Ditto - A System for Opportunistic Caching in Multi-hop Wireless Networks

Fahad R. Dogar, Amar Phanishayee, Himabindu Pucha, Olatunji Ruwase, David G. Andersen

Carnegie Mellon University

{fdogar, amarp, hpucha, oor, dga}@cs.cmu.edu

ABSTRACT

This paper presents the design, implementation, and evaluation of Ditto, a system that opportunistically caches overheard data to improve subsequent transfer throughput in wireless mesh networks. While mesh networks have been proposed as a way to provide cheap, easily deployable Internet access, they must maintain high transfer throughput to be able to compete with other last-mile technologies. Unfortunately, doing so is difficult because multi-hop wireless transmissions interfere with each other, reducing the available capacity on the network. This problem is particularly severe in common gateway-based scenarios in which nearly all transmissions go through one or a few gateways from the mesh network to the Internet.

Ditto exploits on-path as well as opportunistic caching based on overhearing to improve the throughput of data transfers and to reduce load on the gateways. It uses content-based naming to provide application independent caching at the granularity of small chunks, a feature that is key to being able to cache partially overheard data transfers. Our evaluation of Ditto shows that it can achieve significant performance gains for cached data, increasing throughput by up to 7x over simpler on-path caching schemes, and by up to an order of magnitude over no caching.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Design, Performance, Experimentation

Keywords

Multi-hop Wireless Networks, Mesh Networks, Opportunistic Caching, Performance, Throughput

MobiCom'08, September 14–19, 2008, San Francisco, California, USA.

1. INTRODUCTION

Wireless mesh networks are an appealing way to provide Internet access because of their potential low cost and easy deployment. A key challenge in mesh networks is maintaining high transfer throughput. Unfortunately, doing so is difficult because these networks inherently suffer from interference due to the broadcast nature of the wireless medium. This interference manifests itself in several ways: First, increasing the number of hops from source to destination decreases the path throughput because subsequent hops interfere with each other. Second, the probability of loss increases with the number of hops, which can drastically reduce TCP throughput. Third, a common communication pattern in a mesh network exacerbates the situation: Every mesh router obtains Internet access by communicating with a gateway node. Congestion thus increases near the gateway as the medium around the gateway becomes a hot-spot.

This paper presents Ditto[1], a system that improves transfer performance in mesh networks by opportunistically caching data both at nodes *on the path* of a transfer and at nodes that *overhear* the data transfer. This caching is exploited through Ditto-enabled mesh routers, which act as proxies for data requests from clients, satisfying requests from cache when possible and otherwise passing those requests on to the next hop.

Ditto targets traditional applications for content caching (popular Web content, large software and operating system updates, and large data downloads such as those on peer-to-peer networks. etc.) as well as emerging applications such as high-bandwidth video downloads. In this regard, Ditto is similar to much prior work in caching and content delivery, and its target applications and expected cache hit rates are largely the same. Because it operates on lossy lower bandwidth wireless links, however, Ditto's potential performance benefits are much higher. As we show in Section 6, Ditto can improve throughput by up to an order of magnitude.

Ditto leverages content-based naming for opportunistic caching: data objects or parts of data objects are identified by their hash value. This naming mechanism allows Ditto to provide application independent caching at the granularity of *chunks*. Because chunks are typically small (8 – 32 KB), nodes are more likely to overhear a complete chunk compared to a whole file. This makes caching based on overhearing effective in a multi-hop wireless setting where losses would prevent overhearing complete files. Ditto furthermore uses multiple transmissions at different hops to create more op-

[1]The comic strip *Hi and Lois* features two twins, Dot and Ditto, from which our system's name is copied.

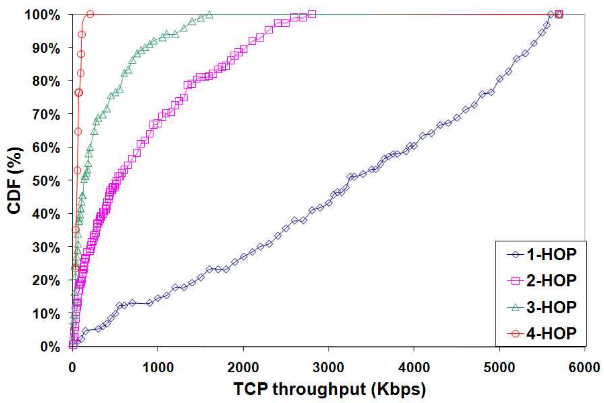


Figure 1: Impact of multi-hop transfers in a mesh network.

portunities for an overhearing node to reconstruct a chunk. To do so, Ditto uses inter-stream TCP reassembly at an overhearing node to construct a chunk that is being transmitted in multiple TCP streams.

We evaluate the effectiveness of Ditto by conducting experiments on two separate wireless network testbeds: MAP [1], a 28 node indoor/outdoor campus-wide wireless mesh testbed and Emulab's indoor 802.11b testbed. We conduct two sets of experiments: 1) analyzing chunk reconstruction efficiency by making each node in the testbed download a file from the gateway and measuring the proportion of chunks that were successfully reconstructed at every other node; and 2) measuring the throughput gain that is achieved by opportunistic caching in Ditto compared to i) no caching and ii) on-path caching by proxies.

Our results show that reconstructing chunks based on overhearing is feasible. Nodes are able to successfully reconstruct chunks of the order of 32KBs in size, although reconstruction efficiency goes down as the chunk size is increased. More specifically, our results show that 25% of the observers in the campus testbed and 50% in Emulab could reconstruct more than 50% of the total chunks transferred. This high reconstruction efficiency also results in significantly improved throughput: Ditto's opportunistic caching increased throughput by up to an order of magnitude compared to the no-caching scenario and 7× compared to on-path caching.

In the next two sections, we discuss the key challenges and opportunities that Ditto faces in the wireless environment (Section 2) and how it overcomes these challenges and exploits the available opportunities (Section 3). We then explain Ditto's design (Section 4) and implementation (Section 5). We present the evaluation of Ditto with respect to reconstruction effectiveness and throughput efficiency (Section 6). Section 7 summarizes the work most closely related to Ditto.

2. CHALLENGES AND OPPORTUNITIES FOR MESH DATA TRANSFERS

We begin by illustrating the challenge presented by increasing hop lengths in a wireless mesh network. Towards this end, we measured the impact of hop length on transfer throughput using a 28 node campus testbed.[2] In these measurements, each node initiated a TCP transfer to every other node using `netperf`. The transfers were conducted one-at-a-time with a five-second idle interval between sessions to avoid interference. The results below show only measurements from the 654 working paths.

Figure 1 shows the CDF of the TCP throughput obtained over all 1, 2, 3 and 4 hop paths in the network. The TCP throughput falls rapidly as the hop count grows. UDP transfers exhibit a similar drop, though their absolute throughput was higher than that of TCP. Our results agree with those observed in other testbeds (e.g., Roofnet [3]). Compounding the problem, one common access pattern in a mesh network consists of many mesh routers communicating with a few gateways to obtain Internet access. This creates hot-spots in the network and the resulting congestion further reduces transfer throughput.

Despite the challenges identified above, there are several opportunities that might be exploited to improve throughput in a mesh network. These opportunities include i) locality and similarity in the workload transferred over a mesh network, and ii) the opportunity to overhear and cache data transferred between other nodes, and possibly using it to improve the performance of a subsequent transfer.

Locality in a workload implies that multiple clients request the same data object over time or that multiple clients from different parts of the network request the same object at the same time. Workload locality is common on the Internet [5]. Because many mesh networks are used to provide Internet access (e.g., Meraki [16]), we believe that this locality will also be present in mesh networks. The recent "MeshCache" project [8] provides evidence of locality in Internet access workloads for client population typically found in a mesh network. Popular transfers such as software updates also create significant opportunities to exploit locality.

