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

Jiaxin Cao, Chuanxiong Guo, Guohan Lu, Yongqiang Xiong,
Yixin Zheng, Yongguang Zhang, Yibo Zhu, Chen Chen
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).

Motivated by recent advances on data center network topology designs (multiple edge-disjoint Steiner trees for RGDD) and innovations on network devices (practical in-network packet caching), we propose *Datacast* for RGDD. Datacast explores two design spaces: 1) Datacast uses multiple edge-disjoint Steiner trees for data delivery acceleration. 2) Datacast leverages in-network packet caching and introduces a simple soft-state based congestion control algorithm to address the scalability and efficiency issues of RGDD.

Our analysis reveals that Datacast congestion control works well with small cache sizes (e.g., 125KB) and causes few duplicate data transmissions (e.g., 1.19%). Both simulations and experiments confirm our theoretical analysis. We also use experiments to compare the performance of Datacast and BitTorrent. In a BCube(4, 1) with 1Gbps links, we use both Datacast and BitTorrent to transmit 4GB data. The link stress of Datacast is 1.01, while it is 1.39 for BitTorrent. By using two Steiner trees, Datacast finishes the transmission in 16.9s, while BitTorrent uses 52s.

1. INTRODUCTION

Reliable Group Data Delivery (RGDD) is widely used in cloud services (e.g., GFS [15] and MapReduce [5]) and applications (e.g., social networking, Search, scientific computing). 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 bulk data from the source to all the receivers.

Existing solutions for RGDD can be classified into two categories: 1) Reliable IP multicast. IP multicast suffers from scalability issues, since it is hard to manage a large number of group states in the network. Adding reliability is also challenging, due to the ACK implosion problem [12]. 2) End-host based overlays. Overlays are scalable, since devices in the network do not maintain group states. Reliability is easily achieved by using TCP in overlays. However, overlays do not use network bandwidth efficiently. The same copy of data may traverse the same link several times, resulting high link stress. For example, ESM [18] reported that the average and worst-case link stresses are 1.9 and 9, respectively.

Motivated by the recent progresses on data center network (DCN) topologies and network devices, we explore new opportunities in supporting RGDD for DCN: 1) Recently proposed DCN topologies have multiple edge-disjoint Steiner trees, which has not been well studied before. These multiple Steiner trees may enable full utilization of DCN bandwidth. 2) There is a clear technical trend that network devices are providing powerful packet processing abilities by integrating CPUs and large memory. This makes in-network packet caching practical. By leveraging in-network packet caching, we can address the scalability and bandwidth efficiency issues of RGDD.

However, it is challenging to take advantage of these opportunities. It has been proved that even calculating a single Steiner tree is NP-hard [16]. In RGDD, we have to calculate multiple Steiner trees within a short time, which makes the problem even harder. Although network devices are becoming capable of in-network packet caching, the resource is not unlimited. We need to use as small caches as possible for each group to maximize the number of simultaneously supported groups. At the same time, we need to increase bandwidth efficiency by reducing duplicate packets transmitted in the network.

In this paper, we design *Datacast* to address the above challenges. Leveraging the properties of the DCN topologies, Datacast introduces an efficient algorithm to calculate multiple edge-disjoint Steiner trees, and then distributes data among them. In each Steiner tree, Datacast leverages the concept of CCN [13]. To help Datacast achieve high bandwidth efficiency with small cache size in intermediate nodes, we design a rate-based congestion control algorithm, which follows the classical Additive Increase and Multiplicative Decrease (AIMD) approach. Datacast congestion control leverages a key observation: the receiving of a duplicate packet request

at the source can be interpreted as a congestion signal. Different from previous work (e.g., TFMCC [26] and pgmcc [21]), which uses explicit information exchanges between the source and receivers, Datacast is much simpler. To understand the performance of Datacast, we build a fluid model. By analyzing the model, we prove that Datacast works at the full rate when the cache size is greater than a small threshold (e.g., 125KB), and also derive the ratio of duplicate data sent by the data source (e.g., 1.19%). We have built Datacast in NS3, and also have implemented it with the ServerSwitch [8] platform. Simulations and experiments verify our theoretical results, which suggest that Datacast achieves both scalability and high bandwidth efficiency.

This paper makes the following contributions:

- We design a simple and efficient multicast congestion control algorithm, and build a fluid model to understand its properties.
- We propose a low time-complexity algorithm for multiple edge-disjoint Steiner trees calculation.
- We implement Datacast with the ServerSwitch platform, and validate its performance.

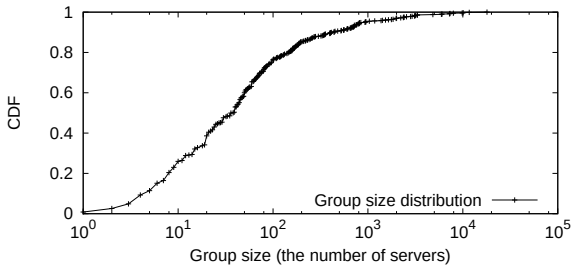
The rest of this paper is organized as follows. In Section 2, we introduce the background. We briefly overview Datacast design in Section 3. We design the Steiner trees algorithm in Section 4, and build the Datacast transport protocol in Section 5. We present simulation results in Section 6, implementation and experiments in Section 7. Finally, we discuss the related work and conclude the paper in Section 8 and 9, respectively.

2. BACKGROUND

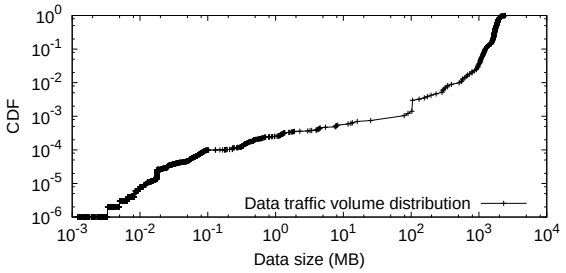
2.1 Reliable group data delivery

In data center applications and services, Reliable Group Data Delivery (RGDD) is a pervasive traffic pattern. The problem of RGDD is, *given a data source, Src, and a set of receivers, R_1, R_2, $\cdots$, R_n, how to reliably transmit bulk data from Src to all the receivers.* A good RGDD design should be scalable and achieve high bandwidth efficiency. The following cases are typical RGDD scenarios.

Case 1: In data centers, servers are typically organized as physical clusters. During bootstrapping or OS upgrading, the same copy of the 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 is assigned 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.



(a) The group size distribution in a large data center.



(b) The traffic volume distribution for a large distributed execution engine.

Figure 1: RGDD groups and traffics in data centers.

Case 2: In distributed file systems, e.g., GFS [15], a chunk of data is replicated to several (typically three) servers to improve reliability. The sender and receivers form a small replication group. A distributed file system may contain tens of Peta bytes using tens of thousands machines. Hence the number of replication groups is huge. In distributed execution engine, e.g., Dryad [19], 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 OSes and application images need be delivered to all the physical servers that host these virtual machines.

Figure 1(a) and 1(b) show the group size and traffic volume distributions for a RGDD service in a large production data center. We use these two figures to illustrate the challenges in supporting RGDD.

The system should be scalable. As we have mentioned in the above scenarios, we need to support a large number of RGDD groups in large data centers. Figure 1(a) further shows that the group size varies from several servers to thousands of servers and even more. The large number of groups and the varying group sizes pose scalability challenges, since maintaining a large number of group states in the network is hard (as demonstrated by IP multicast).

Bandwidth should be efficiently and fully used. Figure 1(b) shows the traffic volume distribution for

group communications. It shows that the groups transmitting more than 550MB data contribute 99% RGDD data traffic volume. Due to the large number of groups and the large data sizes, RGDD contributes a significant amount of traffic. This requires that RGDD uses network bandwidth efficiently. On the other hand, the new DCN topologies (e.g., BCube [7] and CamCube [9]) provide high network capacity with multiple data delivery trees. An RGDD design should take full advantage of these new network topologies to speedup data delivery.

In what follows, we introduce recent technology progresses on DCN topologies and network devices, which we leverage to address the above challenges.

