameters δ, T, and RTO are 5Mb/s, 1ms, and 20ms, respectively. The initial rate of a single tree is set to 500Mb/s. The two Steiner trees Tree_A and Tree_B are shown in Fig. 4 by pruning the two spanning trees. The group transmits a 4GB file from the source to all the 6 receivers. We use ramdisk to avoid disk I/O bottleneck.

Micro benchmark. We run the experiment with no background traffic. We use 8kB Jambo Ethernet frame for data packets. We first study the effect of cache size. We set the link rate from switch <1,3> to server 23 to 200Mb/s. This is to limit the rate of Tree_A to 200Mb/s and to create heterogenous receiving rates. We vary the cache size in the intermediate nodes from 128 to 16384 packets. Table 1 shows the results. In the table, dup means duplicated interest ratio perceived at the source side. Our testbed results align well with the simulation results (see Appendix B) and demonstrate that the performance of Datacast is not sensitive to the size of the content store. Even when the cache size is only 128 packets, Datacast can still achieve high throughput.

The reason that why Datacast achieves almost zero duplicate interests in Tree_B is because all the receivers get synchronized very quickly. But when the receivers's receiving rates are different, the number of duplicate interests is much larger. This is because the receivers can be temporarily asynchronized, and the slowest receiver 23 needs duplicate interests to slow down the sender.

In the rest of experiments, we use cache size 2048. We study the finish time, goodput, and efficiency of Datacast. We restore all the links to 1Gb/s. All the receivers receive the 4GB data in 23 seconds. The source achieves 1.74Gb/s goodput, which is very close to the maximum 2Gb/s of the two 1GbE Steiner trees. The source receives 512129 interest packets. The total number of bytes (including all data and interests packets) transmitted on all the links is 53.9GB. We also have studied the forwarding performance of the intermediate switches. Using switch <0,2> as an example, it is a branching node with two children. It uses 50% cycles for 2 CPU cores for forwarding. Server 32 is a non-branching Steiner node, and packet forwarding is offloaded to ServerSwitch hardware, hence its CPU usage for forwarding is 0%.

Comparison study. We compare Datacast with Bittorrent and the single-tree based approach. We use three metrics: link stress, network stress, and finish time. For a specific link l, link stress is $\frac{B_x^l}{DS}$, where B_x^l is the number of bytes transmitted on link l using scheme x, and DS is the total data size. Network stress is defined as $\frac{\sum B_x^l}{\sum B_{IP}^l}$, where B_{IP}^l is the number of bytes transmitted on link l using IP multicast.

For Bittorrent, we use μtorrent [27]. We setup a tracker at server 00 and let the the source and receivers be neighbors with each other. In this experiment, the receivers finish downloading in 41-52 seconds. Datacast is 2.3X faster. The source sends out 9.25GB data, or with source link stress 2.3. We count the total bytes transmitted on all the links. The value is 53.9GB for datacast and 72.5GB for Bittorrent. The network stress of Bittorrent is 1.35 times larger.

We use original CCN to represent single-tree based approach. In this approach, receivers independently send interests to the source using shortest-path routing, the source sends back data packets along the reverse path. Data packets are cached along the path. Hence a data delivery tree is formed implicitly. In this experiment, the tree is formed as follows. 00→ <0,0>→ 01→ <1,1>→ 21; <1,1>→ 31; <0,0>→ 02; <0,0>→ 03 → <1,3>→ 23; <1,3>→ 33; 00→ <1,0>→ 10. We can see the quality of the constructed tree is similar to those of the two Steiner trees in Fig. 4, with 6 Steiner nodes. However, since the two links of 00 are used in the tree, a second edge-disjoint tree is not possible. We also use it to transmit 4GB, the finish time is 39.1 seconds, which is 1.7X slower than Datacast. Furthermore, non-branching nodes 01, 03, and <1,0> needs to cache data in this approach.