Similarity exists between parts of different data objects. This observation applies across general IP packets on network links [21], within email messages [9, 23], remote filesystem use [19, 24], software [7], Web pages [11, 18], and media files [20]. A transfer system that can opportunistically exploit data similarity at a granularity finer than that of an entire object has even more opportunities to exploit locality in access patterns in order to improve transfer performance.

3. DITTO OVERVIEW

Ditto improves throughput in mesh networks by using the broadcast nature of the wireless medium to exploit content similarity and access locality. Ditto does so using application independent caching at nodes on the path of a data transfer as well as at nodes overhearing the data transfer. While earlier efforts have looked at on-path caching to improve throughput in mesh networks [8], Ditto's novelty comes from the synergistic combination of caching using content-based chunk naming and caching *overheard* data. Ditto's caching can reduce the load on the gateway and decrease the average hop length of data transfers. Ditto's use of content-based naming allows it to aggressively use multiple opportunities of overhearing a given data transfer provided by data transmissions that span multiple hops and by retransmissions, turning sources of performance degradation into additional caching opportunities.

[2]details in Section 6.1

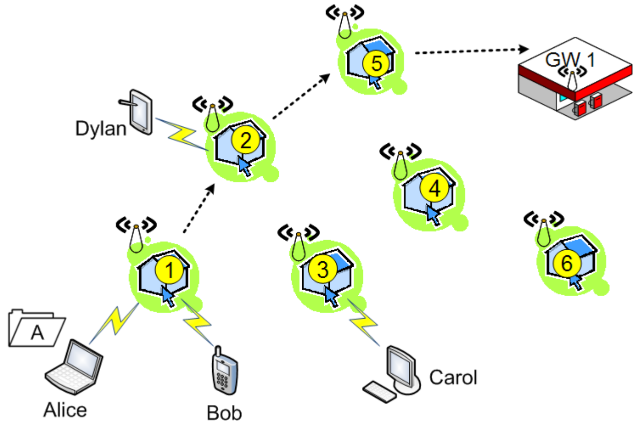


Figure 2: Example illustrating Ditto's effectiveness.

3.1 An Example Scenario using Ditto

We illustrate how Ditto can exploit locality and similarity to improve transfer performance using an example. The details of these mechanisms are presented in the following sections.

Consider the wireless mesh network shown in Figure 2. Alice requests a video object of the documentary "Planet Earth" in English using a p2p file sharing application. The request results in a cache miss in the network and is obtained via the gateway and returned to Alice. During this transfer, Ditto caches the video at router 1 (Alice's access mesh router), at routers 2 and 5 (which form the multi-hop path to the gateway) and at the gateway router GW 1. Ditto also overhears and caches part of the video at routers 3 and 4, which are in radio range of routers 2 and 5 respectively.

If Carol subsequently requests the same video file, her request can be satisfied by router 3, which overheard the transfer via router 2 to Alice earlier, instead of traveling three hops to the gateway node. Similarly, Bob and Dylan benefit from on-path caching at routers 1 and 2 if they request the same video.

Because Ditto uses application independent caching at the granularity of chunks, it can benefit a) transfers that are made by different applications: e.g. Bob might view "Planet Earth" through a web-app rather than using p2p software, and b) transfers that contain similarity *between* objects requested: e.g. Dylan views a trailer for the same documentary, possibly in a different language. In both cases, as the contents of the data being requested are either the same or similar to the data that Alice requested earlier, Ditto can improve transfers by identifying chunks in the requests that might have been cached at routers, thereby avoiding going all the way to the gateway for the entire data.

4. DITTO DESIGN

The core of Ditto's design is its mechanisms for proxying, caching, and overhearing data transferred through the mesh network. This design encompasses three key points: (1) The mechanism that nodes use to perform file transfers; (2) The way Ditto proxies service requests for data; and (3) The mechanisms that Ditto nodes use to cache overheard data. In our design of Ditto, we consider a multi-hop mesh network in which all nodes are Ditto-enabled. We assume that the sender and receivers use the same transfer mechanisms that Ditto does, or that they are suitably equipped with proxies that translate existing protocols (e.g., HTTP) into Ditto's protocol.

4.1 Ditto Data Transfers

In order to cache data at a fine granularity, Ditto uses existing content-based naming techniques for its data transfers. (As we explain further in Section 5, we use Data-Oriented Transfer, or DOT, for this purpose [23].) From our perspective, the pertinent features of content-based naming are:

1. **Chunk-based transfers**: Objects are split into smaller (2-64KB) chunks, and each chunk is transferred independently. A common technique is to define chunk boundaries using Rabin fingerprinting, a technique pioneered in the LBFS filesystem to increase the chances of finding identical chunks in similar files [19]. Before the actual transfer begins, the receiver typically obtains a list of chunks that make up the object it desires, and then requests each chunk in order.

2. **Hash-based chunk naming**: Chunks are named in a *self-verifying* manner by their content hash. As a result, a receiver can verify that it received the correct data for a chunk.

Each of these properties is important to Ditto's effectiveness. First, content-based naming allows Ditto proxies to be application independent, though those applications must be modified or proxied to work with Ditto's chosen transfer service.

Second, chunk-based transfers permit Ditto to operate at a much finer granularity than whole-object caching systems do. This property is critical in the lossy wireless environment: the chance of correctly overhearing the *entire* file decreases exponentially with the size of the file. While the direct receiver of the transfer benefits from both link-level and transport-level retransmissions to recover from losses, an overhearing node does not have the same luxury as its losses may not be correlated with losses at the direct receiver. It therefore is vanishingly unlikely that the overhearing node would properly reconstruct an entire large file. (This shrinking probability is also relevant to the choice of chunk size, a choice we evaluate in more detail in Section 6.)

Finally, Ditto leverages the self-verifying nature of content-based naming to ensure that its data transfers are correct even if it overhears a corrupted packet.

4.2 Ditto Proxies

Ditto transfers chunks using per-hop proxying. This mechanism is much like hierarchical web caching: Each Ditto proxy serves the data to its previous hop, either from its cache or by requesting it from its next-hop Ditto proxy. Each proxy caches all chunks that it transfers during this process. As we discuss in more detail below, the sniffer module also adds overheard chunks to the proxy's cache. Figure 3 outlines the Ditto proxy structure.

Locating data: One important design decision for the proxy architecture was how to locate data. Several nodes in the mesh might have cached a particular chunk. When a proxy needs to locate a chunk, from which other proxy should it request the data? We chose to design Ditto with the simple and robust choice of always requesting data from

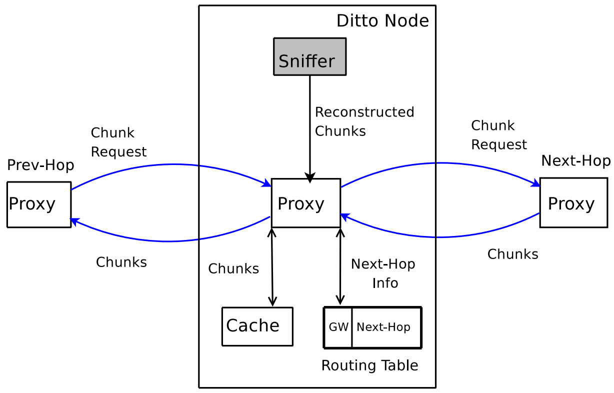


Figure 3: Ditto proxy design.