2.2 New opportunities

Multiple edge-disjoint Steiner trees. Different from the Internet, DCNs are owned and operated by a single organization. As a result, DCN topologies are known in advance, and we can assume that there is a centralized controller to manage and monitor the whole DCN. Leveraging such information, we can improve RGDD efficiency by building efficient data delivery trees. Furthermore, several recently proposed DCNs (e.g., BCube [7] and CamCube [9]) have multiple edge-disjoint Steiner trees which can be used to further accelerate RGDD.

In-network packet caching becomes practical. Recently, we observe a clear technical trend for network devices (switches and routers). First, powerful CPUs and large memory are being included in network devices. The new generation of devices are equipped with multicore X64 CPUs and several GB memory, e.g., Arista 7504 has 2 AMD Athlon X64 Dual-Core CPUs and 4GB DRAM. Second, the merchant switching ASIC, CPU and DRAM can be connected together by using the state-of-the-art PCI-E interface, as demonstrated by research prototype (e.g., ServerSwitch [8]) and products (e.g., Force10 S7000 [20]). With the new abilities of network devices, many in-network packet processing operations (e.g., in-network packet caching) become practical. In this paper, we explore in-network packet caching. By turning hard-states for group managements in intermediate network devices into soft-states based packet caching, we address the scalability and efficiency issues of RGDD.

However, technical challenges exist to take advantage of these opportunities. First, given the network topology, calculating one single Steiner tree with minimal cost is NP-hard [16]. What is more challenging is that we have to calculate multiple Steiner trees, and the calculation has to be fast enough (otherwise it may be more time consuming than data dissemination). Second, we have a large number of RGDD groups to support and have limited resources in intermediate network devices. How to use as few resources as possible to support more RGDD groups is a challenge.

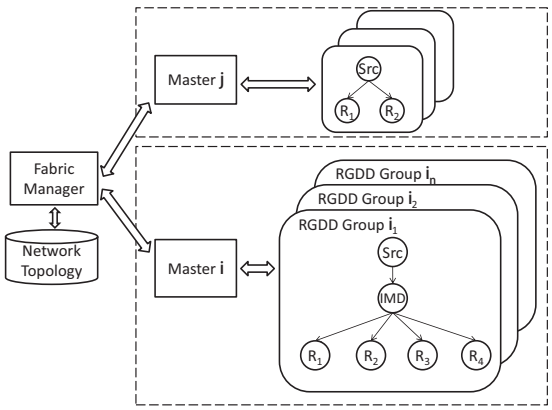


Figure 2: The architecture of Datacast. There are five major components in Datacast, Fabric Manager, Master, data source (Src), receivers (R_i), and intermediate devices (IMD).

We design Datacast to explore the new design spaces provided by the new opportunities. The design goal of Datacast is *to achieve scalability and also high bandwidth efficiency*. In what follows, we first introduce the architecture of Datacast, then describe how Datacast addresses the above technical challenges.

3. DATACAST OVERVIEW

Figure 2 shows the architecture of Datacast. There are five components in Datacast: Fabric Manager, Master, data source, receivers, and intermediate devices (IMD). Fabric Manager is a centralized controller, which maintains a global view of the network topology. When we need to start an RGDD group, we first start a Master. The Master will get topology information from Fabric Manager and then calculate multiple edge-disjoint Steiner trees. After that, the Master will send the tree information and other signalling messages (e.g., which file to fetch) to receivers via a signalling protocol. Then data transmission begins. When transmitting data, the data source will run our congestion control algorithm. During the whole process, intermediate devices do not interact with Fabric Manager, Master, the source or any receivers. These devices just cache and service data based on their local decisions.

To deliver signalling messages efficiently, we have built a signalling protocol, which uses a hierarchical transmission tree structure (generated by the Breadth First Search algorithm) to transmit signalling messages. It encodes the transmission tree into the message. Each node in the transmission tree decodes the signalling message, splits the tree into subtrees and forwards each subtree to its corresponding children. When the signalling messages reach the leaves, ACKs are generated and aggregated along the paths from leaves to the root.

Using the message split and aggregation, signalling messages can be reliably and efficiently delivered.

In large data centers, failures are inevitable. Different from BitTorrent [4], which achieves fault tolerant in a distributed way, Datacast handles network failures in a centralized manner. In Datacast, Fabric Manager monitors the network status in real time. When network failures happen, Fabric Manager will send the new topology information to all the Masters, and each Master will recalculate the Steiner trees and notify the affected receivers accordingly. Due to space limitation, we omit the details of the signalling protocol and the failure handling details, but we do use real implementation and experiment to evaluate their performance in Section 7.2.3.

In the following sections, we will present two key designs of Datacast: the fast calculation of multiple edge-disjoint Steiner trees, and the Datacast congestion control protocol which helps Datacast achieve scalability and high bandwidth efficiency.

4. MULTIPLE EDGE-DISJOINT STEINER TREES IN DCN

In this section, we first present the algorithm on multiple Steiner trees calculation, then discuss how to use these multiple Steiner trees for data delivery.

4.1 Calculation of multiple Steiner trees

It has been known that using multiple Steiner trees can improve the transmission efficiency [3]. However, constructing multiple edge-disjoint Steiner trees in a given (data center) topology has not been investigated before. The problem is, *for a given network $G(V, E)$, where V is the set of nodes and E is the set of edges, and a group D containing one source and a set of receivers, how to calculate the maximum number of edge-disjoint Steiner trees.* This is the well known multiple edge-disjoint Steiner trees problem, which has been studied for decades. Unfortunately, calculating Steiner trees is NP-hard [16].

We therefore turn our attention to heuristic algorithms. One reasonable approach is as follows. There are algorithms for calculating multiple edge-disjoint spanning trees (e.g., [6]). We can first find the multiple edge-disjoint spanning trees, and then prune the unneeded edges and nodes to get the Steiner trees.

However, the generic multiple spanning trees algorithms do not work well for our case. First, the time complexity of calculating the spanning trees is high. The best algorithm we know is Po's algorithm [24]. Its time complexity is $O((k')^2|V||E|)$, which is too high for RGDD (we will see that in Section 6.1.1). 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.

```
// G is the DCN network, D is the Datacast group.
CalcSteinerTrees(G, D):
  // 1) construct multiple spanning trees
  SPTSet = G.CalcSpanningTrees(D.src);

  // 2) prune each spanning trees
  foreach (SPT in SPTSet)
    SteinerTree = Prune(SPT, D);
    SteinerTreeSet.add(SteinerTree);

  // 3) repair Steiner trees if they are broken
  foreach (SteinerTree in SteinerTreeSet)
    if (SteinerTree has broken links)
      if (RepairSteinerTree(SteinerTree, G) == false)
        Release(SteinerTree);
        SteinerTreeSet.remove(SteinerTree);
  return SteinerTreeSet;
```

Figure 3: The algorithm for multiple edge-disjoint Steiner trees calculation.

Fortunately, we observe that DCNs, e.g., Fattree, BCube and multi-dimensional Torus, are well structured topologies. These topologies are also well studied. Multiple spanning trees construction algorithms for these topologies are already known (e.g., [7, 22]), and these spanning trees have good qualities, e.g., small tree depths. However, network failures (e.g., link failures) are common in real networks. Without reorganizing the spanning trees, network failures could possibly break all the trees generated by these algorithms. In order to solve the problem, we propose a multiple edge-disjoint Steiner trees algorithm, which is shown in Figure 3. The algorithm contains three parts. In what follows, we will discuss the each part separately.

4.1.1 Spanning trees construction algorithms

Fattree. We denote the Fattree [1] as Fattree(n, k), where k is the number of switch layers and n is the number of switch ports. For Fattree, there is only one spanning tree, since there is only one link for each receiver. Fig. 4 shows a spanning tree for Fattree(4, 3) in red lines. The construction of this spanning tree is straightforward. We first select a up path to a core switch, and then all the switches in the path selects all the children nodes. The time complexity is $O(|V|)$.

