DECISION-CENTRIC RESOURCE-EFFICIENT SEMANTIC INFORMATION MANAGEMENT

BY

SHAOHAN HU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Tarek F. Abdelzaher, Chair
Professor Marco Caccamo
Associate Professor Romit Roy Choudhury
Professor Ramesh Govindan, University of Southern California

Abstract

For the past few decades, we have put significant efforts in building tools that extend our senses and enhance our perceptions, be it the traditional sensor networks, or the more recent Internet-of-Things. With such systems, the lasting strives for efficiency and effectiveness have driven research forces in the community to keep seeking smarter ways to manage bigger data with lower resource consumptions, especially resource-poor environments such as post-disaster response and recovery scenarios. In this dissertation, we base ourselves on the state-of-the-arts studies, and build a set of techniques as well as a holistic information management system that not only account for *data* level characteristics, but, more importantly, take advantage of the higher *information semantic* level features as well as the even higher level *decision logic structures* in achieving effective and efficient data acquisition and dissemination.

We first introduce a data prioritization algorithm that accounts for overlaps among data sources to maximize information delivery. We then build a set of techniques that directly optimize the efficiency of *decision making*, as opposed to only focusing on traditional, lower-level communication optimizations, such as total network throughput or average latency. In developing these algorithms, we view decisions as choices of a course of action, based on several logical *predicates*. Making a decision is thus reduced to evaluating a Boolean expression on these predicates; for example, "if it is raining, I will carry an umbrella". To evaluate a predicate, evidence is needed (e.g., a picture of the weather). Data objects, retrieved from sensors, supply the needed evidence for predicate evaluation. By using a decision-making model, our retrieval algorithms are able to take into consideration historical/domain knowledge, logical dependencies among data items, as well as information freshness decays, in order to prioritize data transmission to minimize overhead of transferring information needed by a variety of decision makers, while

at the same time coping with query level timeliness requirements, environment dynamics, and system resource limitations. Finally we present the architecture for a distributed semantic-aware information management system, which we call Athena. We discuss its key design choices, and how we incorporate various techniques, such as interest book-keeping and label sharing, to improve information dissemination efficiency in realistic scenarios.

For all the components as well as the whole Athena system, we will discuss our implementations and evaluations under realistic settings. Results show that our techniques improve the efficiency of information gathering and delivery in support of post-disaster situation assessment and decision making in the face of various environmental and systems constraints.

To my family.

Acknowledgments

It has been my distinctive privilege and honor to have had the opportunity to work with my advisor, Professor Tarek F. Abdelzaher, through my Ph.D. career. His vision and passion for research have always inspired me. Without his invaluable guidance and generous support, it would not have been possible at all for me to come this far.

I also owe my gratitude to Professor Marco Caccamo, Professor Romit Roy Choudhury, and Professor Ramesh Govindan, with whom I have taken classes, collaborated on various projects, and/or coauthored research papers, throughout the years. I am incredibly honored to be able to have them on my Ph.D. committee, and am utmost grateful for all their help and support.

I am also fortunate enough to have studied and worked in highly interdisciplinary and collaborative environments, with exceptionally talented mentors and colleagues from across multiple universities and research labs. I would like to take this opportunity to express my deepest appreciation to Dr. Reginald Hobbs, Prof. John Stankovic, Mr. William Dron, Prof. Amotz Bar-Noy, Prof. Pedro Szekely, Dr. Suman Nath, Prof. Lui Sha, Prof. Lu Su, Prof. Jing Gao, Prof. Hengchang Liu, Prof. Pan Hui, Dr. Wei Zheng, Prof. Xiaoguang Niu, Mr. Yitao Hu, Mr. Xiaochen Liu, Mr. Nooreddin Naghibolhosseini, Prof. Feng Liang, Dr. Lakshman Krishnamurthy, Dr. Rong Yang, Dr. Jun Li, and Ms. Hongyan Wang, for the countless insightful discussions, their generous help, and the many exciting and fruitful collaborations.

Being part of the Cyber-Physical Computing Group at UIUC has also given me the opportunity to meet great people throughout the years. We have been close colleagues as well as good friends. Here I would like to give a shout out to Shen Li, Shiguang Wang, Yunlong Gao, Chris Xiao Cai, Tanvir Amin, Chenji Pan, Siyu Gu, Hongwei Wang, Haiming Jin, Prasanna Giridhar, Shuochao Yao, Yiran Zhao, Huajie Shao, and Fatemeh Saremi for all the fun moments

and hardworking days that we have shared together.

I would also like to thank the National Science Foundation and U.S. Army Research Labs for their financial support and assistance.

Lastly, I would like to thank my parents Youfang Li and Wuting Hu, my mother- and father-in-law Guilan Sun and Xiaoping Zhang, my wife Deyin Zhang, as well as my entire family, for their unconditional love and support throughout my journey. To them I owe too much, and to them I dedicate this dissertation, humbly.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Natural disasters, civil unrests and alike, oftentimes dictate chaotic and dynamic environments. Decision makers thus need to rely on sensor network systems deployed to acquire and gather information, in order to assess the situation and in turn carry out proper actions in response to such events. Sensor networks that are deployed and the individual nodes in use are generally limited in resource (e.g. narrow network bandwidth, low battery power, etc). Therefore, in designing systems in support of decision makings under such environments, we need to pay special attention to how data objects are to be collected and shared at the very edge of the network, what information should be transmitted across the network and when, and ultimately how the decision making processes should be carried out for maximum efficiency.

In collecting data about the environment, different sources (participants sharing sensor data) often overlap in information they share. For example, through lack of coordination, they might collect redundant pictures of the same scene or redundant speed measurements of the same street. In improving data collection efficiency under system resource limitations, we design a data prioritization scheme that maximizes information delivery by reducing redundancy, taking into account the non-independent nature of content. We implement our algorithm on Android platforms, and evaluate through both smartphone-based experiments and large-scale real data driven simulations. Results show that our prioritization algorithm outperforms other candidates in terms of information coverage.

In addition to the more generic coverage metric discussed above, we also develop techniques that exploit logical dependencies among data items for reducing the underlying system resource consumption, while maintaining the level of service in support of decision making tasks. For example, during a flooding event, parts of a city might tend to get flooded together because of

a correlated low elevation, and some roads might become useless for evacuation if a bridge they lead to fails. Such dependencies can be encoded as logic expressions that obviate retrieval of some data items based on values of others. Our algorithm takes logical data dependencies into consideration such that application queries are answered at the central aggregation node, while network bandwidth usage is minimized. The algorithms consider multiple concurrent queries and accommodate retrieval latency constraints. Simulation results show that our algorithm outperforms several baselines by significant margins, maintaining the level of service perceived by applications in the presence of resource-constraints.

Environment dynamics is another key that needs to be considered when gathering information for situation assessment and decision making. Dynamic situation evolves and a decision-maker must decide on a course of action in view of latest data. Since the situation changes, so is the best course of action. One should of course be able to successfully compute the course of action in response to the task at hand, more importantly, at the time the course of action is computed, the data it is based on must be fresh (i.e., within some corresponding validity interval). This data freshness constraint creates an interesting novel problem of timely data retrieval, especially under resource-scarce environments, where network resource limitations require that data objects (e.g., pictures and other sensor measurements pertinent to the decision) generally remain at the sources. Hence, one must decide on (i) which objects to retrieve and (ii) in what order, such that the cost of deciding on a valid course of action is minimized while meeting data freshness constraints. Such an algorithm is developed. The algorithm is shown in simulation to reduce the cost of data retrieval compared to a host of baselines that consider time or resource constraints. It is applied in the context of minimizing cost of finding unobstructed routes between specified locations in a disaster zone by retrieving data on the health of individual route segments.

Next we give an overview of the major components covered in this dissertation.

1.1 Data Collection under Information Redundancy

Maximizing data throughput has been an important goal of traditional sensor network systems. We argue that, however, this does not equate delivering the maximum information, which actually is the more desired behavior for the crowdsensing applications today. Be it post-disaster recovery scenarios or everyday crowdsourced sensing settings, resource constraints is frequently, if not always, a key system limiting factor. For example, nodes have limited battery powers, network is of low bandwidth capacities, etc. Therefore, when sensory data are gathered or transmitted, we need to design techniques that maximize the information delivered during the short communication time through through the narrow transmission channel. Crowdsensing scenarios typically exhibit information overlap among sources. For example, vehicles waiting in the same traffic jam may collect very similar observations about traffic. Redundancy in data collection thus leads to inefficiency, which motivates a system that can recognize and eliminate the redundancy. Such a system would maximize information flow, as opposed to mere data throughput.

We in this work design an information maximizing data prioritization scheme. It transmits data in an order that maximizes information flow. Hence, if data transfer is interrupted before all data are transmitted, a notion of information coverage is maximized for the given transfer size. The scheme is suitable for mobile environments where connectivity between nodes may be interrupted due to the nodes mobility patterns and limited battery capacities. We show that without knowing the data transmission time in advance, which is the common case, no prioritization scheme can guarantee the optimal information throughput. Instead, an approximation bound is derived that is achieved by our prioritization algorithm, making it provably near-optimal.

We also separate application specific components from application-independent components. We recognize that information is a measure that may mean different things to different applications. To keep the information-maximization support as application independent as possible, we ask the programmer to define only one application-specific function per collected content type; namely, a map function, which takes a data object as operand and returns its position in a

virtual space, called the information space, where objects that are closer to each other have more information overlap. When two nodes meet, they exchange content in an order that maximizes coverage in the information space.

An example map function could be one that places objects (that constitute sensor readings) in a space whose dimensions are the location and time of data capture. Hence, sensor readings at closer locations and times would be closer in the virtual space (which designates that such readings are more redundant). In general, other features may be considered as dimensions of information space. For example, in an application that measures temperature in campus buildings, a more meaningful set of features to consider might be time, building name, and room number. Hence, readings from the same time, the same building, and the same room number would be more redundant that readings from different times, different buildings or different room numbers. The map function would then map such measurements to a space where feature similarity translates into proximity. Once a map function is defined, information maximization, informally, becomes a problem of selecting points that are far enough apart in the information space, so that they are not redundant. The design of a good map function is an important application-specific problem. To keep the discussion application- independent, we assume that a good map function, for the application at hand, has already been designed and consider how to use it in order to maximize information.

We evaluate our algorithm through smartphone-based experiments as well as a large-scale simulation using the T-Drive dataset collected by Microsoft Research Asia (MSRA) [23]. Evaluation results demonstrate that our prioritization algorithm outperforms other candidates in terms of completeness of information delivered. This work was published in ICCCN 2013 [61].

1.2 Data Acquisition under Timeliness Requirements

Applications, such as disaster response, stability and support operations, or humanitarian assistance missions often do not have the infrastructure and bandwidth to collect large amounts of data about the current state on the ground due to severe resource constraints (e.g., due to depletion, destruction, or acts of war). More data might be collected by the sources than what

the infrastructure would allow them to communicate. To enhance situation awareness, a key question becomes to reduce the amount of resources needed to answer queries about current state. We specifically consider queries issued at a central command center that serves as an aggregation point for collected data. Some of the pertinent data needed to answer a query might have already been collected. The question lies in collecting the remaining data at minimum cost.

Data dependencies and mission specifics can be exploited to minimize the cost of answering command center queries. In the scenarios mentioned above, applications don't make random requests. They typically have very specific information objectives derived from well-defined protocols and doctrine. A disaster recovery team might follow well-defined protocols for carrying out their rescue objectives. These protocols call for specific information objectives that allow first responders to systematically perform their tasks. For example, in recovering from an earthquake, a disease control team might want to find out where the worst afflicted neighborhoods are, what structures can be used as shelters that are closest from these afflicted neighborhoods, whether they are currently occupied or not, and how to get there to carry out disease prevention procedures. To find an occupied, accessible shelter, one might use the following boolean expression: $(shelterOccupied) \wedge (path1IsGood \vee path2isGood \vee path3isGood)$, where the shelter's occupancy can be estimated using pictures taken from cameras installed in the shelter, and the road conditions by retrieving and examining aerial images taken by a different team. Hence, given a specific protocol and terrain knowledge, application requests can be automatically converted into series of boolean expressions comprised of logically inter-dependent tests on data from multiple sources.

We therefore argue that crowdsensing systems can be designed to exploit *application knowledge and logic dependencies among data items* to potentially reduce the underlying network bandwidth consumption. We shows that when one knows application logic and data dependencies, it is possible to plan the retrieval of objects in such a way that the cost of answering queries is significantly reduced. The practice is close to what an optimizing compiler might do when evaluating complex logical expressions. Namely, if possible, it will evaluate first the variables that might short- cut much of the rest of the logic expression. For example, a predicate that evaluates to $true$, ORed with a large expression, is sufficient to yield a positive answer, obviating

the need for evaluating the rest of the expression. The contribution lies in applying this insight to the domain of crowdsensing, thereby significantly reducing resource demand attributed to data retrieval.

The approach is further enhanced by exploiting additional information about the likely values of variables that have not yet been retrieved. In the above example, if the disease prevention team, by looking at the population distribution data, can tell that the particular shelter is unlikely to be occupied, then they don't need to bother asking for aerial images covering the candidate paths—they can focus their attention on the next shelter. A datastore is maintained that keeps historical data and various application-specific information, which can be used for making educated guesses about cost of object retrieval and likelihood that specific tests on data would evaluate to $true$ or $false$. Evaluation shows that our approach is effective at reducing resource consumption while meeting application information needs. This work was published in DCoSS 2015 [26].

1.3 Data Acquisition under Freshness Constraints

Continuing on the problem of timely data retrieval in resource-poor environments under data freshness constraints, we can also investigate from a different perspective. In resource poor environment, retrieving the requisite data to evaluate a condition takes time and consumes some amount of resources (i.e., has cost). Moreover, conditions change dynamically. Hence, if a decision takes too long, conditions may change, calling for a re-evaluation of the course of action, thereby further delaying the decision. Therefore, besides satisfying query level timeliness requirements, it is also important that we make timely decisions at minimum cost while satisfying *freshness constraints* of the underlying data.

To give a concrete example, consider a disaster management scenario where an active threat is spreading, while members of a rescue team must find their way to a set of destinations that need help. The team leader must decide on a valid route to each destination such that help can be sent along. Making the wrong decision will cause back-tracking and delay, which may jeopardize mission success. The set of alternative routes to a given destination constitutes the

set of alternative courses of action that the corresponding decision must choose from. For a given route to be valid, each segment on the route must be obstruction-free. This creates multiple conditions (one per segment) that must be jointly satisfied. However, the state of various road segments is not always available or not always fresh. Just because a segment was available an hour ago does not mean it is available now. In a weather-related disaster, a tree might fall across it, it might get flooded, or a major accident might render it blocked. Hence, a sufficiently recent condition for each segment must be retrieved. This requirement gives rise to data freshness constraints. Different segments might have different freshness requirements. For example, the state of segments that are far enough from the threat might change more slowly than that of segments in harm's way. Also, once a segment becomes unavailable, its state persists for a duration that depends on the type of damage incurred. Car accidents might clear in hours, but a collapsed building may block a road for days. The rescue team in question may be sharing scarce network resources with other teams. Hence, data pertinent to the aforementioned decisions must be retrieved at minimum cost.

We offer a general formulation to the above problem. An algorithm is developed that minimizes the cost of decisions while reducing the chance of timeouts and ensuring the freshness of data. The algorithms is shown in simulation to outperform several baselines for data retrieval such as retrieving the lowest cost object first.longest-freshness-interval object first.

The novelty of this work lies in exploiting the *structure of the decision* to significantly reduce data retrieval cost. There are two aspects to the exploited decision structure. First, the decision considers alternative courses of action. Hence, the algorithm has a degree of freedom in deciding which alternative to evaluate first. For example, an alternative that is more likely to be valid (and can be evaluated at a lower cost) is a good start. Second, evaluating an alternative entails retrieval and evaluation of multiple conditions that must all be satisfied for the alternative to be deemed a valid course of action. The algorithm again has a choice, deciding which condition to evaluate first. In particular, evaluating the most risky condition first (i.e., the one most likely not to be satisfied) might be better in that it minimizes effort wasted on evaluating ultimately invalid alternatives. The above considerations must be balanced against freshness requirements. Specifically, retrieving data objects with short freshness intervals first is a bad idea because

by the time all other pieces are retrieved, the freshness intervals of these objects might expire, forcing repeat retrieval. Instead, it is better to retrieve data objects with the longest freshness interval first. The above insights give rise to an optimization problem solved and evaluated, demonstrating promising results. This work was published in RTSS 2015 [29].

1.4 Distributed Semantic-Aware Information Management

Post-disaster environments can suffer from severe resource constraints, such as a largely reduced network transmission bandwidth brought about by infrastructure outages. We consider the problem of supporting decision-making over such a resource-limited network. Utilization of such constrained networks is critical to allow civilians and first responders the ability to not only interact, but also share vital status information. Decision-makers in this scenario could be the commander in charge of overseeing rescue efforts, or could be a regular civilian trying to decide the best route to use for evacuation in their particular situation.

Past research efforts have focused on caching and improving data delivery over resource-constrained networks through exploitation of data-level characteristics, such as sparsity, redundancy, or popularity to improve efficiency of delivery. For example, researchers have investigated techniques such as sample compression [12, 47, 58], source selection [25, 37, 65, 68], distance/similarity-based transmission [14,15,41,56], and various cache management schemes [8, 10] to reduce resources spent on data delivery. In this dissertation, we take a new and complementary viewpoint of exploiting *decision models* in designing an efficient information management system. The shift in focus from data-level properties to decision-level structures arises from the rationale that, after all, *aiding vital human decision-making* is the ultimate goal for a disaster-response system.

We view decisions as choices of a course of action among multiple alternatives. The viability of each individual alternative depends on satisfaction of several predicates. Making a choice therefore reduces to the evaluation of a Boolean expression over multiple predicates. For example, "if it is (i) sunny and (ii) warm, I will not wear a wind-breaker". Evaluating a predicate requires acquisition of corresponding evidence. Data objects such as images, videos, or sound

clips, generated by appropriate sensors, can supply the needed evidence. Often, a piece of evidence (i.e., a picture that shows whether it is sunny or rainy) can be supplied by any of several alternative sources, such as multiple cameras overlooking the scene. Athena uses models of decisions (that specify the underlying Boolean expressions) and models of sources (that specify who can supply which pieces of evidence) to optimize delivery of objects that maximally help the decision-maker choose the right course of action.

Athena is a *distributed* architecture, where any user can make queries, as well as contribute, request, or help forward data. This distributed nature differentiates our work from prior studies that adopted a similar decision model [26, 29], but considered only centralized scenarios, where the decision maker resided at some fixed, central node. We show that by accounting for models of decisions and sources, Athena carries out information collection and transmission in a more efficient and timely manner to support decision making. We evaluate Athena through experiments involving network simulation under realistic settings to demonstrate its ability to efficiently manage information gathering and delivery for post-disaster situation assessment. Experiment results show that, under high environment dynamics, when competing techniques struggle to even reach 10% query resolution rate, Athena is able to achieve over 90%, while at the same time incurring the lowest system resource consumption.

Chapter 2

Data Collection under Information Redundancy

2.1 System Design

This section describes the detailed system design. We first present the system model for social sensing applications, then explain the programming framework based on this model.

2.1.1 System Model

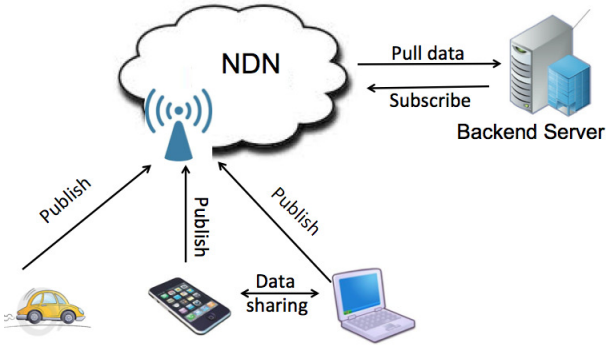


Figure 2.1: System Model

Figure 2.1 depicts the system model for our proposed social sensing applications. Mobile devices participate in these applications by generating and sharing sensory data, which are stored locally and uploaded to a backend server via opportunistic WiFi offloading. Hence mobile devices serve as *publishers*, and the backend server acts as the *subscriber*. Opportunistic peer-to-peer communication might also be enabled to allow information to transparently propagate from one participant to another when they meet, in hopes of finding an offloading opportunity to the server faster. We adopt the NDN framework [31], thus data generated by users are identified by descriptive names.

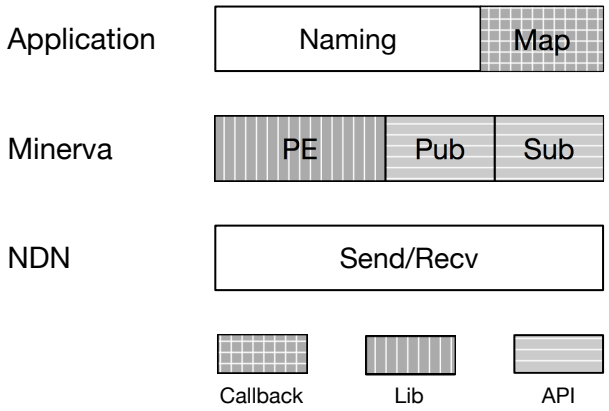


Figure 2.2: Programming Framework

2.1.2 Programming Framework

The programming support in Minerva is straightforward. Minerva provides a publish and a subscribe interface. Additionally, the application provides a callback function (one per content type), called map(), that takes as operand the name of a content object of a particular type and returns the corresponding position and coverage in a virtual information space. The position and coverage of a data point are used to compute its priority in transmission that maximizes information coverage in a resource-constrained environment. Objects are transmitted in the order of largest increase in marginal coverage as discussed in detail in Section 2.2.

As shown in Figure 2.2, an operational system would consist of three different layers: the application layer, the Minerva layer, and the network layer.

The application layer would take care of application-specific functions, such as content naming and publishing. Maintaining uniqueness of names is an application-specific concern not addressed in this dissertation. A fully specified name refers to a unique item. Names can also be partially specified to designate a collection of items that share a common name prefix. In Minerva, publishers and subscribers refer to content collections by name when expressing availability of or interest in content.

In general, applications that use our publish-subscribe system own "subdirectories" in the global name space. For example, an application called *GreenGPS* might own the subdirectory "/root/GreenGPS". The application might publish multiple types of content. Each part could start with "/root/GreenGPS/content-type". Following the content type in the name comes a

listing of content attributes of relevance to the map function. A type-specific map function can therefore parse the name to determine the attributes, and compute the coordinates of the object in virtual space accordingly. For example, an object might be called "/root/GreenGPS/content-type/location/time/filename", where the location and time are the features of the data object.

In addition to the publish and subscribe functions, Minerva internally has a core function, *PE*, short for Prioritization Engine. *PE* reflects our optimization algorithm described in Section 2.2 to compute priorities for data objects such that they are transmitted in an order that contributes to maximum coverage.

The underlying network layer provides the communication functions across a network. In our implementation, we use NDN as the underlying network layer. Our solution does not require any changes to the standard NDN library (developed by PARC). Thus, it is general enough to be compatible with other existing NDN applications.

In the next section, we describe in detail the prioritization algorithm.

2.2 Information-maximizing Prioritization

In this section, we first introduce the definition of information coverage and formulate the information coverage maximizing problem. Then, we present the design and analysis of our algorithm.

2.2.1 Information Coverage

Data collected in social sensing applications are not independent; they exhibit correlations as discussed in the introduction. For the purpose of theoretical problem formulation, we assume that each data point covers a region in the *information space*, referred to as the *data coverage* region, defined below.

Definition 1 (Data Coverage)**.** *Suppose that there are k features of the data collected in a social sensing application. The Cartesian product*[1] *of domains of the k features forms a k-D* information space*. Any data point X with coordinates $\langle x_1, x_2, \cdots, x_k \rangle$ covers an interval*

[1]The Cartesian product of two sets A and B is a set C, such that $C = \{\langle x, y \rangle \mid x \in A, y \in B\}$. Similarly, we can define the Cartesian product of k sets.