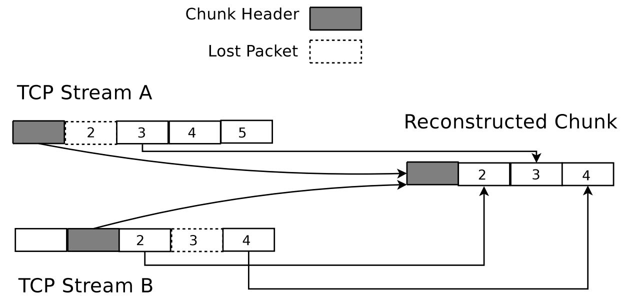


Figure 4: Inter-stream reassembly of chunks.

the next-hop proxy (the one through which traffic would go to reach the gateway). As a result, requests can never move "backwards" away from the ultimate source of the data. In effect, this choice turns Ditto's dissemination into a tree, rooted at the gateway, potentially sacrificing rare uphill caching opportunities for robustness and simplicity. Ditto obtains the identity of the next-hop by periodically querying the operating system for the next-hop route.

Per-hop TCP connections: A consequence of Ditto's per-hop proxying is that data is transferred over a series of different TCP connections, one per hop. As we discuss below, this makes the design of the Sniffer module more complicated. The module must be able to reassemble chunks where it hears part of the chunk on one TCP stream, and part of the chunk on a *different* TCP stream.

4.3 The Sniffer Module

The Ditto sniffer module constructs chunks from overheard data transmissions. It passes these reconstructed chunks to the proxy for caching. The sniffer must address three challenges: First, it must re-assemble overheard TCP streams. Second, it must identify data chunks within those streams. Finally, as noted above, it must be able to re-assemble chunks that were heard partly from one TCP stream and partly from another, a process we term *inter-stream reassembly*.

Stream and Chunk Reconstruction: The sniffer module reassembles TCP streams using standard techniques: it identifies each flow using the (`src, dst, src port, dst port`) tuple, and reassembles the flow based on its TCP sequence numbers. The streams, of course, are likely to be missing bytes, may have corrupted packets, and the sniffer must deal appropriately with retransmissions and duplicates.

As it creates a contiguous stream of the transfer, the Sniffer scans for chunk start markers in the stream. It passes this section of the reconstructed byte-stream to a module that decodes the transfer protocol and extracts the chunk. For this purpose, we modified the DOT protocol to include the chunk name and length at the beginning of its response, a feature that greatly simplifies the system's ability to take advantage of overheard data while adding only small additional overhead.

Inter-stream reassembly: The Sniffer module uses multiple TCP streams to reconstruct a chunk. In a multi-hop transfer, an overhearing node may have several chances to hear the same data as it traverses different hops. As noted above, however, these hops each use a different TCP stream, which complicates reassembly.

Figure 4 shows how inter-stream reassembly works in Ditto. The sniffer module identifies that the same chunk is being transmitted in two different streams by looking at the Ditto header, which contains the chunk id. Therefore, as a first step the sniffer needs to overhear the ditto header of a particular chunk in each stream in order to make use of inter-stream re-assembly. Once the chunk is identified, the sniffer only requires at least one of the streams to overhear a packet and it can use that to fill gaps in the chunk. For example, in Figure 4 TCP stream A missed packet 2 while TCP Stream B missed packet 3. In both cases, the other stream was able to contribute the missing packet towards chunk reconstruction. If a packet is received in multiple streams, as is the case with packet 4, only one copy of this needs to be maintained and the rest can be flushed from the memory. Therefore, Ditto's inter-stream re-assembly improves the chances of successfully reconstructing chunks and also reduces the memory overhead by eliminating similar content that is present in multiple streams.

5. IMPLEMENTATION

Ditto's implementation consists of two major parts: the DOT enabled proxies and the sniffer module.

5.1 DOT enabled Proxies

Ditto proxies use Data-Oriented Transfer, or DOT [23], as their data transfer protocol.[3] DOT uses an RPC-based protocol to transfer chunks across the network. We made one change to this transfer protocol to add the chunk ID and length at the beginning of every response. This information is unnecessary for the actual receiver, since it is known from context, but it greatly simplifies the process of reassembling chunks based on overhearing.

Figure 5 shows the steps involved in a typical DOT enabled file transfer in the presence of proxies. The receiver makes an application-level request for a file to the sender application. The sender application contacts its local DOT service to get the *OID* and *Hints* corresponding to the requested file. An "OID" is an object ID, which is simply the hash of the file being requested. The "Hints" tell the receiver who to ask for in order to retrieve the object in question. The OID and Hints information is returned back to the receiver. The receiver's

[3]Our implementation is based on the DOT snapshot `dot_snap_20070206`.

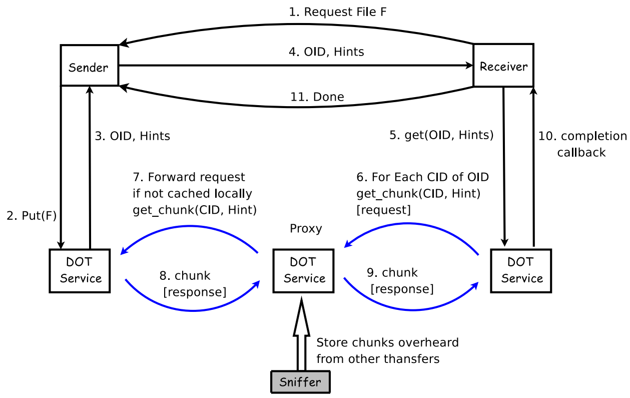


Figure 5: A file transfer in Ditto

DOT service gets the chunk IDs (*CID*s) corresponding to the OID (this step is not shown in the figure for simplicity). The DOT service at the receiver then requests the desired chunks from its next hop Ditto proxy. The proxy is a straightforward extension to DOT that checks the (existing) DOT cache for a chunk and, if not found, passes the request on to the next hop. This process is repeated until the chunk is found at some proxy/source. Once a chunk is received, each proxy stores it in its DOT cache and serves it to its upstream neighbor. Finally, once the data is received by the DOT service at the receiver, it is served to the receiver application.

5.2 Sniffer Implementation

Ditto's sniffer module is a stand-alone process that runs on the same host as a Ditto proxy module. It uses `libpcap` in promiscuous mode to overhear TCP packets and attempts to extract Ditto chunks from reconstructed TCP streams (flows). It sends overheard chunks to the Ditto proxy using RPC over a unix domain socket.

Data Structures: The chunk reconstruction algorithm maintains three main data structures, shown in Figure 6. The first is a table of overheard flows (`flowTable`), indexed by `flowID`. Corresponding to each flow, the `flowTable` includes overheard packets that do not belong to any chunk being currently reconstructed. We store these packets because of the possibility of re-ordering; a late arrival of a re-ordered Ditto header packet may make these packets useful.

The second data structure is a table of partially reconstructed chunks (`chunkTable`), which is indexed by `chunkID`. This table maintains information about all the `flowIDs` that are contributing to a chunk along with the sequence number of the first packet of the chunk in each flow. The third data structure, `flowChunkMap` maintains a list of all the `chunkIds` observed in each flow.

Flows and partially reconstructed chunks are represented as a linked list of blocks of contiguous bytes. Each block includes the following

- Offset of the first byte of the block within the chunk/flow.
- Length of the block.
- Actual bytes that make up the block.