BCube. As is shown in [7], for a BCube(n, k), there are $k+1$ edge-disjoint spanning trees. The spanning trees in [7] are constructed in a server-centric way, i.e., servers use switches as layer-2 crossbars and server-to-server unicast to implement the spanning trees. In our group data delivery system, we assume switches can perform packet caching. So when a server wants to reach all the rest $n-1$ servers under the same switch, instead of using the pipeline proposed in [7], we use the switch to direct connect the rest servers. Fig. 5 shows the

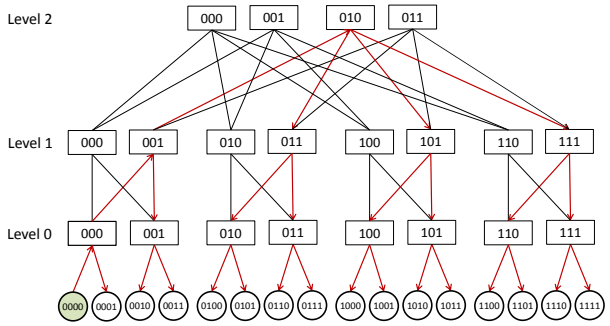


Figure 4: A spanning tree for Fattree(4, 3) with server 0000 as the root. The red links form the spanning tree.

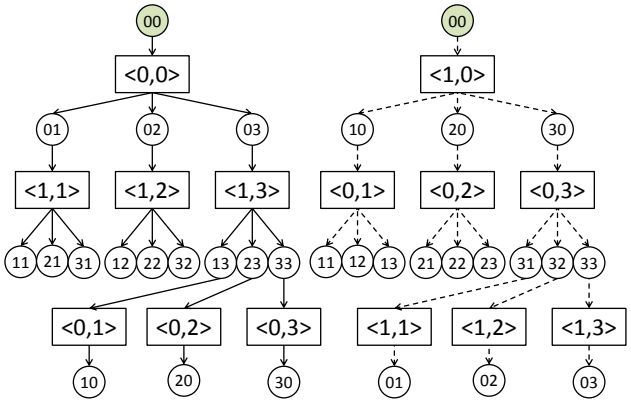


Figure 5: Spanning trees for BCube(4, 1) with server 00 as the roots.

constructed two spanning trees for BCube(4, 1). The time complexity of this algorithm is $O((k+1)|V|)$.

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 proposed in [22] to calculate the $2k$ spanning trees. Fig. 6 shows the constructed four spanning trees for a Torus(4, 2) with server 00 as the roots. The time complexity of this algorithm is $O(k|V|)$.

Compared with the generic algorithms [6], these algorithms are more suitable for data center networks. Its time complexity is much lower, and it also offers better qualities, e.g., small tree depths.

4.1.2 *Spanning trees pruning*

After spanning trees are calculated, we need prune them to Steiner trees (i.e., prune the links that are not needed to deliver data). For example, in Fig. 4, when server 0001 is not receiver, the link from switch 000 to server 0001 will not be used. To prune the spanning tree, we just calculate the paths from the receivers to

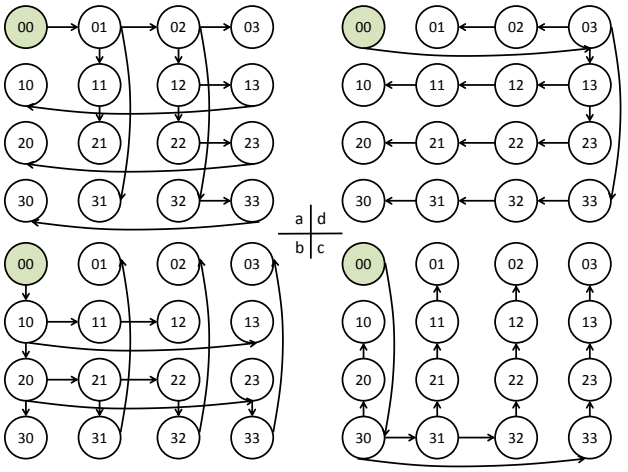


Figure 6: Four spanning trees for Torus(4, 2) with server 00 as the roots.

the source in the spanning tree. Then the set of links involved in the paths form a Steiner tree. The time complexity of pruning all the spanning trees is $O(|E|)$, since each link will only be traversed once.

4.1.3 *Steiner trees repairing*

The construction and pruning of spanning trees do not take the network failures into consideration. But in the worst case, network failures could possibly break all the calculated Steiner trees. In this case, we need to repair the broken trees. The core idea of repairing a Steiner tree is: *we first release the broken tree, and then try to use Breadth First Search (BFS) to traverse the free and active links to construct a new Steiner tree.* The repairing algorithm applies this idea to the broken trees one by one as shown in Fig. 3. Although this idea is simple, it has the following benefits: 1) It guarantees at least one Steiner tree if all the receivers are connected. 2) The depth of the tree is locally minimized due to the use of BFS. The time complexity of repairing all the trees is $O(k'|E|)$, where k' is the number of Steiner trees to be repaired.

Our multiple Steiner trees calculation algorithm is fast. The time complexity of the algorithm is $O(k|V|)+O(|E|)+O(k'|E|)$, which contains the construction and pruning of spanning trees and the repairing of Steiner trees.

Our algorithm has good performance (in terms of the number of Steiner trees) and is fault tolerant. Even if there are network failures, we can still create a number of Steiner trees. We have derived an upper bound of the number of Steiner trees, and found that the number of Steiner trees generated by our algorithm is very close to the upper bound (details will be shown in Section 6.1.2). For Fattree, although it has only one Steiner

tree, our algorithm guarantees that it always generates a Steiner tree as long as the network is connected.

In this paper, we evaluate our algorithm on Fattree, BCube and Torus. However, our algorithm is not constrained to these topologies. To use our algorithm for other topology, we only need to change the spanning trees construction part. Since data center networks are well structured and have been studied extensively, the spanning tree algorithms have already been proposed (e.g., HyperCube [27]), it is easy to adapt our algorithm for these topologies.

4.2 Data distribution among multiple Steiner trees

To use multiple Steiner trees for data delivery, we first split the data into blocks, and then feed each tree with a block. When a Steiner tree finishes transmitting the last data packet of the current block, we know that the transmission of the current block is finished. Then the data source will use our signalling protocol to deliver the information of the next block to be transferred, e.g., the name of the block, to the receivers. After that the Steiner tree will start to transmit the next block. This process repeats until all the blocks are successfully delivered.

5. DATACAST TRANSPORT PROTOCOL

In this section, we introduce in-network packet caching in Datacast, present the Datacast congestion control algorithm and discuss the cache management mechanism. By building a fluid model for the congestion control, we also derive the condition under which Datacast operates at the full rate, and its efficiency.

5.1 Data transmission with in-network caching

In-network packet caching has been used in many previous works, including Active Networking [23], RE (redundancy elimination) [2], and CCN [13]. Datacast is built on top of CCN. In CCN, every single packet is assigned a unique, hierarchical name. A user needs to explicitly send an interest packet to ask for the data packet. Any intermediate device that has the requested data along the routing path can respond with the data packet. The network devices along the reverse routing path then cache the data packet in their content stores for later uses. CCN therefore turns group communication into in-network packet caching.

Datacast improves CCN as follows: 1) Datacast introduces a congestion control algorithm to achieve stability and high bandwidth efficiency. 2) Datacast only caches data packets at branching nodes, which helps the whole system save memory. 3) Datacast uses source routing to enforce routing paths, so no Forwarding Information Base (FIB) management is needed at the intermediate devices.

Figure 7 shows an example of data delivery with in-network caching supports. The green node, 00, is the data source. The blue nodes, 12, 13, 21 and 33, are the receivers. The two Steiner trees calculated by the algorithm proposed in Section 4 are shown in solid lines and dashed lines separately. The transmission in Steiner tree A could take the following steps: 1) Node 21 sends an interest packet to node 00 through the path {21, 11, 01, 00}. Node 00 sends the requested data back along the reverse path. Then the data packet is cached at the branch node 01. 2) Node 12 sends an interest packet along the path {12, 02, 01, 00} asking for the same data. When the interest arrives at node 01, node 01 finds that it has already cached the data packet, so it terminates the interest and sends back the data packet. Then the data are cached at node 02 and 12. 3) Node 13 sends its interest along the path {13, 12, 02, 01, 00}. Then the data is replied by node 12, since it has cached the data. 4) Node 33 sends its interest along the path {33, 32, 02, 01, 00}, and node 02 returns the data packet.

Note that the execution order of the four steps in the example is not important. They can be executed in an arbitrary order, and still achieves the same result. The reason is that, in the end, all the steps together cover the same Steiner tree by traversing every link of the tree exactly once.