Datacast performance. We use the same topology

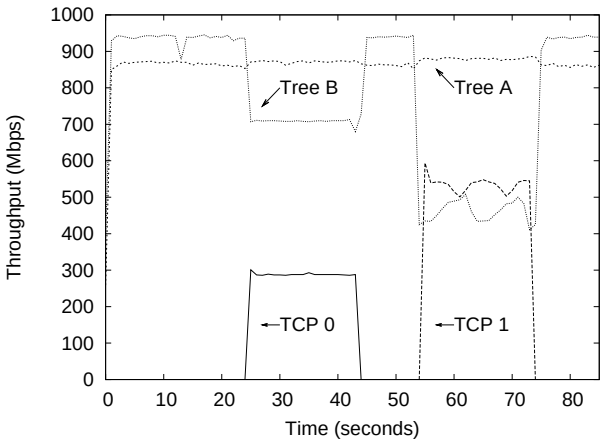


Figure 6: Datacast adapts to dynamically changing network conditions.

setup as in the micro benchmark. All the links are 1Gb/s. During time 23-43, we start a TCP connection TCP_0 between servers 20 and 13 with 64KB receive window. TCP_0 sends traffic along path {20, <1,0>, 10, <0,1>, 13} as fast as possible. During time 53-73, we start another TCP connection TCP_1 with 128KB receive window along the same path. These two TCP connections are set to compete with Datacast Tree_B. Fig. 6 shows the Datacast sending rates of Tree_A and Tree_B together with those of TCP_0 and TCP_1. We use different TCP window sizes to study how TCP and Datacast share bottleneck bandwidth. We found that the sharing ratios are 288Mb/s:709Mb/s for TCP_0 and Datacast Tree_B and 535Mb/s:458Mb/s for TCP_1 and Tree_B, respectively. The sharing ratio is roughly equal to (TCP receive window size 64KB or 128KB):(Datacast credit 16*8KB=128KB). Meanwhile, the sending rate of Tree_A is not affected since the two TCP connections do not compete with Tree_A.

We further test Datacast's failure handling ability. After we recover from TCP interference, we manually disable the link from switch <0,3> to server 32. Both <0,3> and 32 report the link failure to Fabric Manager after 3 seconds (3 is a configurable parameter to damp oscillation). Fabric Manager sends topology update to the master. The master then repairs Tree_B, by reconnecting switch <1,2> to server 22 and notifies server 02 the updated routing path. All these operations finish in 35ms. Since devices along the new path do not have cached data for 02, source 00 begins to receive duplicate interests from 02. Server 02 then begins catch up and the rest receivers slow down. After 4 seconds, all the receivers sync up.

To summarize, using real experiments, we have demonstrated that Datacast works well under varying network conditions and failures due to its robust congestion control. Due to its ability of using multiple Steiner trees, Datacast is around two times faster than both Bittorrent and the single-tree approach. Datacast also achieves higher efficiency than Bittorrent.

8. RELATED WORK

Active network [29, 32] uses "capsules" to carry both data and code. Network devices along the path run the carried code to provide flexible, application specific processing. Active network can be used to implement Datacast (and other new applications). But due to the sophisticated processing involved, active network can only be implemented with software routers. Datacast focuses on RGDD in data center environment. We show that by using ServerSwitch, we can implement Datacast using commodity network device. Furthermore, functions such as Datacast packet filtering and source routing can all be offloaded to hardware.

Active reliable multicast (ARM) [31] uses active network concept. Switches also cache packets in ARM. But the cached packet is used only for re-transmission, hence most likely the cached packets will not be used even once. Furthermore, re-transmitted packets are broadcasted along the whole sub-tree in ARM, whereas they are delivered only to the needed receivers in Datacast.

Content centric networking (CCN) [19] uses named data instead of named host for better networking abstraction. Datacast is built on top of CCN. By leveraging DCN specific characteristics (single operator and known topology), Datacast improves CCN for group data delivery in several aspects: Datacast uses multiple trees for data delivery speedup; it uses source routing to simplify unicast routing; it differentiates branching and non-branching nodes and uses hardware based packet forwarding for non-branching nodes. Besides these DCN specific improvements, Datacast further introduces the DTP protocol for splitting traffic among multiple trees and for congestion control in a single tree.

Reliable IP multicast has been studied extensively [17, 16, 26, 28]. All these designs are based on the unreliable IP multicast [7], hence need to maintain group state in all the intermediate network devices. In order to provide reliability, various designs have been introduced. Due to space limitation, here we only review two of them: SRM [17] and pgmcc [28].

In SRM, when receivers detect a loss, they wait a random time before multicast their repair requests. This is to suppress requests from other members sharing the same loss. Members that have the data use multicast to reply, again use random timer to avoid collision. In Datacast, data repairing is performed locally. When a receiver detects a data packet loss, it simply re-sends the interest packet without any coordination with others. The re-sent interest packet is treated just as a normal one by all the rest intermediate nodes.