Blocks are maintained in increasing order of block offsets. The block data structure makes it easy to support gaps in the flow or partially reconstructed chunks. Blocks are merged whenever possible. Chunk reconstruction is complete when

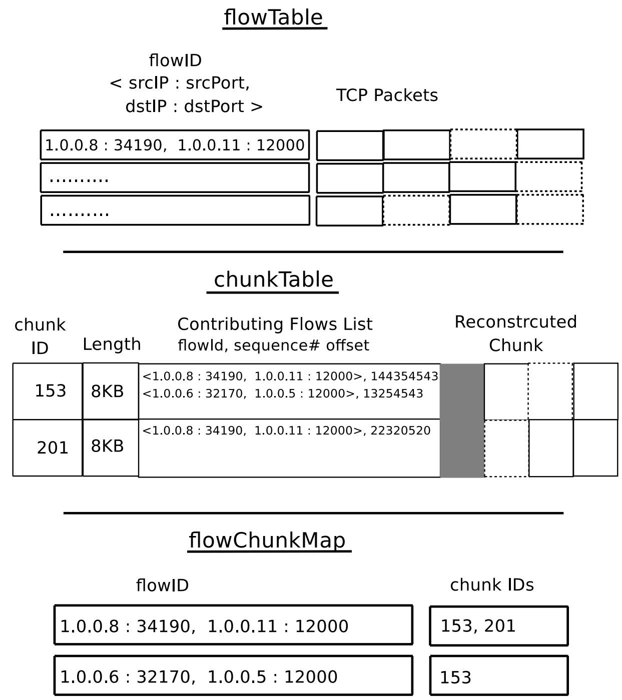


Figure 6: Data Structures used in Sniffer implementation

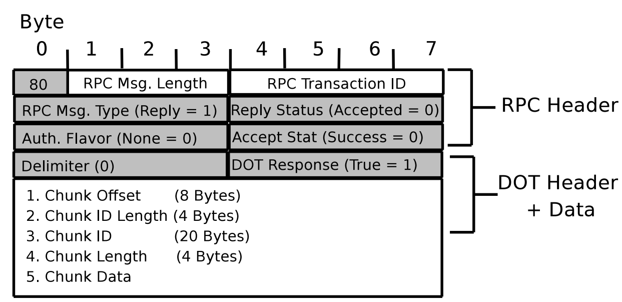


Figure 7: Ditto packet format. Shaded fields are invariant across Ditto responses and the Sniffer uses these to identify a Ditto chunk.

there remains only one block in the list which is of the length of the chunk. At this point, the chunk entry is removed from the `chunkTable` and the chunk is sent to the proxy module.

Packet Processing: The inter-stream re-assembly algorithm performs the following actions for each overheard packet. The packet is added to the corresponding flow in the `flowTable` entry if one exists, else a new flow is created. The newly modified (or created) flow is checked for the existence of a Ditto header. Figure 7 shows the format of a Ditto response. The shaded fields are used to identify a Ditto chunk within the stream. It suffices to check only the contiguous region between packets preceding and following the new packet. If a Ditto header is found, the *chunkID* is extracted, and if a corresponding chunk entry is found in the `chunkTable`, then the flow is added to the list of flows contributing to that chunk. If no corresponding entry is found, a new entry is created, and all the packets corresponding to the new chunk are copied from the `flowTable` into the `chunkTable` to form

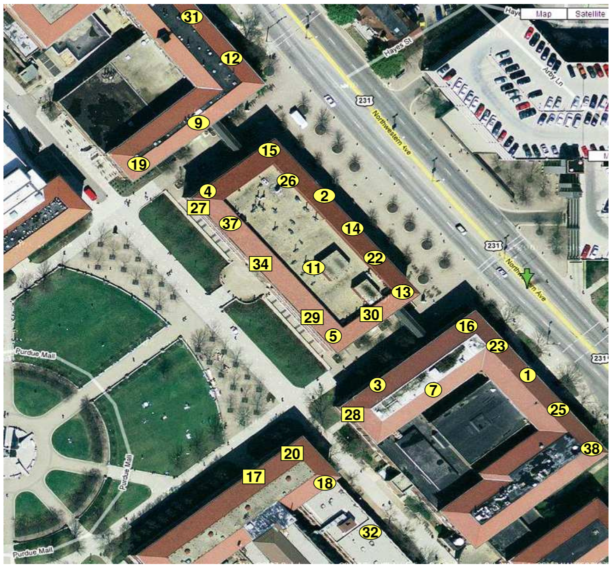


Figure 8: MAP testbed. Nodes represented with a circle are on the second floor while those with a square are on the third floor. Node 11 is the gateway node.

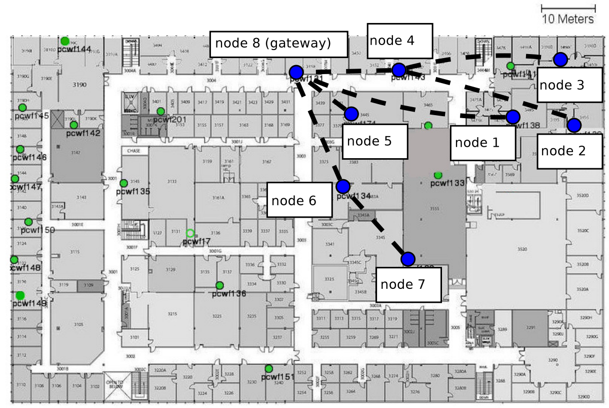


Figure 9: Emulab testbed. Nodes represented with a filled circle are on the third floor while those with a hollow circle are on the fourth floor. Nodes represented by dark circles are the nodes we used for our experiments and the dashed black lines indicate the routes taken by the nodes while communicating to the gateway node (node 8).

the partially reconstructed chunk. Since the Ditto header contains the length of the chunk, all the packets that belong to a chunk can be correctly identified within the flow.

If on the other hand, a Ditto header was not found, the packet could still aid in chunk reconstruction as follows. All the chunks to which the flow is contributing, obtained from the `flowChunkMap`, are scanned to identify the chunk the new packet belongs to. For each `chunkTable` entry, the length of the chunk and starting sequence number of the Ditto header corresponding to the flow are compared against the sequence number of the new packet to identify the relevant chunk. The chunk is then updated with the relevant bytes from the new packet. If the packet is unable to aid in chunk reconstruction immediately it is left in the `flowTable`.

The algorithm has the limitation that it requires that the Ditto header be overheard on any flow before that flow can contribute to chunk reconstruction. However it correctly handles the corner cases where a Ditto header spans two TCP packets and the length of the chunk is not a multiple of packet length.

6. EVALUATION

Our evaluation of Ditto shows that: (1) caching using opportunistic overhearing is practical and effective in wireless multi-hop networks, (2) the performance of chunk reconstruction is improved both by using smaller chunk sizes and by proximity to transfer paths, and (3) Ditto significantly improves transfer throughput in a real testbed deployment.

6.1 Method

Testbeds. We evaluate Ditto's performance using two wireless testbeds: (1) The MAP testbed at Purdue University [1] – a campus wireless mesh testbed, and (2) Emulab's indoor wireless testbed.

The campus testbed consists of 28 mesh routers spread across four academic buildings (Figure 8). The mesh nodes are small form factor desktops, with heterogeneous hardware specifications. Each node has two PCI-card radios—an Atheros 5212 based 802.11a/b/g wireless card and a Senao 802.11b card using the Prism2 chipset. Each radio is attached to a 2dBi rubber duck omni-directional antenna with a low loss pigtail. Each node runs Mandrake Linux 10.1 with the open-source madwifi and hostap drivers. IP addresses are statically assigned. The testbed deployment environment is not wireless friendly, having floor-to-ceiling office walls instead of cubicles as well as some laboratories with structures that limit the propagation of wireless signals. Apart from structural impediments, the testbed experiences interference from other 802.11b networks. We configured the testbed to use the Atheros radio in the 802.11b mode. We used channel 11 because it was the band furthest away from those being already used on the campus. This testbed has both indoor and outdoor links.

We use Emulab's wireless testbed [25] as our second testbed. This entirely indoor wireless testbed consists of 24 nodes located in different rooms across two floors. We picked 8 available nodes for our experiments as depicted in Figure 9. All nodes are "pc3000w" machines with a 3.0 GHz processor and 1 GB RAM. They are equipped with Atheros 5212 based 802.11 a/b/g wireless cards and are operated in 802.11b mode. Each node runs Fedora core 4 with the open-source madwifi driver.