The benefits of in-network caching are two-fold: 1) By leveraging in-network caching, the network devices do not need to maintain hard group states. The intermediate devices do not even know that they are part of a group communication session. Hence Datacast is inherently scalable. 2) In-network caching makes reliable transmission easier to achieve. When a packet is dropped, the receiver will resend an interest packet requesting the same data packet (after timeout). This interest packet could be served by the nearest node that caches the requested data packet. The sender does not even necessarily know that there are packet losses in the network, so the ACK/NAK implosion problem does not occur.

5.2 Datacast congestion control algorithm

Datacast congestion control algorithm works for a single Steiner tree. It is one of the most important part of Datacast to realize its design goal, i.e., to achieve scalability and high bandwidth efficiency. Since Datacast turns hard group states into soft-state based packet caching, it is natural to require that the cache size in intermediate devices for each group is as small as possible (so as to support more groups), and the rates of receivers are synchronized (so as to improve bandwidth efficiency). If the rates of receivers are synchronized, only one copy of each packet is delivered in a Steiner tree. When receivers have different receiving bandwidths, we

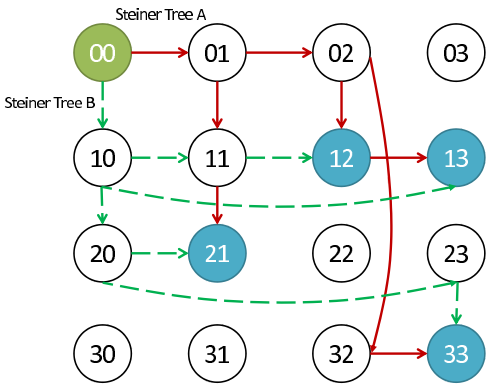


Figure 7: An illustration of in-network caching. The green node, 00, is the data source, while the blue ones, 12, 13, 21 and 33, are the receivers. Two Steiner trees are calculated, shown in the solid and dashed lines, respectively.

expect all the rates of receivers are synchronized to the receiving rate of the slowest receiver.

A synchronized scheme may suffer from significant throughput degradation if a receiver in the group has a small receiving rate. In this case, we may either kick out the very slow receivers, or split the data delivery group into multiple ones. These topics are our future work.

Datacast uses the classical AIMD for congestion control. This is not new. What is new in Datacast is how congestion is detected. Datacast uses *duplicate interests* as congestion signals. A duplicate interest is an interest requiring the same data which has been asked before. The source receives a duplicate interest under the following two cases: 1) The network is congested, so some packets are dropped. Then the receiver will retransmit the interest, which serves as a duplicate interest. 2) Receivers are out of sync. When slow receivers cannot keep up with the fast ones, their interests will not be served by the cache of the intermediate devices. The interests will finally be sent to the data source, which serves as duplicate interests. In these cases, the source needs to slow down its sending rate. On the other hand, if there is no congestion and the rates of receivers are well synchronized, there will be no duplicate interests, and the source should increase its sending rate.

After congestion is detected, the rate adjustment becomes easy: when the source receives a duplicate interest, it decreases its sending rate by half; when no duplicate interest is received in a time interval T, the source increases the sending rate by δ. Datacast congestion control is therefore rate-based. The source maintains and controls a sending rate r[1]. Note that the sending rate of the duplicate data packet is not constrained by the congestion control, since the corresponding duplicate interest packets are from the slowest receiver, and the receiving rate of the slowest receiver should not be further reduced.

At the receivers' side, each receiver is given a fixed number of credit, w, which means that one receiver can send at most w interests into the network. When a receiver sends out an interest, the credit is decremented by one. When it receives a data packet, its credit is incremented by one. In Datacast, the guideline for setting w is to saturate the pipe. In a DCN with 1Gbps link, when the RTT is 200us (which is a typical network latency in a data center environment), $w = 16$ can saturate the link. To achieve reliability, the receiver retransmits an interest if the corresponding data packet does not come back after a timeout. The timeout is calculated in the same way as TCP.

To summarize, Datacast congest control algorithm works as follows.

$$r = \begin{cases} \frac{r}{2} & \text{when a duplicate interest is received.} \\ r + \delta & \text{when there is no duplicate interest in T.} \end{cases}$$

As we can see, the Datacast congestion control algorithm is simple. The source does not need to know which receiver is the slowest one, and what is the available bandwidth of that slowest receiver. In Section 5.4, we will show analytically that Datacast uses small caches size and results in few duplicate data transmissions.

5.3 Cache management

To prevent cache interferences among different transmission trees, we use a *per-tree based* cache replacement algorithm. Each device uses a per Datacast tree based cache with size C. This is possible due to the following reasons: 1) A Datacast tree can be uniquely identified by a global unique tree transmission id (assigned by Master). 2) The cache size needed by each tree is small (as we will show in the next subsection).

In each tree, we find that the most popular data packets are the new ones, since new data packets will always be accessed by other receivers in the future. To keep new data packets in caches and erase old data packets, Datacast chooses First In First Out (FIFO) as its per-tree cache replacement policy. To prevent unpopular data packets from being put into caches, Datacast does not cache duplicate data packets.

Note that although this is a per-tree strategy, it is a scalable solution. The reasons are: 1) Compared with IP multicast, we do not need any protocol (e.g., IGMP) to maintain Datacast's per-tree states. Switches just use local decisions to manage its cache. 2) Datacast

[1]To be exact, this is the rate of the source's token bucket. The source cannot achieve this rate if there are not enough interests from the receivers.

can work efficiently with small caches, e.g., 125KB, and large memory is expected for future network devices, e.g., 16GB memory for a switch. If it uses 4GB as Datacast cache, a network device can support up to 32k ($\approx \frac{4GB}{125KB}$) simultaneous trees.

5.4 Properties of Datacast congestion control algorithm

In this subsection, we study the following questions: 1) What is the condition for Datacast to work at the full rate (i.e., the receiving rate of the slowest receiver)? 2) When Datacast works at the full rate, how much duplicate data will be sent from the data source? We define the *duplicate data ratio* as the ratio of the duplicate data sent by the source to all the new data sent. To answer these questions, we have built a fluid model and derived the following theorems. (Details are presented in Appendix.)

THEOREM 1. *Datacast works at the full rate, i.e., the rate of the slowest receiver, R, if the cache size, C, satisfies*

$$C > \frac{R^2 T}{2\delta}$$

THEOREM 2. *When Datacast works at the full rate, the duplicate data ratio of Datacast is*

$$\frac{\frac{\delta}{T}}{\frac{\delta}{T} + \frac{R^2}{2MTU}}$$

Theorem 1 tells us Datacast works at the full rate when the cache size is greater than $\frac{R^2T}{2\delta}$. For example, when δ = 5Mbps, T = 1ms, and R = 100Mbps, Datacast works at the full rate when the cache size C is larger than 125KB. Theorem 2 reveals the bandwidth efficiency of Datacast. In the above example, the duplicate data ratio is 1.19%. Theorem 1 and 2 tell us that Datacast can achieve the goal of high bandwidth efficiency, and at the same time it also meets the requirement of using small cache size in the intermediate devices.

6. SIMULATION

In this section, we use simulations to study Datacast. First, we evaluate our multiple Steiner trees algorithm. Second, we design micro benchmarks to study Datacast congestion control algorithm. Third, we compare the performance of Datacast with the most widely used P2P overlay, BitTorrent.

6.1 Evaluation of the multiple Steiner trees algorithm

To study the performance of the multiple Steiner trees algorithm, we use a Dell PowerEdge R610 server, which has two E5520 Intel Xeon 2.26GHz CPU and 32GB RAM. We study our algorithm under three topologies, Fattree(24, 3), BCube(8, 3) and Torus(16, 3). The BCube and Torus contain 4096 servers, while the Fattree contains 3456 servers. For each simulation, we randomly generate link failures. The link failure rates (LFR) include 1%, 3% and 5%. We ignore the cases when the network is not connected.

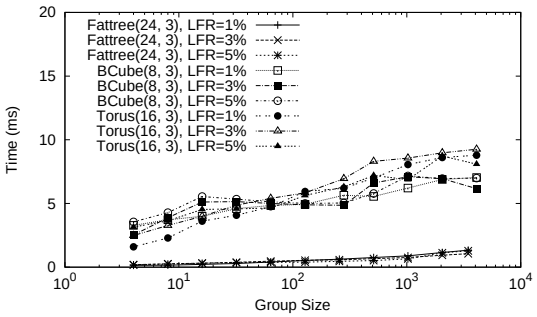


Figure 8: Running times of our Steiner tree algorithm.

