Content Centric Peer Data Sharing in Pervasive Edge Computing Environments

Xintong Song*, Yaodong Huang†, Qian Zhou†, Fan Ye†, Yuanyuan Yang† and Xiaoming Li*
*School of Electronics Engineering and Computer Science, Peking University, Beijing
{songxintong, lxm}@pku.edu.cn
†Department of Electrical and Computer Engineering, Stony Brook University, New York
{yaodong.huang, qian.zhou, fan.ye, yuanyuan.yang}@stonybrook.edu

***Abstract*—The proliferation and daily congregation of modern mobile devices have created abundant opportunities for peer edge devices to share valuable data with each other. The short contact durations, relatively small sharing sizes, and uncertain data availability, demand agile, light weight peer based data sharing. In this paper, we propose Peer Data Sharing (PDS) that enables edge devices to discover which data exist in nearby peers, and retrieve interested data robustly and efficiently. PDS uses novel lingering queries, mixedcast and en-route message rewriting techniques to minimize redundant transmissions and maximize opportunistic overhearing thus caching in data discovery and retrieval. Extensive evaluations based on an Android prototype show that PDS discovers and retrieves almost 100% data in tens of seconds, and remains robust despite wireless contention, simultaneous consumer requests and user mobility.**

***Index Terms*—Peer Data Sharing; Mobile Sensing; Data Discovery; Data Retrieval;**

I. Introduction

The proliferation of modern sensor-rich mobile devices (e.g., smartphones) and opportunistic congregation of users have created novel opportunities for peer data sharing. Many times spontaneous, agile data exchange among nearby users is desired. For example, during large outdoor events (e.g., music festivals, university commencements), smartphones carried by people can capture diverse data, including human activities, their locations, and image/video clips. When shared among peer devices, such data can help people avoid food stands of long lines, discover interesting souvenirs and artifacts, or enjoy images, video clips of special, memorable moments.

Peer data sharing in such pervasive edge environments has some unique characteristics. Each user may possess certain data and need data by others. However, which devices are around, and what kinds of data they carry, occur opportunistically and cannot be foretold. The limited durations (e.g., a few to tens of minutes) devices are in proximity, and the modest amount yet unforeseeable kinds of data, stipulate fast, light weight discovery and exchange on a *peer basis*. This decentralized sharing differs from most crowdsensing [1] applications where a central backend receives data from all devices and then distributes among them. A dedicated backend demands significant monetary, operational costs and overheads in its development, deployment and maintenance. If peer mobile devices can discover and retrieve desired data collaboratively, such costs and overheads are eliminated. [1]

[1]Some popular apps provide sharing (e.g., SHAREit [2]) but only between two neighbors and requires manual discovery and retrieval.

We propose *Peer Data Sharing (PDS)* that enables mobile devices to quickly discover what data exist in nearby peers and retrieve desired data from possibly multiple devices. PDS achieves robust, efficient and timely data discovery and retrieval, despite the dynamic and uncertain environment where nodes may move in/out and data appear or disappear (created, deleted or carried away) frequently. It discovers all existing data and retrieve required data faithfully across opportunistically gathered peers, under limited wireless bandwidth and potentially frequent message losses, with low overhead and latency.

PDS differs from existing data discovery and sharing work in mobile ad hoc networks [3]–[5] by adopting a content centric design [6], [7] where data are self-contained entities that can be referenced, stored and accessed independently from their original producers. Thus data can be widely cached at and retrieved from any willing and capable nodes. This decoupling offers great performance and robustness opportunities. The consumer can retrieve data from a close by cached copy, or different chunks from multiple cached copies, to reduce latency and aggregate bandwidth. When a node moves and carries away its data, other nodes can cache the data and preserve the availability. Compared to mobile ad hoc routing [8], [9] that maintains paths to destination addresses, PDS derives paths to data instead of addresses. It eliminates the complexities in mapping data to node addresses, which is difficult to accurately track in dynamic and uncertain pervasive edge environment. It also differs from existing content centric networks [6], [7] due to the wireless medium and network scale, which will be discussed in Section VIII.

We make the following contributions:

- We devise robust and efficient pervasive data discovery (PDD) that returns all data existence information faithfully, despite dynamic changes in both device and data sets. Compared to existing content centric designs [6], [7], it uses *lingering queries* each can retrieve a continuous stream of returning metadata entries, *mixedcast* to deliver partially overlapping data efficiently to multiple consumers, and *en-route message rewriting* to minimize redundant metadata collection.
- We design two-phase pervasive data retrieval (PDR) that retrieves different portions of data from multiple cached copies robustly and efficiently. It gathers *chunk distribution information* to derive data reachability, and

recursively divides retrieval queries to precisely choose closest data chunk copies to minimize overhead.
- We implement a PDS prototype on android phones that supports opportunistic caching and mixed cast by overhearing. Using characteristic parameters from the prototype, we conduct extensive simulations and show that consumers can discover almost 100% data and retrieve sizable files (20MB) in tens of seconds, in both static and mobile scenarios, even under heavy traffic.

The rest of this paper is organized as follows: In Section II, we present assumptions and preliminaries about peer data sharing. Then in Section III and IV, we introduce our detailed design of Peer Data Discovery (PDD) and Peer Data Retrieval (PDR). We describe a few implementation issues in Section V, and present a comprehensive evaluation in Section VI. We discuss limitations of this work and plans for future works in Section VII, compare related work in Section VIII, conclude and discuss future work in Section IX.

II. Assumptions and Preliminaries

A. Assumptions and Goals

We make the following assumptions: the environment is uncertain and dynamic. Which devices are in proximity and what data they possess, are opportunistic and not known beforehand. At any time, devices may join or leave, bringing in or carrying away their data. Although users are free to move in/out any time, many of them stay for extended periods of time from a few to tens of minutes. Thus the mobility is low to moderate. The geographical area where users congregate (e.g., restaurants, parks, airports) and thus the network size are usually limited. Devices have reasonable storage (e.g., 16GB or higher) and they can cache others' data, both relayed or overheard. The amount and duration of data exchange are usually moderate (e.g., a few MBs and minutes), due to the limited bandwidth and contact durations. To enable opportunistic caching, we assume nodes will overhear transmitted frames whenever possible (e.g., network/MAC broadcast, pseudo-broadcast [10], or monitor mode [11]) and act on the content. We do not assume any specific radio technology. Devices can connect to each other through different technologies (e.g., Wi-Fi ad hoc, Wi-Fi Direct [12], D2D [13], Bluetooth, etc.). All devices are cooperative and play by the rules. Only publicly sharable data are exchanged and we do not consider security or privacy issues in this work.

Each device can be a *consumer* that requests desired data, or a *producer* that provides them (either generated locally or cached). We focus on two typical scenarios: the consumer needs many small data items meeting certain criteria (e.g., air pollution samples in certain area), or one large, possibly popular data item (e.g., a video clip) consisting of many small *chunks*, each available from multiple nearby devices. Due to the uncertainty, a consumer has to discover what data exist in nearby devices. Otherwise he may be blindly requesting non-existing data. This is similar to customers requesting a menu so they only order what a restaurant can serve. The Peer Data Discovery (PDD) provides such a "menu" of available data as completely and faithfully as possible. Peer Data Retrieval (PDR) should return at least one copy of each requested data item/chunk. Both are best effort: occasionally missing existing or reporting disappeared data is allowed because applications are not mission-critical.