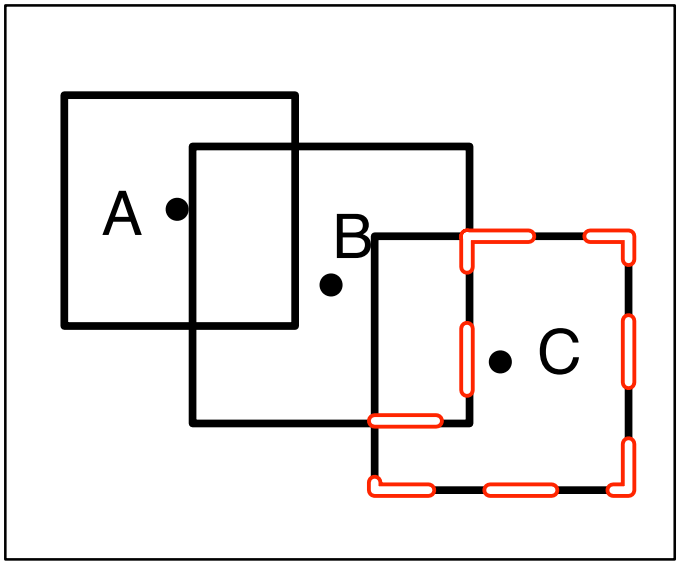


Figure 2.3: Illustration of Coverage and Marginal Coverage with 2 Features

I_j centered at x_j on the j-th dimension, where $1 \leq j \leq k$. The coverage *of X is $\mathcal{C}_X = I_1 \times I_2 \times \cdots \times I_k$, where $\times$ is the Cartesian product.*

Please note that data of different applications might have different coverage intervals. Given a particular application, the notion of coverage is usually clear. For example, when measuring temperature in a corn field, the "coverage" in the time dimension might be, say, 10-20 minutes, since weather does not noticeably change in such a short time. Similarly, coverage in space might be 200-300 meters, since these distances are small enough not to dramatically affect temperature. Hence, given a temperature measurement at some time and location it can be assumed to remain valid for the entire coverage interval (in space and in time).

By Definition 1, the coverage of a data point is a k-D box as illustrated in Fig. 2.3. The coverage of a dataset $\mathbb{S}$ is defined as $\mathcal{C}_{\mathbb{S}} = \bigcup_{S \in \mathbb{S}} \mathcal{C}_S$. The coverage of the intersection (*resp.* union) of two datasets $\mathbb{S}_1, \mathbb{S}_2$ is defined as $\mathcal{C}_{\mathbb{S}_1 \cap \mathbb{S}_2} = \mathcal{C}_{\mathbb{S}_1} \cap \mathcal{C}_{\mathbb{S}_2}$ (*resp.* $\mathcal{C}_{\mathbb{S} \cup \mathbb{S}_2} = \mathcal{C}_{\mathbb{S}_1} \cup \mathcal{C}_{\mathbb{S}_2}$).

We define the *marginal coverage* of a data point X *w.r.t.* a dataset $\mathbb{S}$ in Definition 2.

Definition 2 (Marginal Coverage)**.** *The marginal coverage of a data point X* w.r.t. *a dataset $\mathbb{S}$ is the region in the information space covered by X but not covered by $\mathbb{S}$,* i.e.*, $\mathcal{MC}_{X|\mathbb{S}} = \mathcal{C}_X - \mathcal{C}_{\{X\} \cap \mathbb{S}}$.*

As illustrated in Fig. 2.3, the area surrounded by the dashed red line is the marginal coverage of data point C *w.r.t.* the dataset $\{A, B\}$. By definition, $\mathcal{MC}_{X|\emptyset} = \mathcal{C}_X$.

We define the value of the coverage of a data point in Definition 3.

Definition 3 (Coverage Value)**.** *The coverage value of a data point X in a k-D information space is the* size *of its k-D coverage region, defined as $\mathcal{V}(\mathcal{C}_X) = \prod_{i=1}^{k} I_i$, where I_i is the coverage interval in the ith dimension as in Definition 1.*

For example, if $k = 2$, the value of the coverage of a data point is simply the area of its coverage region in the information space. Similarly, definitions of the coverage value of a dataset and the marginal coverage value of a data point *w.r.t.* a dataset follow.

2.2.2 Problem Definition

A common goal of social sensing is to gather *information* that is as complete as possible. One trivial solution is that when a connection is established between two participants they sync all data, and when connecting to the backend server, a participant offloads its entire local data. However, due to the mobility and resource constraints (*e.g.*, energy), it is not always possible to sync or offload the entire dataset in a single transmission. Thus, in each transmission session, we aim to maximize the marginal information coverage value of the subset of data that can be transmitted, referred to as the MaxInfo problem.

In the rest of this chapter, we shall assume that all data objects of the same type are of the same size. This is a common assumption in sensing applications. For example, in the context of a particular navigation application, all GPS readings have the same format and size. Similarly, in the context of a particular environmental sensing application, all temperature and humidity, measurements have the same format and size. In general, if the data format for sensory measurements is fixed, then all data records have the same size. This assumption simplifies terminology, allowing us to represent connection duration by a corresponding number of transmitted objects. It can be easily generalized to arbitrary object sizes simply by weighting each object by its size. Assuming same size objects, the the MaxInfo problem is formulated as follows:

Problem 1 (MaxInfo)**.** *Suppose that there is a dataset* $\mathbb{S}_1$ *(*resp. $\mathbb{S}_2$*) on the data receiver (*resp. *the data provider).* MaxInfo *is to determine an order based on which the receiver should pull data from the provider such that for any data transmission size the receiver's information coverage is maximized. In other words, let* $\mathbb{R} \subset \mathbb{S}_2$ *with cardinality* n *is the dataset pulled by the order, then* $\forall n, \forall \mathbb{T}, |\mathbb{R}| = |\mathbb{T}| = n, \mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{R}}) \geq \mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{T}})$.

Unfortunately, the unpredictability of the duration of each transmission session makes it *impossible* to find an order that is optimal for any n, which can be proved by a counter example as illustrated in Fig. 2.3. When $n = 1$, the optimal order is to select data object B first, since its coverage value is the highest. When $n = 2$, the optimal order is to select data objects A and C first, which conflicts with the optimal order when $n = 1$. Hence, we need to quantify the best one can do to approximate the optimal MaxInfo solution when one does not know connection duration in advance.

We first define the optimal solution OPT for MaxInfo to be an offline coverage-maximizing solution that assumes knowledge of the cardinality n in advance. It will constitute a theoretical upper bound, since such knowledge is generally not available online. Note that, as illustrated above, OPT may return different optimal orders for different values of n. Let us define the approximation ratio of an online solution A as follows:

Definition 4 (Approximation Ratio)**.** *Consider a dataset* $\mathbb{S}_1$ *(*resp. $\mathbb{S}_2$*) on the data requester* m_r *(*resp. *the data provider* m_p*). Let solution* A *of* MaxInfo *represent a fixed priority order for data object to pull from* m_p*. Let* $\mathbb{A}^n \subset \mathbb{S}_2$ *denote the subset of data transmitted from* m_p *with cardinality* n *during the transmission session. Let* $\mathbb{OPT}^n$ *denote the subset output by* OPT *with* n *known in advance. The approximation ratio of* A *is*

$$\tau = \min_{\forall n, 0 \leq n \leq |\mathbb{S}_2|} \frac{\mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{A}^n})}{\mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{OPT}^n})}.$$

Please note that for any fixed n, when $\mathbb{S}_1 = \emptyset$,the MaxInfo is exactly the weighted *Max n-Cover* problem [18].

Theorem 1. *[18] If Max* n*-Cover can be constructively approximated in polynomial time within a ratio of* $(1-1/e+\epsilon)$ *for some* $\epsilon > 0$*, then* $NP \subset TIME(p^{O(\log \log p)})$*, where* p *is the cardinality*

Algorithm 1 Prioritization Algorithm

Input: Two sets $\mathbb{S}_1$ and $\mathbb{S}_2$

Output: An order of elements in $\mathbb{S}_2$

Set $\mathbb{T} \leftarrow \mathbb{S}_1$
FIFO Queue $\mathcal{Q} \leftarrow \{\}$
while $\mathbb{S}_2 \neq \emptyset$ **do**
    $X \leftarrow \arg\max_{X \in \mathbb{S}_2} \mathcal{V}(\mathcal{MC}_{X|\mathbb{T}})$
    $\mathbb{T} \leftarrow \mathbb{T} \cup \{X\}$, $\mathbb{S}_2 \leftarrow \mathbb{S}_2 - \{X\}$, $\mathcal{Q}$.enqueue(X)
end while
Return $\mathcal{Q}$

of the set (as $|\mathbb{S}_2|$ in Definition 4).

Theorem 1 directly implies that achieving a better approximation ratio than $(1 - 1/e)$ for MaxInfo is NP-hard. Thus, we have the following Corollary.

Corollary 1 (Approximation Bound for MaxInfo)**.** *Achieving approximation ratio* $(1 - 1/e + \epsilon), \forall \epsilon > 0$ *for* MaxInfo *is NP-hard.*

2.2.3 Greedy Algorithm

In this section, we outline our prioritization algorithm. The idea of the algorithm is to give higher transmission priority to data with larger marginal coverage value *w.r.t.* the dataset at the receiver side.

We now prove that the approximation ratio of Algorithm 1 is $(1 - \frac{1}{e})$.

Lemma 1. *For any $n \leq |\mathbb{S}_2|$, if $\mathbb{R}$ is the set of the first n elements of the queue output by Algorithm 1, we have*

$$\mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{R}}) \geq (1 - 1/e) \cdot \mathcal{V}(\mathcal{C}_{\mathbb{S}_1 \cup \mathbb{R}'}), \forall \mathbb{R}', |\mathbb{R}'| = |\mathbb{R}| = n,$$

*where $\mathbb{S}_1$ (*resp. *$\mathbb{S}_2$) is the dataset at the data receiver (*resp. *provider) side.*

The proof of Lemma 1 is similar as that in [18], except that in [18] $\mathbb{S}_1 = \emptyset$. Hence, we do not repeat the proof here. Lemma 1 directly implies the following theorem.

Theorem 2 (Approximation Ratio)**.** *The coverage value of the transmitted set based on the order output by Algorithm 1 is a $(1 - 1/e)$-approximation of* MaxInfo*.*

Note that the approximation ratio matches the approximation bound in Corollary 1.

2.2.4 Transmission Protocol Design

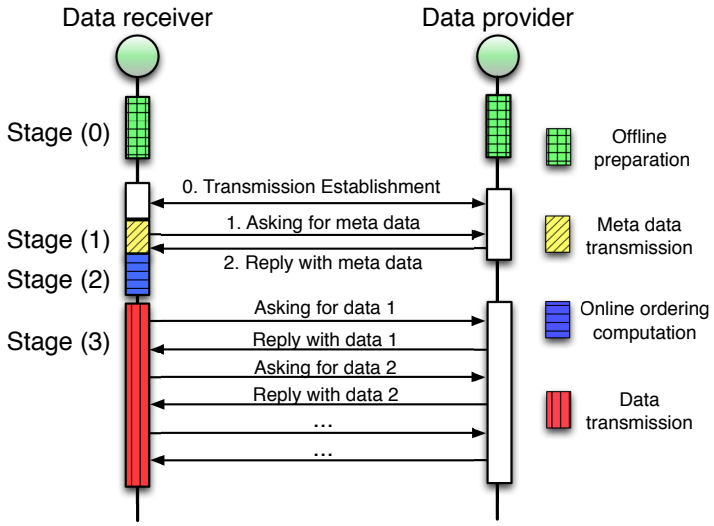


Figure 2.4: Transmission Protocol Illustration

In this section, we present the transmission protocol, as illustrated in Fig. 2.4. Since we target mobile platforms, the transmission is occurs in a disruption-tolerant (DTN) fashion; a device shares its data with a peer or offloads to a backend server when the corresponding connection is established. Thus, each transmission session (in which case we say the device is *online*) is followed by an idle session (when we say the device *offline*).

Each transmission session consists of three stages; (1) meta data transmission, (2) online ordering, and (3) data transmission. Stages (1) and (2) are the transmission overhead. Metadata, here, refers to the (information space) coordinates of data objects available at each node. In an NDN-based implementation, these coordinates can be computed from data names (using the map function). Hence, in our implementation, metadata refers to data object names. The idea being that data object names are generally much shorter than the data objects themselves. Hence, it makes sense to exchange the names first, then let each node specifically request from the other the named data objects it deems complementary (i.e., not redundant with) its own.

Before transmission, an offline preparation operation that generates the meta data needs to be carried out to reduce the overhead of the online ordering computation. We now present the

offline preparation algorithm and online prioritization algorithm in detail as follows.

Offline Preparation

The offline preparation stage outputs a meta data file which contains a list of data names as well as the overlap set of each *heavily informative* data point as described below. (The overlap set of data X contains any data Y *s.t.* $\mathcal{C}_Y \cap \mathcal{C}_X \neq \emptyset$.)

Consider two data points X and Y. If the coverage of X greatly overlaps with that of Y, then after X has been transmitted, Y carries little extra information. Thus, we introduce a constant threshold $\beta > 1$, such that, when the distance between X and Y is smaller than $\frac{1}{\beta}$ we only need to consider one of them (*w.l.o.g.*, say X) in the online prioritization algorithm. The other is put in the lowest priority bin for transmission. We apply this rule repeatedly until no more points can be assigned lowest priority. The surviving points (not assigned lowest priority) are called *heavily informative* data points. Note in particular that if X has no neighbors Y whose distance from X is smaller than $\frac{1}{\beta}$, then X will never be assigned lowest priority and is therefore a heavily informative data point. The heavily informative dataset contains all the data points like X.

Our offline preparation algorithm determines for each new data point whether or not it is heavily informative. If it is heavily informative, it adds the point to the metadata file to be exchanged on contact with another node. It is incremental in the sense that when new data points arrive, we do not need to redo the preparation for old data. The pseudocode for the offline preparation algorithm is presented below.

In our online ordering stage, we only need to consider the heavily informative dataset. The parameter β controls the cardinality of the set of highly informative data. The smaller β is, the smaller the cardinality of this set. In practice, we can use β as a knob to trade-off the accuracy and the time efficiency in the online prioritization as discussed below.

Online Prioritization

Online prioritization is described in Algorithm 3. After metadata (i.e., names of heavily informative objects) have been exchanged, the receiver calculates the marginal coverage value of

Algorithm 2 Preparation Algorithm

Input: Existing dataset $\mathbb{S}$, existing meta data file, newly arrived dataset $\mathbb{T}$
Output: Updated meta data file

From meta data, get the heavily informative dataset $\mathbb{H}$ of $\mathbb{S}$
Sort $\mathbb{H}$ based on the lexicographical order of data coordinates in the k-D information space
$\mathbb{D} \leftarrow \emptyset$, $\mathbb{N} \leftarrow \emptyset$
for $\forall S \in \mathbb{T}$ **do**
    Use binary search to find its overlap set $\mathcal{O}_S \subseteq \mathbb{H}$
    if $\exists E \in \mathcal{O}_S$, *s.t.* $S \simeq E$ **then**
        $\mathbb{D} \leftarrow \mathbb{D} \cup \{S\}, \mathbb{T} \leftarrow \mathbb{T} - \{S\}$, **continue**
    end if
    Add S to the overlap set of any element in $\mathcal{O}_S$
    Insert S into $\mathbb{H}$ *s.t.* $\mathbb{H}$ remains sorted
    $\mathbb{N} \leftarrow \mathbb{N} \cup \{S\}$
end for
Add the name following by the overlap set of each data in $\mathbb{N}$ to the front of the meta data file
Append names of data in $\mathbb{D}$ to the end of meta data file
Return the meta data file

each data object in the highly informative set *obtained from the data provider*, and puts the these values into a max heap. Then, it sends a request for the data object D popped from the max heap, and at the same time updates the marginal coverage value of each data object in the overlap set of D and do the standard heapify. This process continues until the max heap is empty, then, if the connection is still up, the receiver starts to pull data that not in the highly informative set in FIFO order.

Overhead Analysis

The time complexity of Algorithm 3 is $O(m \log n + mn^{\frac{k-1}{k}} + m\beta^k \log m)$, where n is the number of highly informative data in $\mathbb{S}_1$, m is the number of highly informative data in $\mathbb{S}_2$, and β^k is the size bound of a overlap set as stated in the offline preparation section. In the worst case, $n = |\mathbb{S}_1|$ and $m = |\mathbb{S}_2|$, however both of them are related to the parameter β. Thus β serves as a knob to trade-off prioritization accuracy and computational efficiency in our algorithm. We will further study the parameter β in the overhead evaluation.

Algorithm 3 Online Prioritization Algorithm

Input: The highly informative dataset $\mathbb{S}_1$ on the receiver side which is sorted based on the lexicographical order of data coordinates in the k-D information space, meta data from the data provider

Output: The transmission order (represented by a FIFO queue $\mathcal{Q}$)

Initiate a Max-heap $\mathcal{H}$ that stores and sorts data points according to their associated values
for Each heavily informative S in $\mathbb{S}_2$ **do**
    Use binary search to get its overlap set $\mathbb{T} \subseteq \mathbb{S}_1$
    Calculate $v_S = \mathcal{V}(\mathcal{MC}_{S|\mathbb{T}})$
    $\mathcal{H}$.add(S, v_S)
end for
Initiate a FIFO queue $\mathcal{Q}$
while $\mathcal{H} \neq \emptyset$ **do**
    $X \leftarrow \mathcal{H}$.popMax()
    ▷ At the same time send request X to the data provider.
    $\mathcal{Q}$.enqueue(X)
    For each data S in X's overlap set $\mathcal{O}_X$, update $v_S = \mathcal{V}(\mathcal{MC}_{S|\mathbb{T}\cup\mathcal{Q}})$, and update the heap $\mathcal{H}$
end while
Append all other data in $\mathbb{S}_2$ to $\mathcal{Q}$
Return $\mathcal{Q}$

2.3 Evaluation

In this section, we study the performance of Minerva. We first describe the experimental setup and evaluation methodology of Minerva for both real-phone based experiments and simulations using the T-Drive dataset collected by MSRA [3]. Then we present evaluation results.

2.3.1 Experimental Setup and Methodology

Minerva is designed for social sensing applications with resource constraints. Thus, we need to evaluate two aspects of the system: (i) the overhead of data prioritization, and (ii) the application performance, measured in terms of application-level metrics; namely, information coverage. The following methodology was used for evaluation:

- *Data prioritization overhead:* In order to measure overhead under a wide set of workloads, we generate synthetic load (i.e., synthetic data to be transmitted) that can be easily parameterized to represent a large set of relevant properties. These properties include the size of the data set, the dimensionality of the data, and the degree to which the data is

redundant. We then test the overhead of prioritizing such data on real phones.

- *Application-level performance:* In order to evaluate application-level performance metrics (namely, coverage), we find an actual data trace of a participatory sensing application. We then compare coverage when Minerva is used and when other data transmission schemes are used, given a simplified network simulator.

To accomplish the above, we implemented Minerva on Google Galaxy Nexus smartphones [1], equipped with a 1.2 GHz dual-core CPU, 1GB RAM, and 802.11a/b/n WiFi radio with Android OS 4.1. Minerva is implemented using the Java language on top of PARC's CCNx prototype software [2]. The overhead study is conducted in an outdoor environment on real phones. The application performance study uses a real-world taxi trace dataset, the T-Drive dataset [3,66,67], which contains the GPS trajectories of 10,357 taxi cabs during the period from February 2nd to February 8th, 2008 in Beijing. The total number of points in this dataset is about 15 million and the total distance of the trajectories is around 9 million kilometers. The trajectories covered are shown in Figure 2.5.

Figure 2.5: Simulation Data Illustration
(Borrowed from T-Drive dataset [3])

2.3.2 Overhead of Minerva

A key goal in the overhead study is to understand the overhead of data prioritization (which includes the overhead of meta-data transmission for purposes of computing priorities) for a wide set of workloads. Hence, synthetic data is used. In this study, workload generation does not attempt to mimic characteristics of any specific application. Rather, it attempts to investigate overhead under a broad range of conditions that affect it. These include, the size of the data set (in terms of the number of objects), the dimensionality of data, and the degree of redundancy among data items.

To explore the effect of data dimensionality, we generate data by (uniformly) sampling from a k-dimensional box of unit size in the information space, where k is a configurable parameter that represents the number of features (i.e., information space dimensions) considered in the Minerva prioritization algorithm. We use a unit box and uniform sampling because it allows us to easily control the degree of data redundancy by tuning the value of coverage interval associated with individual data points. We focus on overhead only (as opposed to the time it takes to send the data objects). Hence, we measure the overhead of sending meta-data and computing priorities only. At the of this overhead all objects are properly prioritized and ready for transmission. The results of the overhead study are shown in Table 2.1.

The data set parameters considered in the table are (1) the number of features, (2) the number of data points, (3) the coverage interval per point (and hence degree of redundancy of data), and (4) the value of $1/\beta$ as defined in Section 2.2. The more data points are considered, the more computation is needed for data prioritization. Similarly, the larger the coverage interval of individual data points, the higher the redundancy (or the probability that two data points overlap in coverage), and hence the higher the computation overhead of redundancy-minimizing prioritization. The value of $1/\beta$ is a parameter of our algorithm that indicates its tolerance of imprecision in redundancy minimization. The higher the $1/\beta$, the more approximate the prioritization, and the lower the data prioritization overhead, as discussed in Section 2.2. Rows 1-12 of Table 2.1 show the total time in meta-data exchange and prioritization between two android phones that have the same amount of data points on both phones. Rows 13-16 show the corresponding overhead when an android phone uploads data to a back-end server with a

3.10GHz CPU and 8GB RAM.

Table 2.1: Overhead of Minerva

index	dataset features (# dim, # points, interval, $1/\beta$)	overhead(s)
1	2 , **250** , 0.05 , 0.1	0.321 ± 0.165
2	2 , **500** , 0.05 , 0.1	0.837 ± 0.208
3	2 , **750** , 0.05 , 0.1	3.070 ± 1.240
4	2 , **1000** , 0.05 , 0.1	7.205 ± 2.579
5	2 , 500, **0.01**, 0.1,	0.339 ± 0.071
6	2 , 500 , **0.03** , 0.1	0.582 ± 0.104
7	2 , 500 , **0.05** , 0.1	0.837 ± 0.208
8	2 , 500 , **0.07** , 0.1	1.667 ± 0.320
9	2 , 500 , 0.05 , **0.15**	0.700 ± 0.140
10	2 , 500 , 0.05 , **0.2**	0.626 ± 0.145
11	**3** , 500, 0.05 , 0.1	0.257 ± 0.093
12	**4** , 500, 0.05 , 0.1	0.204 ± 0.040
13	2, (10000, 500), 0.01, 0.1	0.076 ± 0.044
14	2, (100000, 500), 0.01, 0.1	1.152 ± 0.093
15	2, (1000000, 500), 0.01, 0.1	4.773 ± 0.537
16	2, (1000000, 500), 0.01, 0.2	0.727 ± 0.104
17	WiFi connection establish time	2.002 ± 0.106

From the table, we observe that the overheads increase as the number of data points increase (rows 1-4) or as the coverage interval increases (rows 5-8), but decrease as $1/\beta$ increases (rows 9-10), which corroborates expectations. When the number of data points is 1000 on both phones (row 4), Minerva takes about seven seconds to prioritize and all objects, which is unacceptable. The overhead drops off sharply with size of the data set (rows 1-3). With 500 objects (row 2), the overhead is less than one second, which is tolerable. Measurements reported in the next section show that one can send roughly $250K$ bytes during that time. Hence, if objects are $250K$ bytes long, the overhead of prioritizing 500 objects is roughly equal to the transmission time of one object. In other words, the prioritization overhead is acceptable as long as the individual objects are sufficiently long.