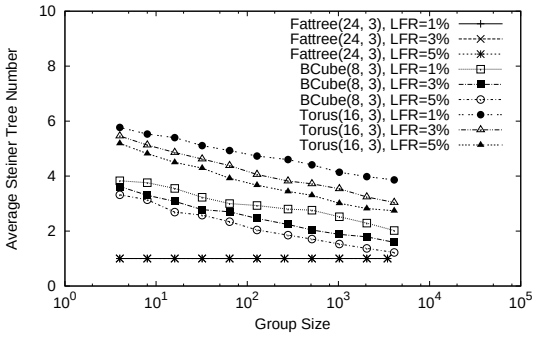


Figure 9: The numbers of Steiner trees with different failure rates and group sizes.

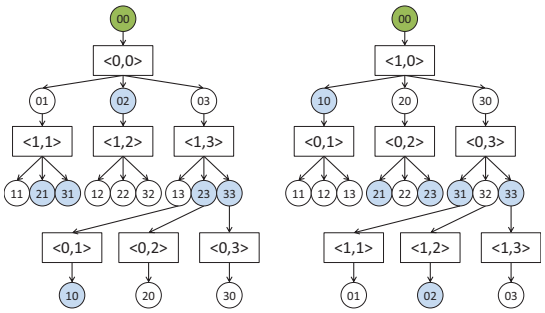
6.1.1 Running time

Figure 8 shows the running times of our algorithm. From the results, we can see that our algorithm can finish all of the tree calculations within 10ms.

We compared our algorithm with the generic algorithm which first calculates the spanning trees using Po's algorithm [24], then prunes them to get Steiner trees. The time complexity of the generic algorithm is dominated by the spanning tree calculation. The times needed for calculating spanning trees for Fattree(24, 3), BCube(8, 3) and Torus(16, 3) are 1, 39 and 42 seconds respectively. This algorithm therefore cannot be used in Datacast.

6.1.2 Steiner tree number

Figure 9 shows the numbers of Steiner trees constructed by our algorithm. For BCube and Torus, the numbers of Steiner trees decrease as the group size and the link failure rate increase. This is expected, since



(a) Steiner Tree 1 (b) Steiner Tree 2

Figure 10: The simulation and experiment set-up. The green node, 00, is the source, while the blue ones, 02, 10, 21, 23, 31 and 33, are the receivers.

a large group would experience more link failures, and more link failures will break more trees. Though Fat-tree has only one Steiner tree, our algorithm helps on failure recovery when the original tree is broken by link failures.

To check whether our algorithm can create enough Steiner trees, we have derived a bound of the Steiner tree number, which is the minimum value of the out-degree of the source and the in-degrees of all the receivers. The Steiner tree numbers produced by our algorithm are only 0.8% less than the bounds on average.

6.1.3 *Steiner tree depths*

Our algorithm also guarantees small tree depths. For example, when the link failure rate is 1%, the average Steiner tree depths for BCube, Torus and Fattree, are 9.99, 24.31 and 6.00, respectively.

6.2 Micro benchmarks for Datacast congestion control algorithm

We have built Datacast in NS3. In this subsection, we use micro benchmarks to study Datacast congestion control algorithm in a BCube(4, 1). We use a single multicast tree shown in Figure 10(a). The green node, 00, is the source, while the blue ones, 02, 10, 21, 23, 31 and 33, are the receivers. δ = 5Mbps, T = 1ms and MTU = 1.5KB. The link rates are 1Gbps, and the propagation delays are 5us. We slow down the link from switch <0,0> to node 02 to 100Mbps to make node 02 the slowest receiver. The queue size for each link is 100 packets. The headers of the interest and data packets are both 16 bytes. The initial rate of the source is 500Mbps.

6.2.1 *Efficiency study*

We first verify Theorem 1. We vary the cache sizes from 8KB to 2048KB. Based on Theorem 1, Datacast works at the full rate when the cache size is larger than 125KB. The simulation results are shown in Table 1. From the results, we can see that Datacast works at the full rate when the cache size is larger than 125KB. Its throughput, 98.799Mbps, is very close to the optimal results, which is 98.933Mbps (= 100Mbps $\times \frac{1500-16}{1500}$). The table also shows that the throughput of Datacast degrades gracefully when the cache size is smaller than 125KB.

Cache Size (KB)	Throughput (Mbps)	Duplicate Data Ratio (%)
8	91.380	1.15
32	95.076	1.14
128	98.799	1.11
512	98.799	1.10
2048	98.799	1.12

Table 1: Datacast's performance under different cache sizes.

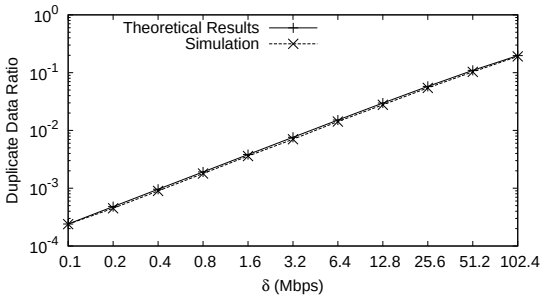


Figure 11: Duplicate data ratio vs. δ.

The table shows that the duplicate data ratio is about 1.12%. This is close to the result produced by Theorem 2, 1.19%. It also shows that the duplicate data ratio does not depend on the cache size once Datacast works at the full rate. To examine the accuracy of Theorem 2, we also vary the rate increase, δ, from 0.10Mbps to 102.40Mbps. From the results shown in Figure 11, we can see that the duplicate data ratio derived from our model is consistent with the simulation results.

6.2.2 *Performance under packet losses*

To see whether Datacast is resilient to packet losses, we randomly drop data packets at the link from switch <0,0> to node 02. The packet drop rate ranges from 0.001% to 4.096%. The cache sizes are set to 128KB. The results in Figure 12 shows that Datacast is quite resilient to packet losses. Even when the packet loss rate is 1.02%, the finish time only increases by 2.76% and the duplicate ratio is 1.23%.

6.2.3 *Fairness*

In this simulation, we set all the links back to 1Gbps. To study intra-protocol fairness, we set up multiple Datacast groups. The first group, which is shown in Figure 10(a), starts at time 0s. At time 10s, we start three new groups, whose source is node 00 and receivers are

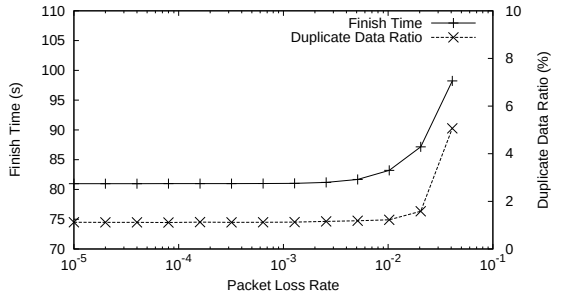


Figure 12: Datacast's performance under different packet loss rates.

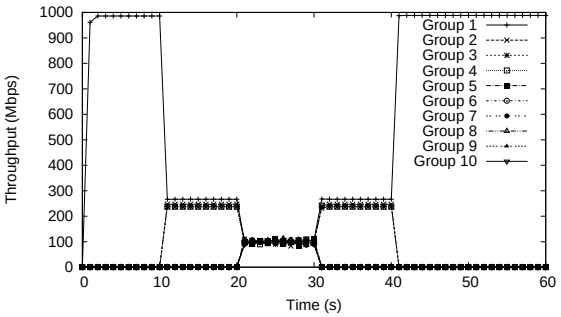
node 12, 22, 32. These three groups stop at time 40s. At time 20s, we start another six groups, whose source is node 00 and receivers are node 11 and 21. These groups end at time 30s. The ten groups share a congestion point at the link from node 00 to switch <0,0>. From the results shown in Figure 13(a), we can see that the first four groups equally get about 250Mbps throughput after time 10s, and the ten groups equally get about 100Mbps after time 20s. This means that Datacast congestion control algorithm achieves good intra-protocol fairness.