B. Data Descriptors

When generating a new data item, a node creates and associates to it a ***data descriptor*** (i.e., metadata) consisting of multiple ***attributes*** each having a name and taking a certain value of some primitive type (e.g., string, integer, float, Unix time). For example, an NO_x pollutant sample may have *data type* of `NO`$_x$, *time* of the sample generation at `2016-01-01 08:00:00`, and *location* the GPS coordinates of the sample. To avoid conflicts, a *namespace* where the data type is defined (e.g., `environment monitoring`) can be added. For a large data item divided into many chunks, an attribute `total chunks` indicates how many chunks exist. The descriptor of each chunk is simply the data item descriptor appended by a `chunk id` attribute.

C. Metadata and Queries

Each descriptor is a ***metadata entry*** that indicates the *potential* availability of the corresponding data item/chunk. Thus all such entries together describe what data may exist in the environment. Because metadata entries have small sizes and are frequently requested by many consumers, they are widely cached. Any node receiving, relaying or overhearing metadata entries will cache them to serve potential future requests.

On any device, a metadata entry exists as long as the corresponding data item (or any chunk of the data item) exists. If an entry is cached by a node without respective payload, an expiration time is added to this entry. Upon expiration, the node removes the entry if it does not yet have the payload. These simple rules make metadata and corresponding data roughly synchronized in the network. The existence of a metadata entry indicates respective data item is likely available (or at least partially available) somewhere in the network.

A consumer sends ***queries*** to specify desired data and retrieve them from other devices. A query consists of a collection of ***predicates*** specifying desired values on attributes using a relation (e.g., $=$, $>$, $\in$, etc.) to a value or value range. Queries can be specified for data items, chunks and metadata. The retrieval of them follows similar query-response mechanisms, to be presented in Section III and IV.

III. Peer Data Discovery (PDD)

Peer Data Sharing (PDS) consists of two components: ***Peer Data Discovery (PDD)*** and ***Peer Data Retrieval (PDR)***. They share similar message formats, processing procedures and routing mechanism. We present PDD in this section, and introduce PDR design in Section IV focusing on its differences.

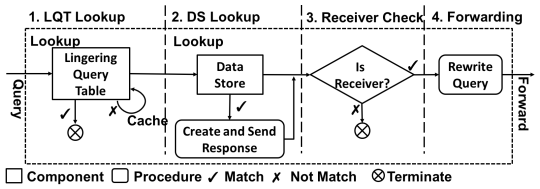

Fig. 1. PDD Query Processing: When receiving a query, a node should perform 4 steps: Lingering Query Table Lookup, Data Store Lookup, Receiver Check and Forwarding.

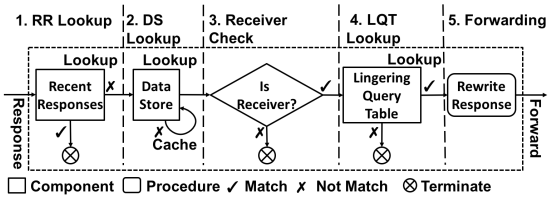

Fig. 2. PDD Response Processing: When receiving a response, a node should perform 5 steps: Recent Responses Lookup, Data Store Lookup, Receiver Check, Lingering Query Table Lookup and Forwarding.

Algorithm 1 PDD Query Processing

Input: $query$
{LQT Lookup}
if $lingering\_query\_table$.Exist($query.id$) **then**

  return
else
  $lingering\_query\_table$.Insert($query$)
end if
{DS Lookup}
$matching\_metadata \leftarrow data\_store$.Match($query$)
if not $matching\_metadata$.Empty() **then**
  $response \leftarrow$ CreateResponse($matching\_metadata$)
  $response.receiver\_ids$.Insert($query.sender\_id$)
  SendResponse($respone$)
end if
{Receiver Check}
if $query$.HasReceiverIds() **and not** $query.receiver\_ids$.Exist($self.id$) **then**

  return
end if
{Forwarding}
UpdateReceiverIds($query$)
$query.sender\_id \leftarrow self.id$
SendQuery($query$)

PDD collects metadata through multi-round requests. In each round the consumer sends a query message requesting metadata, and waits for response messages carrying metadata entries to return. Each node receiving that query should reply all the metadata entries it holds back to the consumer. The consumer dynamically decides whether and when to start a new round, or terminate the data discovery if it determines that almost all data entries are returned.

A. Basic Peer Data Discovery

A metadata query contains the namespace (set to `system`), data type (set to `metadata` since metadata is also a type of data), a globally unique query ID to detect redundant copies, an expiration time beyond which the query is removed, the ID of the node transmitting the query (at the current hop) for returning the response, and an optional list of receiver IDs of the intended next hop receivers (when not all neighbors). This will retrieve all metadata entries. If the consumer is interested in a particular type of data in certain spatial-temporal scope, it can include filters on those attributes. A response contains a namespace (`system`), data type (`metadata`), an optional set of attributes corresponding to filters in the query, a random thus globally unique response ID to detect redundant copies,

Algorithm 2 PDD Response Processing

Input: $response$
{RR Lookup}
if $received\_response\_ids$.Exist($response.id$) **then**

  return
end if
{DS Lookup}
for all $metadata\_entry$ **in** $response.payload$ **do**
  if not $data\_store$.Exist($metadata\_entry$) **then**
    $data\_store$.Insert($metadata\_entry$)
  end if
end for
{Receiver Check}
if not $response.receiver\_ids$.Exist($self.id$) **then**

  return
end if
{LQT Lookup}
$matching\_queries \leftarrow lingering\_query\_table$.Match($response$)
if $matching\_queries$.Empty() **then**

  return
end if
{Forwarding}
$response.receiver\_ids$.Clear()
for all $query$ **in** $matching\_queries$ **do**
  $response.receiver\_ids$.Insert($query.sender\_id$)
end for
SendResponse($response$)

a list of receiver IDs of the intended next hop receivers, and metadata entries as the payload.

1) Query Processing: Figure 1 and Algorithm 1 show how PDD processes incoming queries. A node first examines whether the query has been received before (**LQT Lookup**). It checks the Lingering Query Table (LQT) that holds all recent received, unexpired queries. A redundant copy having the same query ID should be discarded. Otherwise, the new query is inserted into the LQT. Then the node examines whether it has matching data in its Data Store (**DS Lookup**). Since metadata are requested, it creates and sends a response message that contains all its metadata entries.

Next, the node examines the receiver list of the query (**Receiver Check**). If the receiver list contains the ID of itself, or the list is empty (indicating all neighbors should relay), the node should continue to forward the query. Before relaying the query, it updates the receiver ID list with intended next hop receivers (or empty list if all neighbors should relay), and changes the sender ID to that of its own (**Forwarding**). Thus its neighbors know whether to forward the message,

and to which node to return response messages. Only for newly formed networks or generated data where no routing information is available, a query is flooded; otherwise it is forwarded only in directions where the requested data exist. As we discussed in Section II, the size of network that PDS targets is usually limited, thus we do not limit the scope of query propagation. However, such limiting can be achieved easily with a hop counter if needed.