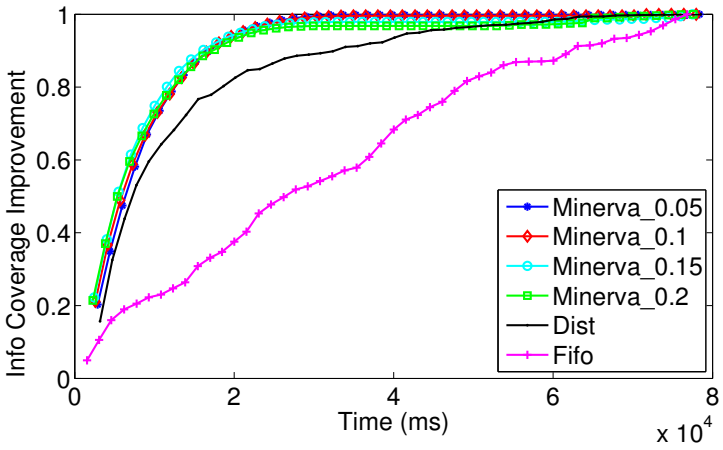
The effect of the number of features on overhead is shown in row 11 and 12 in the table. It can be seen that the overhead decreases in higher-dimensional spaces (all else being equal), because for the same number of data points and the same coverage interval, higher dimensionality means a sparser space, and hence less redundancy, and less overhead for redundancy-minimizing

prioritization.

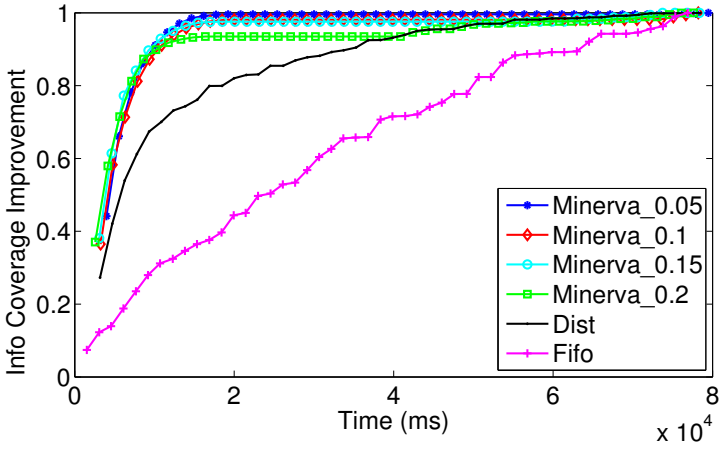
The overheads when data is offloaded from a mobile device to the back-end server are shown in Row 13 to 16. The number of data points is set to 500 on the participant side, and the number of data points on server side is set to be $10,000$, $100,000$, and $1,000,000$. We observe that as the number of data points increases on the server side, the overhead grows. We can also observe that the slope of overhead increase becomes smaller when the number of data points at the server side becomes larger. The reason is that the coverage improvement grows submodularly; when the server already got a large enough amount of data, the probability that a new data point is redundant is close to 1.

Next, we study (in Fig. 2.6) the coverage achieved with Minerva transmissions between two smartphones using the synthetic data. The number of data points is set to 500 on both phones, and the coverage interval is set to be 0.063, 0.077, and 0.089 (thus, in expectation, one data point overlaps with 2, 3, and 4 other points respectively). For each interval value, we plot the coverage improvement using Minerva (with $1/\beta \in \{0.05, 0.1, 0.15, 0.2\}$), Dist (a distance-based data selection algorithm used in PhotoNet [55]) and FIFO. The x-axis is the time in milliseconds that starts right after the connection is established. The y-axis denotes the normalized information coverage at the receiver side, where a coverage of 1 is equivalent to transmitting all sender data. Remember that the objective of prioritization with Minerva is maximize the coverage of transmitted data (i.e., achieve close to 1 coverage as early as possible during transmission).

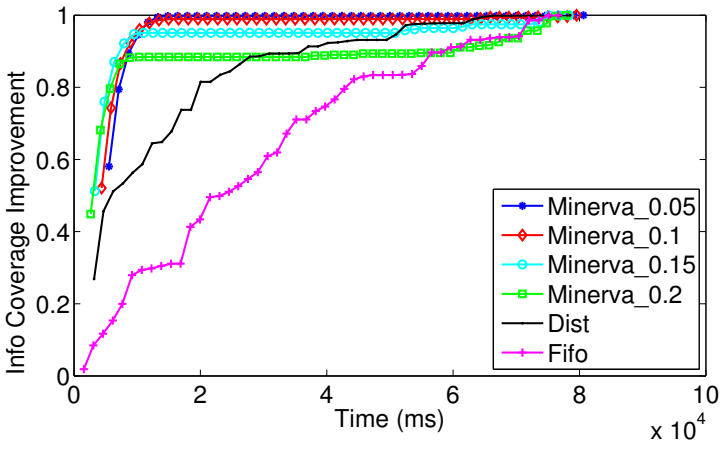
From Fig. 2.6, we observe that Minerva outperforms the other algorithms in general in that it achieves higher coverage earlier on. The larger the coverage interval of individual objects, the better Minerva performs. From the figure, to get 80% coverage, Minerva uses 10 (8 and 5 *resp.*) seconds for a coverage interval of 0.063 (0.077 and 0.089 *resp.*). Dist uses around 20 seconds to achieve 80% coverage, while FIFO takes more than 50 seconds. Minerva coverage also decreases somewhat with increased $1/\beta$ due to the approximation involved.



(a) Coverage Interval 0.063



(b) Coverage Interval 0.077



(c) Coverage Interval 0.089

Figure 2.6: Performance of Minerva with Different Coverage Intervals and $1/\beta$ Values in Phone-based Experiments with Synthetic Data

2.3.3 Large-scale Trace-based Evaluation Results

In order to test application-level performance with Minerva in large-scale applications, we emulate a hypothetical *real-time street view* application. This application applies social sensing in a future where vehicles are equipped with cameras. Participants are requested to send pictures from their car's cameras to the base station when they encounter an access point. These pictures are then used on the server to provide a real-time view of city streets on demand.

In this evaluation, we run simulations on real taxi traces in Beijing (the T-Drive dataset). We consider data within the area from latitude 39.5^oN to 40.5^oN and from longitude 116^oE to 117^oE, where most data resides. In the simulation, we assume that there are two sinks that collect data for the back-end server as indicated by the two stars in Fig. 2.5. They are located in two relatively busy roads, where cars can have a higher chance of offloading their data. Cars are assumed not to share data with each other. In our simulation, if the distance between a car and a sink is smaller than 200 meters, we assume that the car can offload data. We assume that each data point (a picture) is 1M bytes long.

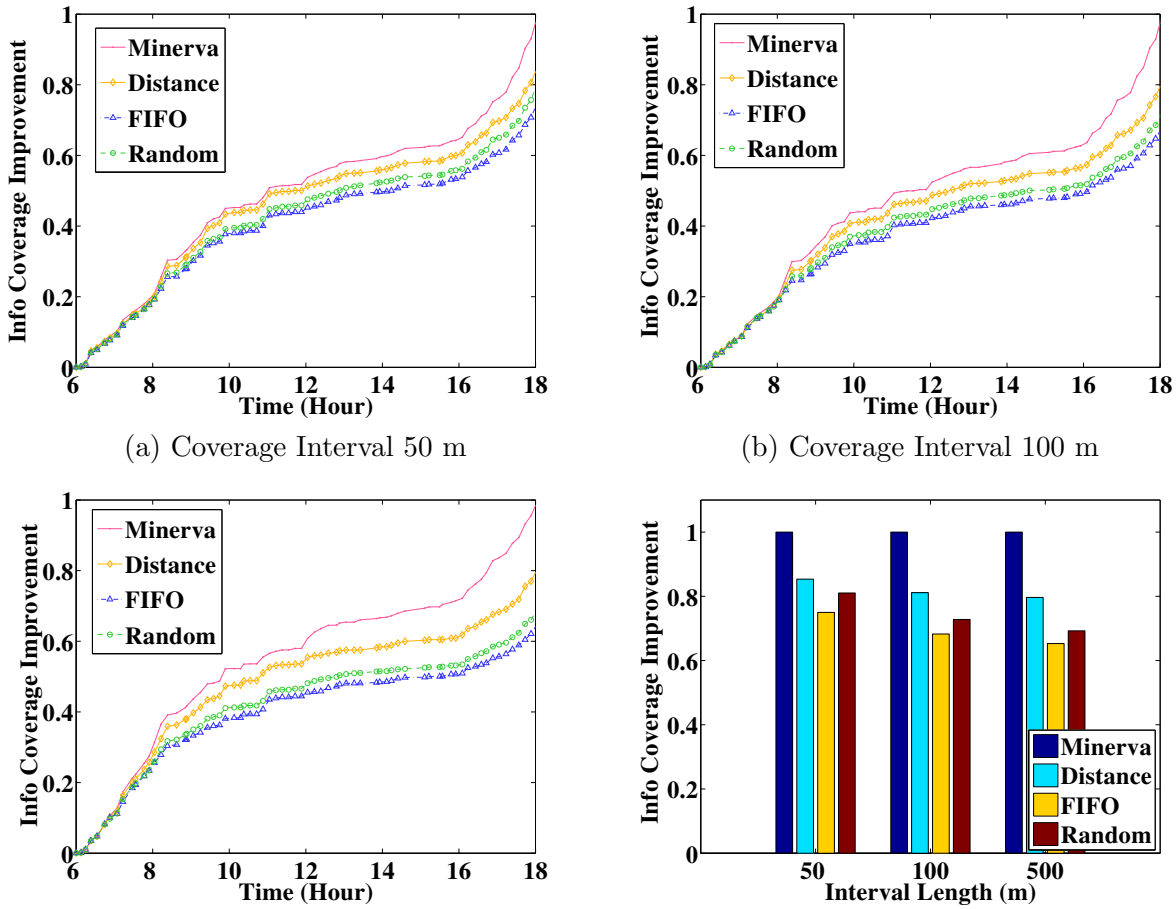
We determine the transmission duration of each car by examining its speed when it enters the transmission range of a sink; the speed can be estimated from the time and location information of the latest GPS samples.

In order to obtain a realistic WiFi transmission rate, we conduct a small experiment where we use a smartphone to send a long file over WiFi in an outdoor environment to a desktop in our lab. The average data transmission rate is found to be approximately 250KB per second, and is set accordingly in the simulation.

In our simulations, we consider two features per data sample, the latitude and the longitude. The coverage interval for each data point is set to 50, 100, and 500 meters respectively in subsequent simulations. We simulate for 10 hours' data (1,500,000 points) and assume at the very beginning the server does not have any data.

The results are shown in Fig. 2.7. From Fig. 2.7, we observe that at the beginning of data collection, the four algorithms compared yield similar performance. Minerva is slightly worse than the others due to its overhead. After collecting data for one hour, Minerva begins to outperform the others. The reason is that Minerva is designed for eliminating redundant data.

At the beginning, there is little redundancy since the server has only a limited amount of data. Hence, the overhead is not paying off in terms of application-level metrics. As more data is collected, more redundancy is exploited by Minerva. Eventually, Minerva outperforms all other algorithms in terms of attained coverage. Note that the slope of the coverage curves changes over the course of the day (where the x-axis is time of day). This is because more data is collected during rush hour, roughly between 6-8 am and 4-6 pm. In summary, evaluation shows that redundancy-minimizing data prioritization has promise in terms of improving coverage of the physical environment in social sensing. However, overhead remains an issue, which reduces applicability except where sensed objects are sufficiently large (e.g., multimedia objects).



(a) Coverage Interval 50 m

(b) Coverage Interval 100 m

(c) Coverage Interval 500 m

(d) Final Coverage Improvement

Figure 2.7: Performance of Minerva with Different Coverage Intervals in Large-scale Real-world GPS Trace Simulations

Chapter 3

Data Acquisition under Timeliness Requirements

3.1 System Overview

Reducing information redundancy and maximizing coverage, as discussed so far, is a rather general goal when carrying out crowd sensing tasks. Oftentimes, more specific application objectives can be exploited to further improve information delivery efficiency, which we explore from hereon.

We consider crowdsensing applications (such as disaster response and recovery), where information at data sources is not immediately accessible to the collection point due to resource constraints. Retrieval of data from sources needs to be carried out very carefully such that the least amount of resource is used. Decisions based on received data need to be made at different time-scales depending on their criticality. Hence, in addition to resource bottlenecks, we have different application deadlines on information acquisition.

The goal is to fetch data from sources in a way that minimizes system resource consumption while meeting request deadlines. In general, a decision-maker might have several information needs, each translating into a query. Hence, multiple queries could be issued at the same time. We assume that each query is translated into a set of objects that must be retrieved. A retrieved object can be subjected to a test that evaluates a predicate. For example, an image of a building can be inspected to determine if the building is occupied. Given application logic and data dependencies, it is desired to determine which objects to retrieve in order to answer all queries with minimum resource consumption while meeting their deadlines.

In the rest of the discussion, we describe cost as the bandwidth needed for retrieving a data item (i.e., the size in bytes of the retrieved data). However, our actual algorithm is not restricted to this cost definition—it can operate on any cost metric definition that is additive.

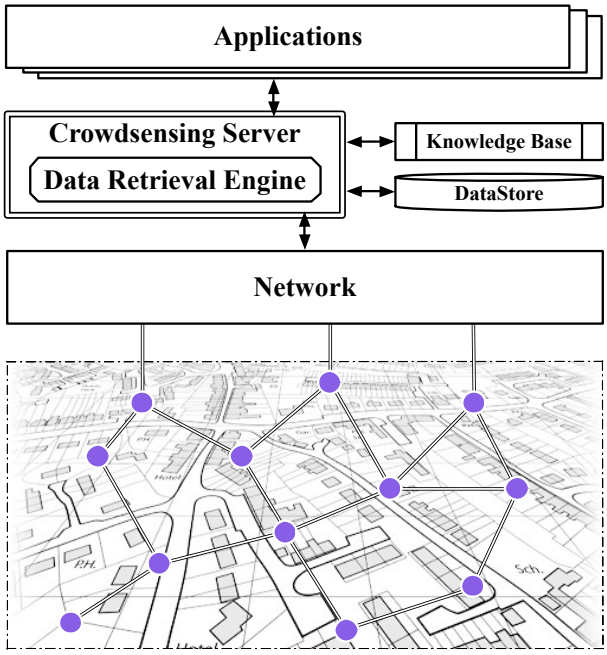


Figure 3.1: System Design

For example, in a scenario where energy is the most important resource, we can easily define cost to be a node's energy consumption. We assume that the concurrent retrieval latency of multiple data object is the maximum single retrieval latency among all the single objects.

Our resource-limited crowdsensing system design is depicted in Fig. 3.1. As seen, as application requests arrive, our crowdsensing engine uses the knowledge-base to convert the requests into boolean logic expressions. It then queries the datastore for the likelihood that different predicates in those expressions would evaluate to $true$ or $false$, to guide object retrieval accordingly for best odds of shortcutting logic expressions. It also fetches the estimated retrieval latencies between the collection point and the remote data sources. The data retrieval engine takes these estimates as input, computes a minimum cost retrieval plan, and performs the actual data retrieval. As data comes in, requests are updated, and the datastore is enriched with the latest data. Note that, in general, some data pertinent to a query might already be available. If so, part of the expression is immediately evaluated. The above applies to the remaining part, if any. Hence, without loss of generality, we focus on retrievals of missing data only.

Q	all requests	$n_E(q)$	the src leads to min $c_E(q)$
q	a request	$n_E(q, \mathcal{R})$	$n_E(q)$ given $\mathcal{R}$
$\mathcal{R}$	srcs to retrieve	$l_E(q)$	**E**[latency] of q
r_k	k^{th} src in $\mathcal{R}$	$l_E(q, \mathcal{R})$	$l_E(q)$ given $\mathcal{R}$
n_j	data source j	$c_E(q)$	**E**[cost] of q
s_j	true prob. of n_j	$c_E(q, \mathcal{R})$	$c_E(q)$ given $\mathcal{R}$
c_j	cost of n_j	$q_{\|\bullet}$	apply data of src $\bullet$ to q
$\mathcal{M}$	retrieved srcs	q_l	request of least laxity
p_i	test i	b_i	short-circuit prob. of p_i

Table 3.1: Notation Table

3.2 Analyses and Algorithms

In this section, we first talk about what analyses and building-block operations need to be done for each individual crowdsensing application request, and then discuss how the crowdsensing system carries out data retrievals when serving multiple requests concurrently under our targeted resource-constrained settings.

3.2.1 Single-Request Analyses

Given a single request q (please refer to Table 3.1 for all notations), we would like to compute the best (cost-minimizing) next source $n_E(q)$ for retrieving data from and the corresponding expected bandwidth cost $c_E(q)$ assuming sequential retrieval for all the rest of the involved data sources. We focus our discussion on bandwidth only, as the same methods can be used for the expected retrieval latency. Notice that this is generally *not* true because latencies for concurrent retrievals are not additive. But since all expected values that we discuss in this section are under the assumption of sequential retrievals, this *is* true.

Please note that we default to sequential retrieval because of our original objective of minimizing costs—more parallel retrievals lead to higher probability of unnecessarily retrieving data objects that could otherwise be avoided due to short-circuiting. Only when a query might miss its deadline if we stick to sequential retrieval do we schedule concurrent retrievals in order to decrease the total query resolution time.

We need to not only be able to compute the next best source and the expected cost when given just the request, but be able to do so when also facing a set $\mathcal{R}$ of (zero or more) sources

that have already been selected for the next round of data retrieval. Formally, for a request q, let P be its set of tests, where each test p_{q_i} corresponds to a particular data source n_j, and thus has associated with it a retrieval bandwidth cost c_{n_j} and success (evaluating to $true$) probability s_{n_j}. Given a set $\mathcal{R}$ of data sources that are already selected for retrieval for the next round, we want to compute for request q the next best source $n_E(q, \mathcal{R})$ to retrieve data from and q's corresponding expected cost $c_E(q, \mathcal{R})$. The reason why we need to do this will be more clear, if not already so, when we talk about the actual data retrieval algorithm for multiple concurrent requests in Sec. 3.2.2.

If set $\mathcal{R}$ is empty, then the problem becomes the standard PAOTR problem, already known to be NP-hard. It is quite obvious that it is still so with an non-empty $\mathcal{R}$. Casanova et al. [9] give a greedy algorithm for computing q's best data retrieval plan without such a set $\mathcal{R}$ present. We denote it as $c_E(q, \varnothing) = c_E(q)$, and briefly introduce how this greedy algorithm works. For a test p_i with cost c_i and success probability s_i, we define the test's short-circuit probability to be the probability that performing it will cause its parent conjunction or disjunction to be resolved without needing to perform its sibling tests. Then p_i's short-circuit probability to cost ratio is $b_i = (1 - s_i)/c_i$ in a conjunction, or $b_i = s_i/c_i$ in a disjunction. As each boolean expression can be converted into its disjunctive normal form (i.e., a disjunction of one or more conjunctions of one or more tests), for a conjunction we order its tests according to their (descending) b values. With this order $j_1, j_2, \ldots, j_m$, the expected cost of the conjunction can then be computed as

$$c_{conj} = c_{j_1} + s_{j_1}(c_{j_2} + s_{j_2}(c_{j_3} + \ldots + s_{j_{m-1}} c_{j_m}) \ldots),$$

which, together with its derived success probability $\prod s_i$, yields the current conjunction's short-circuit probability to cost ratio $\frac{\prod s_i}{c_{conj}}$ for its parent disjunction, which can then be used to select the best conjunction to process first, the same way how the best test for a conjunction is selected.

After each test p_t is selected, the next best test p_{t+1} to be selected can depend on the evaluation result of p_t. So, both possibilities $p_{t+1}^{(T)}$ (upon $p_t == true$) and $p_{t+1}^{(F)}$ (upon $p_t == false$) are considered. The final retrieval plan computed is therefore a binary decision tree, where the evaluation result of each test performed dictates which source to retrieve data from and test

to perform next. For ease of presentation, we use $n_E(q)$ and $c_E(q)$ to denote the algorithms for computing a request q's next best source to retrieve data from (i.e., the root node of the binary decision tree) and the corresponding expected bandwidth cost. Please note that data sources' costs and the corresponding tests' success probabilities are the inputs to the mentioned algorithms. We, however, assume their static availabilities throughout our discussions, and thus omit them as being part of the input.

Note that due to the binary decision tree nature of the algorithm $n_E(q)$, it has exponential run time in request q's number of tests. This is not a problem in practice where a request contains say 5 to 10 tests. If, however, cases need to be handled where inputs are quite large (for example, a request consists of hundreds or thousands of tests in it), a simplified (and computationally more efficient) version of algorithm $n_E(q)$ can be used, which simply ranks all the conjunctions first and then proceeds with data retrievals according to the conjunction order and then the data source order within the same conjunction. This effectively reduces the computation complexity from exponential growth to $\mathcal{O}(|q| \lg |q|)$, and only slightly underperforms the original version in terms of ratio to optimal retrieval plan [9].

Algorithm 4 $c_E(q, \mathcal{R})$ - Request's Minimum Expected Bandwidth Cost Given A Predetermined Retrieval Set

Input: Request q, predetermined retrieval set $\mathcal{R}$
Output: The minimum expected bandwidth cost of q given $\mathcal{R}$

1: **if** q is already resolved **then return** 0
2: **else**
3: **if** $\mathcal{R} == \varnothing$ **then**
4: **return** $c_E(q)$
5: **else**
6: $r_l \leftarrow$ last element of $\mathcal{R}$
7: $\mathcal{R} \leftarrow \mathcal{R} \setminus \{r_l\}$
8: $q_{|r_l^{(T)}}, q_{|r_l^{(F)}} \leftarrow$ request q with $r_l = true, false$ values applied, respectively
9: **return** $s_{r_l} c_E(q_{|r_l^{(T)}}, \mathcal{R}) + (1 - s_{r_l}) c_E(q_{|r_l^{(F)}}, \mathcal{R})$
10: **end if**
11: **end if**

In our case, with the predetermined retrieval set $\mathcal{R}$ present, we can compute request q's expected cost in a recursive fashion. The base case would be when set $\mathcal{R}$ is empty, where the algorithm $c_E(q)$ mentioned above can compute q's cost directly. The recursive case goes as

follows. For a set $\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$ with l elements, we remove from it its last element r_l, apply r_l's two possible values $true/false$ to the request and get two sub-requests $q_{|r_l^{(T)}}$ and $q_{|r_l^{(F)}}$. Then the expected cost of q given $\mathcal{R}$ can be computed recursively as follows,

$$c_E(q, \mathcal{R}) = s_{r_l} c_E(q_{|r_l^{(T)}}, \mathcal{R} \setminus \{r_l\}) + (1 - s_{r_l}) c_E(q_{|r_l^{(F)}}, \mathcal{R} \setminus \{r_l\}),$$

where s_{r_l} is the success probability of the test r_l. The pseudo code is depicted in Algorithm 4.