We also investigate whether Datacast congestion control algorithm is friendly to TCP. Similarly, we set up a Datacast group shown in Figure 10(a), which starts at time 0s. At time 10s, three TCP connections from node 00 to node 02 start. The three connections end at time 40s. At time 20s, six TCP connections from node 00 to node 01 start, which end at time 30s. The Datacast and TCP connections congest at the link from node 00 to switch <0,0>. The results are shown in Figure 13(b), which suggest that Datacast congestion control algorithm has good inter-protocol fairness with TCP.

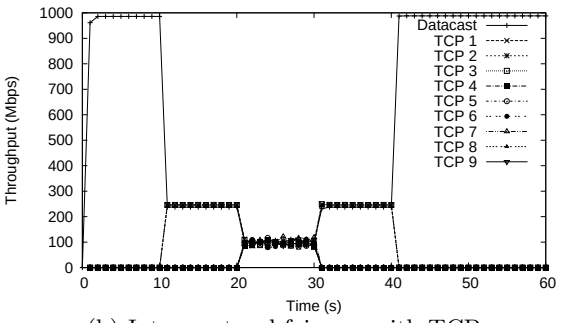
Datacast achieves good inter-protocol fairness with TCP, since their additive increase parts are at the same magnitude. In this simulation, we measure that the RTT of TCP is about 1ms when there are nine TCP flows and one Datacast group. TCP increases its rate at the speed of 12Mbps ($= \frac{MTU}{RTT}$) per RTT (1ms), while Datacast increases its rate at the speed of 5Mbps per millisecond. Therefore, Datacast and TCP achieve good inter-protocol fairness.

6.2.4 *Cache replacement algorithms*

We study the performance of Datacast when different cache management policies are used. We find that the cache management algorithms affect the duplicate data ratio. We evaluate three representative cache management algorithms, Least Recently Used (LRU), Least Frequently Used (LFU) and First In First Out (FIFO). The three algorithms differ in the replacement of cache items. When we need to replace an old item with a new one, LRU replaces the one that has been least used recently, LFU replaces the one that has the lowest used frequency, and FIFO replaces the one that first enters the cache.



(a) Intra-protocol fairness.



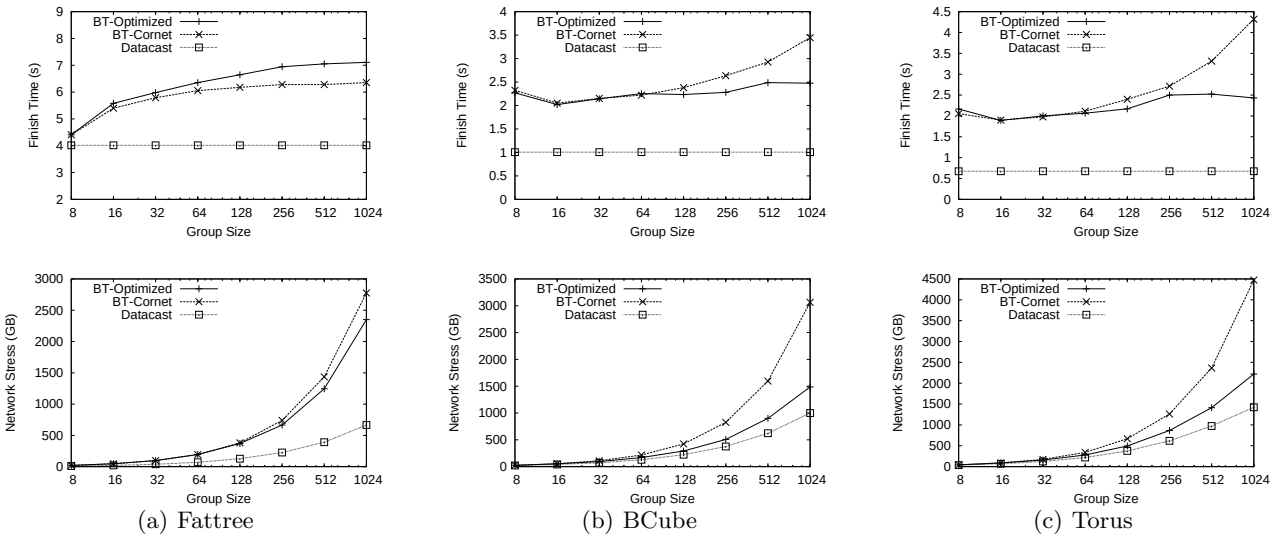
(b) Inter-protocol fairness with TCP.

Figure 13: Intra-protocol and inter-protocol fairness.

The cache miss ratios for LRU, LRU and FIFO are 3.90%, 1.63% and 1.12%, respectively. LRU and LFU cause larger duplicate data ratios due to the following reason. When the slowest receiver cannot keep up with the fast ones, its interests will be sent to the data source as duplicate interests, and then the source slows down its sending rate. During the process that the slowest receiver is catching up with the fast ones, it accesses the old data packets in the caches and makes its next data packet as the least recently (or frequently) used when LRU (or LFU) is used. So if the source sends out a new packet for the fast receivers, it will erase the next data packet for the slowest receiver in intermediate devices' caches, resulting in more cache misses and duplicate data.

6.3 Performance comparison

BitTorrent was originally designed for P2P file sharing in the Internet. Since a data center is a collaborative environment and the network topology can be known in advance, we use techniques similar to Cornet [10] to improve the original BitTorrent. Cornet improvements include: a server does not immediately leave the system after it receives all the content; no SHA1 calculation per block; use large block size (4MB). Cornet suggests using



(a) Fattree (b) BCube (c) Torus

Figure 14: Performance comparison of Datacast and BitTorrent.

large block size (4MB). Our simulations demonstrate that smaller block size results in better performance. We choose 108KB as the block size in the simulations. We call the Cornet optimized version BT-Cornet. Similar to Cornet, we also consider the topology awareness. Since the topologies we use have rich topological information, we design the following neighbor selection algorithm: a server selects 10 peers (when the group size is less than 10, all the members are peers). It sorts the group members via the distance. It prefers peers with a small distance, but guarantees that at least one member (if it exists) is selected as its peer at each distance range. Similar to Cornet, tit-for-tat and choke-unchoke are disabled. We call the optimized version BT-Optimized.

We use two metrics for the comparison. The first metric is the network stress, which is the sum of all the bytes transmitted on all the links. The second is the finish time.

In all the simulations, the source sends 500MB data. Figure 14 shows the performances of Datacast, BT-Cornet, BT-Optimized under different group sizes for three different topologies, Fattree(24, 3), BCube(8, 3) and Torus(16, 3). The group size varies from 8 to 1024. Our results clearly demonstrate that Datacast is better than BT-Cornet and BT-Optimized in terms of the network stress of the finish time. On BCube and Torus, Datacast is much faster since each server has multiple 1GbE ports. In all the simulations, the network stresses of BT-Optimized are 1.2-3.5X than Datacast, and Datacast is 1.1-3.7X faster than BT-Optimized.

We also note that in our simulations, when the topology is Fattree, the finish time with BT-Cornet is smaller than with BT-Optimized. This is because with BT-Optimized, we prefer peers that are close with each other. This preference may result in small cliques which may not be fully connected. BCube does not have such an issue because its structure does not have hierarchy.

In the experiments, Datacast's finish times are quite close to the ideal cases. There is one Steiner tree in Fattree(24, 3), and there are four Steiner trees in BCube(8, 3), and six in Torus(16, 3). Therefore the ideal finish times are 4s, 1s and 0.67s for Fattree(24, 3), BCube(8, 3) and Torus(16, 3), respectively. The finish times of Datacast are 0.67% larger than the ideal cases on average. Datacast is also efficient. The average link stress of Datacast is only 1.002, which means that each packet only traverse each Steiner tree link 1.002 times on average.

7. IMPLEMENTATION

7.1 ServerSwitch based implementation

We have implemented Datacast using the design shown in Figure 2. Fabric Manager, Master, data source and receivers are all implemented as user-mode applications. Each node in the data center runs a Datacast daemon, which is responsible for forwarding and receiving signalling messages. When Datacast is trying to start a group for data transmission, it first starts a Master process. The Master process calculates multiple Steiner trees, and then sends signalling messages to the group members. The daemons on these nodes will start the data source process and the receiver processes. Then the transmission starts.