In pgmcc, receivers report RTT and loss rate to the source, and source selects one receiver to act as the

acker, which is the one that has the lowest throughput. Then the source and acker run congestion control similar to TCP. Datacast differs from pgmcc in several ways. First, receivers in Datacast do not need to send feedback on packet loss rate and round trip time. Second, source does not need to select an acker, hence does not need to handle acker selection.

In order to solve the scalability and deployment issues of IP multicast, end-host based overlays and P2P systems have been designed (e.g., [4, 3, 5] and citations thereafter). These designs keep group state solely at end hosts and use TCP unicast for data delivery. Hence they easily achieve reliable data delivery and scale well. But overlays introduces high link and network stresses. Experiments in [4, 3] showed that the average link and network stresses are 1.9 and 1.9 for ESM, and 1.3 and 2.92 for Splitstream, respectively. And we have shown in 7 that the network stress of Bittorrent is 2.35.

9. CONCLUSION

We have presented the design, implementation, and evaluation of Datacast for reliable group data delivery in data centers. Datacast eliminates group state management in intermediate network devices by turning one-to-many communication to in-network packet caching. It accelerates data delivery by using multiple Steiner trees. By introducing a reliable Datacast transport protocol (DTP), Datacast splits traffic among multiple trees, achieves reliable data delivery, and adapts to varying network conditions by interpreting duplicate interest packets as congestion signals.

We have implemented Datacast using ServerSwitch, a programmable and high performance platform that integrates a merchandize switching chip and a commodity server. With ServerSwitch, we can implement source routing and offload packet forwarding for non-branching nodes using hardware. We further leverage server CPU for packet processing and server memory for packet caching.

Our implementation, simulation, and real testbed evaluation demonstrate that Datacast achieves scalable group state management and efficient and reliable data delivery, and that it can be implemented using commodity network devices, as designed.

10. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] Arista Networks. `www.aristanetworks.com`.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP*, 2003.

[4] Y. Chu, S. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. *IEEE JSAC*, Oct 2002.

[5] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[7] S. Deering. Host Extensions for IP Multicasting, August 1989. RFC1112.

[8] J. Edmonds. Edge-disjoint branchings. In R. Rustin, editor, *Combinatorial Algorithms*, pages 91–96. Algorithmics Press, New York, 1972.

[9] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, Aug 2009.

[10] C. Guo et al. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *SIGCOMM*, 2008.

[11] C. Guo et al. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[12] C. Guo et al. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CONEXT*, 2010.

[13] G. Lu et al. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI*, 2011.

[14] H. Abu-Libdeh et al. Symbiotic Routing in Future Data Centers. In *SIGCOMM*, 2010.

[15] J.S. Yang et al. Parallel Construction of Optimal Independent Spanning Trees on Hypercubes. *Parallel Computing*, 33, 2007.

[16] M. Handley et al. The Reliable Multicast Design Space for Bulk Data Transfer, August 2000. RFC2887.

[17] S. Floyd et al. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE trans. Networking*, Dec 1997.

[18] S. Tang et al. Independent Spanning Trees on Multidimensional Torus Networks. *IEEE Trans. Computers*, January 2010.

[19] V. Jacobson et al. Networking Named Content. In *CoNext*, 2009.

[20] Force10 Networks. S7000 Ű next-generation 10/40 gbe top-of-rack system. `www.force10networks.com/products/s7000.asp`.

[21] P. Fraigniaud and C.T. Ho. Arc-Disjoint Spanning Trees on the Cube-Connected-Cycles Network. In *ICPP*, 1991.

[22] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[23] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *SOSP*, 2003.

[24] R. L. Graham and L. R. Foulds. Unlikelihood That Minimal Phylogenies for a Realistic Biological Study Can Be Constructed in Reasonable Computational Time. *Mathematical Bioscience*, 1982.

[25] M. Isard, M. Budiu, and Y. Yu. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[26] Microsoft. Windows Deployment Services, 2009. http://technet.microsoft.com/en-us/library/cc772106(WS.10).aspx.

[27] μtorrent. www.utorrent.com.

[28] L. Rizzo. pgmcc: a TCP-friendly Single Rate Multicast Congestion Control Scheme. In *SIGCOMM*, 2000.

[29] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *SIGCOMM CCR*, April 1996.

[30] Po Tong and E. L. Lawler. A Fast Algorithm for Finding Edge-disjoint Branchings. *Information Processing Letters*, August 1983.

[31] Li wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse. Active Reliable Multicast. In *infocom*, 1998.