Again, due to the structure of the binary decision tree, the computational complexity of Algorithm 4 grows exponentially with the cardinality of the set $\mathcal{R}$. When fast decision makings on large inputs are needed, applications can opt to the static version of Algorithm 4 which only explores the most probable value of r_l, avoiding the exponential explosion. As the static algorithm travels a single path from root to leaf on the binary decision tree, the computational complexity is linear in $|\mathcal{R}|$. Therefore the total complexity is $\mathcal{O}(|\mathcal{R}| + |q| \lg |q|)$

Algorithm 5 $n_E(q, \mathcal{R})$ - Request's Best Next Source for Retrieval Data from Given A Predetermined Retrieval Set

Input: Request q, predetermined retrieval set $\mathcal{R}$
Output: q's best next data source for retrieval given $\mathcal{R}$

```
1: if q is already resolved then return ∅
2: else
3:     if R == ∅ then
4:         return n_E(q)
5:     else
6:         V_R ← the vector of most probable outcomes of tests corresponding to R
7:         q_{|V_R} ← request q after applying all values from V_R
8:         return n_E(q_{|V_R})
9:     end if
10: end if
```

Regarding how to compute, for request q, the best source $n_E(q, \mathcal{R})$ to retrieve data from next given the set $\mathcal{R}$, we let each member $r_l \in \mathcal{R}$ assume its most probable outcome determined by its success probability s_{r_l}, and apply this set of most probable values to q and get the resulting request q'. After that we can compute the best source as $n_E(q, \mathcal{R}) = n_E(q')$. The pseudo code is depicted in Algorithm 5. Similar to the notation $c_E(q, \mathcal{R})$ and $n_E(q, \mathcal{R})$, which denote the algorithms for computing request q's best data source and expected cost given predetermined retrieval set $\mathcal{R}$, we use $l_E(q, \mathcal{R})$ to refer to the algorithm for computing q's expected retrieval

latency given a retrieval set $\mathcal{R}$.

The computational complexity of Algorithm 5 is dominated by the sorting operation in $n_E(\cdot)$, resulting in $\mathcal{O}(|q| \lg |q|)$.

3.2.2 Data Retrievals for Multiple Concurrent Requests

Having established several building block algorithms, we are now ready to talk about the data retrieval algorithm for serving multiple concurrent requests. We first describe the various component stages, and then give the complete algorithm at the end.

At each round, the data retrieval engine needs to determine, from the set of all relevant data sources, a subset to retrieve data from. It proceeds by first selecting the one that leads to the minimum total expected cost. This is done by taking the Cartesian product of the set of all data sources and set of all requests, and for each source-request pair (n_i, q_j), computing the expected cost $c_E(q_j, \{n_i\})$. Then, the best source n_r is selected as the data source n that minimizes the sum of its own cost c_n plus the collective sum of the expected costs of all the requests, assuming the predetermined set of sources being the current retrieval set with n inserted, mathematically (and more clearly) as follows,

$$n_r = \arg\min_n \left(c_n + \sum_j c_E \left(q_j, \mathcal{R} \cup \{n\} \right) \right).$$

After n_r is selected, it is added to the retrieval set for this round $\mathcal{R} = \mathcal{R} \cup \{n_r\}$.

The data retrieval engine then checks the expected laxity (laxity = deadline − expected retrieval latency) of all requests. If the minimum expected laxity is negative, then it means the corresponding request q_l is expected to miss its deadline if we proceed with retrieval using the current $\mathcal{R}$ as is. Since all sources in $\mathcal{R}$ are to be retrieved in parallel, having more of q_l's relevant data sources in $\mathcal{R}$ for parallel retrieval might then shorten its expected retrieval latency and in turn make its laxity positive again. Under this intuition, we then pick q_l's best next source $n_E(q, \mathcal{R})$ to add to $\mathcal{R}$ and carry out this laxity checks again with the updated $\mathcal{R}$. This process terminates when no request is expected to miss its deadline given the latest set $\mathcal{R}$. It might also happen that at some point some request q_l cannot be saved from having to miss its deadline, for

example when the retrieval latency of set $\mathcal{R}$ already exceeds q_l's deadline, therefore adding more sources to $\mathcal{R}$ won't help in terms of saving q_l. Under such circumstances we have two options, either backtrack in $\mathcal{R}$ to try to save q_l, or accept that q_l might miss its deadline and disregard it for this current round of planning. In this study we take the latter approach.

As we have taken care of bandwidth and deadlines at this point, it might seem like we are ready to carry out the retrieval of $\mathcal{R}$ for this round; our data retrieval engine, however, actually goes one step further here in preventing unnecessary request delays, as we explain next. We say two sources n_1 and n_2 are *related* if they co-appear in at least one request q. It then follows that the retrieval of n_1 will affect how n_2 is to be chosen in the future, because n_1's data upon retrieval will be applied to the request q, causing $c_E(q, \{n_2\}) \neq c_E(q_{|n_1}, \{n_2\})$. It is easy to see that this effect is transitive: If we build a graph of all the sources where each pair of sources share an edge iff they are *related*, then sources of the same connected component can affect each other in this fashion, either in the next round or after multiple rounds in the future. A pair of sources that are *disconnected*, however, will never affect each other. We then say two sets of sources are *disconnected* if no edge exist between them.

Now coming back to our multi-request data retrieval problem, if all sources in a request q_d are disconnected from the set $\mathcal{R}$, then there is no benefit for us not to start retrieving data from some source relevant to q_d at this current round, because we will not be saving total network bandwidth usage, and are just delaying the processing of this request, adding to the danger of it missing its deadline. Therefore, our data retrieval algorithm behaves as follows. At the beginning of each round (at which point $\mathcal{R} == \varnothing$), a set $\mathcal{Q}$ is initialized to contain all requests. As $\mathcal{R}$ is populated, elements of $\mathcal{Q}$ are removed from $\mathcal{Q}$ only if necessary s.t. $\mathcal{R}$ and $\mathcal{Q}$ remain disconnected. The previously mentioned cost-minimizing and deadline-miss-prevention operations are actually always carried out on the set $\mathcal{Q}$, until $\mathcal{Q}$ becomes empty, at which point, the data retrieval engine is actually ready to retrieve data from all sources in $\mathcal{R}$ in parallel. Upon receiving the data, the engine updates all the requests, and continues onto the next round of planning. The above detailed discussions are also collected and depicted in Algorithm 6.

Assuming there are n sources and m requests. Then the outer loop repeats at most m times. In each iteration, the computation on line 4 costs $\mathcal{O}(n^2)$ time when the static version

Algorithm 6 Multi-Request Serving

Input: The set $\mathcal{M}$ of retrieved data from previous round, the set $\mathcal{Q}$ of all unresolved requests, current time T

Output: The set $\mathcal{R}$ of sources to retrieve data from for this round

1: Use values in $\mathcal{M}$ to update all requests in $\mathcal{Q}$, remove all resolved requests from $\mathcal{Q}$
2: Initialize $\mathcal{R} \leftarrow \varnothing$
3: **repeat**
4: $n_{min} \leftarrow \arg\min_n \left(c_n + \sum_j c_E(q_j, \{n\})\right)$ the source that minimizes the total expected retrieval cost
5: $\mathcal{R} \leftarrow \mathcal{R} \cup \{n_{min}\}$
6: $\mathcal{L} \leftarrow \{q \in \mathcal{Q} | q.deadline - l_E(q, R) - T < 0\}$ the set of requests that are expected to miss their deadline after this round of retrieval
7: **repeat**
8: $q_{urgent} \leftarrow \arg\min_{q \in \mathcal{L}} (q.deadline - l_E(q, R) - T)$
9: $n_{q_{urgent}} \leftarrow n_E(q_{urgent}, \mathcal{R})$
10: **if** $n_{q_{urgent}} == NULL$ **then**
11: $\mathcal{L} \leftarrow \mathcal{L} \setminus \{q_{urgent}\}$
12: $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{q_{urgent}\}$
13: **else**
14: $\mathcal{R} \leftarrow \mathcal{R} \cup \{n_{q_{urgent}}\}$
15: **end if**
16: **until** $\mathcal{L} == \varnothing$
17: $\mathcal{Q} \leftarrow \{q \in \mathcal{Q} | q$ is disconnected from $\mathcal{R}\}$
18: **until** $\mathcal{Q} == \varnothing$
19: **return** $\mathcal{R}$

of algorithm $c_E(q, \mathcal{R})$ is in use, dominating the computational cost from line 4 to 6. The inner loop repeats n times in the worst case. Line 8 costs $\mathcal{O}(mn)$ time to enumerate all requests and compute their latencies, which is the most expensive operation in each inner iteration. Therefore, the computational complexity of Algorithm 6 is $\mathcal{O}(m(n^2 + n(mn))) = \mathcal{O}(m^2n^2)$. In practice, this algorithm is highly parallelizable, more specifically the operations on Line 4, 6, and 8. Thus in case of extremely large inputs (e.g. hundreds of concurrent requests with thousands of relevant data sources), parallel computing techniques (e.g. GPGPU) can be utilized to boost performance. For more realistic input sizes the computing time is far smaller than the network transmission delay time for our targeted scenarios.

3.3 Evaluation

In this section we evaluate our proposed data retrieval algorithm (which we will mark as *greedy* for ease of presentation) for resource-limited crowdsensing through simulations. First, we examine our algorithm's performance against several baseline methods under various settings. Then we specify a concrete application scenario and present our results and findings.

We compare our algorithm to the following four baseline methods in the evaluation.

- *Random* (*rnd*) — The elements of the retrieval set for each round are randomly selected from the set of all relevant data sources of the remaining requests. Here we experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time.

- *Lowest Cost Source First* (*lc*) — All requests' relevant data sources are put together and sorted according to their costs in ascending order. The retrieval set is then populated using the lowest cost sources. We also experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time.

- *Most Urgent Request First* (*uq*) — All requests are sorted according to their expected laxities in ascending order. The retrieval set is then populated by randomly selecting sources from the most urgent (least laxity) request. We also experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time. Note that if the most urgent request's relevant sources have been exhausted, we move to the next urgent request in line.

- *Most Urgent Request's Lowest Cost Source First* (*uqlc*) — The same as the *uq* approach, except that when selecting sources from the urgent request, the lowest cost source is selected first, as opposed to random selection.

3.3.1 Algorithm Behaviors

We examine our algorithm's behaviors under the following experiment settings. All requests are randomly generated. We experiment with different numbers of tests per request ($\{4, 8, 12, 16, 20\}$, indicating how many remote data sources are generally involved in a single crowdsensing application request), different numbers of conjunctions per request ($\{1, 3, 5, 7\}$, which could possibly

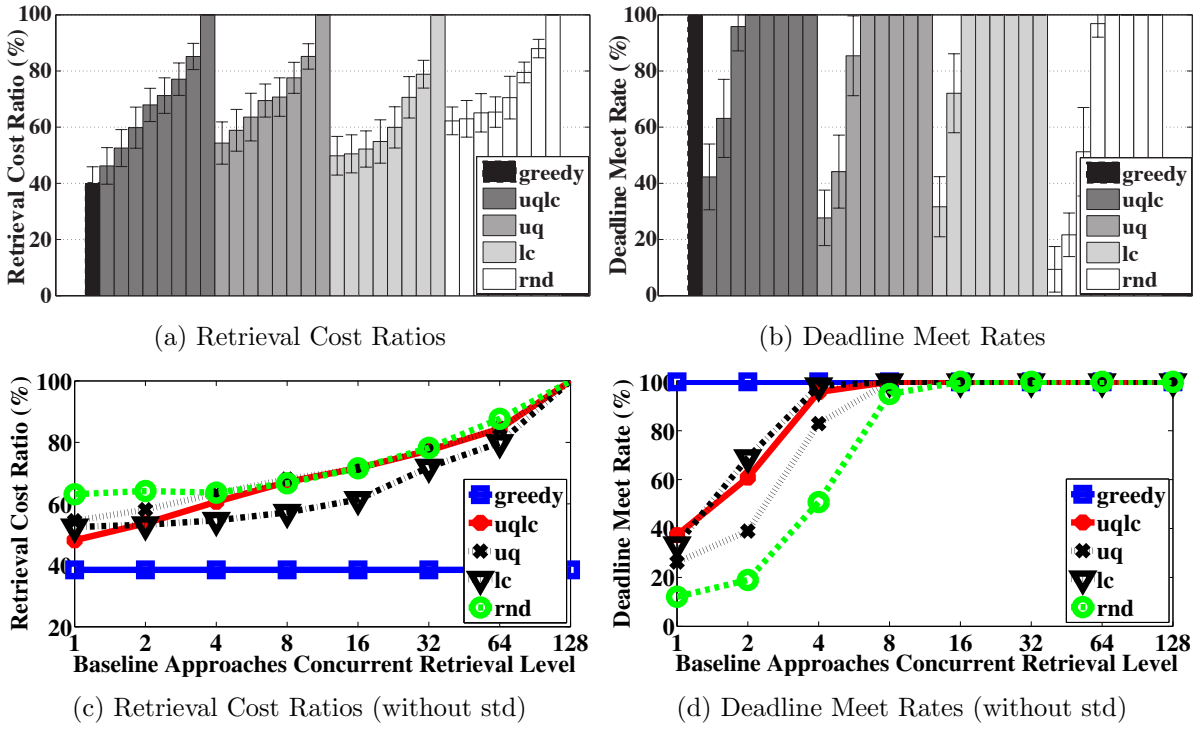
correspond to the number of different options that a sensing application can take), and different numbers of concurrent requests that the crowdsensing system is serving ($\{5, 10, 15, 20, 25\}$, the number of sensing applications are being served all at the same time). All tests' success probabilities are uniformly randomly generated. Besides the uniform distribution on $(0, 100\%)$, we also experiment with ranges $(0, 10\%) \cup (90\%, 100\%)$ and $(40\%, 60\%)$, which represent situations where tests' success probabilities are generally less and more ambiguous in indicating how likely they are to evaluate to either $true$ or $false$ than the $(0, 100\%)$ case. Regarding sources' retrieval costs, we experiment with uniformly on $[1, 10]$, and 2^p where the power p is uniformly randomly generated on $[1, 10]$. Compared to the former, the latter represents a highly heterogeneous network where data collected by different sources can have orders of magnitude of differences in bandwidth costs. We set a source's retrieval latency to be proportional to its bandwidth cost plus a transmission setup latency, and for the actual request deadlines, we experiment with the following different settings: For a request, we take the sum of all its sources' individual retrieval latencies, and multiply it by 0.5, 1, 2, 4, or 8 to use as the request's deadline, representing different levels of "tightness" of the request deadlines.

We first take a look at how our algorithm's performance compares to the baseline methods in general. We set the total number of data sources in the field to be 100, whose bandwidth costs are uniformly distributed on $[1, 10]$ and success probabilities $(0, 100\%)$. We set the number of concurrent requests to be 20. Each request involves 8 individual tests grouped in 3 different conjunctions, and has its deadline set to be twice the sum of its relevant sources' individual retrieval latencies. Each set of experiments is repeated 20 times, for which we report the means and standard deviations of both the retrieval cost ratio, computed as

$$\frac{\text{total bandwidth cost of actually retrieved data}}{\text{total bandwidth cost of all sources involved in all requests}},$$

and the deadline meet rate—the percentage of requests whose resolutions are within their deadlines. Therefore, ideally we want low retrieval cost ratios and high deadline meet rates. Results are shown in Fig. 3.2, of which Fig. 3.2a shows the retrieval cost ratios and Fig. 3.2b the deadline meet rates. In each plot, the heights of the bars represent the mean values, and the error

windows indicate the standard deviations. Results of our algorithm are captured by the leftmost bar, whereas each of the four baseline methods is represented by a cluster of adjacent bars with the same greyscale level, from left to right: *uqlc*, *uq*, *lc*, and *rnd*. Within the same baseline method cluster, the different bars stand for different concurrent retrieval levels: from left to right in the order 1, 2, 4, 8, ..., 128. For the sake of easier comparisons, the same results are also presented in a different manner without the standard deviation error windows, as illustrated in Fig. 3.2c and 3.2d.



(a) Retrieval Cost Ratios

(b) Deadline Meet Rates

(c) Retrieval Cost Ratios (without std)

(d) Deadline Meet Rates (without std)

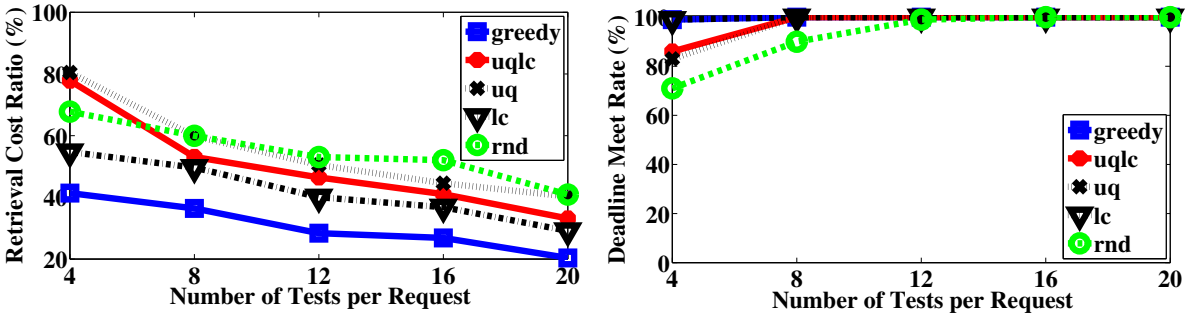
Figure 3.2: Retrieval Cost Ratios and Deadline Meet Rates of Our Algorithm vs Baseline Methods

As can be seen, our data retrieval algorithm achieves a low retrieval cost ratio of about 40%, while maintaining a 99.8% average deadline meet rate. In comparison, among all the baseline strategies, the lowest retrieval cost ratio with 99+% deadline meet rate is achieved by *lc* with 4 concurrent retrievals at each round. Its retrieval cost ratio is about 52%, quite a bit higher than the 40% of our algorithm. For the rest of the baseline methods, in order to achieve 90+% deadline meet rate, the retrieval cost ratios have to be 60% or even greater.

After getting a general sense of how our algorithm and all the baseline strategies perform against each other, we would next like to investigate how various aspects affect the data re-

trieval algorithm's behaviors. Thus for each of the next sets of experiments, we tune one of the system and problem parameters while fixing all the rest to their default values as set in the previous experiment, and look at how our algorithm stacks up against the baseline methods. Note that for clarity of presentation, we pick, from each baseline strategies, a representative concurrent retrieval level. After examining results from all baseline strategies, we notice that with a concurrent retrieval level of 4, usually at least one of the four baseline methods can achieve 100% deadline meet rate without incurring additional retrieval costs. Therefore, we pick the concurrent retrieval level of 4 to be the representative of each of the baseline strategies. Each experiment is repeated 20 times. In order to better show the trend of changes reflected from the results, we omit showing standard deviations, and focus on the average results, similar to Fig. 3.2c and 3.2d.

First we look at how the number of tests in a request affects the data retrieval algorithm's performance. This in practice could indicate the general complexity of the requesting crowdsensing applications. For example, an application request from a user who just wants to find out if she could go get gas from either of the two gas stations near her home could be $gasStationAIsOpen \vee gasStationBIsOpen$, which is rather simple; another application request from a disaster response team that needs to find out how to reach their destination given three candidate routes could look like $(roadAIsGood \wedge roadBIsGood) \vee (roadAIsGood \wedge roadCIsGood \wedge roadDIsGood) \vee (roadEIsGood \wedge roadFIsGood)$, which is a bit more complex than the previous gas station example.



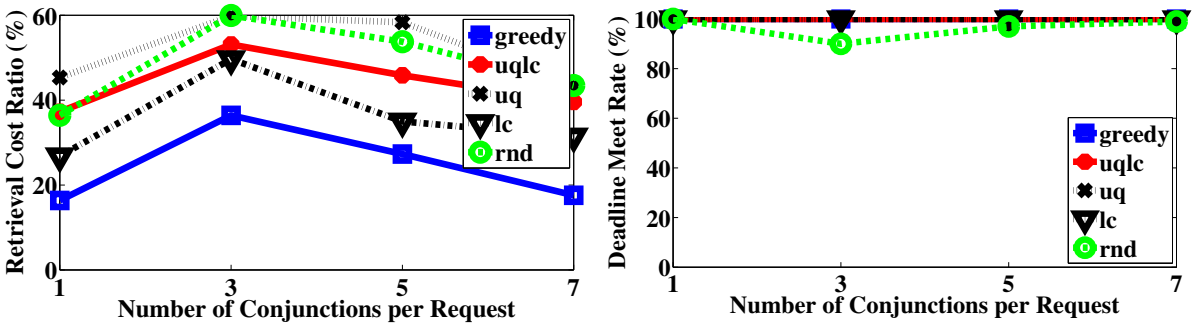
(a) Retrieval Cost Ratios

(b) Deadline Meet Rates

Figure 3.3: Varying Number of Tests per Request

Results are shown in Fig. 3.3. As seen, our algorithm always achieves the lowest retrieval

cost ratios with perfect/near-perfect deadline meet rate. We can observe that as the number of tests per request increases, the retrieval cost ratios of all approaches decrease. This is because more tests per request means generally more tests per conjunction (the number of conjunctions per request is fixed in this experiment). Having more tests included in a request increases the deadline of the request, but then there are more chances for tests within the same conjunction to be short-circuited by a sibling of theirs. Therefore, the total retrieval cost for resolving a request might not increase with the additional tests. So, we observe an downward trend of the retrieval cost ratio and an upward trend of deadline meet rate, as we increase the number of tests in requests. Among the baseline methods, the *rnd* (random) approach generally gives the worst results, which is expected. The *lc* (least cost sources first) approach leads to the second best retrieval cost ratio, which is understandable as minimizing cost is its sole objective. It is interesting to see that *lc* also leads to perfect deadline meet rate, considering the method itself does not concern requests' deadlines at all. One explanation we can think of is that since it always picks the lowest-cost sources, which generally also leads to small retrieval latencies, *lc* can therefore retrieve more data and fast, which helps it with request serving abilities.



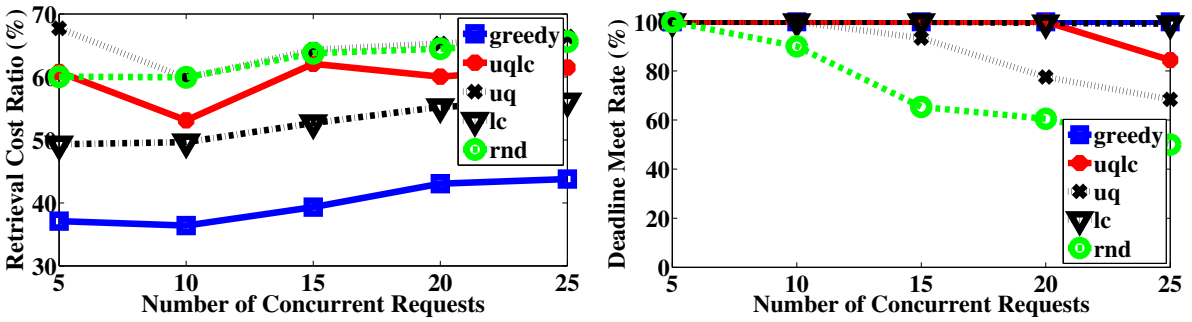
(a) Retrieval Cost Ratios

(b) Deadline Meet Rates

Figure 3.4: Varying Number of Conjunctions per Request

Besides the number of tests, we also look at how the number of conjunctions within a request affects data retrieval algorithm's behavior. This in practice could correspond to the number of different options an application request can be satisfied. Taking the previous disaster-response-team-finding-routes example, the three conjunctions in the request correspond to 3 alternative routes that can be taken (of course different routes could possibly share common road segments

among them). We experiment with 1, 3, 5, and 7 conjunctions per request, and show results in Fig. 3.4. Nothing interesting going on in Fig. 3.4b for deadline meet rates, we focus our attention to the retrieval cost ratios in Fig. 3.4a. Besides the obvious fact that our algorithm achieves the best retrieval cost ratios, we do see an interesting common trend among all approaches—the retrieval cost ratios generally decrease with a growing number of conjunctions per request except for when there is only one conjunction present in a request. This makes sense because the more conjunctions there are within a request (which has a fixed number of tests), the more the request itself looks like a disjunction, which means the more likely it is for a single test's evaluating to *true* to short-circuit all other tests and completely resolve the request. When there is only one conjunction, however, the request itself is then just, well, a conjunction, which means a single test's evaluating to $false$ would completely resolve the request, and hence the low retrieval cost ratio.