To cache data packets in intermediate nodes, we use the ServerSwitch platform [8]. ServerSwitch 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's switching chip is programmable. It uses a TCAM table to define operations for specific types of packets. To implement data packet caching in switches, we use User Defined Lookup Keys (UDLK) to forward data packets to the Datacast kernel mode driver at branch nodes. The driver is used to do the in-network data packet caching. At non-branch nodes, the data packets are directly forwarded by hardware.

7.2 Evaluation

In this subsection, we use our real testbed implementation to evaluate Datacast. We use a BCube(4, 1) with 1Gbps links for our study. 8KB jumbo frame is used in the experiment.

7.2.1 *Efficiency study*

We study Datacast's performance when different cache sizes are set for branching nodes. We use a single Steiner tree shown in Figure 10(a) and slow down the link from switch <1,3> to node 23 to 100Mbps. We let $\delta = 5$Mbps and $T = 2$ms. Based on Theorem 1, Datacast works at the full rate when the cache size is larger than 256KB. When we use 64KB cache, the average throughput is 91.998Mbps, which is still acceptable due to the graceful throughput degradation of Datacast. When the cache size is 256KB, the average throughput is 99.595Mbps, and the duplicate data ratio is 3.48%, which is close to the theoretical result of Theorem 2, 3.10%.

7.2.2 *Performance comparison*

We compare the performance of Datacast with BitTorrent (we use μtorrent). In this experiment, we use both Datacast and BitTorrent to transfer 4GB data. The cache size on each branch node is 512KB. For Datacast, $\delta = 125$Mbps and T = 1ms.

Datacast finishes the transmission within 16.9s. The source achieves 1.89Gbps throughput on average, which is close to the 2Gbps capacity of the two 1GbE Steiner trees. The link stress of Datacast is 1.01. This means that Datacast achieves high bandwidth efficiency, since each packet only traverses each Steiner tree link 1.01 times on average. We compare Datacast with BitTorrent. Using BitTorrent, the receivers finish the downloading in 41-52s, and the link stress is 1.39. So BitTorrent is 2.75 times slower than Datacast on average, while its link stress is 1.38 times larger.

7.2.3 *Failure handling*

To study the failure handling of Datacast, we manually tear down the slow link. Our Fabric Manager detects the link failure in 483ms, and then notifies all the Masters. The Master uses the signalling protocol proposed in Section 3 to deliver the signalling messages to all the receivers in 2.592ms. (As a comparison, using TCP to send the signalling messages to receivers in parallel takes 20.122ms.) Then the transmission continues.

8. RELATED WORK

RGDD is an important traffic pattern, which has been studied for decades. Existing solutions can be classified into two categories.

Reliable IP multicast. The design space of reliable IP multicast has been nicely described in [11]. IP multicast has scalability issues for maintaining a large number of group states in the network. Adding reliability to IP multicast is also hard due to the ACK implosion problem [12].

We compare Datacast with two representative reliable multicast systems: pgm congestion control (pgmcc) [21] and Active Reliable Multicast (ARM) [25]. Pgmcc needs to explicitly track the slowest receiver for congestion control, and the congestion control protocol needs to be run between the sender and the slowest receiver. Datacast does not need to track which receiver is the slowest. This is because Datacast uses the duplicate interest packets as congestion signals, hence congestion control becomes the local action of the sender. ARM uses the active network concept and network devices also cache packet, but the cached packets are used only for re-transmission. Hence most likely the cached packets will not be used even once. Furthermore, retransmitted packets are broadcasted along the whole sub-tree in ARM, whereas they are delivered only to the needed receivers in Datacast.

End-host based overlay system. End-host based overlay system overcomes the scalability issue by transmitting data among peers. No group states are needed in network devices, and reliability is easily achieved by directly using TCP. It is widely used in the Internet. However, end-host based overlay systems suffer from low bandwidth efficiency. For example, the worst-case link stress of SplitStream can be tens [3], and the average and worst-case link stresses of End System Multicast (ESM) [18] are 1.9 and 9, respectively.

Recently, in the work of Orchestra [10], Cornet is proposed, which is an optimized version of BitTorrent for DCNs. Different from the distributed manner of Cornet, Datacast is a centralized approach. Due to the fact that a data center network is built and managed by a single organization, centralized designs become possible (e.g., software-defined networking [17]). Due to its centralized nature, Datacast is able to utilize multiple Steiner trees for data delivery, and achieve minimum finish time. Since the routing path from a receiver to data source is predetermined, high cache utilization is achieved. Furthermore, as we have demonstrated in the paper, the intermediate device only needs to maintain small cache per Steiner tree. All these benefits are hard,

if not totally impossible, to be achieved by distributed approaches like Cornet.

9. CONCLUSION

In this paper, we have presented the design, analysis, implementation and evaluation of Datacast for RGDD in data centers. Datacast first calculates multiple edge-disjoint Steiner trees with low time complexity, and then distributes data among them. In each Steiner tree, by leveraging in-network packet caching, Datacast uses a simple, but effective congestion control algorithm to achieve scalability and high bandwidth efficiency.

By building a fluid model, we show analytically that the congestion control algorithm uses small cache size for each group (e.g., 125KB), and results in few duplicate data transmissions (e.g., 1.19%). Our analytical results are verified by both simulations and experiments. We have implemented Datacast using the ServerSwitch platform. When we use Datacast to transmit 4GB data in our 1GbE BCube(4, 1) testbed with two edge-disjoint Steiner trees, the link stress is only 1.01 and the finish time is 16.9s, which is close to the 16s lower bound.

10. REFERENCES

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

[2] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *SIGCOMM*, 2008.

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

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

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

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

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

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

[10] M. Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra. In *SIGCOMM*, 2011.

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

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

[13] Van Jacobson et al. Networking Named Content. In *CoNEXT*, 2009.

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

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

[16] 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.

[17] K. Greene. Special reports 10 emerging technologies 2009. MIT Technology Review, 2009. http://www.technologyreview.com/biotech/22120/.

[18] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A Case for End System Multicast. *IEEE JSAC*, Oct 2002.

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

[20] Force10 networks. Force10 s7000. www.force10networks.com.

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

[22] Shyue-Ming Tang, Jinn-Shyong Yang, Yue-Li Wang, and Jou-Ming Chang. Independent Spanning Trees on Multidimensional Torus Networks. *IEEE Trans. Computers*, Jan 2010.

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

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

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

[26] J. Widmer and M. Handley. TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification, Auguest 2006. RFC 4654.

[27] J.S. Yang, S.M. Tang, J.M. Chang, and Y.L. Wang. Parallel Construction of Optimal Independent Spanning Trees on Hypercubes. *Parallel Computing*, 33, 2007.

Notation	Meaning
t	The current time.
$x_s(t)$	The data sequence position of the data source.
$x_r(t)$	The data sequence position of the slowest receiver.
R	The rate of the slowest receivers.
C	The size of the cache (the content store).
MTU	The size of a full Datacast data packet.
δ, T	The two parameters of Datacast congestion control, which are proposed in Section 5.2.
t_a	The start time of state 0.
t_b	The end time of state 0, and the start time of state 1.
t_c	The end time of state 1.
$\Delta x(t)$	$x_s(t) - x_r(t)$

Table 2: Notations used in the fluid model.

APPENDIX

A. PROOF OF THE DATACAST THEOREMS

We build a fluid model to analyze the performance of Datacast. We make the following assumptions: 1) The (desired[2]) rate of the slowest receiver, R, does not change over time. 2) Network latencies and queueing delays are negligible. In data center environment, network latency is small and around several hundreds of microseconds. 3) The credit number w is large enough to saturate the pipe. Table 2 shows the notations that are used in the analysis. Our fluid model can be de-

[2]Here "desired" means that the rate of the slowest receiver is not constrained by the sending rate of the data source.