The *lingering query* is very different from *Interest* messages in some content centric work [6], [7]. A lingering query stays in the LQT until its expiration, upon which it is removed. Because many nodes have metadata entries and they will come back over extended period of time, one lingering query can direct the continuous stream of returning responses back to the consumer. In contrast, the Interest is removed upon one single response message. Thus many Interest messages are needed to retrieve all matching metadata entries. By setting appropriate expiration, PDD incurs only one or a few lingering queries.

2) Response Processing: Figure 2 and Algorithm 2 show how PDD processes returning responses. A node first examines whether the response has been received before (**RR Lookup**) from other neighbors (e.g., overheard). It examines the response ID against the IDs of recently received responses. A redundant one will be discarded. To enable opportunistic caching, it detects if any new metadata entries exist in the response. They will be added to its data store (**DS Lookup**). Thus any overheard metadata entry could be served to other consumers in the future.

Then the node examines whether the receiver list contains itself (**Receiver Check**). If so, it is the intended receiver back to the consumer, and should continue to relay the response. The receiver ID checking ensures only nodes on the right path (e.g., reverse) will relay the response. Without it, the response would be flooded throughout the whole network, causing severe contention and overhead.

Before relaying the response, it finds unexpired matching lingering queries in LQT (**LQT Lookup**), and sets the Receiver IDs to neighbors who transmitted these queries, then sends the message (**Forwarding**). When overhearing is possible, one response can be overheard and relayed by multiple neighbors on return paths to possibly different consumers. This avoids sending multiple responses with same content to different neighbors.

B. Efficiency and Robustness Enhancements

1) Mixedcast: The above assumes every query asks for all metadata entries. If some of them have filters, metadata *entry pruning* is needed. An entry is kept in the response only if it has at least one matching lingering query (i.e., requested by at least one consumer). The Receiver IDs are set to neighbors who sent matching queries (according to LQT). Thus the payload is the *union* of desired entries of consumers generating those queries: an entry might be needed by one or multiple of these consumers, while one response message carries all these entries. Thus any entry, regardless of requested by how many consumers, is transmitted only once. This pruning is conducted by all nodes on return paths. Thus only entries desired by downstream consumers are forwarded at each hop.

We call the above ***mixedcast***. Multiple consumers may request partially overlapping sets of metadata. One "joint" response message contains the union of entries requested by consumers, while each entry is sent only once, even needed by multiple consumers. This differs from multicast where the same content is delivered to multiple receivers. Mixedcast ensures all needed entries are included for each receiver (consumer) while pruning ensures only needed ones are returned towards the consumers.

2) Multi-round Discovery: Depending on network quality and node mobility, messages can get lost and nodes can be temporary disconnected. Thus the above single round method may fail to discover some data items. PDD adopts a multi-round discovery algorithm to obtain as many metadata entries as possible.

The consumer makes two decisions: when the current round is finished, and whether to start the next round. Upon each response, it computes the ratio of number of responses received within a recent time window $\mathtt{T}$ to that since sending the query. If the ratio is less than a threshold $\mathtt{T}_r$, the current round is considered (almost) finished. As time goes, less and less responses return. Thus the rule detects the "diminishing" of this trend. It then computes the proportion of new metadata entries received in this round compared to all received, including previous rounds. If the proportion is greater than a threshold $\mathtt{T}_d$, showing many new entries are received in the current round and more might be out there, the consumer starts a new round.

To avoid receiving redundant entries, the consumer applies a ***redundancy detection*** technique. It appends to the query a *Bloom filter* [14] including entries already received. [2] Bloom filter is a space-efficient data structure representing a set of elements and widely used to test whether a given element is in the set, with low and controllable false positive rate. Upon receiving a query, the Bloom filter is cached together with the lingering query.

The Bloom filter is used to *rewrite* response and query messages en-route to reduce redundancy. When sending back or relaying a response, a node should test each metadata entry against the Bloom filter in the matching query. It should send back only those not included in the Bloom filter (thus not yet received by the consumer); it also inserts them in the Bloom filter of the lingering query in LQT, thus the same entries returned by other nodes later will not be transmitted again.

A node possessing local data may return a response before further propagating a query. It should rewrite the query by inserting new entries that it just sent into the Bloom filter of the query. It then forwards the updated query. Thus downstream nodes will not return the same entries. Both response and query are rewritten en-route at each hop. Such *message*

[2]We have compared histogram, wavelet [15] and Bloom filter and found Bloom filter has the highest compression ratio for discrete and unrelated individual items like metadata entries. We do not elaborate due to space limit.

rewriting can significantly reduce redundant metadata returned and contention losses.

IV. Peer Data Retrieval (PDR)

After nearby data are discovered, a consumer can retrieve interested data items. PDD handles two typical scenarios: retrieving one large data item, or collecting many small data items that satisfy a query (e.g., air pollution samples in a radius). The latter follows almost the same process as metadata discovery because these samples are of small sizes. The only change is the filters in queries specify desired data type, and location/durations instead of metadata. Therefore, we focus on retrieval of large data items. We are aware that there are other potential scenarios: retrieving many large data items, or subscribing to a data item that keeps growing (e.g., live video streams). While the former can be achieved by applying PDD for each data item separately, the latter brings more challenges such as real time performance and quality of service, which we plan to address in future work.

PDR has two phases: ***chunk distribution information (CDI) retrieval*** and ***chunk retrieval***. In phase 1, the consumer requests the large data item's CDI, which describes where the nearest copy of each chunk can be found. The CDI is built on demand by propagating a query in the network and soliciting responses. In phase 2 the consumer requests and retrieves each chunk from its nearest provider.

The purpose of CDI retrieval is to build routing entries for different chunks of the requested data item. The principle is similar to Distance-Vector Routing [16]. However, instead of finding one shortest path to a given address, PDR maintains that to a given *data chunk*. Such information is used to find the nearest copy among all candidates, and recursively rewrite the query to divide requested chunks en-route.

The use of CDI ensures each chunk is retrieved only once from a nearest copy to minimize message overhead. The cost is building and collecting CDI. When data items have small sizes (e.g., pollution samples), CDI retrieval may have comparable overhead. However, for large data items such overhead is much smaller than transmitting redundant data chunks. Thus the two-phase mechanism is intended mainly for data items of many chunks (e.g., video clips).

A. Phase 1: Chunk Distribution Information Retrieval

When chunk routing entries do not exist or outdated, CDI retrieval is conducted in manner similar to PDD. We focus on the differences: the query specifies namespace "`system`", data type "`cdi`" and "`descriptor`" whose value is the requested data item's metadata, which includes possibly its unique name. A node creates a response if its Data Store (DS) has chunks or unexpired CDI entries of the requested data item. An entry contains a ***chunk id***, a ***hop count*** to the nearest chunk copy, and a ***neighbor id*** via which the copy can be retrieved. The latter two are set to 0 and the node's own ID if its DS contains the chunk. When a chunk can be retrieved with the same least hop count via multiple neighbors, a CDI entry is created for each neighbor. If a node does not have the chunk in its Data Store, the respective CDI entry is removed after an expiration time. Thus obsolete CDI entries do not stay forever.