[32] David Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *SOSP*, 1999.

APPENDIX

A. SPANNING TREE ALGORITHMS

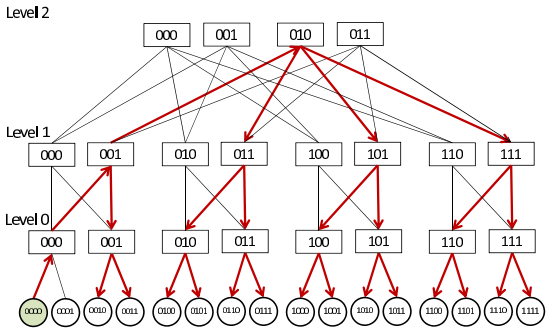


Figure 7: Spanning tree construction for fat-tree. The figure shows a spanning tree with server 0000 as the root for fat-tree(4,3).

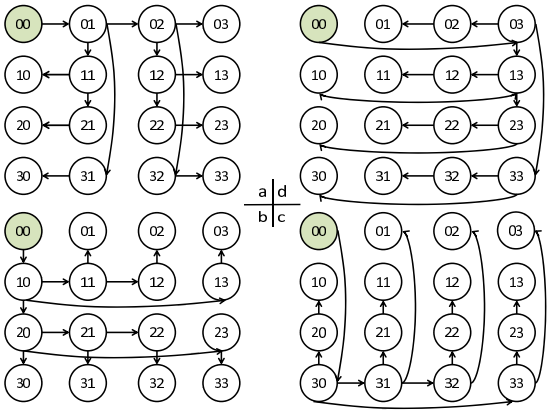


Figure 8: The four spanning trees for a torus(4, 2) with server 00 as the root.

A.1 Fat-tree

We denote a fat-tree [1] as fat-tree(n, k), where k is the number of switch layers of the fat-tree and n is the number of switch ports. There are $2(\frac{n}{2})^k$ servers in a fat-tree(n, k). Fig. 7 shows a fat-tree(4,3) with 16 servers. There is only one spanning tree since the servers have only one network interface. The algorithm for calculating the spanning tree for fat-tree is as follows. We divide the links as *up-links* and *down-links*. A low level device uses its up-links to connect to high level devices, and a high level device uses its down-links to reach low level devices. When building a spanning tree for a source server, the source server uses its up-link to reach its switch. The switch in turn uses one of its up-links to reach another high-level switch. The procedure is repeated until one of the top-level switches is reached. Then all the devices recursively use all their down-links to reach all the rest devices. Fig. 7 shows a spanning tree using the thick red lines. The time complexity of constructing such a spanning tree T is $O(|T|)$, where $|T|$ is the number of devices in T. The depth of the tree is $2k$. This algorithm can be applied to any generic multi-rooted trees.

A.2 BCube

As shown in [11], a BCube network is denoted as BCube(n, k), where n is the number of switch ports and k is the number of switch layers. There are k edge-disjoint spanning trees in a BCube(n, k). The spanning trees in [11] are constructed in a server-centric way. Servers use switches as layer-2 crossbars and server-to-server unicast is used to implement the spanning trees. The disadvantage of the construction is that the tree depth is $2(n-1)k+2$. When $n = 8$ and $k = 4$, the tree depth is 58 hops! In Datacast, we assume switches can perform packet caching, hence we can construct better spanning trees. When a server needs to reach all the rest $n-1$ servers under the same switch, instead of using the pipeline as described in Fig.5 of [11], we use the switch to directly reach the $n-1$ servers. We can still have k edge-disjoint spanning trees and the depth of each tree is only $2(k+1)$. The time complexity of constructing the k spanning trees is $O(k|V|)$. Fig. 4 shows the two edge-disjoint spanning trees for a BCube(4,2) network. The detailed algorithm is omitted due to space limitation.

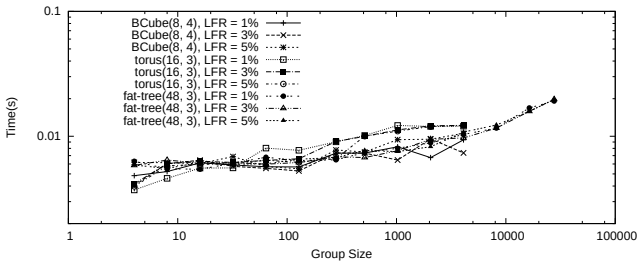
A.3 Torus

We use torus(n, k) to denote a torus network, where $n(n > 2)$ is the number of servers in one dimension and k is the number of dimensions. For a k dimensional torus, there are $2k$ edge-disjoint spanning trees. We use the algorithm designed in [18] to calculate the $2k$ spanning trees. Fig. 8 shows the four spanning tress constructed using the algorithm in [18]. The time complexity of the algorithm is $O(E)$. The tree depth is $(k+1)\lfloor\frac{n}{2}\rfloor$.