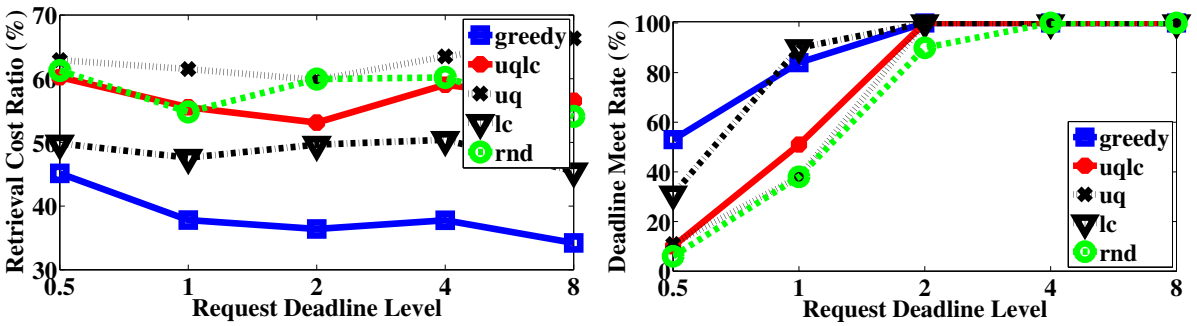
(a) Retrieval Cost Ratios

(b) Deadline Meet Rates

Figure 3.5: Varying Number of Concurrent Requests

We next look at how the number of concurrent requests (which can be a measure of the load on the crowdsensing system) affects performance of various methods. Results are shown in Fig. 3.5. As seen, the increase in the number of concurrent requests has a pronounced effect on (increasing) the retrieval cost ratios and (decreasing) the deadline meet rates on all baseline methods, which is quite expected. We also observe that our algorithm still always achieves the lowest retrieval cost ratio, and, unlike many of the baseline methods, is able to maintain its perfect deadline meet rate in spite of the increase in the request quantities.

Next we look at how different request deadline settings affect the behaviors of our algorithm and the various baseline methods. Different deadlines obviously express different levels of ur-



(a) Retrieval Cost Ratios

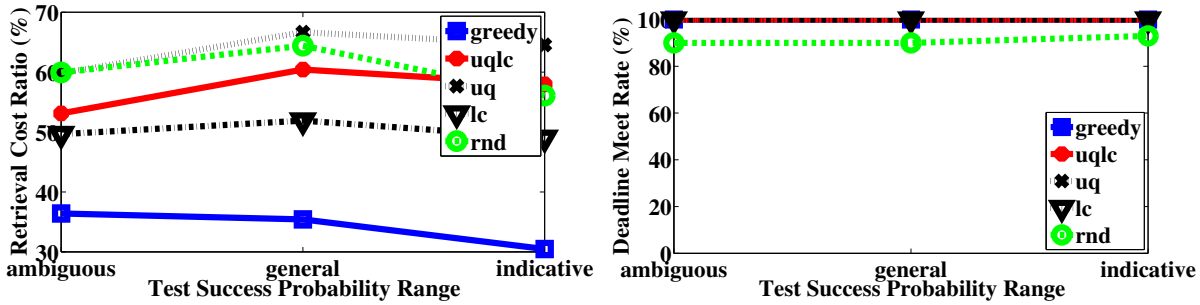
(b) Deadline Meet Rates

Figure 3.6: Varying Request Deadlines

gency of the application requests. For example, a medical team wanting to find ways to get to a collapse site to save lives obviously would issue their route finding request with a tight deadline; a family looking for the nearest under-occupied shelter after losing their home to a hurricane probably doesn't need their request fulfilled with the same level of urgency.

We set a request's deadline to be a multiple of the sum of its sources' individual retrieval latencies. We experiment with multipliers 0.5, 1, 2, 4, and 8. Results are shown in Fig. 3.6. First of all, none of the baseline approaches concern requests' absolute deadline values, which is why in Fig. 3.6a we do not see a clear trend of change among baseline approaches. Looking at Fig. 3.6b, it's quite clear that the multiplier setting of 0.5 and 1 represent quite strict deadlines—even our algorithm results in significant deadline misses. Under a multiplier setting of 2, most approaches start to no longer struggle with avoiding deadline misses. As deadlines become more and more relaxed, we observe a trend of decrease in terms of the retrieval cost ratio of our algorithm. This is because the more relaxed requests' deadlines are, the less likely our algorithm is forced to include sources in each round for parallel data retrieval just to avoid deadline misses for some requests, which might otherwise be unnecessary and suboptimal.

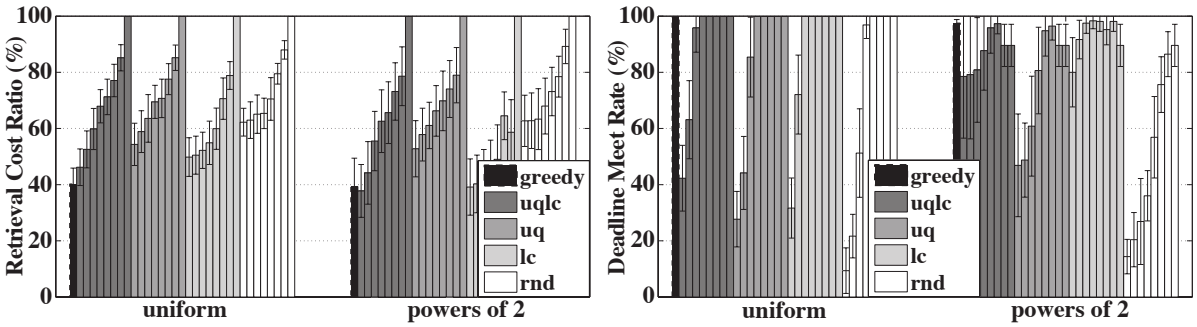
As our algorithm exploits the logic relations among the tests within requests, it would be interesting to see how tests' success probabilities affect the performance of our data retrieval algorithm. These probabilities represent a measure of how good of a prior knowledge we already have regarding the crowdsensing environment. Intuitively, if the probabilities of all the tests evaluating to $true$ (or $false$) are around 50%, our algorithm will make more wrong guesses when it tries to retrieve data from sources in hoping they can help short-circuit other sources.



(a) Retrieval Cost Ratios (b) Deadline Meet Rates

Figure 3.7: Varying Test Success Probability Ranges

On the other hand, if the probabilities are very close to 100% (or 0), indicating that a test is extremely likely to evaluate to *true* (or $false$), then our algorithm will be correctly picking the right sources to retrieve data from for short-circuiting other sources more often. We experiment with uniformly generating success probabilities for all the tests within three different ranges, namely the quite ambiguous $(40\%, 60\%)$, the general $(0, 100\%)$, and the quite indicative (less ambiguous) $(0, 10\%) \cup (90\%, 100\%)$. Results are shown in Fig. 3.7. As baseline methods do not take into consideration these probabilities, we only look at our algorithm. As seen, our algorithm's retrieval cost ratio gradually improves as tests' success probabilities become less ambiguous, coinciding with our intuition.



(a) Retrieval Cost Ratios (b) Deadline Meet Rates

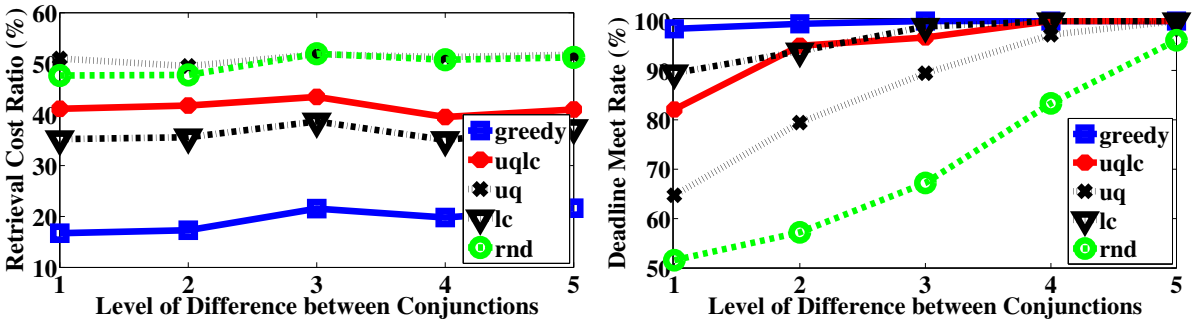
Figure 3.8: Varying Levels of Sources' Retrieval Costs

One other interesting aspect to look at is the level of retrieval cost heterogeneity among different data sources. In other words, in a more homogeneous setting, sensors/sources tend to be of similar types with each other, generating data of closer natures and comparable retrieval bandwidth costs. In a highly heterogeneous setting, however, we can imagine sources being

of drastically different natures. For example, for a network composed of carbon monoxide sensors, cameras, human reporters, and microphones, data generated by different sources could have retrieval costs differing from each other over multiple orders of magnitude. In order to capture this difference, we experiment with two different ways of setting sources' retrieval costs, one being uniformly on $[1, 10]$, and the other 2^p where p is uniformly randomly generated on $[1, 10]$. We include all concurrent retrieval levels for all baselines, and use the bar-with-error representation similar to Fig. 3.2. Results are shown in Fig. 3.8. As seen, when retrieval costs differ vastly from sources, all approaches' retrieval cost ratio standard deviations increase. From looking at Fig. 3.8a alone, it might seem like the single-source-per-round *uqlc* scheme and *lc* scheme slightly outperform our algorithm in terms of retrieval cost ratio, but it can be seen from Fig. 3.8b that these two schemes lead to about 20% deadline miss rates, whereas for our algorithm only about 2.5% requests experience deadline misses. Even though our algorithm still gives the best performance, we do observe that the highly heterogeneous setting causes larger degree of deviations, compared to when sources are less different from each other. We are currently working on improving our algorithm to better handle crowdsensing scenarios where data sources' costs are substantially different from each other.

One last interesting aspect that we look at is the similarity between conjunctions, or how similar an application's different options are. Looking again at the previous route finding example, in an urban area where roads are highly interconnected, all alternative routes might share a lot of common road segments among each other; whereas for a more rural setting, the different possible routes might share few common road segments because roads are quite sparsely connected in the first place.

We set the number of conjunctions per query to 6 and the number of test per query to 6 as well. With 10 concurrent queries, we let, in each request, adjacent conjunctions differ by 1, 2, 3, 4, or 5 tests, and examine the data retrieval algorithm's performance using the different approaches. Results are shown in Fig. 3.9. As seen, our algorithm still outperforms all baseline methods, achieving perfect/near-perfect deadline meet rates, and reducing the network bandwidth cost by half compared to the second best competing method (*lc*). Looking at the trend of changes, we can see similarities to Fig. 3.3. As conjunctions share less common tests with



(a) Retrieval Cost Ratios

(b) Deadline Meet Rates

Figure 3.9: Varying Levels of Conjunction Differences

each other, the number of different tests involved increases. We therefore see from Fig. 3.9b a general trend of rising deadline meet rates for all approaches, similar to what we have observed in Fig. 3.3b. However, rather than decreasing retrieval cost ratios as seen from Fig. 3.3a, we observe either roughly unchanged cost ratios from all baseline approaches, or an ever so slightly increase in cost ratios from our algorithm, as shown in Fig. 3.9a. From the route-finding perspective, as candidate routes become more different from each other, it becomes less likely for the bad condition of a single road segment to rule out multiple routes. As all baseline methods disregard the logic relations within a request, the benefit of having more tests roughly cancels out the damage of not being able to resolve multiple conjunctions with a single source. For our method, which does exploit the logic relations, the damage is likely slightly overshadowing the benefit, hence the higher retrieval cost ratios.

3.3.2 An Application Scenario

Now we adopt a concrete post-disaster application scenario and evaluate our algorithm's performance as compared to baseline methods. In particular, we assume the Urbana-Champaign, Illinois area has just been hit by an earthquake. The first responder team has made an initial damage report and deployed an emergency sensor network for further damage assessments and information passings. Various recovery teams have just arrived nearby and would like to go to different parts of the region to carry out their different recovery work (e.g., restore power-grid, search and rescue from collapsed building sites, etc.). We consider the following particular situations that three teams are facing:

1. The Department of History building suffered from severe damage due to its lack of maintenance, causing quite a few serious injuries. A medical team needs to rush send the injured personnels to McKinley Health Center.

2. The communication team has just arrived from the University Airport. They are currently at the southwest corner of the area, and would like to get to the Department of Computer Science Siebel Center to set up an emergency communication center.

3. The evacuation team has learned that the University High School has not been fully evacuated. They plan to move the remaining kids to the South Quad open area.

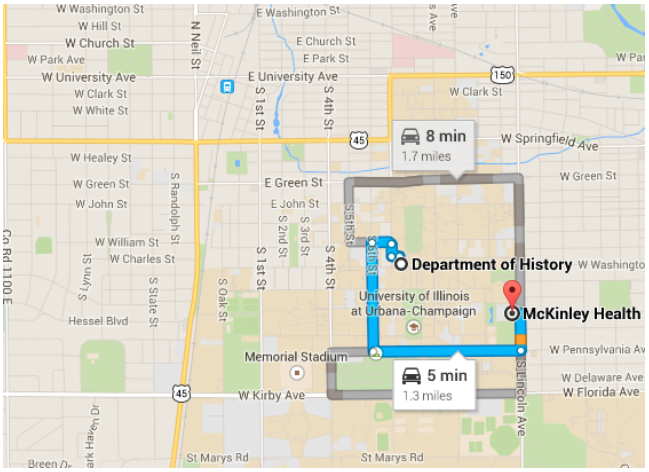
One big problem is that some roads within the region are damaged, preventing recovery teams' vehicles to pass, thus they need to find out fast what paths to take to get to their respective destinations—if there is no viable paths for any particular team, they will have to call in nearby military helicopters for assistance. They can make guesses regarding the damage of roads using the reports generated by the first responder team, but to actually determine if a road is in a reasonable condition for vehicles to pass, a picture needs to be acquired from the camera sensor deployed along the road on the emergency network. The network is of low bandwidth and is shared by various teams and agencies, thus, it is desirable that the recovery teams use as little resource as possible when figuring out their paths.

Using Google map navigation service, each of the three teams has determined their candidate routes according to their respective source-destination pairs, as illustrated in Fig. 3.10. In particular, below are the exact routes.[1] For the medical team, from the Department of History to McKinley Health Center:

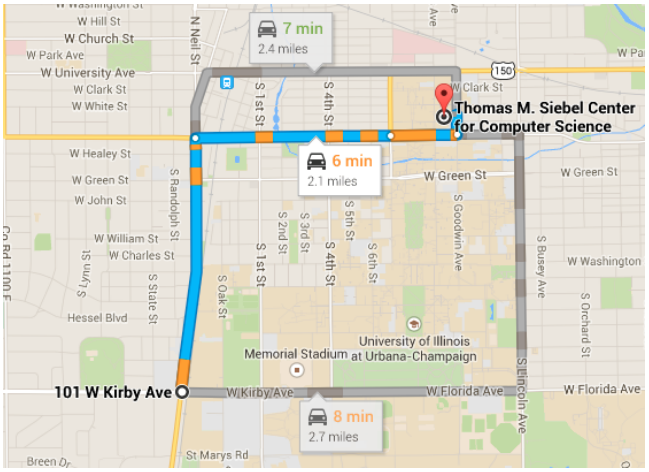
- **6th**→**Penn**→**Lincoln**, or
- *Chalmers*→*5th*→Green→**Lincoln**, or
- **6th**→**Penn**→*4th*→*Florida*→**Lincoln**,

For the communication team, from Kirby to Siebel Center:

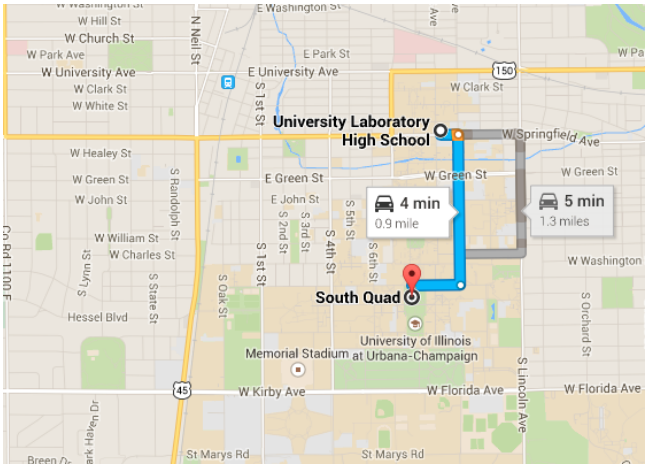
[1]Different font-faces indicate: non-italic—road is good; *italic—road is down*; **bold—retrieved by** *greedy*; underlined—retrieved by *lc*.



(a) Medical Team



(b) Communication Team



(c) Evacuation team

Figure 3.10: Candidate Routes Map Illustrations
(All candidate routes are shown, with the shortest routes highlighted.)

- *Florida*→**Lincoln**→*Springfield*→***Goodwin***, or
- Neil→*Springfield*→***Goodwin***, or
- Neil→*University*→***Goodwin***,

For the evacuation team, from High School to South Quad:

- *Springfield*→**Lincoln**→*Oregon*→***Goodwin***→*Gregory*, or
- *Springfield*→***Goodwin***→*Gregory*.

Every route consists of several road segments, and each road is covered by a camera sensor connected to the emergency network. A request for each team is formed, where different conjunctions represent the different candidate routes, and each test within a single conjunction corresponds to a single road along the path being in OK condition for vehicles to pass. Due to the urgent nature of the post-disaster situation, we set the request deadline level to 1. The emergency network's topology is generated randomly, from which each camera sensor's individual retrieval latency is estimated and fed to the data retrieval engine. All retrievals are then handled by a network simulator running using the topology. Of all the baseline approaches, *lc* (lowest cost source first) achieves the best performance, so we compare our algorithm just to the *lc* approach. For the actual road conditions and exact camera data retrieval details, please refer to Footnote 1. The retrieval cost ratio and deadline meet rate results are shown in Tab. 3.2. The information of which approaches choose which roads' conditions to retrieve data for is also illustrated in Fig. 3.11. As seen, though no deadline misses are observed from either strategy, our data retrieval algorithm is able to finish all route-finding tasks using less than half of the number of road-side cameras that the competing *lc* approach uses, consuming only half the retrieval bandwidth cost.

	greedy	*lc*
Number of Road Conditions Retrieved	4	9
Retrieval Cost Ratio	28.95%	56.58%
Deadline Meet Rate	100%	100%

Table 3.2: Multiple-Route-Finding Application Result

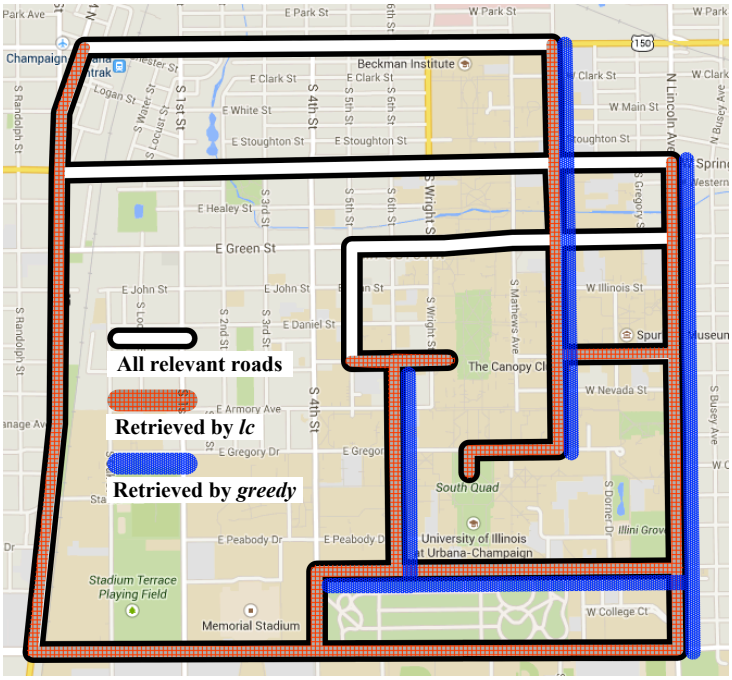


Figure 3.11: Multiple-Route-Finding: All candidate routes of all teams are highlighted, different road segments whose conditions are retrieved by either approach (*greedy* and *lc*) are marked accordingly.

Chapter 4

Data Acquisition under Freshness Constraints

4.1 System Overview

We aim to design a sensing system specifically tailored for resource constrained dynamic environments, such as disaster response and recovery. Under resource limitations (e.g., network bandwidth, node battery power, etc), data stays at sources, not to needlessly use the network. Only upon explicit requests do sensors take/share measurements of the environments and transmit back the sensory data or results. Under such settings, retrieval of sensor data from sources needs to be carried out with care such that the minimum amount of resources is consumed.

Adding to the complexity of the problem, the environment that is of interest is dynamic at different timescales. For example, in a politically unstable region, every now and then roads are taken up by protesters or rioters, preventing safe passage. As the crowds move around, the traffic blockage situation evolves with time; road information obtained that is old may no longer reflect the actual states of the environment. As another example, an earthquake and its aftershocks can affect a region differently under different situations: If a segment of a road gets blocked because it is flooded due to damage to its sewage system, then perhaps in a day or two it will get repaired and return to functional status. If a bridge over a river collapses during the earthquake, then it might take months before it can be repaired or an new one constructed. Therefore, data retrieved from remote sources may have very different *freshness* characteristics. In carrying out sensing tasks to help with decision making, we need to take into consideration such data freshness characteristics in order to avoid reaching invalid or inconsistent decisions made from partially stale sensor data.

The goal is then to fetch data from sources in a way that minimizes system resource consumption while reaching answers using data items all within their freshness intervals. In general, a

decision-maker's information need is formed into a query, which is then translated into requirements for a set of relevant data objects. A retrieved object can be subjected to a test that evaluates a condition on the object. For example, an audio clip from an road acoustic sensor might be used to determine if there is currently a convoy passing by; an image of the inside of an emergency shelter can be inspected to determine if the occupancy is reaching its limit. The goal of the decision-maker is to find the best course of action. Each possible course of action requires multiple conditions to be evaluated to determine if this course of action is valid. It is our objective to compute a *good* data retrieval plan that makes efficient use of network resources to help choose the right course of action from sufficiently fresh data.

We use network bandwidth consumption (i.e., the size in bytes of the retrieved data) as the cost measure for supporting the decision-making task. We do, however, want to point out that the algorithms developed in this work can be used on any cost metric definition as long as it is additive. For example, in a scenario where energy is the critical resource, we can easily redefine the cost metric to optimize energy consumption.

In our design, requests from decision makers upon arrival are converted and encoded to boolean logic expressions, where the alternative courses of action are represented by a disjunction (OR) of terms, whereas the conditions that need to be satisfied for a course of action to be valid are represented by a conjunction of variables (AND). A directory, called semantic store is assumed to exist that knows which data source has what information. Given the menu of available information objects at different sources, a source selection module [6] is used to select a suitable source to contact for each needed object. A data retrieval planner then computes an optimal retrieval order. The latter module is the focus of this work.

4.2 Problem Description

For ease of discussion, we introduce several key notations first. In making a decision, let $\{a_i\}$ be the set of alternative courses of action decided on, and $\{t_{i_j}\}$ the set of conditions for a particular a_i to be valid. Each condition t_{i_j} has cost c_{i_j} (e.g. data retrieval bandwidth cost), latency l_{i_j} (e.g. data retrieval delay), probability of being satisfied p_{i_j}, and freshness interval d_{i_j}, after

which it needs to be reevaluated.