A CDI response has namespace "`system`", data type "`cdi`", the same "`descriptor`," and a list of ChunkId-HopCount pairs each indicating which chunk can be retrieved at the specified hop count from the transmitting node. Upon a response, a node creates a new CDI entry for each received ChunkId-HopCount pair, with ***hop count*** $= HopCount + 1$, and ***neighbor id*** set to the transmitting neighbor. The new CDI entry replaces existing ones in the DS if it has smaller distance for the same chunk, or is added in the DS if no CDI entries exist for that chunk. Responses will return to the consumer along reverse paths of query propagation. Eventually CDI entries are created on demand at each node, indicating which neighbors have the shortest paths to which chunks.

B. Phase 2: Recursive Chunk Retrieval

Since one large data item may have many chunks, the consumer sends multiple chunk queries, each requesting a subset of the chunks and directed at a different neighbor closest to those chunks. A node receiving a chunk query will reply requested chunks that it holds, and further divides the subset of remaining chunks into multiple sub-queries, each directed at a different neighbor. This recursive query division allows simultaneous requests of different chunks from different (and nearest) neighbors, both aggregating the bandwidth and reducing latency.

Given CDI entries, each chunk should always be retrieved from the neighbor with the least hop count. When multiple such neighbors exist for one chunk, any one is fine. This may lead to unbalanced loads among neighbors, thus more traffic, contention and losses in some directions. PDR tries to balance the loads when assigning chunks by minimizing the maximum load among neighbors. The problem can be formally represented:

$$\begin{aligned} & \min_{X} \max_{i \in N} \sum_{j \in C} d_{ij} x_{ij} \\ s.t. \quad & x_{ij} \in \{0,1\} \\ & x_{ij} \leq e_{ij} \\ & \sum_{i \in N} x_{ij} = 1 \end{aligned} \tag{1}$$

where N and C denote the sets of neighbors and requested chunks respectively. While $e_{ij} \in \{0,1\}$, $e_{ij} = 1$ indicates chunk j can be retrieved from neighbor i with the least distance, 0 otherwise. $d_{ij} \in D$ is the least hop count to retrieve chunk j from neighbor i. x_{ij}'s are decision variables. $x_{ij} = 1$ indicates assigning chunk j to neighbor i, 0 otherwise. The above problem assigns chunks among neighbors to minimize the maximum load among neighbors. The constraints guarantee that each chunk is always assigned to only one neighbor where it can be retrieved with the least distance.

The above problem formulation balances the load among all immediate neighbors but not necessarily all downstream nodes. The latter would require global knowledge about the distribution of copies of all chunks, which incurs too much complexity and overhead.

The constraint $x_{ij} \leq e_{ij}$ can presented as $(1 - e_{ij})x_{ij} \leq 0$, equivalent to a max-min version ***Generalized Assignment Problem (GAP)***, which is proved to be NP hard and many approximate algorithms are proposed [17]. We use a simple heuristic algorithm. It first assigns chunks to neighbors with the least hop counts. Then it finds the neighbor of the highest load, moves one chunk from it to another neighbor that can retrieve the chunk at the (possibly next) smallest hop count. This is repeated until the highest load no longer decreases. Its complexity is $O(|N||C|^2)$, which is acceptable because both $|N|$ and $|C|$ are small to moderate (e.g., ~ 10) in each query.

V. PDS Prototype Implementation

PDS is designed at application level above the network stack. It can leverage different underlying network (e.g., IP) and link technologies (Wi-Fi infrastructure/ad hoc mode, Bluetooth, ZigBee, Wi-Fi Direct [12], D2D [13], etc.) with proper adaption. At application level, PDS treats all network/link technologies as "faces" [6], [7]. Such abstraction provides a uniform high-level interface while hiding heterogeneous lower level details of different network/link technologies.

To enable opportunistic overhearing critical to reduce message overhead thus efficient caching, PDS should take advantage of the broadcast nature of the wireless medium whenever possible. Many PDS query/response messages are intended for multiple neighbors, and if supported, overhearing allows non-intended neighbors to cache the overheard content. Thus overhearing transmissions is key to improve data availability and retrieval performance. Reliable wireless broadcast and multicast techniques [10], [18]–[20] have been studied and should be leveraged if available. However, many of them require changes at network/link level, thus inconvenient for average users of commodity mobile devices (e.g., smartphones), which constitute a large fraction of edge devices.

Recent work [21], [22] has proposed methods to create multi-hop networks among commodity devices using Wi-Fi Direct [12], supported natively in many smartphones. The network is formed by interconnecting multiple single-hop Wi-Fi Direct groups. Certain devices in each group serve as gateways providing connectivity across groups. Thus opportunistic overhearing can be enabled by network level (e.g., UDP) broadcast within *one hop* neighborhood, without changes to network/link levels. By design, PDS messages may contain an explicit list of the ID(s) of intended neighbor(s). Thus while all neighbors overhear the same transmission from UDP broadcast, only those intended receivers (i.e., those whose IDs appear in the intended receiver list) continue to transmit, while others only cache useful content. Thus a PDS UDP broadcast message does not cause a network wide "storm".

We build a PDS prototype on Android phones to measure the practical performance of such single-hop performance. The parameters are plugged into our simulator later to ensure realistic large-scale simulation. For simplicity in enabling overhearing, all messages are sent by UDP broadcast. We will discuss further strategies dealing with different network mechanisms in Section VII. We present a few implementation issues:

1) Per Hop Ack/Retransmission: UDP broadcast is unreliable, and may suffer high loss rate in wireless networks. We adopt application level ack/retransmission to improve per hop reception. After sending a message, a node waits for acks from intended receivers. A receiver should send back an ack, including the ID of the response and its own ID, so that the sender knows which receiver has received which response. Upon a `RetrTimeout`, if the sender has not received acks from all intended receivers, it transmits the message again with receiver IDs set to those not yet acknowledged only. A message is retransmitted up to `MaxRetrTime` times.

2) Leaky Bucket to Pace Sending Rate: We observe that the Android non-blocking UDP send API has very low reception ratio. Even with one phone sending to another phone in a quiet wireless environment without other senders, only about 14% messages are received. We find that the low reception is caused by an internal buffer overflow in UDP send API. When the API is called to send a message, the message is put into an internal buffer. However, the rate that the MAC can send data in broadcast mode is low (e.g., 7.2 Mbps in 802.11n 20MHz [11]). If the application sends UDP packets at data rates much higher than MAC broadcast data rate, messages arrive at the buffer much faster than they can leave. When the buffer is full the phone's OS simply discards newly arrived messages. Thus those lost messages are never transmitted by the radio. We validate this observation by having 4 receivers (1 laptop running Wireshark [23] and 3 phones) listening to the same sending phone. We find that almost all of the first 658 messages (about 1MB) are received by all receivers (while the buffer is not yet full). After that, messages start getting lost. Lost messages are never heard by any receiver, indicating that they were not transmitted.