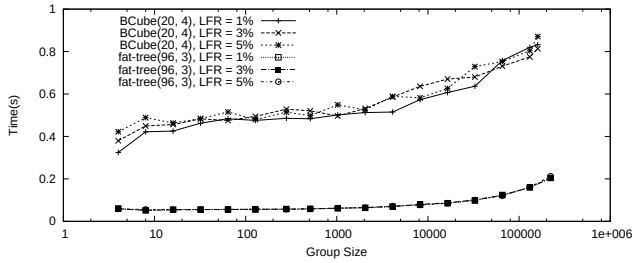
We note that there are fast algorithms for calculating spanning trees for hypercube and other structures (e.g., [15, 21]). We omit the discussions on these structures due to space limitation. After we construct the spanning trees, by prunning them, we can get the Steiner trees. But this approach cannot work when there are link failures. For example, four broken links $01 \rightarrow 02$, $10 \rightarrow 20$, $30 \rightarrow 31$, and $03 \rightarrow 13$ in Fig. 8 can destroy all the four spanning trees. Next, we show how we handle link failures.

A.4 simulation

We use simulation to study the performance of our algorithm for multiple Steiner trees calculation. We use a Dell PowerEdge R610 server for all the simulations. It has two E5520 Intel Xeon 2.26GHz CPU and 32GB RAM. We use the following network structures: a BCube(8, 4) with 4096 4-port servers and 2048 8-port switches, a torus(16,3) with 4096 6-port servers, a fat-tree(48, 3) with 27,648 servers and 2880 48-port switches. To study the performance of our algorithm for



(a) Results for BCube(8,4), torus(16,3), and fat-tree(48,3).



(b) Results for BCube(20,4) and fat-tree(96,3).

Figure 9: The running time of our multi-Steiner trees calculation algorithm.

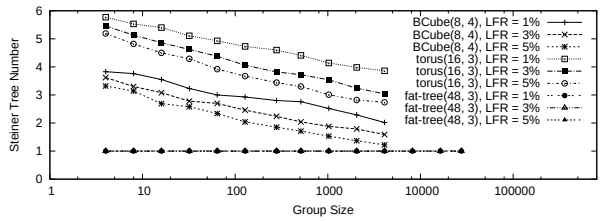
large networks, we further use a fat-tree(96,3) which has 221,184 servers, and a BCube(20,4) with 160,000 servers.

We have compared our algorithm with the generic algorithm which first calculates the spanning trees using Po's algorithm[30], then prunes the spanning trees to get the Steiner trees. The time complexity of the generic algorithm is dominated by the spanning tree calculation. The times needed for calculating spanning trees for BCube(8,4), torus(16,3), and fat-tree(48,3) are 44, 38, 118 seconds. The average spanning tree depths are 1301, 816, and 98, respectively. The worst-cast depth for BCube can be as large as 2099 hops. The generic algorithm therefore cannot be used in Datacast due to the low quality of the generated spanning trees and its high time complexity. In the rest of the section, we focus on the performance of our algorithm.

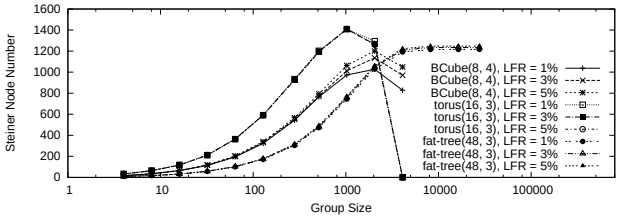
A.5 Running time

Fig. 9(a) shows the running time of our algorithm under various group sizes and link failure rates (LFR for abbreviation) for BCube(8,4), torus(16,3), and fat-tree(48,3). The link failures are randomly generated, but the cases when the network is not connected are ignored since it is impossible to generate spanning trees in that case. We run each experiment 100 times and show the worst-case running time. From the figure, we can see that our algorithm can finish in 20ms for various groups sizes for all the three network structures. Note that the running time can be further reduced by pre-calculating the spanning trees for all the sources.