In both networks, there is no other traffic in the mesh network apart from the traffic from our experiments. We use the wired network to control the experiments and collect logs. We use the OLSR routing protocol with distributed link quality information to setup routes between nodes. Once the routes are setup and all the nodes in the network can reach the gateway node, we terminate OLSR on all the nodes to avoid route flapping.

Metrics. To capture the effectiveness of overhearing and Ditto's chunk reconstruction schemes, we use three metrics:

(1) **Average reconstruction efficiency.** The average percentage of chunks completely reconstructed at every node across multiple runs. (2) **Memory overhead.** The average percentage unused bytes of total Ditto traffic overheard at each node. These unused bytes represent packets that could not be successfully used to reconstruct a chunk. Retaining these packets for as long as possible improves the opportunities to reconstruct the incomplete chunk at a later instant. Thus, the lower the memory overhead from a reconstruction scheme, the higher the chance of retaining more unused bytes. (3) **Throughput.** To demonstrate the performance improvement using Ditto over existing schemes, we present throughput across receivers for a given traffic pattern.

To capture summary statistics, we present the percentage of observers that achieve a given value of the metric. For every transfer between a given gateway-receiver pair, we define the remaining nodes in the testbed as **observers** that can passively overhear the ongoing communication. For example, in the 28-node campus testbed, a transfer between node 11 (gateway) and node 1 (receiver) has 26 observers. When we change the receiver to node 2, we have 26 new observers. For an entire experimental run with each node receiving once, we have $26 \cdot 27 = 702$ observers.

Schemes. We compare Ditto's performance to three other approaches:

- **End-to-end:** In this scheme, the transfer request from a receiver is sent directly to the the gateway, with intermediate nodes in the network acting as IP routers. The transfer throughput does not benefit from caching or overhearing.

  This scheme is valuable because it represents an ideal scenario for chunk reconstruction efficiency—because the sender and the receiver use a single end-to-end TCP connection, the same TCP packets for a given chunk transfer traverse multiple hops. As a result, a node that overhears the transfer more than once will observe the same packet multiple times; if it misses the opportunity to capture the packet on one of the attempts, it might capture it from the transmission on the following hops.
- **On-path caching:** The on-path caching scheme uses per-hop proxying similar to that in Ditto. However, the nodes in this scheme only cache chunks for which they are the intermediate hops, and not the chunks they overhear. This scheme is an improvement over an existing approach, MeshCache [8], which also benefits from on-path caching. While MeshCache's caching is on a per-file basis, our scheme provides chunk-level caching.
- **Ditto without inter-stream reassembly**: To evaluate the design choices in Ditto's chunk reconstruction, we compare Ditto with a variant that does not perform inter-stream reassembly. The overhearing node reconstructs chunks by *independently* analyzing each overheard TCP stream.

We transfer a 1 MB file and present the average across three runs unless otherwise specified. All experiments use node 11 as the gateway in the campus testbed and node 8 in the Emulab testbed. The routes to the gateway for the Emulab and MAP testbeds are shown in Figures 9 and 10 respectively.

```
node11---node30---node13---node1
node11---node37---node2
node11---node30---node3
node11---node37---node4
node11---node5
node11---node30---node29---node7
node11---node30---node13
node11---node14
node11---node22---node15
node11---node30---node13---node16
node11---node30---node28---node18
node11---node37---node4---node19
node11---node22
node11---node30---node13---node1---node25
node11---node34---node27
node11---node30---node28
node11---node30---node29
node11---node30
node11---node34
node11---node37
node11---node30---node13---node16---node38
```

Figure 10: Routes to the gateway node (node 11) in the MAP testbed.

6.2 Overhearing Effectiveness

This section characterizes the potential benefit from Ditto's passive overhearing and caching by measuring the reconstruction efficiency across all observers. Each node in the testbed performs a transfer, one at a time, by requesting data from the gateway. The caches on the nodes are reset after each transfer.

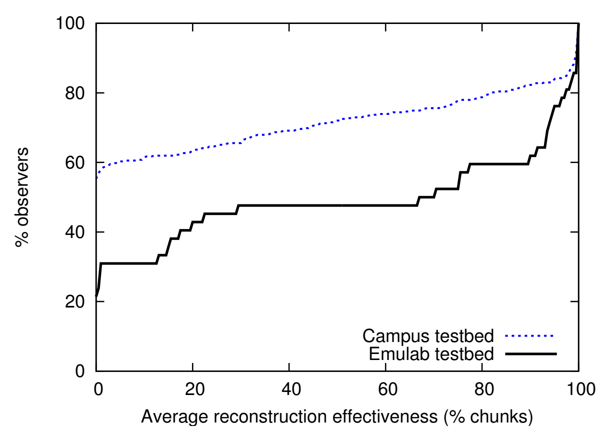


Figure 11: CDF of average reconstruction efficiency in Ditto across observers for the campus testbed and Emulab.

Figure 11 shows that observers in both testbeds can potentially benefit from Ditto: 40% of the observers in the campus testbed and 70% of the observers in the Emulab testbed could reconstruct complete chunks from passive overhearing. In fact, 25% of the observers in the campus testbed and 50% in Emulab could completely reconstruct more than 50% of the total chunks transferred. The campus testbed obtains less benefit than Emulab. We believe that this is a result of the larger scale of the campus testbed: when nodes closer to the gateway are the receivers, the nodes at the edge of the testbed cannot benefit from overhearing. We investigate this effect further by analyzing the impact of proximity on the nodes in the campus testbed.

6.2.1 Effect of proximity

Figure 12 confirms our intuition that nodes spatially closer to the gateway (nodes 14, 30, 34, 5, 37, 22 are all one hop away) overhear transfers from several receivers and hence have a high median reconstruction efficiency across all their observations. On the other hand, nodes at the edge of the network have fewer opportunities to overhear and hence exhibit a median reconstruction efficiency of zero.

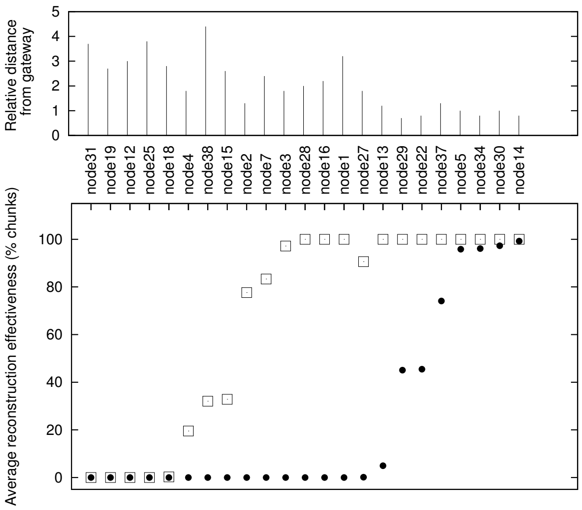


Figure 12: Impact of proximity on the median and the maximum per-node reconstruction effectiveness (MAP testbed). Nodes (X-axis) are sorted by their median reconstruction effectiveness (primary) with a secondary sort on the max value.

6.2.2 Effect of chunk size

Due to losses in the wireless network, the probability of overhearing the entire file decreases exponentially with the file size, and opportunities from overhearing parts of a file are wasted. Caching smaller chunks improves the chances of a node to overhear the entire chunk transfer.

Figure 13 shows this effect for transfer of a 1MB file from the gateway to the receivers. In the campus testbed, 20% of the observers were able to reconstruct atleast 60% of the chunks when the chunk size is 32KB. In contrast, for a chunk size of 8KB, 20% of the observers were able to reconstruct atleast 80% of the chunks. Reducing chunk size similarly improves reconstruction efficiency in Emulab.