We use a classical leaky bucket mechanism [24] at application layer to pace the PDS data sending rate. The bucket has a `BucketCapacity` and `LeakingRate`, which are the size of internal buffer PDS plans to use and the data rate that messages (if any) are taken from the buffer (i.e., transmitted). We conduct experiments to find the leaky bucket parameters for the best performance.

3) Bloom Filter Size: Given estimated number of elements in the set and desired false positive probability, the proper size of a Bloom filter can be calculated [14]. When generating a query, the consumer examines how many entries are already received, then computes a small Bloom filter size to achieve a small (e.g., < 0.01) false positive rate. When the amount of received entries is large, the Bloom filter size may still be big. To address this issue, the consumer uses different hash functions to build Bloom filters in each round. With more rounds, thus more different hash functions, the probability that an entry remains a false positive becomes smaller and smaller (e.g., 0.02 in 2 rounds and 0.003 in 3 rounds for 10,000 entries). Thus the size of Bloom filters can be limited.

4) Prototype Performance Characteristics: We use 5 Android phones (3 Samsung Galaxy Nexus, 1 LG Nexus 5 and

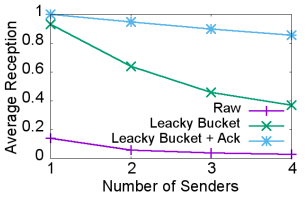

Fig. 3. UDP broadcasting reception.

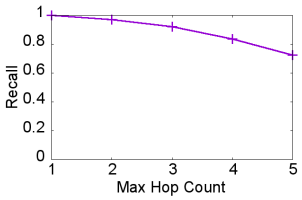

Fig. 4. Impact of max hop count on recall of single round PDD.

1 Motorola Nexus 6) to measure single-hop performance. All phones are within one hop radio range. Packets each 1.5KB size are sent as quickly as possible.

One sender and one receiver phone in a quiet wireless environment shows only 14% reception rate (Figure 3) due to internal buffer overflow. We further explore leaky bucket parameters to improve the reception (figure omitted). When `LeakingRate` grows (1-5Mbps), the reception first stays high ($> 97\%$) for a while and then starts to drop, because too high leaking rates exceed how fast radios can broadcast. A large `BucketCapacity` also lowers the reception, because it may overestimate the available internal buffer size, thus causing overflow. We find 300KB `BucketCapacity` and 4.5Mbps `LeakingRate` achieves a balance between reception and data rates.

Leaky bucket addresses message losses by internal buffer overflow. Ack/retransmission help reduce those by external wireless collisions. We explore how `RetrTimeout` or `MaxRetrTime` impact the reception for two concurrent senders sending to one receiver (figure omitted). We find that as `RetrTimeout` or `MaxRetrTime` increases, the reception first improves then plateaus. This is because longer `RetrTimeout` allows more time for ack to return. Thus the sender does not prematurely retransmit to cause more contention. More retries directly improve the chances of reception. However, the benefits diminish beyond 0.2s `RetrTimeout` or 4 `MaxRetrTime`.

We compare the data rate (network layer throughput) and reception rates of raw UDP broadcast, leaky bucket only and with ack. Multiple phones send data to one phone concurrently. Figure 3 shows the reception rate have great improvements with leaky bucket (from $\sim 10\%$ to $40 \sim 90\%$), and increases further to $85 \sim 99\%$ when ack/retransmission is also added. We conclude that proper leaky bucket and ack/retransmission parameters can achieve reasonable one hop data rates and high reception even with multiple concurrent senders. These parameters are used in multi-hop simulations next.

VI. Performance Evaluation

A. Methodology

We implement PDS in NS-3 [25], a popular network simulator including lower layer Wi-Fi MAC stack. Single-hop data rate and reception measured from the prototype are ported as parameters to the simulator. We simulate both static and mobility scenarios. For static scenario, we distribute 100 nodes as a 10 by 10 grid at proper neighboring distances such that each node can communicate directly with its 8 surrounding neighbors. A consumer is at the center of the field; for multiple consumers, they are randomly located in the center 5 by 5 subgrid.

We generate mobility traces based on 8-hour observation of two real world scenarios in a university, and consumers are picked randomly from all nodes. Each metadata entry is 30 bytes, enough to cover most common data type, time and location attributes. Each data chunk is 256KB, large enough to avoid too many chunks while small enough to fit in the internal buffer and transmit as a unit.

We use several metrics: ***Recall*** is the fraction of distinct metadata entries or chunks received by the consumer, representing the correctness of PDS. ***Latency*** is the time from the consumer sending the query to the arrival of the last returned metadata entry or data chunk, which is strongly related to the system performance and user experience. For energy efficiency, the main consumption of the communication intensive PDS design comes from wireless network communication. Therefore, for simplicity we use ***message overhead***, which is the number of bytes of all messages, to show the cost of data transmission. We distribute metadata entries or data chunks among all nodes uniform randomly at the beginning of simulation. Several factors impact the performance: ***metadata amount*** is the number of different metadata entries; ***data item size*** is the size of the complete data item; ***redundancy*** is the number of copies of each metadata entry or data chunk. We also evaluate how parameters in different components (e.g., `MaxRetrTime`, `RetrTimeout`, $\mathtt{T}_d$) affect their performance. Unless specified, results are averaged over 5 runs.

B. Multi-hop Simulation

We first run single round PDD without ack/retransmission under different metadata amounts and redundancy. We observe a saturation point around 10,000 total metadata entries, beyond which the recall becomes much lower. E.g., with one copy, the recall remains around 0.35, but decreases obviously beyond 10,000 entries (0.20 at 20,000 entries); with two copies, it remains around 0.55 before 5,000 distinct entries. Unless specified, in the following we use 5,000 distinct entries as the normal load and those beyond 10,000 for stress tests. Redundancy is always set to 1 since each entry initially has only one copy on its original producer.

1) Single Round PDD: Next we study how ack/retransmission and multi-round affect the performance. Single round PDD (with ack/retransmission) achieves 76% recall, 3.2s latency and 1.54MB message overhead (figure omitted). Despite the significant improvement in recall brought by ack/retransmission (76% vs. 32%), there are still 1/4 of data items that are not discovered. According to Figure 3, ack/retransmission can improve the reception to higher than 90% when two devices send to one receiver concurrently, all within one hop. However, the reception between provider and consumer can decrease sharply when there are more hops, and even one hop reception might be lower since there might be more than 2 concurrent senders in the simulation scenario.

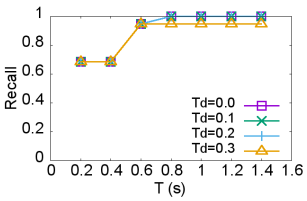

Fig. 5. Recall of multi round PDD

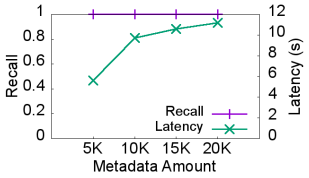

Fig. 6. Impact of metadata amount on PDD recall and latency.