As previously discussed, for a decision maker a decision is made by choosing one among multiple courses of action, each of which consists of multiple conditions that must simultaneously hold. To evaluate a condition, some data must be retrieved over the resource-poor network, and in a timely fashion due to environment dynamics, as different conditions have corresponding freshness intervals. At the time the decision is made all underlying data must be fresh, which means if the retrieval order was $O = \langle t_{i_1} t_{i_2} \dots t_{i_n} \rangle$ then the freshness interval d_{i_j} of any of the retrieved data item t_{i_j} is greater than the sum of the retrieval latencies of the data item itself and all subsequent ones in the retrieval order:

$$d_{i_j} > \sum_{s=j}^{n} l_{i_s}.$$

Given this setting our goal is then to minimize expected decision cost.

Sequential processing is generally more cost efficient than its parallel counterpart, as it can minimize the probability of unnecessary data fetches caused by parallel retrievals, which can otherwise be avoided by evaluation short-circuiting in sequential processing. Therefore, towards the goal of minimizing the total cost, we want to plan the evaluation of conditions in a sequential fashion whenever plausible. However, this might not always be possible, for example when data items associated with an request all have rather short freshness intervals and sequential processing would always lead to old data items expiring before the whole request can be resolved. In situations as such, we would need to make our solution capable of adapting parallel processing s.t. the request *can* at least be resolved in time, because otherwise talking about cost optimization is of little value.

Furthermore, for the post-disaster scenarios we consider, oftentimes *a* valid course of action does exist for the decision making task—for example, when looking for a route for a medical team to go from location A to B after an earthquake, many or even most of the candidate routes might be blocked, but some *will* be in reasonable condition—the focal point here is that we want to find *a* positive resolution as soon as possible with minimal resource consumption. Having said that, we make the following observations. i) Making a choice among multiple courses of action

is accomplished by finding *a* single valid one, and ii) Once we've started verifying the validity of a particular course of action, there is no reason to turn to another one before we are finished with the current one (i.e., verified that either all its conditions hold or at least one fails) under the goal of minimizing cost.

The above discussions serve as a guideline of how we go about solving our problem at hand. In order to devise an actual solution, we need to answer the following questions,

1. Among all courses of action, which one should we examine first,
2. For verifying the validity of a particular course of action, how should we plan the evaluation of its conditions, and
3. If sequential processing does not suffice, how should we employ parallel processing to try to avoid causing freshness interval violations.

Before diving into the detailed algorithm that answers the above questions, we give a rough solution sketch here: We first pick the course of action that has the highest valid probability per unit (expected) cost. We then schedule its conditions in an EDF-inspired sequential retrieval order. If verifying them all in that order does not lead to freshness constraint violation, we rearrange the order to try to decrease the expected cost; otherwise, we increase the level of parallel retrievals in order to try to avoid the freshness deadline violation. If this course of action is valid, we are done; otherwise we move onto next best course of action in the request.

4.3 Algorithms

As our ultimate goal is to minimize the excepted cost of carrying out a sensing task, we first need to rank all the alternatives (courses of action) according to their cost effectivenesses, computed as the validity probability per unit cost. This should feel natural as, for example, the more probable a candidate route is in good condition and the lower the cost is for verifying its condition, the sooner we should examine it for a route finding task.

Revisiting some similar concepts from our previous discussion, for a particular action a_i, the most cost effective processing order is then computed by always picking first the condition with

the highest probability of short-circuiting its siblings per unit cost. If we use $t_{i_u} \succ_t t_{i_v}$ to denote that condition t_{i_u} should precede t_{i_v} in the optimal order, we have

$$t_{i_u} \succ_t t_{i_v} \Leftrightarrow \frac{1 - p_{i_u}}{c_{i_u}} > \frac{1 - p_{i_v}}{c_{i_v}}.$$

With the order $\langle t_{i_{o0}}, t_{i_{o1}}, t_{i_{o2}}, \ldots \rangle$, we can compute a_i's expected cost as

$$c_i = c_{i_{o0}} + p_{i_{o0}}(c_{i_{o1}} + p_{i_{o1}}(c_{i_{o2}} + p_{i_{o2}}(\ldots))),$$

(as also mentioned previously) and its validity probability as

$$p_i = \prod_j p_{i_j}.$$

Therefore, the order of the courses of action is constructed as

$$a_i \succ_a a_j \Leftrightarrow \frac{p_i}{c_i} > \frac{p_j}{c_j},$$

similar to the order of conditions, with the only difference of the short-circuiting probability being the validity probability, as opposed to the failure probability.

Having established the order $\langle a_{o0}, a_{o1}, a_{o2}, \ldots \rangle$, we can then start processing the request by following that order. For a particular action a_i, we want to find out its validity status with the least cost, but at the same time without violating any of its component conditions' freshness constraints. Thus we proceed as follows. Inspired by EDF, we first order the a_i's conditions according to their freshness intervals, latest first

$$t_{i_u} \succ_d t_{i_v} \Leftrightarrow d_{i_u} > d_{i_v},$$

which we call a_i's *Latest Deadline First* (*LDF*) order.

Theorem 3. *If LDF order cannot avoid data freshness violation, no sequential order can.*

Proof. We use proof by contradiction. For retrieving all conditions for some action a_i, let's

suppose its LDF order

$$O = \langle t_{i_1} t_{i_2} \dots t_{i_n} \rangle$$

causes data freshness constraint violation(s), i.e., for some condition t_{i_f}, its freshness interval is shorter than the sum of the retrieval delays of itself and of all subsequent ones in the LDF order,

$$d_{i_f} < \sum_{k=f}^{n} l_{i_k}. \tag{4.1}$$

We assume, for contradiction, that there exists a different (from LDF) sequential order O' that causes no data freshness constraint violations. So in O', t_{i_f} must *not* still be the front of all elements of the set $S_f = \{t_{i_k} | f \leq k \leq n\}$, which means some other $t_{i_s} \in S_f$ must be the new front of S_f in order O'. Since t_{i_s} is behind t_{i_f} in O, so we have

$$d_{i_s} < d_{i_f}. \tag{4.2}$$

Chaining the Inequalities (4.1) and (4.2) together, we have

$$d_{i_s} < \sum_{k=f}^{n} l_{i_k},$$

which means that test t_{i_s} will miss its freshness deadline in O'. Contradiction reached. □

Now, assuming we were to retrieve *all* tests' data items of this action by following the LDF order, we can check to see if any freshness constraint violation would've occurred by the end of the retrievals. If not, it means this course of action *can* be checked by sequential processing, and we might be able to further decrease its expected cost; otherwise, it *cannot* be checked by sequentially retrieving its data items, and we will need to add parallel processing in order to decrease the total retrieval latency and make the action resolvable without violating any data item's freshness constraints. We next discuss each of the two cases in detail.

Algorithm 7 When a_i's LDF order already satisfies freshness constraints, rearrange the LDF order to minimize the resolution cost

Input: Action a_i's conditions $\{t_{i_j}\}$, the corresponding costs $\{c_{i_j}\}$, retrieval latencies $\{l_{i_j}\}$, probabilities of being true $\{p_{i_j}\}$, and freshness intervals $\{d_{i_j}\}$

Output: The retrieval order

$Q_c \leftarrow \emptyset$, $Q_d \leftarrow$ LDF order
$L \leftarrow \{t_{i_j}\}$ sorted in descending order of $\frac{1-p_{i_j}}{c_{i_j}}$
while $Q_d \neq \emptyset$ **do**
    for t_l in L **do**
        $Q_H \leftarrow Q_c + \langle t_l \rangle + Q_d \setminus \langle t_l \rangle$
        if Q_H meets freshness constraints **then**
            $Q_d \leftarrow Q_d \setminus \langle t_{i_j} \rangle$
            $Q_c \leftarrow Q_c + \langle t_{i_j} \rangle$
            break
        end if
    end for
end while
return Q_c

4.3.1 Cost Minimization

If a_i's LDF retrieval order does not violate any test t_{i_j}'s freshness constraint, we can possibly rearrange it in order to decrease the expected resolution cost. Note that if a_i is valid (i.e. all its conditions are true), we wouldn't be able to decrease the cost, as all data items do need to be retrieved for verifications. However, if it turns out that a_i actually is invalid (i.e., one or more of its conditions are false), then we want to detect this failure with as little cost as possible. Therefore, the rearrangement proceeds as follows: We initialize $Q_d =$ LDF order and $Q_c = \emptyset$. Then we compute the order L for all of a_i's tests according to their per unit cost failure probabilities $\frac{1-p_{i_j}}{c_{i_j}}$ (in descending order). We then pick L's first condition $t_{i_j} \notin Q_c$ and check if the hypothetical order

$$Q_c + \langle t_{i_j} \rangle + Q_d \setminus \langle t_{i_j} \rangle$$

would violate any freshness constraint (where + denotes concatenation). If not, we remove t_{i_j} from the LDF order Q_d

$$Q_d = Q_d \setminus \langle t_{i_j} \rangle,$$

and append it to the end of Q_c

$$Q_c = Q_c + \langle t_{i_j} \rangle.$$

We terminate when $Q_d == \emptyset$. This whole process essentially tries to make failures appear as soon as possible (in the sense that least amount of cost has been incurred) if there *is* a failure, without violating any freshness deadline. The algorithm pseudo-code is shown in Alg. 7.

The computational complexity is dominated by the nested loops and the check for freshness constraint violations for each test, of order $O(n^3)$ where $n = |a_i|$.

4.3.2 Freshness Constraint Violation Avoidance

Algorithm 8 When an action's LDF order does not meet freshness constraint, transform the LDF order to add parallel retrievals in order to eliminate constraint violations

Input: Action a_i's conditions $\{t_{i_j}\}$, the corresponding costs $\{c_{i_j}\}$, retrieval latencies $\{l_{i_j}\}$, probabilities of being true $\{p_{i_j}\}$, and freshness intervals $\{d_{i_j}\}$
Output: The retrieval order

```
1: Q_d ← LDF order, S_p ← ∅
2: while |Q_d| > 0 do
3:     t_e ← end element of Q_d
4:     Q_d ← Q_d \ ⟨t_e⟩
5:     S_p ← S_p ∪ {t_e}
6:     if Q_d + S_p meets freshness constraints then
7:         return Q_d + S_p
8:     end if
9: end while
10: return NULL
```

On the other hand, if a_i's LDF retrieval order does violate some condition t_{i_j}'s freshness constraint, then we want to add parallelism to the sequential order to try to eliminate the violation. Keeping the parallelism at the end is beneficial in terms of preventing cost from unnecessary increase as any failed condition before the end will still be able to short-circuit the paralleled one. It is quite straightforward to carry out the transformation: We prepare queue Q_d = LDF order and set $S_p = \emptyset$. We take Q_d's end condition t_{i_j} and check if the order

$$Q_d \setminus \langle t_{i_j} \rangle + S_p \cup \{t_{i_j}\}$$

would violate any freshness deadline, where all data items in the set are to be retrieved in parallel. If yes, we remove t_{i_j} from Q_d

$$Q_d = Q_d \setminus \langle t_{i_j} \rangle,$$

and add it to S_p

$$S_p = S_p \cup \{t_{i_j}\},$$

and move on to the new end item in Q_d and continue the process; otherwise, we terminate the transformation as we have successfully eliminated the freshness constraint violation. The algorithm pseudo-code is shown in Alg. 8.

The computational complexity is dominated by the for-loop and the check for freshness constraint violations, of the order $O(n^2)$ where $n = |a_i|$.

Finally, in designing a solution that handles both the cost minimization and deadline violation avoidance, we combine the above two techniques into an unified algorithm. The basic idea is as follows: First we carry out cost-saving rearrangement to the LDF order as much as possible, given that i) if there was no freshness constraint violations before, we do not introduce one now, and ii) if there were, we do not worsen the violation degree (e.g., if there was a deadline miss by 3 minutes, we cannot carry out order rearrangement that increases the miss to 4 minutes). After the rearrangement, we carry out parallel transformation at the end of the retrieval order just as previously described. We collect all above discussions and present the complete sensing algorithm in Alg. 9. We call our algorithm *Variational Latest Deadline First* (vLDF).

Given our separate complexity analyses for Alg. 7 and 8, it is easy to see that our unified algorithm Alg. 9 runs in $O(mn^3)$ time, where m is the number of alternative courses of action in the request, and n the number of conditions in an action.

4.4 Implementation

In order to test our system in a realistic setting, we implement a route planning system that targets post-disaster scenarios, for disaster response teams. In our normal everyday life, we can

Algorithm 9 vLDF — Resolution algorithm for answering a sensing request

Input: The courses of action $\{a_i\}$, and each a_i's conditions $\{t_{i_j}\}$, the corresponding costs $\{c_{i_j}\}$, retrieval latencies $\{l_{i_j}\}$, and success probabilities $\{p_{i_j}\}$, and freshness interval $\{d_{i_j}\}$
Output: Request resolution result

$L_a \leftarrow \{a_i\}$ sorted in descending order of $\frac{p_i}{c_i}$, $\#_{fail} \leftarrow 0$
for a_i in L_a **do**
    $Q_c \leftarrow \emptyset$, $Q_d \leftarrow$ LDF order, $S_p \leftarrow \emptyset$
    $L \leftarrow \{t_{i_j}\}$ sorted in descending order of $\frac{1-p_{i_j}}{c_{i_j}}$
    while $Q_d \neq \emptyset$ **do**
        for t_l in L **do**
            $T_d \leftarrow Q_c + Q_d$'s degree of freshness violations
            $Q_H \leftarrow Q_c + \langle t_l \rangle + Q_d \setminus \langle t_l \rangle$
            $T_H \leftarrow Q_H$'s degree of freshness violations
            if $T_H \leq T_d$ **then**
                $Q_d \leftarrow Q_d \setminus \langle t_{i_j} \rangle$, $Q_c \leftarrow Q_c + \langle t_{i_j} \rangle$
                break
            end if
        end for
    end while
    while $|Q_c| > 0$ **do**
        $t_e \leftarrow$ end element of Q_c
        $Q_c \leftarrow Q_c \setminus \langle t_e \rangle$, $S_p \leftarrow S_p \cup \{t_e\}$
        if $Q_c + S_p$ meets freshness deadlines **then**
            Proceed with resolving a_i by following $Q_c + S_p$
            if a_i succeeds **then**
                return a_i as an successful result
            else
                $\#_{fail} \leftarrow \#_{fail} + 1$
                break
            end if
        end if
    end while
end for
if $\#_{fail} == |L_a|$ **then**
    return request resolves to failure
else
    signal freshness deadline violation unavoidable
end if

easily use Google Maps to compute routes by specifying source and destination locations. But after a natural disaster, routes returned by Google Maps or any other traditional route planning service might not suffice as roads might be blocked or damaged, and the entire environment is dynamic, with various aspects changing over time (e.g., bridges might collapse, roads might

get flooded or blocked, people might setup temporary camps and move around). Therefore, verifications (e.g. visually via pictures taken of the roads) are needed to make sure if a route is in reasonable condition for vehicles to pass, and they need to be carried out in a timely fashion s.t. results do not become stale as the environment changes. The emergency networks set up by first responder teams are likely of very low bandwidth, and are shared by multiple response teams (e.g. infrastructure, medical, etc), thus it is key no single request exhausts the network resource.

Our post-disaster route planing system consists of the following components,

- Router: We use open-source router code Gosmore [21] to compute routes on Open Street Map [46] data. We modify the Gosmore code so that it computes, for specified source and destination locations, any number of candidate routes the user desires.

- Meta Store: A database that hosts the meta data (location, time, size, etc) of pictures taken by remote camera phones. Note that due to goal of minimizing network resource consumption, photos themselves stay on the phones, and only meta data are sent to the central semantic store. For fast indexing and searching, we use ElasticSearch engine [17], one of the most popular enterprise search engines, as the back-end.

- Camera Pipeline: The subsystem that takes care of picture taking and meta data extraction on the remote phones, meta data transmission to the meta store, and sending the actual images upon explicit requests. We use the Medusa/MediaScope systems [33, 49] for this component.

- Source Selector: A road segment can potentially have multiple pictures that can be retrieved for checking its condition. In order to minimize the transmission cost, we perform source selection computation [6] to reduce the set of relevant pictures.

- Request Resolver: The component that uses our algorithms as the underlying engine to plan for optimal request resolution strategies.

- User Interface: A web interface where a user can specify the source and destination locations (by either clicking on the map, or using natural language, which would then go

through an NLP processor for location name extraction, and Google geocoding [20] for conversion to latitude-longitude coordinates), provide human judgments to retrieved images (by specifying if they reflect road segments being in good or bad conditions), and see the final routing result. AJAX is used to pass information between the client web front-end and the server back-end algorithms for iterative interactions and updates.

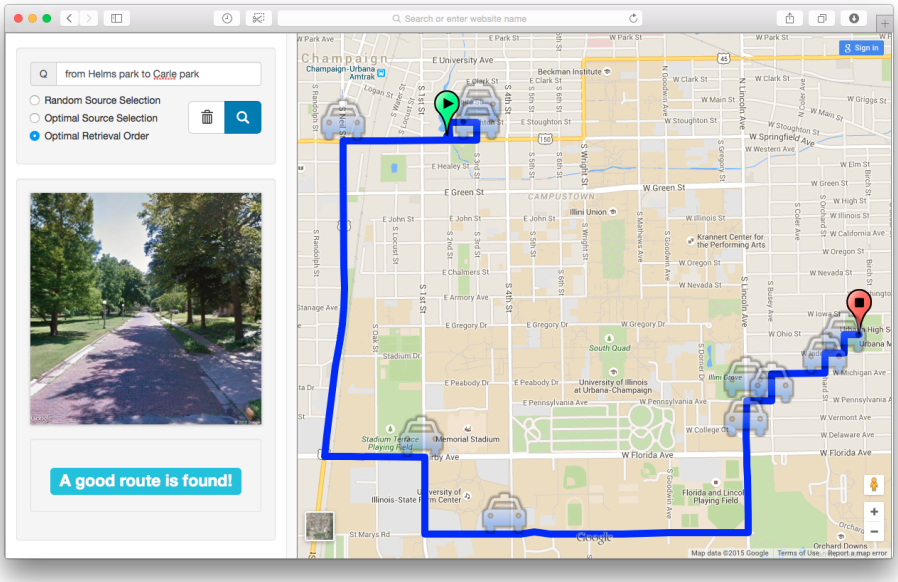


Figure 4.1: Screenshot of the Route Planning System's Web Front-end

Upon receiving the source-destination input, the system computes multiple (>10) candidate routes and extracts all the unique road segments of all the routes. For each road segment, a geo-polygon is computed and then used to query the semantic store for matched meta data. The source selection module then stripes the set of all relevant meta data down to a minimal-cost set that still covers all the relevant road segments. With this set of meta data, our algorithm computes the optimal retrieval order, and iteratively carries out image retrieval and order update according to user input to resolve the routing request. A screenshot of the system finding a good route for a user specified source-destination location pair is shown in Fig. 4.1. The car icons indicate the locations where images have been retrieved and approved by the user; the result route is highlighted by the blue polyline on the map.

4.5 Evaluation

We take two complementing approaches in evaluating our proposed solutions. On one hand we design and carry out extensive simulations that explore how various problem and system aspects affect the behavior of our proposed algorithm. On the other hand we specify concrete application scenarios and demonstrate the effectiveness of our algorithm and system through actual resolutions of route finding requests.

We compare our algorithm (*vLDF*) to the following three baseline methods in the evaluation.

- *Lowest Cost Source First* (*LCF*) — Data retrievals follow the request's data items' costs, in ascending order.

- *Short-circuiting Benefit* (*SCB*) — Data retrieval order is computed according to the short-circuiting benefit (i.e., short-circuit probability per unit cost), as described by Casanova et al. [9].

- *Probability based Prediction* (*PbP*) — We assign to each test a predicted value based on its success probability (i.e., $v(t_{i_j}) = \mathbb{1}_{p_{i_j} \geq 0.5}$). Then each action a_i's value is computed according to its conditions' predicted values: $v(a_i) = \prod_j v(t_{i_j})$. a_i's cost is then the sum of its conditions' costs if $v(a_i) == 1$, and the smallest cost of its 0-valued conditions otherwise. The actual retrieval order is as follows, all 1-valued actions are ordered before 0-valued ones; actions of the same predicted value are ordered according to their predicted costs (ascending); for a 1-valued action, its conditions are ordered according to their costs (ascending); for a 0-valued action, all its 0-valued conditions are placed before 1-valued ones, and conditions of the same value are ordered according to their costs (ascending). After the retrieval of each data item, the request and the retrieval order is updated according to the actual value newly fetched.

As none of the above baseline methods take into consideration data freshness, so upon encountering data expiry, they simply refetch the data if their respective updated retrieval orders dictate so. We set each request to timeout when its lapse has exceeded the sum of all its data items' freshness deadlines, as a way to prevent infinite expiry-refetch loops.

4.5.1 Algorithm Behaviors

We first introduce our simulation settings, and then go through each of the set of experiments.

We experiment with the scenario involving data items of 2 different freshness deadline levels, tight and relaxed, and set a ratio parameter (tights' percentage, from 40% to 100%, default at 70%) to vary the mixture of the 2 types of data items. We experiment with different numbers of alternative courses of action per request (from 4 to 10, default at 8), and different numbers of conditions per action (from 4 to 10, default at 6), indicating the complexity of each alternative for the application requests. All data items' sizes range from 2 MB to 5 MB (default at 3450 KB)[1]. Due to our target post-disaster setting, we experiment with limited network bandwidths from 3.5 KBps to 6.5 KBps (default at 5 KBps), simulating a slow emergency network set up by first responder teams[2]. Conditions' average success probabilities range from 45% to 95% (default at 75%).

Regarding general network topologies, rather than committing to a particular structure, we use a single parameter α to indicate the network's common bottleneck ratios, which is defined as follows: When performing concurrent retrieval of multiple data items, the α portion of each item's transmission time will be spent in a bottleneck, where different data items would queue up and transmit in sequential order; the $1 - \alpha$ portions, on the other hand, proceed in parallel, thus the total delay incurred by those portions are just the maximum among all data items being concurrently fetched. We experiment with α ranging from 0 to 100% (default to 50%). In terms of each data item's individual transmission delay, because of the fluctuations caused by various internal and external factors, predicted transmission delay will never be perfectly accurate. Thus for simulating the actual transmission delay in a noisy network, we add a Gaussian noise to the predicted value, with the mean ranging between ± 3 minutes (default at 0-mean) and standard deviation 0 to 6 minutes (default at 1).

We carry out our simulation experiments by tuning one parameter at a time and fixing all the rest to their default values. We use two metrics for performance measures:

- *Request Resolution Ratio* — The percentage of the number of resolved requests over the

[1]These sizes roughly correspond to image sizes of today's mobile phones.

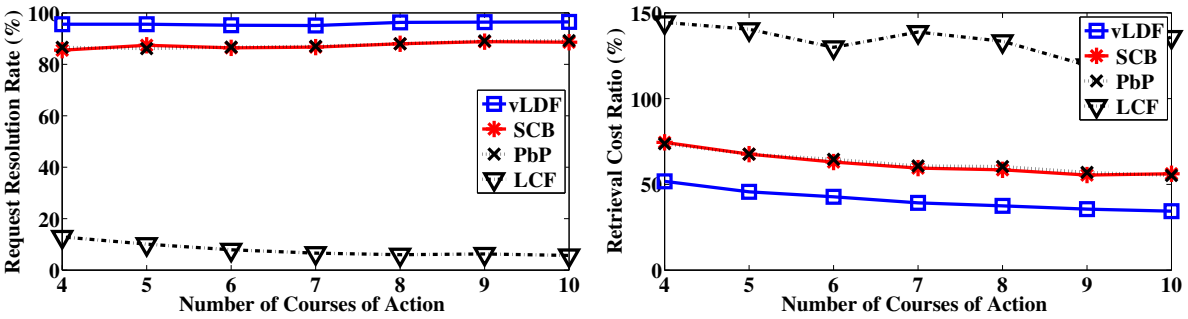
[2]These bandwidths roughly correspond to that of military ad-hoc communication networks under similar settings.

total number of requests attempted (1000 randomly generated for each parameter setting). The higher the resolution ratio is, the better.