The resulting improvement in reconstruction efficiency provides a second advantage: It reduces Ditto's memory overhead, thereby allowing the incomplete chunks to be retained longer. Figure 14 depicts this improvement for the campus testbed. For instance, 60% of observers reduced their memory overhead from less than 70% to less than 35% when the chunk size was reduced from 32KB to 8KB.

Reducing the chunk size, however, increases the number of wireless transmissions: A smaller chunk size increases the overhead from chunk request packets. To balance the overhead and the increased efficiency, we choose a chunk size of 8KB for Ditto.

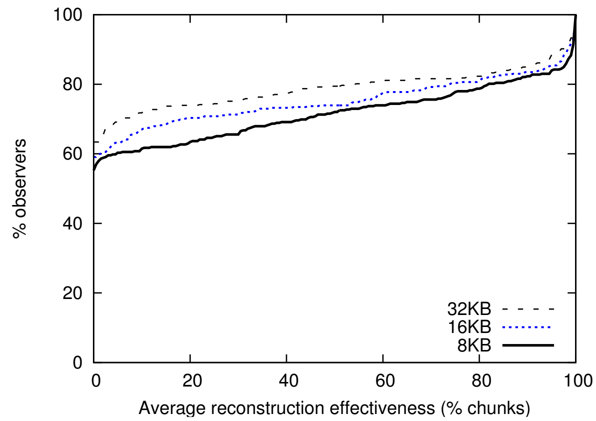


Figure 13: CDF of average reconstruction efficiency across observers showing the effect of varying chunk sizes. (MAP testbed).

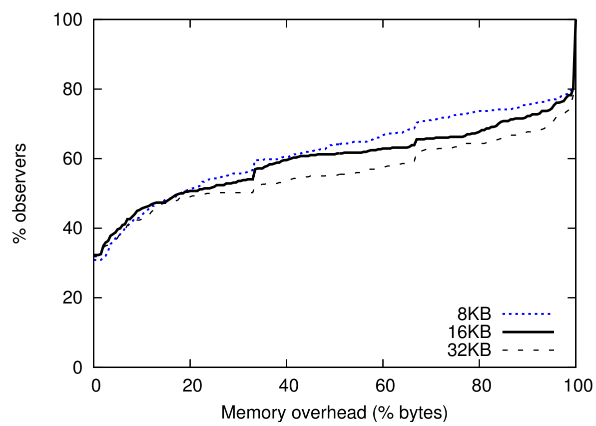


Figure 14: CDF of memory overhead (percentage unused bytes of total Ditto traffic overheard) across observers showing the effect of varying chunk sizes. (MAP testbed).

6.3 Effect of Inter-stream Reassembly

We first investigate the benefit from Ditto's ability to reassemble chunks overheard via multiple TCP streams by comparing Ditto to Ditto without inter-stream reassembly. As seen in Figure 15, Ditto performs similarly with or without inter-stream reassembly. Our inspection of the topology indicates that both testbeds provide very few opportunities for observers to overhear multiple flows. In fact, when there were multiple TCP streams that an observer could overhear, the benefit from inter-stream reassembly is significant. In the campus testbed, for example, when node 1 is communicating with the gateway (node 11) via route 1-13-30-11, node 7 combines TCP streams from all the three proxy hops to obtain 10% better reconstruction efficiency.

Further, Figure 16 shows that the increased reconstruction efficiency reduces Ditto's memory overhead. Thus, Ditto's inter-stream reassembly provides two advantages: (1) improved reconstruction efficiency in the presence of multiple chances of overhearing, and (2) lower memory overhead, both at no performance penalty.

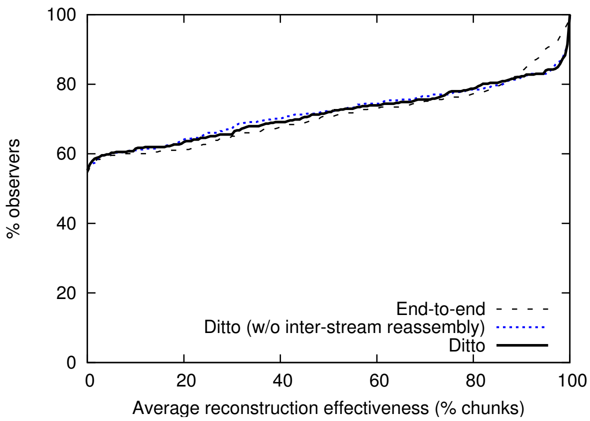


Figure 15: CDF of average reconstruction efficiency across observers for different schemes. (MAP testbed).

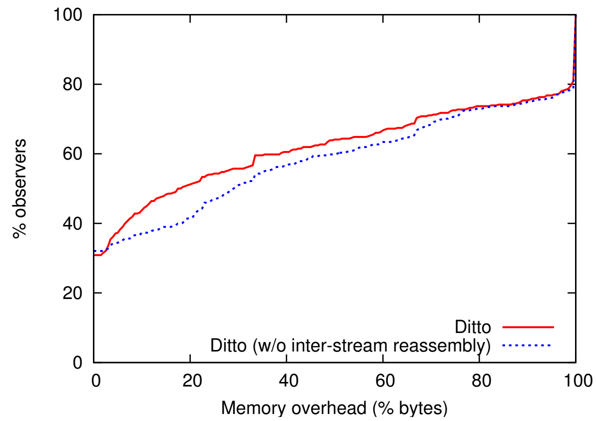


Figure 16: CDF of memory overhead across observers showing the effect of inter-stream reassembly. (MAP testbed).

A deliberate choice in Ditto's design requires a node to overhear the first packet of each chunk (containing the chunk identifier) in every TCP stream that the chunk may appear in order to correlate multiple streams and reconstruct the chunks. This potentially results in lost opportunities and lower reconstruction efficiency in Ditto since, given an ideal implementation, as long as every byte of a chunk is overheard at least once, the chunk should be successfully reconstructed. To analyze the performance impact of this design choice, we compare Ditto to the end-to-end scheme. Figure 15 indicates that in our campus testbed, Ditto performs comparably to the end-to-end scheme. This observation was also true for the Emulab testbed.

We further analyze the effect of this design decision through an offline analysis of `tcpdump` logs of an experimental run. The analysis showed that the requirement to observe the first packet of the chunk on each stream cost ditto the opportunity to reconstruct up to 6% more chunks than it is able to. Due to the complexity of matching packets without the initial chunk identity, we believe this design decision is sound, though we intend to revisit it on more dense topologies that have more overhearing opportunities.

6.4 Ditto Performance

In this section, we compare the throughput improvement from deploying Ditto with existing schemes under a variety of traffic patterns. We first deploy Ditto on all the nodes in the testbed. Leaf nodes in the network then request the same file from the gateway. We vary the time interval between these requests such that the requests are either (1) entirely sequential (one transfer does not interfere with the next), or (2) staggered (the time interval between consecutive transfers is set to 5 seconds). We also vary the order in which the receivers make their requests. Note that because all the nodes are Ditto enabled, even the nodes making the request can overhear prior data transfers.