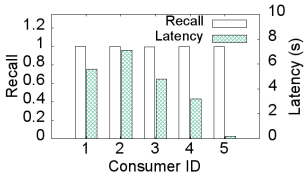

Fig. 7. Recall and latency of PDD with multiple sequential consumers.

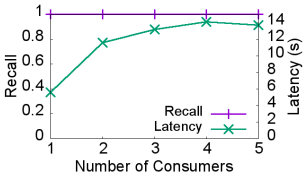

Fig. 8. Recall and latency of PDD with multiple simultaneous consumers.

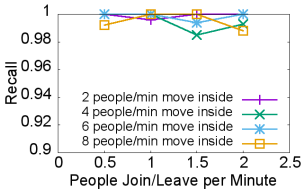

Fig. 9. Impact of node mobility on PDD recall in student center.

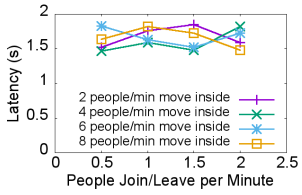

Fig. 10. Impact of node mobility on PDD latency in student center.

To validate this, we change the nodes distribution: while keeping the consumer at the center, the size of grid varies from 3 by 3 to 11 by 11, thus maximum hop count to the consumer from 1 to 5. We keep the average metadata entries at each node to 50 (same average load as 5,000 entries in 100 nodes). Figure 4 shows that as the maximum hop count grows from 1 to 5, recall drops from 100% to 72.3%. Latency and message overhead increase from 0.3s/0.04MB to 3.5s/1.71 MB (figure omitted), because both the network size and metadata amount increase. The result shows that a single round cannot achieve high recall at large network size, because message loss increases over multiple hops. Thus multi-round data discovery is necessary.

2) Multi-round PDD: Three parameters affect the number of rounds and their durations: the recent time window $\mathtt{T}$ in which the number of received entries are counted, the fraction of such entries must be less than a threshold $\mathtt{T}_r$ of all entries in the current round to stop this round, and the ratio of entries in the current round to all received entries must be greater than $\mathtt{T}_d$ to start a new round. Larger $\mathtt{T}$, smaller $\mathtt{T}_r$, $\mathtt{T}_d$ are more aggressive to extend the current round or start a new round.

Figure 5 shows how $\mathtt{T}$ and $\mathtt{T}_d$ impact recall when $\mathtt{T}_r = 0$. Recall increases and becomes stable once $\mathtt{T}$ reaches 0.6-0.8s, because a larger $\mathtt{T}$ extends the current round and receives more entries until there are no more entries to return. Smaller $\mathtt{T}_d$ leads to higher recall (e.g., 1 for $\mathtt{T}_d = 0$ vs. 0.95 for $\mathtt{T}_d = 0.3$). Similar trend can be observed for latency and overhead (figures omitted), which grow fast initially and become gradual or flat. Smaller $\mathtt{T}_d$, thus more rounds, also increases the latency and overhead (e.g., 3.4s and 3.85MB for $\mathtt{T}_d = 0.3$ vs. 5.6s and 5.13MB for $\mathtt{T}_d = 0$). When we keep $\mathtt{T}_d = 0$, varying $\mathtt{T}_r$ does not have significant impact on recall, latency or overhead (thus those figures not presented). After more trials we use $\mathtt{T}_d = \mathtt{T}_r = 0$, and $\mathtt{T} = 1s$ as the best combination.

Next we evaluate multi-round PDD under normal and stress load, multiple consumers and real world mobility. Figure 6 shows when metadata amount increases from 5,000 to 20,000, recall remains at 100%, while latency increases sub-linearly from 5.6s to 11.2s. Message overhead increase almost linearly from 5.13MB to 22.21MB (figure omitted due to space limit). This demonstrates PDD has great robustness against network saturation because: 1) there are less and less metadata entries to collect in subsequent rounds. Redundancy detection filters out already received entries; 2) metadata entries lost in previous rounds leave cached copies along return paths, and more copies are created progressively closer to the consumer. Thus it takes much less hops and latency to retrieve them in later rounds.

Then we evaluate PDD when multiple consumers, either sending queries sequentially or simultaneously. Figure 7 shows that all sequential consumers achieve nearly 100% recall. Latency becomes smaller for later consumers: 5-7s for the first two, 4.8s, 3.2s for the third and fourth. This is due to the overhearing and caching: more redundant copies are created, and closer to consumers. So a consumer has more cached entries and needs to collect less entries from closer copies. The last one takes only 0.2s because it has already cached more than 95% entries even before sending its own query. Figure 8 shows simultaneous consumers also have 100% recall, while latency increases sub-linearly and becomes stable as the number of consumers grows. This is because one mixedcast transmission delivers data for multiple lingering queries, thus less latency for each additional consumer. Message overhead trends for both sequential and simultaneous consumers are similar to respective latency, thus figures omitted.

We further study how node mobility impacts PDD. To make our evaluation as close to real world scenarios as possible, we observe people's movements in a *Student Center* and *Classrooms* in a university. The student center is about $120 \times 120m^2$, while classrooms $20 \times 20m^2$. We monitor how many people stay, how frequently people join, leave and move within the area. The observations last 1-1.5 hours and is repeated 6 times (in total 8 hours). We find that there are usually about 20/30 people stay in the area, on average 1/0.5 people join or leave and 4/0.5 people move inside the area per minute for the two locations. We generate mobility traces based on these observations, and vary the joining/leaving/moving frequencies from 0.5 to 2 times of what are observed.

Figures 9 and 10 show that as node mobility increases, recall remains nearly 100% while latency remains within 2s for the Student Center. Message overhead also remains within 3MB (figure omitted). Simulation in classroom scenario have similar results, whose figures are omitted due to space limit. The results show that PDD is robust under real world mobility scenarios.

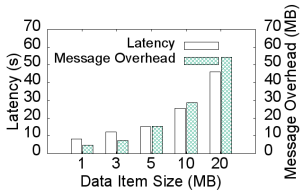

Fig. 11. Impact of data item size on PDR recall and latency.

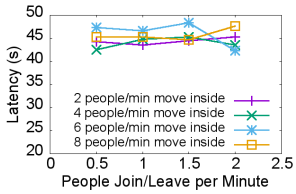

Fig. 12. Impact of node mobility on PDR latency in student center.

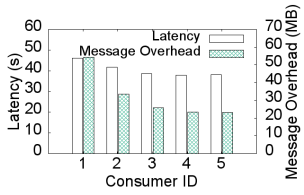

Fig. 15. Latency and message overhead of PDR with multiple consumers.

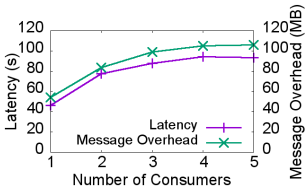

Fig. 16. Latency and overhead of PDR with multiple simultaneous consumers.

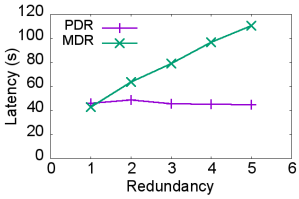

Fig. 13. Impact of chunk redundancy on data retrieval latency.