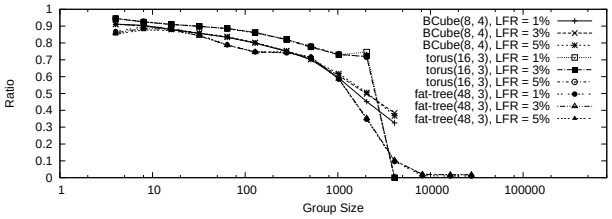
Fig. 9(b) further shows the running time of our algorithm for even larger networks fat-tree(96, 3) and BCube(20, 4). Note that this experiment is to stress test the performance of our algorithm. In practice, we can allocate small groups in small sub-networks. For example, for a small group with 10 members, we can allocate the group into a BCube(20,2) instead of the whole BCube(20,4). Nonetheless, our algorithm can still calculate the Steiner trees in less than one second for both networks. Note that our algorithm performs better for fat-tree since it has only one tree and its spanning tree algorithm has lower time complexity.



(a) Number of Steiner trees.



(b) Number of Steiner nodes.



(c) Non-branching ratio.

Figure 10: Steiner tree quality for various group sizes under different link failure rates.

A.6 Steiner tree quality

In what follows, we study the Steiner tree quality generated by our algorithm. Each experiment is performed 100 times, and the average results are reported. Fig. 10(a) shows the number of Steiner trees. As we stated before, fat-tree has only one Steiner tree. For BCube and torus, the number of Steiner trees decreases as the group size increases. This is expected because a large group may experience more link failures. To check whether our algorithm is effective in repairing Steiner trees, we have derived a bound, which is the minimum value of the out-degree of the source and the in-degrees of all the receivers. The Steiner tree number gotten by our algorithm is only 0.8% less than the bound on average.

Fig. 10(b) shows the number of Steiner nodes. We can see that our algorithm uses only a small amount of Steiner nodes for small groups. When the group size grows up, the number of Steiner nodes increases. For BCube and torus, after a threshold, the Steiner node number drops down because many of the servers are now receivers. For fat-tree, the number of Steiner nodes becomes a constant since almost all the switches are now Steiner nodes. We also study the depth of the Steiner trees. The tree depths (10 for BCube(8,4), 6 for fat-tree(48,3) and 32 for torus(16,3)) under various link failure rates for all the structures are small and close to the diameter of baseline networks.

To further study the Steiner tree quality, we have compared our algorithm with BFS_Steiner, which uses breadth first search (BFS) for Steiner tree calculation. BFS_Steiner generates minimal tree depth and therefore is expected to have small number of Steiner nodes. The results show that our algorithm and BFS_Steiner generate Steiner trees of similar quality. For example, for BCube(8,4) with 0% LFR, the numbers of Steiner nodes for a single tree in a group with 280 members are 525 and 493 using our algorithm and BFS_Steiner, respectively. But we always get 4 Steiner trees whereas BFS_Steiner can only get one Steiner tree when the group size is larger than 32 for all the three networks.

A.7 Branching versus non-branching nodes

Fig. 10(c) shows the non-branching ratio for different networks under different link failure rates. The result shows that different networks have different non-branching ratio curves. But the same network has similar non-branching ratio under different link failure rates. The result also shows that when the group size is small to average, the number of non-branching nodes contributes a significant portion of the total Steiner nodes. For example, for a BCube(8,4) with 1% link failure rate, the non-branching ratios are 88% (56 non-branching nodes over 64 Steiner nodes) and 75% (or 407 non-branching nodes over 543 Steiner nodes) for group sizes 16 and 280, respectively.

B. DTP SIMULATION

We have implemented DTP in NS2. In this section, we use comprehensive simulations to study the performance of DTP under different parameter settings. All of our experiments except the multi-tree one are performed using the first Steiner tree (Tree_A) shown in Figure 4(b). The bandwidth and propagation delay of every link are set to 10Gb/s and 10us, respectively. For every switch, the size of PIT is 64k, which is enough to hold all the flying interests. The header sizes of both interest and data packets are 72 bytes. The interest timeout value for a receiver or intermediate switch is set to 20ms. The total size of data to be delivered is 1GB. For each experiment, we measure the finish time and the total bytes transmitted in the network (which is the sum of the bytes transmitted of every link).