6.4.1 Example

We first analyze Ditto's performance gain in comparison to an end-to-end transfer and to the on-path caching scheme using an example traffic scenario. Figure 17 shows the average throughput achieved by each receiver when requesting a

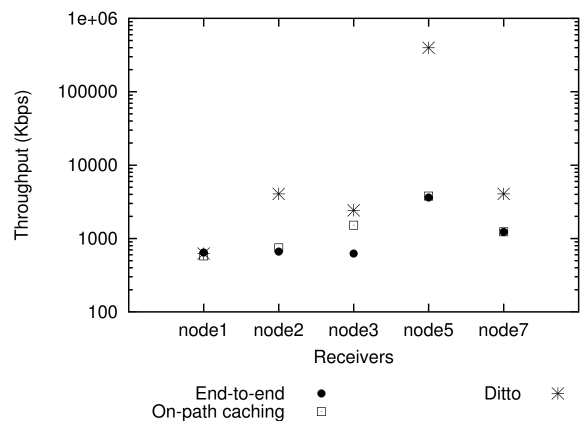
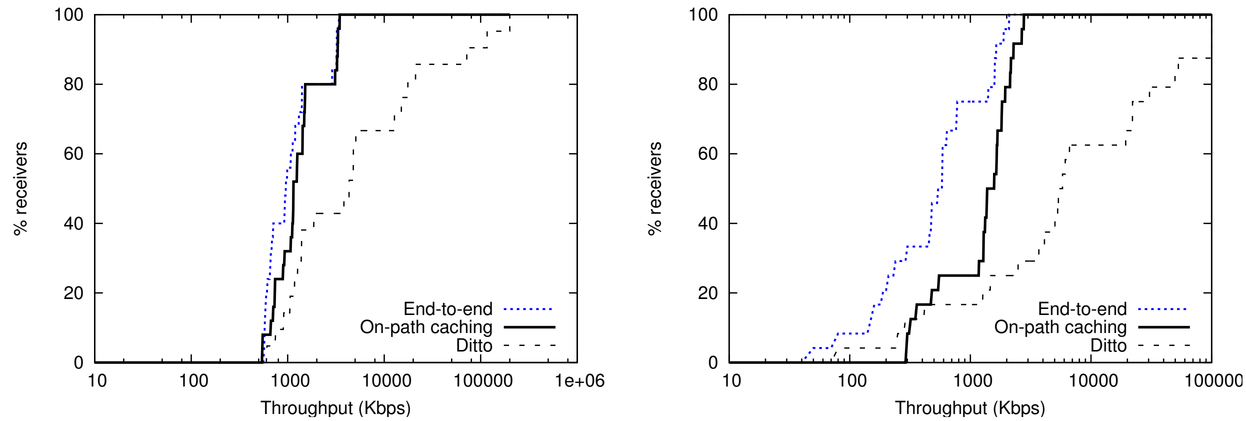


Figure 17: Average throughput achieved by each receiver when requesting sequentially. Note that the receivers are sorted in the request order. (Emulab).

1MB file sequentially. The receivers are arranged in the order in which they made requests i.e., the first request was made by node 1 and node 7 made the last request.[4] The results clearly indicate the differences in the three schemes. The first request, made by node 1, achieved the same throughput in all three schemes because data was not cached anywhere in the network. In all the subsequent requests, Ditto benefited from caching overheard bytes and outperformed end-to-end and on-path schemes. Only node 3 benefited from on-path caching: when node 2 was the receiver, the data was cached at node 4 (its intermediate hop). Node 3 exploited the cache at node 4 to obtain the data in a single hop (vs. the 2 hops in the end-to-end case) and obtained better throughput. For node 5, Ditto performed significantly better because the data was already present in its local cache as a result of overhearing. Thus, Ditto significantly improves throughput performance by obtaining data over fewer hops.

[4]Please refer to Figure 9 for the placement of these nodes.



(a) MAP testbed **(b) Emulab**

Figure 18: CDF of throughput across receivers when requesting sequentially in one chosen request order.

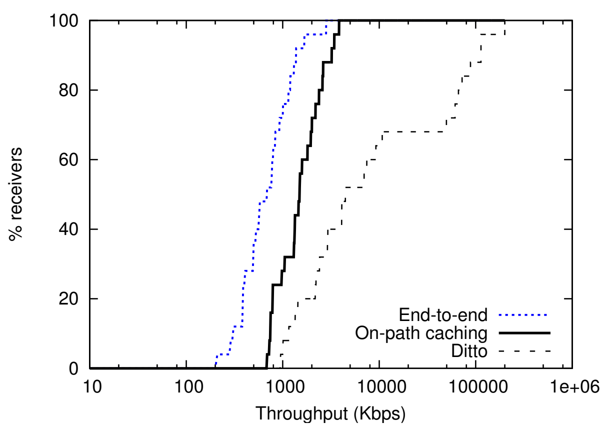


Figure 19: CDF of throughput across receivers with staggered requests in one chosen request order. (Emulab).

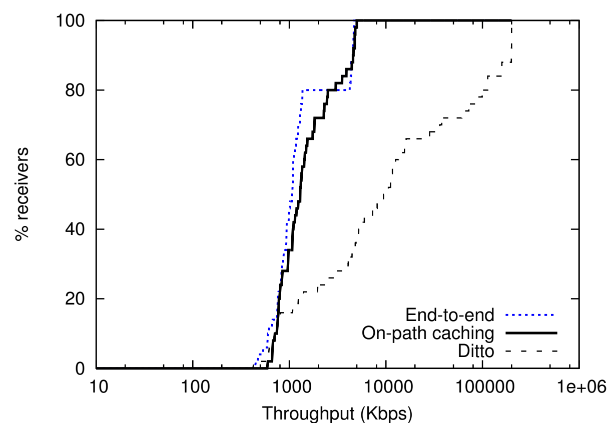


Figure 20: CDF of throughput across receivers when requesting sequentially in randomly chosen request order (Across ten runs in Emulab).

6.4.2 Overall Performance

Figure 18 shows the CDF of the transfer throughput across all the receivers for both the testbeds. We chose one random order of receivers and the requests were sequential. Both the campus and the Emulab testbeds show significantly improved median throughput using Ditto:

	End-to-end	On-path	Ditto
Campus	540 Kbps	1380 Kbps	5370 Kbps
Emulab	960 Kbps	1150 Kbps	4330 Kbps

In Emulab, the median throughput achieved using Ditto is 4.5× better than end-to-end and 3.7× better than on-path caching. Similarly, the median number in the campus testbed is 3.5× better than on-path caching, but 10× better than end-to-end transfers. Since the requests are sequential, the gain in these scenarios is a result of the reduced hop count to the data enabled by Ditto's overhearing and caching mechanisms.

We then changed our request pattern to be staggered across receivers. In this scenario, consecutive transfers potentially contend for the wireless medium. Figure 19 presents the performance gain from Emulab (note that the throughput in this figure is in log-scale). The results show a wider performance gap between Ditto and existing schemes in comparison to the sequential case, with median throughputs of:

	End-to-end	On-path	Ditto
Sequential	960 Kbps	1150 Kbps	4330 Kbps
Staggered	690 Kbps	1500 Kbps	4470 Kbps

Ditto is 6.5× better than end-to-end in the staggered scenario vs. the 3.7× improvement in the sequential case. Here, Ditto's caching reduces the transfer time of requests thereby decreasing the likelihood of contention for the wireless medium due to overlapping requests.

Finally, we vary the request order among receivers in Emulab over ten runs to more exhaustively characterize the range of gains possible using Ditto. Our results show that Ditto significantly outperforms existing schemes (Figure 20): Ditto's median throughput is 8.8× better than end-to-end and 7× better than on-path caching.

7. RELATED WORK

Ditto is most related to four significant areas of work: Caching, particularly hierarchical and packet level caching, and three techniques that attempt to turn wireless broadcast from a liability to an advantage: opportunistic routing, network coding, and partial packet recovery.

Hierarchical caching is used to improve throughput and reduce load on long-distance links by satisfying multiple client requests for the same data from a nearby source [5, 22, 12]. In the caching arena, the project closest to Ditto is the recent MeshCache system, which places a hierarchical Web cache at each node in a wireless mesh network [8]. As our evaluation showed (Section 6), Ditto's overhearing can increase throughput by up to 7x for cached files compared to simple on-path caching. This difference is fundamental to Ditto's use of per-chunk content naming, which frees it from both being application-specific and needing to cache whole files, which limits the opportunities for Web-cache-based approaches such as MeshCache.