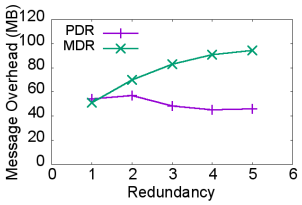

Fig. 14. Impact of chunk redundancy on data retrieval message overhead.

3) PDR Performance: First we vary data item size from 1MB to 20MB, which covers the size of most commonly shared files (e.g., a photo to a 5min 720p video clip). All experiments achieve 100% recall. Thus we only present latency and message overhead. Figure 11 shows that, as data item size grows from 1MB to 20MB, latency and message overhead increase almost linearly from 8.2s/4.83MB to 46.1s/54.22MB. We notice that, with 4.5Mbps broadcast data rate of 802.11n 20MHz, it takes the consumer at least 35.6s just for receiving a 20MB file. Thus PDR's 46.1s total latency (including collecting CDI and data over multiple hops, and potential message loss/retransmissions) is quite small. Message overhead is about 2-3 times of the data item size, because most chunks travel several hops to reach the consumer, adding chunk size overhead at each hop.

We compare PDR with a baseline Multi-round Data Retrieval (MDR) mechanism which is similar to PDD except data chunks instead of metadata entries are retrieved: a consumer sends queries in multiple rounds. Each query requests all the chunks that are not yet received. Nodes receiving queries reply requested chunks they hold, and redundancy detection techniques are used to avoid multiple nodes along a reverse path replying the same chunk.

While both methods achieve 100% recall in all experiments, Figures 13 and 14 show the impact of chunk redundancy on latency and message overhead for a 20MB data item. When only one copy exists, MDR has slightly better performance than PDR (10.7s and 51.34MB vs. 13.5s and 54.22MB). However, in reality, popular files will have more copies exist. MDR shows almost linear increase, while PDR has flat and slight decrease in latency and overhead, about half of MDR's. (e.g., 11.9s and 45.98MB vs. 27.6s and 94.23MB when redundancy is 5).

This is because those redundancy detection techniques can not completely eliminate redundant retrieval, especially those along different reverse paths to the consumer. In contrast, PDR always retrieves exactly one closest copy of each chunk. When more copies of each chunk exist, the nearest copy becomes closer, thus the slight drop in both latency and message overhead. Similar observations are found for other data item sizes. The results show that PDR's two-phase data retrieval mechanism is necessary and can significantly reduce latency and message overhead for popular data items with redundant copies.

We also study how multiple consumers and node mobility impact PDR. Recall is always 100% for both sequential and simultaneous consumers. Figure 15 shows that, from the 1st consumer to the 5th, latency of sequential consumers decreases from 46.1s to 38.1s, while message overhead decreases from 54.22MB to 23.11MB. The significant overhead drop is because more copies of chunks are cached during previous retrieval, thus the average hop each chunk is transmitted become much smaller. Closer chunk distance also decreases latency. Chunks from different directions eventually have to wait for the consumer to receive, thus the drop in latency is somewhat limited. For simultaneous consumers, Figure 16 shows that when the number of consumers increases, both latency and message overhead first increase then become stable. This is because initially there is only one copy of each chunk, thus all simultaneous consumers request the same copy. Consumers at the same direction to a chunk can all benefit from one transmission in that direction.

Finally, we study how node mobility impacts PDR. We present PDR latency retrieving a 20MB data item in Student Center (Figure 12). When mobility increases, latency remains roughly the same (42s-48s). Message overhead remains 24MB-27MB while recall is always 100% (figures omitted). Results in classrooms are similar. The results show that PDR is robust under real world mobility.

VII. Discussion and Future Work

PDS is designed for spontaneous, agile data exchange among nearby opportunistically gathered users. It targets small-scale networks with low to moderate mobility. The proposed mechanism in this paper is our first step towards regional data sharing among nearby mobile devices, especially for crowdsensing data. It is difficult to apply such pure peer based design on a larger scale region (e.g., the whole campus of a university). Increasing amounts of both peer devices and distinct data items require maintaining metadata in a more structured and concise way. Our next step is to leverage edge servers to scale the data sharing service to campus size regions.

PDS can leverage different network/link technologies. In the prototype we choose one hop UDP broadcast to enable

overhearing without changing network/link stacks. By including an intended receiver ID list, only neighbors appearing in the list will retransmit the message (e.g., responses). Only when routing entries for certain data do not exist (e.g., newly produced data), a query may need to be flooded. Well studied mechanisms [26], [27] reducing broadcast and contentions in flooding can be used. When allowed, non-application level techniques can enable overhearing without network level broadcast. Reliable multicast or broadcast techniques at MAC level have been studied [18], [19]. Rooting the device can enable monitoring mode thus overhearing of unicast messages [10], [20] when network/link broadcast is not allowed.

Compared to Wi-Fi ad hoc mode [11] which has little support, Wi-Fi Direct is natively available in commodity mobile devices. It can be used to form multi-hop networks among them [21], [22], with no or minimal change to the OS. Certain devices in single-hop Wi-Fi Direct groups act as "bridges" to interconnect groups. PDS can use the same one hop UDP broadcast to enable overhearing in such networks. Adaptation of query/response delivery may be necessary to avoid overloading those "bridge" devices.

In this paper, we evaluate the performance of PDS by a combination of simulation and a small single-hop network prototype of real phones. There are still some limitations. A large scale prototype testbed would produce more reliable measurements of the performance of proposed mechanisms, such as energy consumption and robustness to mobility. Building and deploying such a testbed is also one future work that we are working on.

The current PDS design does not consider security or privacy issues, and handles publicly sharable data only. In reality a provider may share some data with only certain specific users. Mechanisms that encrypt/sign the data and distribute respective keys to relevant parties have been proposed [28], [29], and studied in content centric networks [30], [31]. Thus encrypted data can still be cached anywhere, but the content accessible to only authorized parties.

Incentive mechanisms that motivate users to consume their resources to participate in such sharing have been studied [32], [33]. With such incentives, PDS provides the data discovery and retrieval part for sharing.

Current PDS caches all metadata entries due to their small size. Data chunks are much bigger, thus cannot always be cached due to limited storage capacity. We plan to study proper data chunk caching strategies based on their popularity and devices' resource availability.

To enable overhearing, the radio must be kept on, which may lead to high energy consumption. Mechanisms for radio schedule synchronization and power management [34], [35] can be used to ensure message reception and overhearing while preserving energy by radio duty cycling.

VIII. Related Work

PDS differs from existing data discovery and sharing work in mobile ad hoc networks [3]–[5]. They are mostly designed for traditional endpoint based networks, where data are bond to specific nodes with certain network address. Sailhan et al. discuss how to discover services in the network, where each service has only one provider and the discovery is actually collecting addresses of service providers [4]. Existing endpoint based ad hoc routing protocols [8], [9] focus on finding one path to a specific destination address. PDS adopts a content centric design where routing entries are for data instead of addresses. Data are cached opportunistically by any capable and willing nodes. Thus consumers do not need to know or care at which addresses the data exist, as long as existing data are discovered and at least one copy (probably the nearest one) is retrieved. PDS focuses on application level mechanisms of lingering queries, mixed cast and en-route message rewriting to find and retrieve data efficiently and robustly.