Parameters	Finish time	Total transferred bytes
T = 100ms, δ = 5Mb/s	62.407s	14.312GB
T = 1ms, δ = 50Mb/s	42.017s	14.469GB
T = 0.1ms, δ = 100Mb/s	42.018s	16.348GB

Table 2: Performance under different parameters δ and T.

In what follows, we study the parameters (δ, T, and the size of the content store) of the protocol, the performance of DTP under different packet loss condition, and speedup when multiple trees are available, and the fairness between DTP and TCP.

B.1 DTP Parameters

In this simulation, we deliberately slow down the link from switch <1,3> to server 23 to 200Mb/s to create receiving rate heterogeneity. Ideally, the goodput is $200Mbps \times \frac{1500-72}{1500} = 190.4Mb/s$, so the ideal finish time is $\frac{1GB \times 8}{190.4Mb/s} = 42s$, and the data to be transmitted is $13 \times 1GB \times \frac{1500+72}{1428} = 14.311GB$.

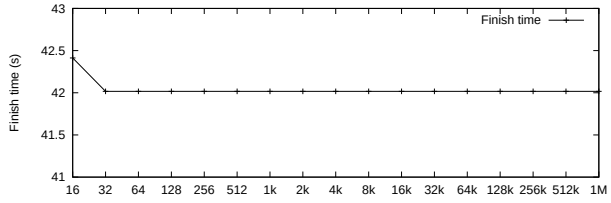
δ and T. We fix the size of the content store to be 2048 packets, and study the performance of the protocol using different δ and T combinations. The results are shown in Table 2.

The results indicate that $T = 1ms$ and $\delta = 50Mb/s$ has the best performance. The rest two groups are used to demonstrate the performance using aggressive and conservative parameter settings. When we use conservative parameters $T = 100ms$ and $\delta = 5Mb/s$, the sending rates of fast ones are not able to cache up with the slow receiver after the source cuts the sending rate by halve. As a result, the sending rate will be cut again. So the total finish time is much larger than the ideal one. When we use more aggressive parameters $T = 0.1ms$ and $\delta = 100Mb/s$, the sending rates of fast receivers grow too fast and the slow receivers cannot cache up. Hence the receivers become unsynchronized, which results in inefficient use of bandwidth.

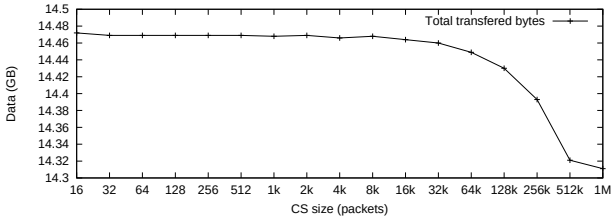
In the rest of the paper, we use $T = 1ms$ and $\delta = 50Mb/s$. These parameters are comparable to the parameter setup in TCP. TCP increases its window by one packet in one RTT. In data center networks, the RTT is around 0.5ms. Hence in 1ms, the rate increase in TCP is $\frac{2 \times MTU}{RTT} = 48Mb/s \approx \delta$.

Content store size. We vary the content store (CS) size from 16 packets to 1M packets for each node. The goodput and the transferred data size are shown in Figure 11.

From the results, we observe that the goodput of our algorithm is quite stable with different CS size. Our algorithm works well even with a very small CS. The transferred data is also close to the theocratical bound.

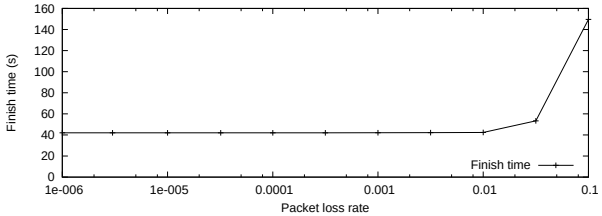


(a) Finish time

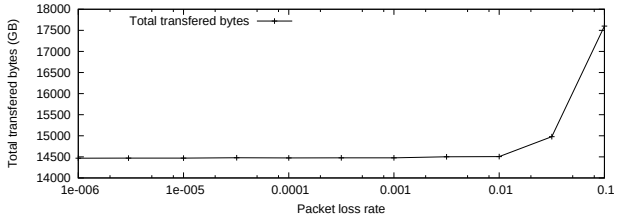


(b) Total transferred data

Figure 11: Performance under different cache sizes.



(a) Finish time



(b) Total transferred data

Figure 12: Performance under different data packet loss rates.

The performance is slightly better when the cache size is larger. This is because, when the CS size is smaller, cache miss will increase and more duplicate interests will be sent to the sender.