- *Retrieval Cost Ratio* — The percentage of the total cost of all retrieved data items over the total cost of all relevant data items, averaged over all resolved requests from the 1000 runs. The lower the cost ratio is, the better.

We first look at how the number of courses of action within a request affects data retrieval algorithms' behaviors. This in practice corresponds to the number of different alternatives an application request can be satisfied. Take the post-disaster route planning scenario as an example, we can easily draw parallels between actions $\Leftrightarrow$ routes, and conditions $\Leftrightarrow$ road segments. We experiment with 4 to 10 actions per request, and show results in Fig. 4.2. As seen in Fig. 4.2a our vLDF scheme achieves the highest request resolution ratios compared to the baseline methods. The SCB (short-circuiting benefit) and PbP (probability based prediction) show very similar performance (which is the case through all simulation experiments), while the simplest LCF (least cost first) scheme achieves the worst performance (which also is the case through all simulation experiments). This general performance comparison is reasonable because our vLDF scheme considers success probability, cost, and freshness deadlines; SCB and PbP both fail to take into consideration the freshness deadlines; and LCF only looks at the costs. We see that LCF's retrieval cost ratios are normally beyond 100%; this is because it needs to deal with data items' expiries and carry out refetches, thus greatly increasing the network resource consumption. We also observe the trend of more courses of action lead to lower retrieval cost ratios, as more actions can get short-circuited by a succeeding one.

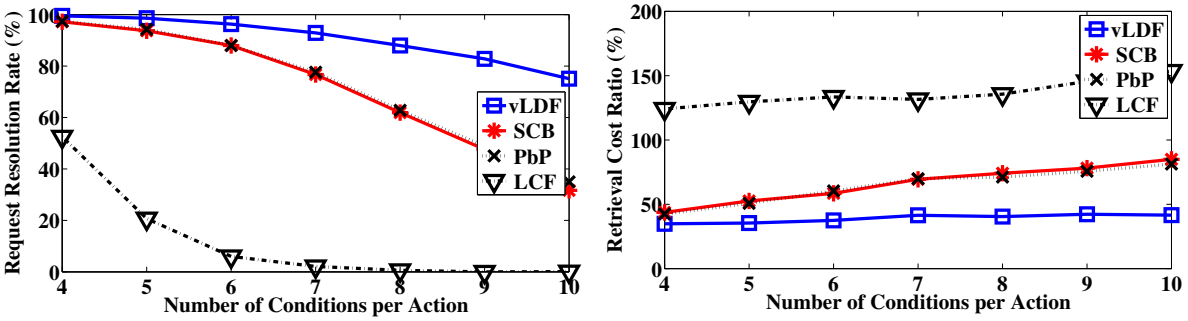
We next look at how the number of conditions per action (which can be a measure of the complexity of each request's resolution alternative is) affects performance of the various schemes. Results are shown in Fig. 4.3. As seen, having more conditions to check for each resolution alternative makes it harder for all schemes to resolves requests, evidenced by the declining resolution rates. But our vLDF scheme shows the slowest dipping trend. On the other hand, for the requests vLDF *is* able to resolve, the retrieval cost ratio is not affected much, compared the the other methods who clearly need to consume more network bandwidth for



(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.2: Number of Courses of Action per Request

their query resolutions.

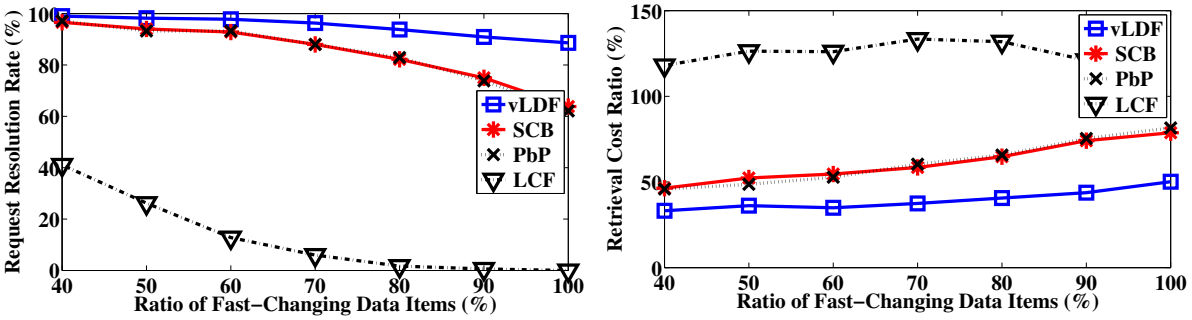


(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.3: Number of Conditions per Action

As previously discussed, we have conditions generally belonging to two different categories, namely slow changing and fast changing. Therefore, we also experiment with how the mixture ratio of the two categories affects the behaviors of all the approaches. The results are shown in Fig. 4.4. As seen, as the proportion of fast-changing data items increases, all schemes gradually show degraded performance, as it is generally harder to resolve requests in time without data freshness violation when we have to deal with more objects that intrinsically changes states fast.

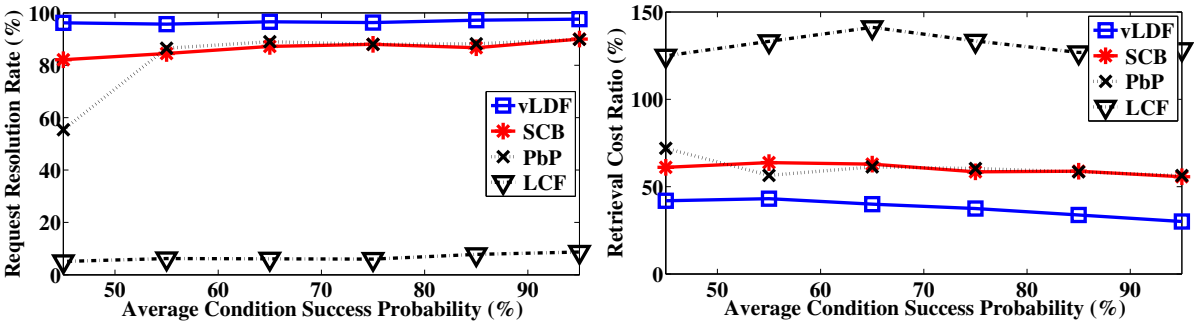
Since our vLDF algorithm, as well as the PbP and SCB baselines, exploits conditions' logic relations, it would be interesting to see how the performance of each of the scheme is affected by different success probability settings. For sensing tasks, we can think of these probabilities as representing a measure of how good of a prior knowledge we already have of the target environment: good and clear knowledge (e.g., probabilities close to 100% certainty) would lead



(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.4: Fast-changing Data Proportion

to algorithms more often making the correct guesses and picking the more optimal data items to retrieve to short-circuit other data items; on the other hand if the prior knowledge is ambiguous (e.g., probabilities close to 50%), algorithms will make more wrong guesses. Our simulation results are shown in Fig. 4.5. The trend of change is rather subtle, but is still visible: generally the closer all conditions' success probabilities are to certainty (i.e., 100%), the higher percentage of requests that can be resolved. This is because as the success probabilities become more certain, all schemes (except for LCF) make the right guesses more frequently, and thus resolve the requests more efficiently. For the case of LCF, the slight increasing resolution ratio comes from the fact that higher certainty (for success) leads to sooner valid course of action being found. Given the above discussion on request resolution ratio, the retrieval cost ratio results shown in Fig. 4.5b should be clear as well.

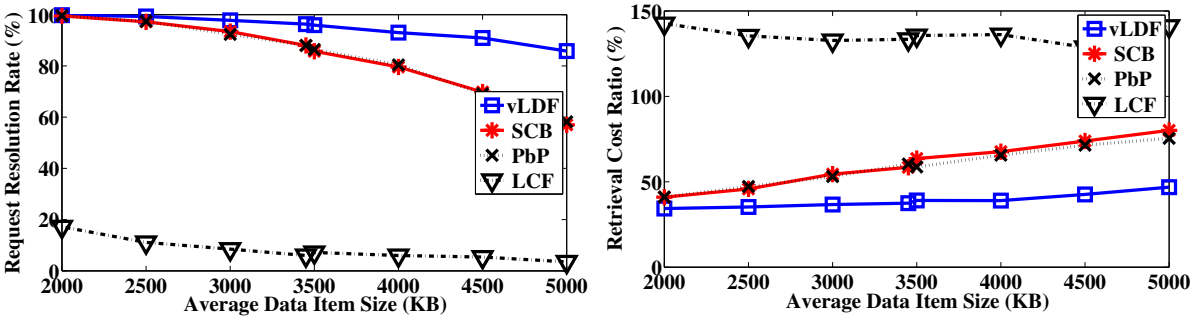


(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.5: Probability of Conditions Being True

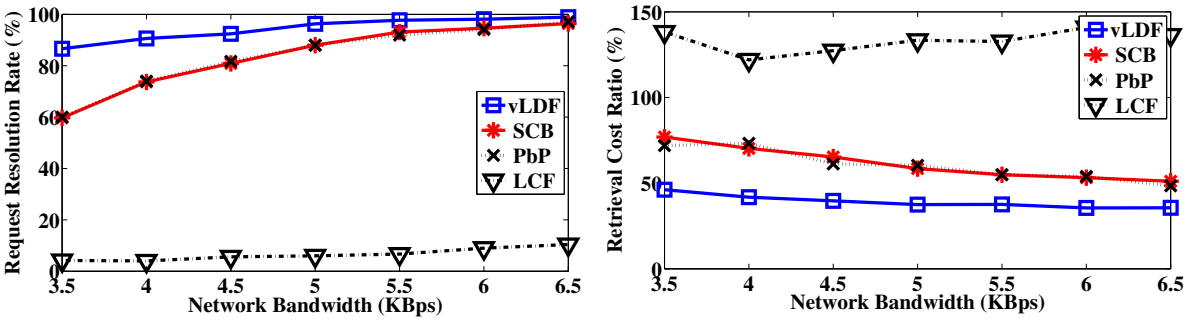
Next up, we experiment with varying requests' data item sizes as well as the network band-

width, and examine how the various schemes' performance changes. Results are shown in Fig. 4.6 and 4.7. As can be seen, the two sets of Figures appear to be mirror images of each other. This makes sense as both of them directly influence the retrieval delays during request resolutions; and the retrieval delays directly affect the various schemes' behaviors. Increasing data item sizes therefore has a similar effect as decreasing network bandwidth. Let's focus our attention on the network bandwidth experiment result as shown in Fig. 4.7: Higher transmission speed lead to more requests being resolved and at the same time less network resource consumed. With more resource available on the network, requests tend to get resolved faster with less freshness violations. Thus the retrieval cost ratios generally decreases for all schemes, except for the LCF approach, which shows fluctuating (but still all poor) performance.



(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.6: Average Data Object Size

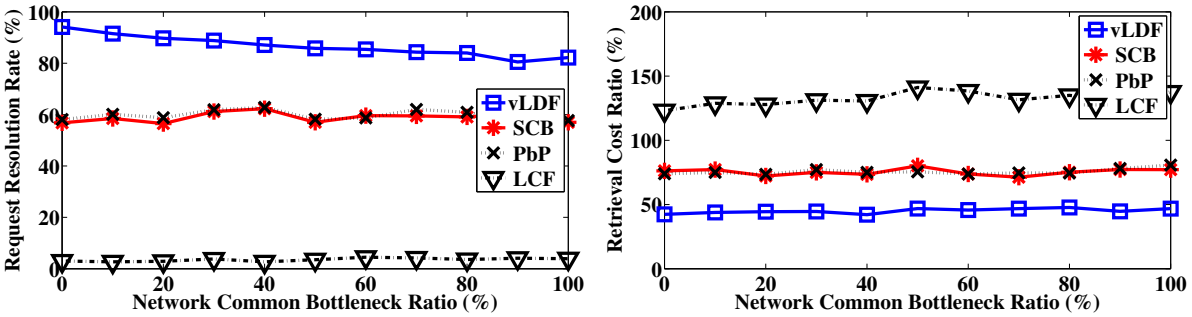


(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.7: Network Bandwidth

For the general network topology, we use the parameter α to indicate the level of shared bottleneck in the network, as also previously discussed when we introduce experiment settings.

Basically, a higher α value indicates an more severe bottleneck shared by all nodes within the network; parallel data retrievals from nodes will be queued up to become sequential processing more severely. Results are shown in Fig. 4.8. Since all three baseline methods have no way of incorporating parallel retrievals, changing α value has no obvious effect on them. For our scheme vLDF on the other hand, a trend of decreasing proportion of requests being resolved can be observed as the level of shared network bottleneck increases. But even then, we still see a clear advantage margin of our algorithm over any other baseline methods. We can also see a slight hint of increase in the network resource consumption of our algorithm from Fig. 4.8b.



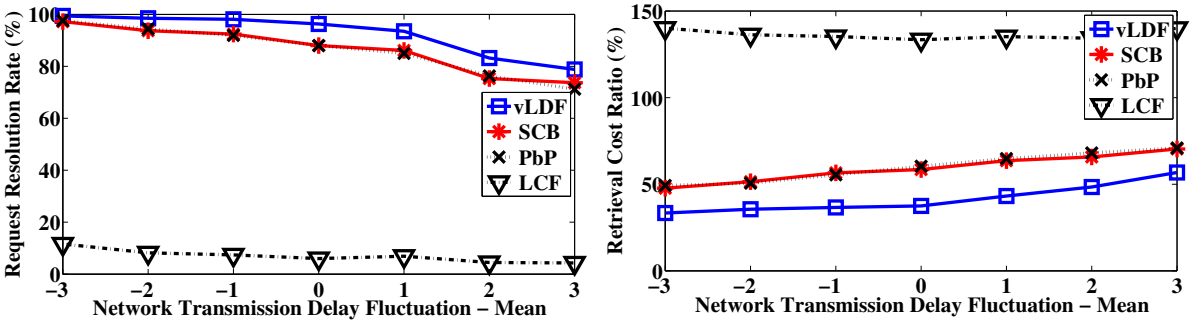
(a) Request Resolution Ratio

(b) Retrieval Cost Ratio

Figure 4.8: Network Common Bottleneck Ratio

Lastly, we look at how network fluctuations (as represented by Gaussian noise added onto estimated transmission delays) affect the performance of the various algorithms. This set of experiments are interesting because network transmission delays need to be estimated through the running of our vLDF algorithm: If the actual transmission takes longer to finish than what's estimated, we expect performance degradation; on the other hand if the transmission finishes sooner than expected, then data retrieval plan computed by vLDF would have been too conservative, in the sense that too much caution would have been taken to prevent freshness deadline violations, and cost minimization could have been carried out more aggressively. These intuitions are confirmed by results shown in Fig. 4.9: the advantage margin of vLDF over SCB and PbP are at its largest when the mean of the network fluctuation is at 0, and shrinks if the mean moves to both the negative and positive directions. The general trend that underestimating transmission time (positive fluctuation mean) leads to poorer performance is understandable, as longer actual transmission time will lead to more data freshness violations and thus potentially

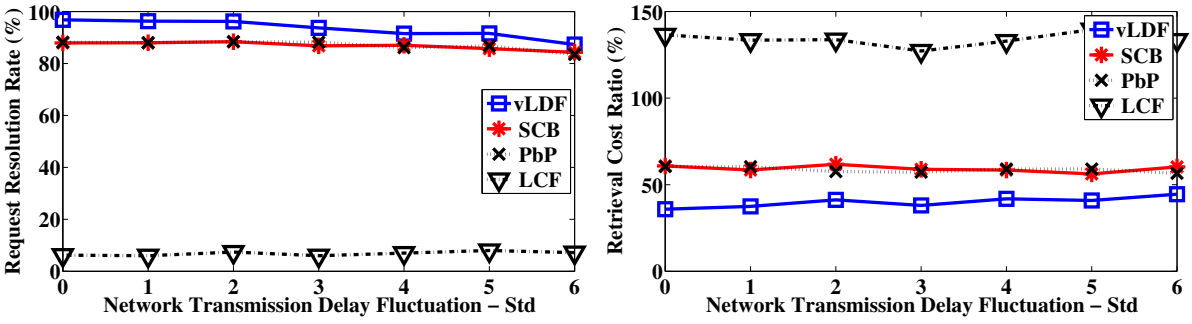
more request resolution timeouts and more data refetches.



(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.9: Network Delay Noise Mean

In addition to the mean, we also experiment with various levels of network fluctuation standard deviations. Results are shown in Fig. 4.10. We can observe for our vLDF algorithm that, as the standard deviation increases, the proportion of requests that can get resolved decreases and the retrieval cost ratio slightly increases. These are reasonable because of the higher level of unexpectedness of the network transmission delays caused by the larger standard deviations. All other baseline methods do not show clear trend of changes, because the mean of the fluctuation is set at 0 for this set of experiments, thus positive and negative effects tend to cancel out over time.



(a) Request Resolution Ratio (b) Retrieval Cost Ratio

Figure 4.10: Network Delay Noise Variance

4.5.2 Route Finding Application

After rather abstract simulation experiments, we now look at a few concrete instances of the route planning application running using our implemented system. As it is difficult to find actual post-disaster scenarios to test our systems in, we take the following steps as a way to emulate the disaster settings: We imagine a chaotic environment in the Urbana-Champaign IL region. We crawl Google Maps Street View images, and Instagram's Urbana-Champaign traffic accidents and road blocks images (all geotagged). We insert all image meta data to the meta store, and use the Emane/Shim [13] network simulator to intercept all image requests (where link bandwidth is set at 5KBps). Each image's probability of showing a road segment of being in good condition is set to be reversely proportional to the road segment's speed limit (accidents tend to happen on high speed roads rather than residential roads). Freshness intervals are also set reversely proportional to speed limit, as whatever abnormalities on higher speed roads tend to get cleared more quickly. We experiment with two different underlying data retrieval algorithms: our vLDF, and the PbP baseline. The end to end query resolution image retrieval cost and latency comparisons are shown in Table 4.1. As seen, vLDF consistently over-performs the baseline. Fig. 4.11 shows the actual route finding results from a particular run. The PbP scheme tests Route I and II first before testing Route III, where our algorithm vLDF selects Route III to check first.

vLDF Cost (KB)	*PbP* Cost (KB)	*vLDF* Time (s)	*PbP* Time (s)
516	685	164	255
343	598	150	206
319	485	160	248
506	1093	165	372
524	1042	175	206

Table 4.1: 5 Example Route Planning Results

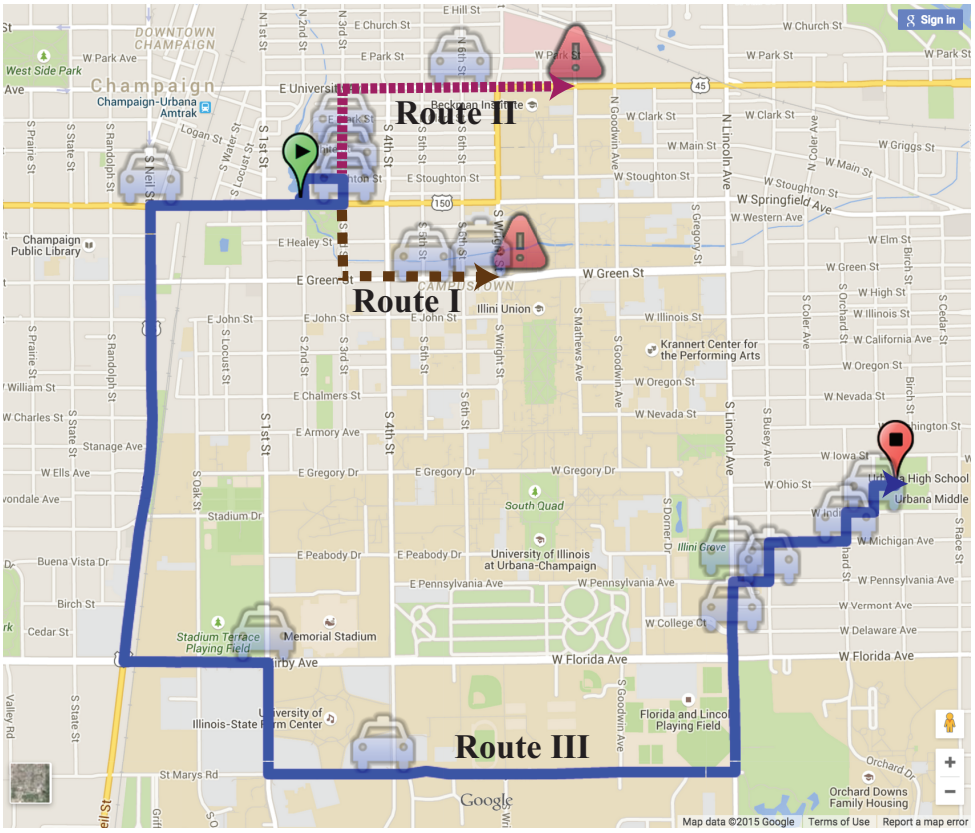


Figure 4.11: Screenshot of Example Results of Performing Route Planning using our Web Front-end Interface

Chapter 5

Distributed Semantic-Aware Information Management

5.1 System Overview & Architectural Design

As mentioned in the introduction, we view decision-making as the process of making a logical choice between multiple alternative courses of action. Making this choice entails gathering and weighing several relevant, but distinct pieces of information. More specifically, in our model, the viability of a course of action is assumed to depend on satisfaction of a set of predicates. Making a decision reduces to evaluating a Boolean expression on such predicates to arrive at an acceptable course of action. Evaluating a predicate requires acquisition of supporting evidence. Typically a data object obtained from a sensing source carries the pertinent information. The role of Athena thus lies in supporting the acquisition of evidence needed to evaluate the viability of different courses of action involved in decision-making. By taking advantage of the higher-level logical decision structures, represented by Boolean expressions, and by observing which sources have which pieces of evidence, Athena significantly improves the efficiency of decision-making.

Athena offers a query-based interface, where each query asks for information to support a decision. It is novel in that queries are backed by Boolean expressions specifying the decomposition of the decision into the predicates that need to be evaluated for the corresponding choice to be made. In principle, there are no limits on the types of queries that can be expressed as long as they can be represented by Boolean expressions over predicates that the underlying sensors can supply evidence to evaluate. A repository of such expressions is thus maintained that helps decompose the queries. Initially, we might only be able to support a limited set of query types for which we have decomposition logic in the repository. This set can progressively be expanded later to enrich Athena's capabilities, without affecting the underlying information collection and dissemination mechanics. While the design of such a repository is itself an inter-

esting and challenging problem, here we focus on the following complementary question: given the decomposition of each query into a known graph of logical predicates to be established, how best to deliver the requisite information?

Let us look at a toy example to help make the picture more concrete: Suppose after an earthquake, there is a shortage of air support and an emergency medical team needs to transport a severely injured person from an origin site to a nearby medical center for surgery. There are two possible routes to take: One composed of segments A–B–C, and the other D–E–F. We need to make sure that the chosen route is in good enough condition for our vehicle to pass, so we want to retrieve pictures from deployed roadside cameras in order to verify the road conditions and aid our decision-making on which route to take. Our route-finding query can be naturally represented by the logical disjunctive norm form $(viable(A) \wedge viable(B) \wedge viable(C)) \vee (viable(D) \wedge viable(E) \wedge viable(F))$, where $viable(X)$ represents the predicate "segment X is viable". This expression signifies that at least all segments of one route need to be viable for the transport to occur. In this example, if road segments A, B and C all turn out to be in good condition (i.e., Route 1 is viable), then there is no need to continue retrieving pictures for road segments D, E, and F. Similarly, if a picture of segment A shows that it is badly damaged, we can then skip examining segments B and C as Route 1 isn't going to work anyway. Instead, we can move on to explore option D–E–F.