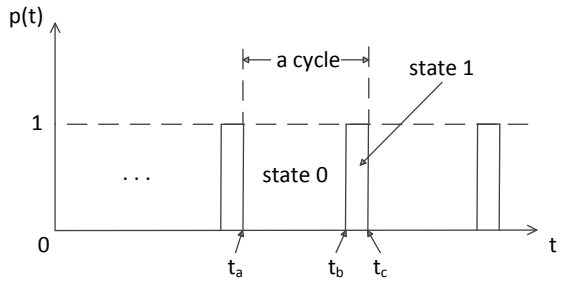


Figure 15: An illustration of the state changes in Datacast.

scribed by the following equations:

$$x_s''(t) = (1 - p(t))\frac{\delta}{T} - p(t)\frac{x_s'(t)}{2}\frac{x_r'(t)}{MTU} \quad (1)$$

$$x_r'(t) = \begin{cases} R & \text{if } x_r(t) < x_s(t) \\ \max\{R,\ x_s'(t)\} & \text{if } x_r(t) = x_s(t) \end{cases} \quad (2)$$

$$p(t) = \mathbb{1}_{\{x_s(t) - x_r(t) > C\}} \quad (3)$$

In this model, Equation (2) captures the slowest receiver's (actual) rate. When the source is ahead of the slowest receiver, the slowest receiver's rate is R. When the slowest receiver catches up with the source, its rate is constrained by both the source's rate and R. Equation (3) is an indicator function. $p(t) = 1$ when the data source receives a duplicate interest, otherwise $p(t) = 0$. Equation (1) models the rate control at the data source. $\frac{\delta}{T}$ captures a constant rate increase δ in every time period T if there is no duplicate interest. The second term is the rate decrease when duplicate interests are received (i.e., $p(t) = 1$). When $p(t) = 1$, the data source receives one duplicate interest from the slowest receiver in every time period $\frac{MTU}{x_r'(t)}$, and decreases its sending rate by half. The decreasing rate therefore is $\frac{x_s'(t)}{2} / \frac{MTU}{x_r'(t)} = \frac{x_s'(t)}{2}\frac{x_r'(t)}{MTU}$.

We say the system is in **state 0** when $p(t) = 0$, in **state 1** when $p(t) = 1$. It is easy to see that the system will oscillate between the two states, since $x_s''(t) > 0$ in state 0, and $x_s''(t) < 0$ in state 1. We call it a **cycle** from the start of state 0 to the end of state 1. Figure 15 gives us an illustration of how state changes in Datacast.

Proof of Theorem 1:

Proof. We first prove that if $C > \frac{R^2T}{2\delta}$, the rate of the slowest receiver is R, i.e., $x_r'(t) = R$. To prove that, we first prove $\Delta x(t) > 0$. It is easy to see it holds in state 1, since it is $\Delta x(t) > C$ in state 1. In state 0, we have $x_s''(t) = \frac{\delta}{T}$, and we can calculate

$$x_s'(t) = x_s'(t_a) + \frac{\delta}{T}(t - t_a)$$

From Equation (2), we have $x_r'(t) \le R$. Then we can derive

$$\Delta x'(t) \ge x_s'(t_a) + \frac{\delta}{T}(t - t_a) - R \quad (4)$$

Based on Inequality (4) and $\Delta x(t_a) = C$, we can derive

$$\Delta x(t) \ge \frac{\delta}{2T}(t - t_a)^2 + (x_s'(t_a) - R)(t - t_a) + C \quad (5)$$

The right side of Inequality (5) achieves its minimum value when $t = t_a + \frac{T}{\delta}(R - x_s'(t_a))$, and $t_a + \frac{T}{\delta}(R - x_s'(t_a))$ is in the region of $(t_a,\ t_b)$. It is greater than t_a, since $x_s'(t_a) < R$. It is lower than t_b, since $t_b = t_a + \frac{T}{\delta}(x_s'(t_b) - x_s'(t_a))$ and $x_s'(t_b) > R$. Put it into Inequality (5), we get

$$\begin{aligned} \Delta x(t) &\ge \frac{\delta}{2T}(\frac{T}{\delta})^2(R - x_s'(t_a))^2 - \frac{T}{\delta}(R - x_s'(t_a))^2 + C \\ &= C - \frac{T}{2\delta}(R - x_s'(t_a))^2 \\ &\ge C - \frac{R^2T}{2\delta} \end{aligned}$$

Since $C > \frac{R^2T}{2\delta}$, we have $\Delta x(t) > 0$ in state 0. $\Delta x(t)$ is therefore always greater than 0 in both states.

Putting $\Delta x(t) > 0$ into (2), we get $x_r'(t) = R$, which means that the slowest receiver's rate is not slowed down. We can further prove that the average sending rate of the data source will converge to R (which is omitted due to the space limitation), i.e., Datacast works at the full rate when $C > \frac{R^2T}{2\delta}$. □

Theorem 1 provides a sufficient condition to guarantee $x_r'(t) = R$. When C is not large enough, $x_r'(t)$ can be constrained by $x_s'(t)$ in state 0. However, $x_s'(t)$ will grow at a constant speed, $\frac{\delta}{T}$. $x_s(t)$ will soon be greater than $x_r(t)$, which means that the slowest receiver's rate is back to R. Even when C is not large enough, the system will experience graceful performance degradation instead of abrupt performance changes, as we have observed in the simulations and experiments.

Proof of Theorem 2:

Proof. The duplicate ratio can be calculated as

$$\frac{(t_c - t_b)R}{x_s(t_c) - x_s(t_a)}$$

when Datacast works at full rate, i.e., $x_r'(t) = R$. $(t_c - t_b)R$ is the amount of duplicate data that the slowest receiver requested in state 1, while $x_s(t_c) - x_s(t_a)$ is the amount of new data sent from the source in the whole cycle. Noticing that $\Delta x(t_a) = \Delta x(t_c) = C$, we have $x_s(t_c) - x_s(t_a) = x_r(t_c) - x_r(t_a)$. Since $x_r'(t) = R$, $x_r(t_c) - x_r(t_a) = (t_c - t_a)R$. The duplicate data ratio can be calculated as

$$\frac{t_c - t_b}{t_c - t_a} \quad (6)$$

To calculate (6), we first derive the durations of two states.

In state 0, $x_s''(t) = \frac{\delta}{T}$ and $x_r'(t) = R$, based on which we get

$$\Delta x(t) = \frac{\delta}{2T}(t - t_a)^2 + (x_s'(t_a) - R)(t - t_a) + C \quad (7)$$

In state 0, since $x_s'(t)$ increases linearly, we have

$$x_s'(t_b) = x_s'(t_a) + \frac{\delta}{T}(t_b - t_a) \quad (8)$$

Combining $\Delta x(t_b) = C$ and Equation (7) and (8), we get

$$x_s'(t_b) = 2R - x_s'(t_a) \quad (9)$$

Putting Equation (9) back into Equation (8), we have

$$t_b - t_a = \frac{2(R - x_s'(t_a))}{\frac{\delta}{T}} \quad (10)$$

In state 1, $x_s''(t) = -\frac{x_s'(t)}{2}\frac{x_r'(t)}{MTU}$ and $x_r'(t) = R$, based on which we can derive

$$\Delta x(t) = \frac{2MTU}{R}x_s'(t_b)(1 - e^{-\frac{R}{2MTU}(t-t_b)}) - (t - t_b)R + C \quad (11)$$

In state 1, since $x_s'(t)$ decreases exponentially, we have

$$x_s'(t_c) = x_s'(t_b)e^{-\frac{R}{2MTU}(t_c - t_b)} \quad (12)$$

Combining Equation (9), (11) and (12), we derive

$$t_c - t_b = \frac{2R - x_s'(t_a) - x_s'(t_c)}{\frac{R^2}{2MTU}}$$

The end of state 1 is also the start of state 0 in the next cycle. In stable state, $x_s'(t_c)$ and $x_s'(t_a)$ are the same, we thus have

$$t_c - t_b = \frac{2(R - x_s'(t_a))}{\frac{R^2}{2MTU}} \quad (13)$$

Combining (6) with the durations of the two states, (10) and (13), the duplicate ratio is

$$\frac{t_c - t_b}{t_c - t_a} = \frac{\frac{2(R-x_s'(t_a))}{\frac{R^2}{2MTU}}}{\frac{2(R-x_s'(t_a))}{\frac{R^2}{2MTU}} + \frac{2(R-x_s'(t_a))}{\frac{\delta}{T}}} = \frac{\frac{\delta}{T}}{\frac{\delta}{T} + \frac{R^2}{2MTU}}$$

□