Loss Rates. To study how packet losses affect the performance of our algorithm, we manually generate losses when data packets are transmitted through the link from switch <1,3> to server 23. In our simulation, the loss rate ranges from 0.0001% to 10%. The results is shown in Figure 12.

The results show that our algorithm is quite resilient to data packet drops. The finish time is only 0.9% larger, and the total transferred bytes are only 1.4% larger than the optimal results when the loss rate is 1%. Our algorithm still performs reasonably well even when the packet drop rate is as high as 3% (which is not expected to happen in the real world).

B.2 Multiple Tree Performance

We also study the performance of DTP when multiple trees are available. The group setup is the same as the one in section 7. Server 00 acts as the source, and servers 02, 10, 21, 23, 31, 33 as the receivers. All the links are 1Gb/s.

We start all the receivers simultaneously. The transmission finishes at time 4.205s. The total bytes transferred in the network are 14.311GB.

These results are also very close to the optimal value. Since the header size is 72 packets, the maximum possible goodput is $2 \times 1Gbps \times \frac{1428}{1500} = 1.904Gbps$, so the optimal finish time is $\frac{1GB}{1.904Gbps} = 4.202s$. Our result is only 0.1% larger than that. Our bandwidth utilization is also efficient. Both Steiner trees contain 13 links, so the minimum possible data to be transferred is $13 \times 1GB \times \frac{1572}{1428} = 14.311GB$, which is identical to our result.

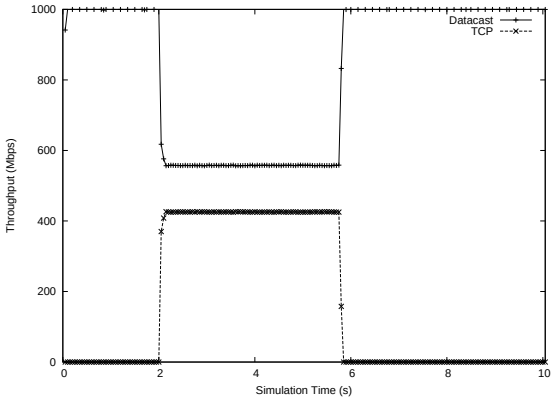


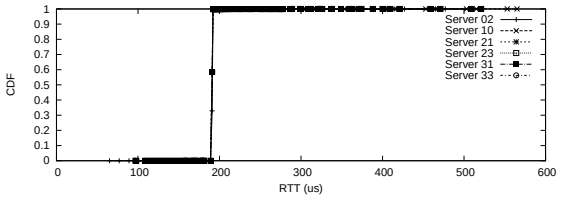
Figure 13: Inter-protocol fairness with TCP.

B.3 TCP Friendliness

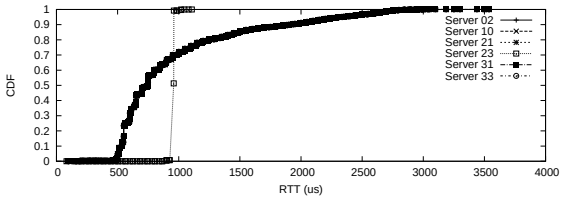
To study the inter-protocol fairness with TCP, we set up a TCP connection between server 00 and server 01, which starts at time 2s. The TCP connection transfers 200MB data. Figure 13 shows the result. Before TCP starts, Datacast Tree_A transfers data at full speed. After TCP starts, the throughput of Datacast Tree_A drops down immediately. TCP achieves transmission rate 425Mb/s, while Datacast gets 557M/ps. The result shows that our DTP achieves rough fairness with TCP.

B.4 Receivers' RTTs

We study the RTT distributions perceived at different receivers. Since DTP synchronizes the receivers to similar receiving rate, we expect that they experience



(a) RTT on the Steiner tree without congestion.



(b) RTT on the Steiner tree with congestion.

Figure 14: RTT distributions of different receivers.

similar RTT. The RTT distribution also can help us choose the value of interest retransmission RTO. Figure 14 plots the CDFs of the RTTs. Figure 14(a) shows the CDF when there is no background traffic. Figure 14(b) shows the CDF when there are congestions. To simulate congestion, we simply slow down the link from switch <1,3> to server 23 to 200Mbps.

From Figure 14(b), we see that although the receivers do not have identical RTT distributions, they are still in the same order of magnitude. So it is feasible to use a fixed timeout for all of them. We will study how to use a dynamic RTO in our future work.