Information centric networks [6], [7] have been studied extensively. PDS shares similar query-response processing to Content Centric Network (CCN) [6] and Named Data Network (NDN) [7]. Casetti et al. focus on establishing connectivity in multi-hop Wi-Fi Direct networks [22] and use content centric routing tables similar to those in [6], [7]. Due to differences in wireless medium, network scale, PDS differs from them in important aspects: 1) Both CCN and NDN are initially intended for wired networks where each "face" is connected to a different neighbor, whereas PDS leverages the broadcast wireless medium to reduce message overheads and enable opportunistic overhearing. Explicit intended receiver list is used to specify which neighbors should continue forwarding the message. 2) Bandwidth is a scarce resource in shared wireless medium. In CCN/NDN, each Interest is removed upon the return of any matching Data, and Interest/Data are delivered as-is. While PDS uses lingering queries each can guide the return of many response messages to avoid repeating a query many times. It further joins the partially overlapping content of multiple response messages in one mixed cast, and rewrite both queries and responses en-route to minimize transmissions of redundant data, and maximize opportunistic caching.

IX. Conclusions

In this paper, we propose content centric data discovery and retrieval among peer edge devices, which is fundamental to many novel applications where opportunistically congregated devices need to share each other's sensing data. We design multi-round data discovery, and recursive data retrieval by combining lingering queries, mixed cast, and en-route message rewriting techniques that minimize redundant transmissions and maximize opportunistic caching. Evaluations based on an Android prototype demonstrate almost 100% data retrieval in short time under multiple consumers and real world mobile scenarios.

Acknowledgment

This work was supported in part by US NSF grant CSR-1513719 and China National 973 grant 2014CB340405.

REFERENCES

[1] R. K. Ganti, F. Ye, and H. Lei, "Mobile crowdsensing: current state and future challenges," *Communications Magazine, IEEE*, vol. 49, no. 11, pp. 32–39, 2011.

[2] "SHAREit," http://www.ushareit.com/.

[3] A. N. Mian, R. Baldoni, and R. Beraldi, "A survey of service discovery protocols in multihop mobile ad hoc networks," *IEEE Pervasive computing*, vol. 8, no. 1, pp. 66–74, 2009.

[4] F. Sailhan and V. Issarny, "Scalable service discovery for manet," in *Third IEEE International Conference on Pervasive Computing and Communications*. IEEE, 2005, pp. 235–244.

[5] G. Ding and B. Bhargava, "Peer-to-peer file-sharing over mobile ad hoc networks," in *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*. IEEE, 2004, pp. 104–108.

[6] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.

[7] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[8] "RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing," http://www.rfc-base.org/rfc-3561.html.

[9] "RFC 4728: The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4," http://www.rfc-base.org/rfc-4728.html.

[10] Y. Park, C. Jo, S. Yun, and H. Kim, "Multi-room iptv delivery through pseudo-broadcast over ieee 802.11 links," in *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*. IEEE, 2010, pp. 1–5.

[11] I. C. S. L. M. S. Committee *et al.*, "Wireless lan medium access control (mac) and physical layer (phy) specifications," 1997.

[12] "Wi-Fi Direct," http://www.wi-fi.org/discover-wi-fi/wi-fi-direct.

[13] A. Asadi, Q. Wang, and V. Mancuso, "A survey on device-to-device communication in cellular networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1801–1819, 2014.

[14] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2004, pp. 30–39.

[15] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

[16] C. L. Hedrick, "Routing information protocol," 1988.

[17] D. G. Cattrysse and L. N. Van Wassenhove, "A survey of algorithms for the generalized assignment problem," *European journal of operational research*, vol. 60, no. 3, pp. 260–272, 1992.

[18] D. Nguyen, T. Tran, T. Nguyen, and B. Bose, "Wireless broadcast using network coding," *IEEE Transactions on Vehicular technology*, vol. 58, no. 2, pp. 914–925, 2009.

[19] P. Li, S. Guo, S. Yu, and A. V. Vasilakos, "Reliable multicast with pipelined network coding using opportunistic feeding and routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3264–3273, 2014.

[20] M. Li, D. Pei, X. Zhang, B. Zhang, and K. Xu, "Ndn live video broadcasting over wireless lan," in *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2015, pp. 1–7.

[21] C. Funai, C. Tapparello, and W. Heinzelman, "Supporting multi-hop device-to-device networks through wifi direct multi-group networking," *arXiv preprint arXiv:1601.00028*, 2015.

[22] C. Casetti, C. F. Chiasserini, L. C. Pelle, C. Del Valle, Y. Duan, and P. Giaccone, "Content-centric routing in wi-fi direct multi-group networks," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*. IEEE, 2015, pp. 1–9.

[23] "Wireshark," https://www.wireshark.org/.

[24] A. S. Tanenbaum, *Computer networks*. Prentice-Hall, 2003.

[25] "NS-3," https://www.nsnam.org/.

[26] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," *Wireless networks*, vol. 8, no. 2-3, pp. 153–167, 2002.

[27] Y. Sasson, D. Cavin, and A. Schiper, "Probabilistic broadcast for flooding in wireless mobile ad hoc networks," in *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, vol. 2. IEEE, 2003, pp. 1124–1130.

[28] J. V. D. Merwe, D. Dawoud, and S. McDonald, "A survey on peer-to-peer key management for mobile ad hoc networks," *ACM computing surveys (CSUR)*, vol. 39, no. 1, p. 1, 2007.

[29] B. Selim and C. Y. Yeun, "Key management for the manet: A survey," in *Information and Communication Technology Research (ICTRC), 2015 International Conference on*. IEEE, 2015, pp. 326–329.

[30] X. Zhang, K. Chang, H. Xiong, Y. Wen, G. Shi, and G. Wang, "Towards name-based trust and security for content-centric network," in *2011 19th IEEE International Conference on Network Protocols*. IEEE, 2011, pp. 1–6.

[31] B. Hamdane, A. Serhrouchni, A. Fadlallah, and S. G. El Fatmi, "Named-data security scheme for named data networking," in *Network of the Future (NOF), 2012 Third International Conference on the*. IEEE, 2012, pp. 1–6.

[32] M. Feldman, K. Lai, I. Stoica, and J. Chuang, "Robust incentive techniques for peer-to-peer networks," in *Proceedings of the 5th ACM conference on Electronic commerce*. ACM, 2004, pp. 102–111.

[33] L. Duan, T. Kubo, K. Sugiyama, J. Huang, T. Hasegawa, and J. Walrand, "Incentive mechanisms for smartphone collaboration in data acquisition and distributed computing," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1701–1709.

[34] R. Jurdak, A. G. Ruzzelli, and G. M. O'Hare, "Radio sleep mode optimization in wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 9, no. 7, pp. 955–968, 2010.

[35] D. Zhang, T. He, F. Ye, R. K. Ganti, and H. Lei, "Eqs: Neighbor discovery and rendezvous maintenance with extended quorum system for mobile sensing applications," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 2012, pp. 72–81.