An alternate approach is to enable caching at a **packet/sub-packet** granularity. In the RTS-id approach [2], nodes maintain a cache of recently overheard packets. The RTS message contains the id of the packet that a node wants to send; if the receiver already has the packet in the cache it acknowledges this via the CTS message. This avoids redundant transmissions by leveraging recently overheard packets that are present in the cache. In contrast, Ditto's caching works on a longer time scale (multiple requests for the same file) and at the granularity of application data chunks. Similarly, Spring & Wetherall also provide sub-packet content caching on a single Internet link [21]. Like Ditto, this technique uses content-based hashing to identify shared chunks between packets. Unfortunately, this technique makes use of a synchronized dictionary between the routers at each end of the link, which makes it challenging to extend their approach to wireless overhearing: a sender has no way of knowing which, if any, other nodes overheard its prior packet transmissions. Ditto can successfully use overheard transmissions, but at the cost of coarser-grained sharing: Ditto uses roughly 8KB sized chunks to amortize the overhead of explicitly transmitting hash values, where the technique of Spring & Wetherall can take advantage of similarities on the order of a few hundred bytes.

Opportunistic routing takes a different approach to using the broadcast nature of wireless networks, by forwarding packets based upon which nodes *actually* received the transmission instead of using one pre-determined path. These approaches include the pioneering ExOR system [4] and the more recent XOR [15] and MORE [6] systems, among others. These systems operate independently of the transport protocol used, but do impose a requirement that the transport protocol be able to tolerate significant (8-100 packet) amounts of batching when performing data transfers. As a result, to date these protocols require a new transport protocol, similar to Ditto's requirement that applications use a data-oriented transfer system. Because their benefits are independent of the longer-term locality exploited by Ditto, we believe that combining these approaches is a promising avenue of future work.

Network coding attempts to optimize the simultaneous transmission of packets by combining multiple packets into a single transmission, such that they can be independently decoded by different receivers. Examples of network coding systems include COPE [15]. Like opportunistic routing, the way that Ditto takes advantage of wireless broadcast is complimentary to the approaches used in network coding. Combining them, however, is a more challenging aspect of future work, because the packets overheard by a node in a network coding system may not be directly decodable by a third party.

Partial packet recovery techniques leverage the fact that different parts of an IP-layer packet may be received by different sets of receivers [17, 14]. Ditto's partial chunk recovery scheme draws directly from these techniques to reassemble application-layer chunks that were received from independent transmissions. The techniques differ in that Ditto must decode the TCP and RPC layers, as appropriate, and deal with the potential that different transmissions of the same chunk may use different packet sizes or be offset to different locations in the TCP stream (and therefore packetized differently).

8. DISCUSSION

Real world traffic study: A key factor that impacts Ditto's performance is the cache size at each node; shared data should not be evicted from the cache before it can even be used. A detailed study on this parameter is challenging since it requires access to traffic traces from real world mesh deployments. We do, however, shed some light on its impact by collecting aggregate statistics in two real world Meraki deployments: (1) one with 20 Meraki Minis, and (2) one with 10. We logged the total traffic volume in these networks over a 24 hour period for a week in March and April. A worst case scenario for cache size (assuming every node can overhear every other node) results in a median value of 3 GBytes and a maximum of 13 GBytes per node per day. Considering a software update scenario, a study from Microsoft Research noted that 80% of software update downloads happen in the first day after a patch is released [13]. Putting these together, we can infer that using Ditto with a cache size larger than 3 GBytes, the networks can benefit by caching the shared data.

Intelligent proxy selection: MeshCache [8] demonstrates throughput improvement from routing away from a gateway by alleviating the congestion around the gateway (a hotspot). Ditto already alleviates hotspots using overhearing; an interesting area of future research would be determining whether further benefit could be obtained in Ditto by selecting proxies not enroute to the gateway.

Convergent encryption: Ditto targets applications that require high transfer efficiency but have little or no privacy requirements. One possible option to provide Ditto's efficiency while providing some confidentiality to data transfers is the use of convergent encryption [10], which uses the hash of the content as an encryption key. This provides a middle ground in the trade-off between high level of privacy and improved transfer efficiency.

On-Path Opportunistic Caching: In Ditto, only later requests benefit from opportunistic caching. As a future extension, nodes on the path of a transfer *might* benefit from overhearing prior hops in the transfer, and could thus cancel their pending request. For example, consider three consecutive nodes on a path: A, B, and C. If A can successfully overhear a complete chunk while it is being transferred from

C to B, it can cancel its request for that chunk from B. This is similar to ExOR routing which employs a similar idea at the granularity of individual packets. Ditto can work at the granularity of chunks while ensuring that existing transport and MAC protocols need not be changed to fully leverage this idea.

9. CONCLUSION

This paper presented the design, implementation, and evaluation of Ditto, a system for opportunistic caching in multi-hop wireless mesh networks. Ditto uses content-based naming to cache chunks of data in an independent manner. This use of content-base naming enables Ditto's key contribution: it caches data either when it sends it to a client or when it overhears it being transferred by other nodes.

Our evaluation on two wireless testbeds showed that not only is this scheme feasible, but that it significantly outperforms prior schemes that only cache along the actual data transfer path. Our evaluation shed light on Ditto's design decisions for supporting inter-stream reassembly, favoring a robust and simple mechanism that works well in practice, and provided insight into the effects of chunk size and proximity. Taken in total, Ditto improved the performance of repeated data transfers by up to an order of magnitude over end-to-end transfers, and by 3-7× compared to simple on-path caching.

Acknowledgments

We are very grateful to the people who made our wireless experiments possible—David M. Johnson and the Emulab team and Y. Charlie Hu, Dimitrios Koutsonikolas and Syed Ali Raza Jafri for the MAP testbed. We also thank Jeff Pang, Brad Karp and the anonymous reviewers for their valuable feedback. This work was supported by NSF award CNS-0546551 and by DARPA grant HR0011-07-1-0025. The data analysis was performed on the NSF-funded Datapository, supported by grant MRI-0619525.

10. REFERENCES

[1] MAP: Purdue University Wireless Mesh Network Testbed. `https://engineering.purdue.edu/MESH`.

[2] M. Afanasyev, D. G. Andersen, and A. C. Snoeren. Efficiency through eavesdropping: Link-layer packet caching. In *Proc. 5th USENIX NSDI*, San Francisco, CA, Apr. 2008.

[3] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Proc. ACM Mobicom*, Cologne, Germany, Sept. 2005.

[4] S. Biswas and R. Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM*, pages 126–134, New York, NY, Mar. 1999.

[6] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading structure for randomness in wireless opportunistic routing. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[7] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.

[8] S. M. Das, H. Pucha, and C. Y. Hu. Mitigating the gateway bottleneck via transparent cooperative caching in wireless mesh networks. *Ad Hoc Networks (Elsevier) Journal*, Special Issue on Wireless Mesh Networks, 2007, 2007.

[9] T. E. Denehy and W. W. Hsu. Duplicate management for reference data. Research Report RJ10305, IBM, Oct. 2003.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. 22nd Intl. Conf on Distributed Computing Systems*, Vienna, Austria, July 2002.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, pages 254–265, Vancouver, British Columbia, Canada, Sept. 1998.

[13] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vonjović. Planet scale software updates. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[14] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[15] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Mèdard, and J. Crowcroft. XORs in the air: practical wireless network coding. In *Proc. ACM SIGCOMM*, pages 243–254, Pisa, Italy, Aug. 2006.

[16] Meraki Wireless Network. `http://meraki.com/`.

[17] A. K. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *Proc. ACM Mobicom*, Cologne, Germany, Sept. 2005.

[18] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[19] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[20] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Cambridge, MA, Apr. 2007.

[21] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, Sept. 2000.

[22] Squid Web Proxy Cache. `http://www.squid-cache.org/`.

[23] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[24] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proc. USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[25] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.