As is hopefully evident from this toy example, exploiting decision structure (represented by the Boolean expression) enables us to take inspiration from heuristics for short-circuiting the evaluation of logical expressions to schedule the acquisition of evidence. Specifically, we can acquire evidence in an order that statistically lowers system resource consumption needed to find a viable course of action. By incorporating additional meta-data (e.g., retrieval cost of each picture and the probability of each road segment being in good or bad condition), we can compute retrieval schedules that better optimize delivery resources expended to reach decisions as will be detailed later in the algorithm description.

We represent each predicate that a query needs with a label. For instance, in the routing example above, the predicate $viable(X)$ can be represented by the label $viableX$, denoting a Boolean variable of value True or False. The decision is associated with labels $viableA$, $viableB$,

..., *viableF*. Sources that originate data, such as sensors, advertise the label names that their data objects help resolve together with the objects that resolve them. These advertisements are propagated in the network, much like routing updates, forming a query routing backbone. Multiple sources may advertise the same label. Note that, this architecture effectively changes the query paradigm from specifying *what* objects to retrieve to specifying why they are needed. Specifically, objects are needed to resolve predicates named by corresponding labels that are explicitly mentioned in the query. This change allows the network to be much smarter when answering a query.

In the current implementation, source advertisements of semantic labels that their objects help resolve are propagated over an overlay structure of servers, called *semantic stores*. At a high level, query resolution works as follows. Athena first determines the set of predicates (i.e., labels) that is associated with a query from the underlying Boolean expression. This is the set of labels whose values need to be resolved. The query source then consults the nearest semantic store to determine the set of sources with relevant objects. Redundancy is eliminated, if multiple sources cover the same label. An algorithm at the query source then decides on the order in which objects must be retrieved to evaluate the different labels. It is inspired by minimum-cost algorithms for Boolean expression evaluation. Say, the algorithm decides to resolve the value of the label, *viableX*. This label is propagated to a source that has the needed data. If the label has already been evaluated in the recent past, its evaluation is cached in the network, in which case the resolved value may be found and returned. This is the cheapest scenario. Otherwise, if the object needed to evaluate the predicate has been recently requested, it may already be cached. The query is propagated to the first node who has the object, or the source who advertised it. The relevant data object is thus found and returned. Previously unresolved predicates are evaluated in view of the arrived evidence object. The resulting new (label, value) pairs are cached in the network with a freshness interval for future decisions to use.

Athena operates in a distributed fashion, where all participating nodes are treated equally and any node can potentially serve as the requester, forwarder, or content source for any particular query. Since multiple requests involving overlapping nodes can be processed concurrently, a node can potentially take on multiple different roles at the same time for different queries.

As mentioned above, Athena caches data objects on their way to the query originator. When evidence arrives and predicates related to a query are evaluated, Athena propagates the resulting values (together with freshness intervals) back into the network, allowing other decisions to make use of the results. By allowing all nodes to cache data objects that pass by (as storage is extremely cheap today, we do not consider it to be a system constraint), we can fulfill data requests before reaching the actual data source node, if some intermediate nodes have a valid, local cached copy of the requested data. What's more, since each predicate's true/false state label value is propagated back into the network, rather than having to always fulfill requests with actual data (evidence) objects, we are able to respond with already evaluated predicate labels. Data objects and evaluated predicate labels (True/False values) are signed by the source who did the data collection or predicate evaluation, respectively. A query can thus specify whose signatures to trust, thereby causing untrusted sources to be ignored.

To help better visualize how Athena operates and how information is processed and propagated, a pictorial illustration of the complete Athena architecture is shown in Fig. 5.1. As seen, a user's query is accepted and then translated to the corresponding logical expression by *Application Semantic Translator*, which uses the nearby *Semantic Store* to identify the set of nodes that have evidence objects. Taking this information, the *Logical Query Resolution Engine* then uses the *Source Selector* to minimize redundancies of the node set, and proceeds with query processing with the minimal node set. Data requests are then scheduled in a resource efficient manner, and handed over to the *Information Collection & Dissemination Engine*, which accesses the underlying sensing and communication stacks to handle all incoming and outgoing requests, data objects, and resolved predicate label values, with proper book-keeping and information updates to its various internal components, which we will discuss in detail next in Sec. 5.2. Please note that, as requested data objects arrive, they are presented to the user for labeling; the human provided labels are then propagated back into the network for opportunistic future uses. And of course, as each query is resolved, the result is presented to the original requesting user.

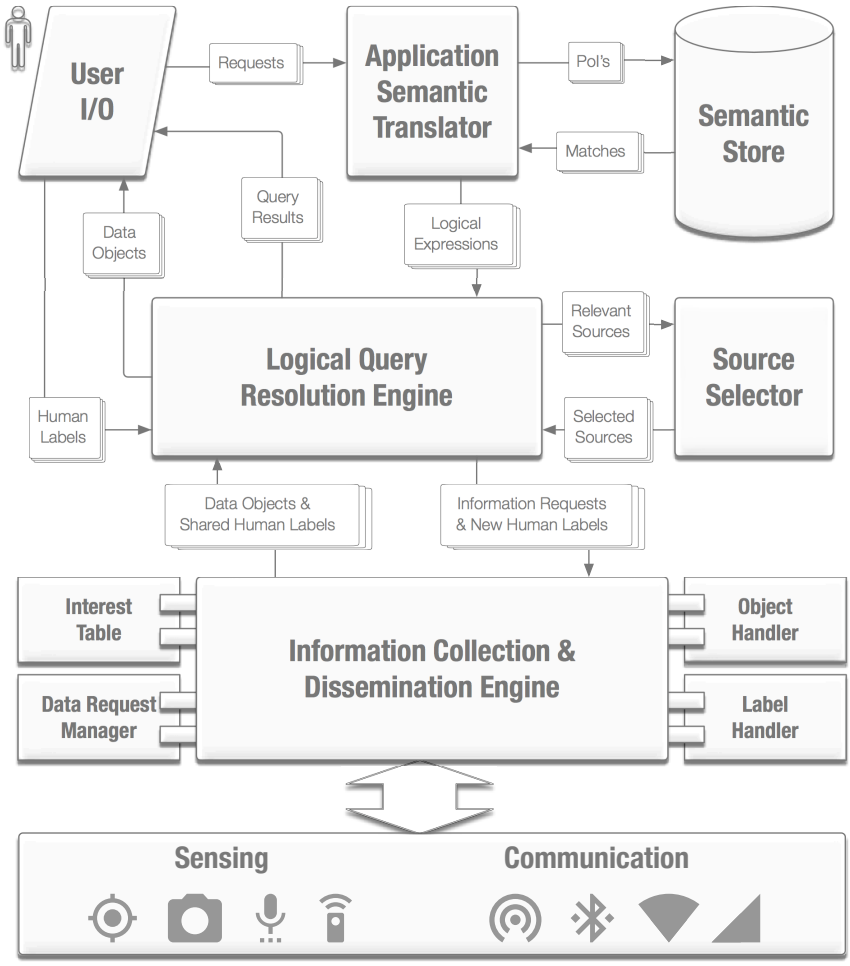


Figure 5.1: Athena's Architectural Design, with Illustration of Information Propagation through Different System Components

5.2 Information Dissemination

Given an overview of Athena's architecture, we next take a closer look at how information dissemination operations are managed in Athena.

5.2.1 Query Requests

A user can issue query request(s) at any Athena node, using a *Query_Init* call. At each node, upon user-query initiation, Athena translates the query into the corresponding Boolean expression over predicates, and starts carrying out necessary evaluation processing. This processing is done in the context of *Query_Recv*. The component reacts to received queries (either initiated locally or propagated from neighbor nodes) by carrying out four execution steps: (i) add the

new query to the set of queries currently being processed by the node; (ii) determine the set of sources with relevant data objects using the nearest semantic store [33, 49], and compute the optimal source subset using a source selection algorithm [6]; (iii) send the Boolean expression of the query to neighbors and (iv) compute an optimal object retrieval order according to the current set of queries. The detailed ordering algorithm is discussed in Sec. 5.3. Requests for those objects that are slated for retrieval are then put in a queue, called the *fetch queue*. Note how, in this architecture, a node can receive the Boolean expression of a query from step iii above before actually receiving requests for retrieving specific objects. This offers an opportunity to prefetch objects not yet requested. A node receiving a query Boolean expression from neighbor nodes will try prefetching data objects for these remote queries, so these objects are ready when requested. Such object requests are put in a *prefetch* queue. The prefetch queue is only processed when the fetch queue is empty. When a queue is processed, an object *Request_Send* function is used to request data objects in the fetch/prefetch queue from the next-hop neighbors.

5.2.2 Data Object Requests

As a query is decomposed, as stated above, into a set of data object requests, each corresponding to a specific label to be resolved. These requests are then sent through the network towards their data source nodes. We need to be careful here because of the potentially severe system resource limitations in a post-disaster setting. Each node maintains an *Interest Table* that keeps track of *which data objects* have been requested by *which sources* for *what queries*, if the requested data objects have already arrived, and *how fresh* each data object is (i.e., the timestamps of when the sensory data was sampled by the corresponding sensor hardware). The interest table helps nodes keep track of upstream requesters and avoid passing along unnecessary duplicate data object requests downstream.

One other important functionality of the interest table is that it serves as a data cache, storing data objects that pass through, so new requests for a piece of data object that is already cached *might not* have to be passed along to the final source node, thus preserving network bandwidth usage. When a forwarder node already has a cached copy of a piece of data, it needs to make the decision as to whether or not this cached copy is still *fresh* enough to serve an

incoming request for this piece of data. If yes, then the forwarder would just respond to this request by returning the cached object, otherwise it would pass along the request towards the actual source for a fresh copy.

Specifically, a *Request_Recv* is called upon receiving an object request from a neighbor. The request is first bookmarked in the interest table. Then, if the object is not available locally, the request is forwarded (using *Request_Send*) closer to the data source node if the request was a fetch (prefetch requests are not forwarded).

Above, we just discussed how data object requests are handled by Athena nodes. Next, we will look at how Athena handles the transmission of the actual requested data content, either from actual data source nodes or intermediate nodes upon cache hits, back towards the requesters.

5.2.3 Data Object Replies

Requested data objects (e.g. a picture, an audio clip, etc) are sent back to corresponding requesters in the similar hop-by-hop fashion as that of the requests themselves. Each data object, as it is being passed through intermediate forwarder nodes, is cached along the way. Since storage nowadays is extremely cheap, we do not worry about evicting cached content due to hitting space limitations. Because of the environment dynamics discussed previously, cached data objects will decay over time, and eventually expire as they reach their freshness deadlines (age out of their validity intervals). In terms of functional interfaces, each Athena node implements the following two function: *Data_Send* is used to send requested data object content back towards the original requesters; and *Data_Recv* is invoked upon receiving a piece of requested data object, which is then matched against all entries in the interest table. If the current node is the original query requester node, the data object is presented to the user for the label value, which is in turn used to update the query. Otherwise, the object will be forwarded to the next hop towards the original requester.

One important note here is that in Athena, a piece of *raw data object* needs to be sent from the source back to the requester only when the predicate evaluation (labeling) has to be done by the requesting source. For example, after an earthquake, a user is using Athena to look for a safe

route to a nearby medical camp. In doing so, Athena retrieves road-side pictures along possible routes for the user to examine. This judgment call—looking at a picture and recognizing it as a safe or unsafe road segment—is put in the hands of the user (the human decision maker) at the original query requester node. Alternatively, predicate evaluation could be made by machines automatically (e.g., using computer vision techniques to label images). If a qualified evaluator is found at a node for a given predicate, the predicate can be evaluated when the evidence object reaches that node. If the source of the query specified that the signature of this evaluator is acceptable, only the predicate evaluation is propagated the remaining way to the source (as opposed to the evidence object). In the implementation, we restrict predicate evaluators to sources of the query.

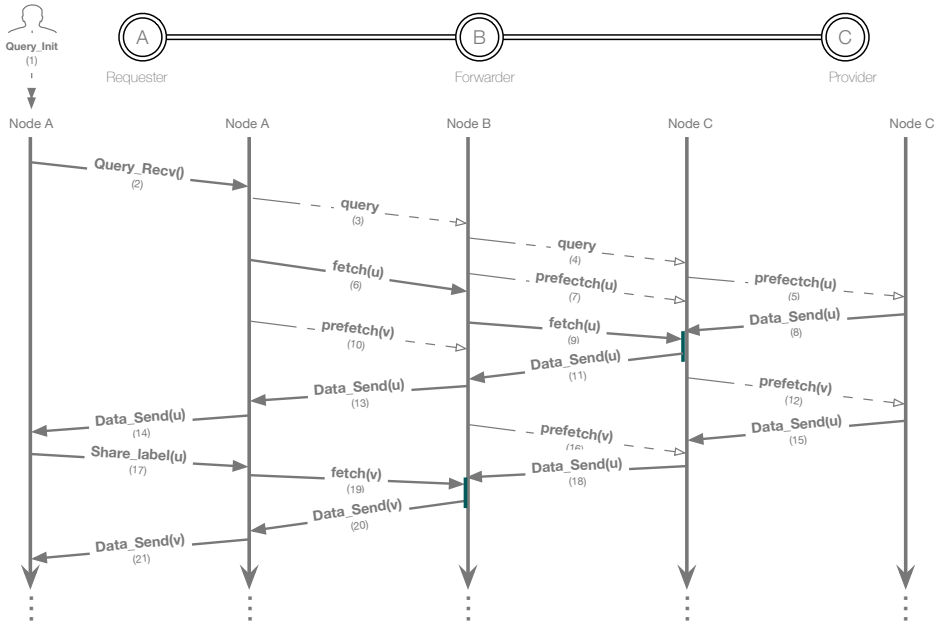


Figure 5.2: A Visualization Showing the Flow of Requests and Data as Nodes in Athena Work Together for Query Resolution

5.2.4 Label Sharing

As requested data objects arrive, the query source can then examine the objects and use their own judgment to assign label values to the objects for the particular query task. These labels are injected back into the network, such that future data requests might potentially be served by the semantic labels rather than actual data objects, which depends on whether the requests need

to evaluate the same predicates, and what trust relations exist among the different entities (e.g. Alice might choose not to trust Bob's judgment, and thus would insist on getting the actual data object when a matched label from Bob already exists). As such human labels are propagated from the evaluator nodes back into the network towards the data source nodes, they are cached along the way, and can be checked against the interest tables and, upon matches, used locally to update query expressions, and forwarded to the data requesters. Compared to sending actual data objects, sharing and utilizing these labels can lead to several orders of magnitude resource savings for the particular requests.

To help better visualize how the various discussed components work together, we show, in Fig. 5.2, an example of requests and data flows for a particular query. In the example shown, the user uses `Query_Init()` to create and issue a query at Node A. The query in our example involves two data objects, u and v. Node A calls `Query_Recv()` locally to start the processing. The query is propagated through the network (edge 3 and 4), reaching Node B and C. Upon receiving the query, Node C attempts prefetching, first for data object u in this particular example. Since Node C is the data source for u, it sends u back towards the requester node (edge 8, 11, 13, and 14), during which, Node A's fetch request meets the returned data at Node C (edge 9). Upon receiving u at Node A, the user examines the data, makes a judgment regarding the corresponding condition state of the query. This state label provided by the human decision maker is then propagated back into the network (edge 17). The handling for data object v follows a similar pattern, for which the fetch request has a cache hit at the forwarder node B, without reaching the actual source node C, due to prefetch requests.

5.3 Information Retrieval Scheduling

We now discuss how information retrieval is planned at each node with the goal of reducing system resource consumption and improving information dissemination efficiency. We build upon the algorithms that we developed in Chapters 3 and 4, which only considered centralized situations and without any caching mechanism. We encourage readers to refer back to the previous chapters for more details; we do also attempt to make the following discussion self-

contained without much repetition.

We first introduce the key notations used: Each decision-making task is to find a valid course of action among several alternative candidates. Thus each query is naturally represented as a logic expression in disjunctive normal form (OR of ANDs). We use $\{a_i\}$ to denote the set of alternative courses of action, and $\{t_{i_j}\}$ the set of conditions that determine viability of a_i. Therefore, a query q takes the general form

$$q = \underbrace{(t_{0_0} \wedge t_{0_1} \wedge \ldots)}_{a_0} \vee \underbrace{(t_{1_0} \wedge t_{1_1} \wedge \ldots)}_{a_1} \vee \ldots .$$

Associated with each condition t_{i_j} are the following pieces of meta information: i) evaluation cost c_{i_j} (e.g. data retrieval bandwidth cost), ii) estimated retrieval latency l_{i_j}, iii) success probability p_{i_j} (i.e., probability of evaluating to TRUE value), and iv) validity interval d_{i_j} (i.e., how long the data object remains fresh). Then, the question we need to answer is *when to retrieve which piece of data object*.

When facing system resource limitations (e.g., low network transmission bandwidth), sequential processing is in general more efficient compared to parallel processing, as it gives us the opportunity to take advantage of the decision logic structure to short-circuit and prune unnecessary retrievals in view of previously retrieved objects. Simply put, when handling an AND,

$$a_i = t_{i_0} \wedge t_{i_1} \wedge t_{i_2} \wedge \ldots ,$$

We want to start with the *most efficient* t_{i_j} and proceed downward. Here, "most efficient" means *highest short-circuit probability per unit cost*

$$\frac{1 - p_{i_j}}{c_{i_j}}.$$

For example, imagine a particular course of action that consists of just two conditions, h and k, which require retrieving and examining a 4 MB and a 5 MB audio clip, respectively. And it has been estimated (e.g., from historic data, or domain expert knowledge, etc) that condition h has a 60% success probability, and k 20%. In this case, we would want to evaluate k first as it has

a higher short-circuiting probability per unit bandwidth consumption

$$\underbrace{\frac{1-0.2}{5}}_{0.16} > \underbrace{\frac{1-0.6}{4}}_{0.1},$$

and this evaluation order would lead to a lower expected total bandwidth consumption compared to the other way around (i.e., evaluating h before k)

$$\underbrace{5+0.2\times 4}_{5.8} < \underbrace{4+0.6\times 5}_{7}.$$

After the data retrieval order for each a_i is determined, their corresponding expected short-circuiting probabilities (relative to the OR) and evaluation costs can in turn be computed. Then, similarly for the handling of the entire query OR

$$q = a_0 \vee a_1 \vee a_2 \vee \ldots,$$

we also start processing from the a_i with the highest short-circuiting probability per unit cost.

As mentioned previously, environment dynamics in the post-disaster scenarios will cause state conditions to change over time due to information decay (hence the validity interval). Therefore, it is important that, at the time a decision is made, all pieces of information involved must still be fresh. Otherwise, decisions will be made based on (partially) stale information. We borrow from our previously introduced greedy algorithm in Chapter 4.3, where all data object requests are first ordered according to their validity intervals (longest first) and then rearrangements are incrementally added, according to objects' corresponding short-circuiting probabilities per unit cost, to reduce the total expected cost. This corresponds to Line 4 through 13 in Alg. 10, and the basic idea stemmed from the simple insight that the shorter a data object's validity interval is (i.e., the more fast-changing the corresponding state is), the later it should be retrieved, in order to avoid its becoming stale while waiting for the retrieval of other slow-aging data objects.

As Athena operates in a distributed fashion, it allows data objects to be cached at all intermediate forwarder nodes as they are sent through the network towards their respective

Algorithm 10 Retrieval Schedule for Dynamic Query Resolution

Input: A query's deadline requirement d_q, its candidate courses of action $\{a_i\}$, and acceptable parallel retrieval level r. For each constituting condition t_{i_j} for a particular a_i, its corresponding evaluation (retrieval) costs c_{i_j}, retrieval latencies l_{i_j}, success probabilities p_{i_j}, and freshness (validity) interval v_{i_j}. And finally the subset of conditions that have been cached O_q, in descending order of $\frac{1-p_{i_j}}{c_{i_j}}$.

Output: Query resolution result

$failure\_counter \leftarrow 0$
$L_a \leftarrow \{a_i\}$ sorted in descending order of $\frac{p_i}{c_i}$
for a_i in L_a **do**
  $Q_c \leftarrow \emptyset$, $Q_d \leftarrow$ longest valid. interval first order, $S_p \leftarrow \emptyset$
  $L \leftarrow \{t_{i_j}\}$ sorted in descending order of $\frac{1-p_{i_j}}{c_{i_j}}$
  while $Q_d \neq \emptyset$ **do**
    for t_l in L **do**
      if moving t_l from Q_d to the end of Q_c does not increase freshness violation degree **then**
        $Q_d \leftarrow Q_d \setminus \langle t_l \rangle$, $Q_c \leftarrow Q_c + \langle t_l \rangle$
        break
      end if
    end for
  end while
  while $|Q_c| > 0$ **do**
    $t_e \leftarrow$ end element of Q_c
    $Q_c \leftarrow Q_c \setminus \langle t_e \rangle$, $S_p \leftarrow S_p \cup \{t_e\}$
    if $Q_c + S_p$ satisfies validity intervals **then**
      for t^o in O_q **do**
        $d_{t^o} \leftarrow t^o$'s absolute validity deadline
        if $Q_c + S_p \setminus \langle t^o \rangle$ satisfies $\min(d_q, d_{t^o})$ **then**
          $Q_c \leftarrow Q_c \langle t^o \rangle$, $S_p \leftarrow S_p \setminus \langle t^o \rangle$
        else if a shortest tail, T_{Q_c} of Q_c's can be moved to S_p to satisfy validity intervals **AND** $|S_p| \leq r$
          $Q_c \leftarrow Q_c \setminus T_{Q_c}$, $S_p \leftarrow S_p \cup T_{Q_c}$
          update d_q
        end if
      end for
      Process a_i with retrieval schedule $Q_c + S_p$
      if a_i succeeds **then**
        return a_i as an successful result
      else
        increment $failure\_counter$ by 1
        break
      end if
    end if
  end while
end for
if $failure\_counter == |L_a|$ **then**
  return request resolves to failure
else
  signal validity interval cannot be satisfied
end if

requester nodes. This gives us the opportunity to let intermediate nodes respond to data requests with their locally cached objects upon cache hits, rather than always having to pass along requests to their targeted data source nodes. Intuitively, whether a cached object could be used or not depends on its "age": if it is still reasonably fresh, we go ahead serving the request with the cached object; otherwise (the cached object is approaching the end of its validity interval, or is already stale), we might need to pass along the request to ask the data source node for a fresh sample of the desired data object.

To make our information retrieval planning method capable of taking advantage of possible cache hits, we observe the different characteristics between the validity intervals of a data object to be freshly sampled from its source node and a cached copy from an intermediate node: For the former, it starts decaying at the time of sampling, which means its validity interval is *relative* to when the requester asks the source for the sample; however, for the latter, the cached object's validity interval is *absolute* in the sense that it has already started decaying irrespective of when the current requester asked for the data object. With this observation, we can treat the absolute validity intervals of cached data objects as deadlines, and combine them with query deadlines when computing the data retrieval schedule. This corresponds to Line 14 through 35 in Alg. 10. A query level deadline is an absolute temporal requirement on when a particular query must be resolved, which could indicate different urgent levels. When facing multiple queries, an Athena node processes them in an earliest deadline first (EDF) order, favoring queries with earlier deadlines.

The complete algorithm pseudo-code is shown in Alg. 10.

5.4 Evaluation

In this section, we first give a brief description of our prototype Athena implementation, then discuss the set of experiments we carried out in testing Athena's various aspects, and finally present our results and findings.

In our prototype Athena implementation, each of the functional components (as discussed in Chapter 5.2) runs in its own separate thread within the process for the corresponding node.

Three main data structures, local/remote query logs, fetch/prefetch queues, and the interest table, are asynchronously protected and shared among the functional component threads. A node's main event loop simply waits on a TCP socket for incoming messages, and dispatches received messages, according to their headers, to spawn the corresponding functional threads.

To simulate a network, we ran the actual protocol code in a separate process per emulated node. Each node was uniquely identified by its IP:PORT pair. We were also able to run multi-node simulation experiments from a single workstation, where each node used a separate port, while sharing the same local host.

As we have not had the opportunity to test our system in an actual post-disaster environment, we adopted a set of simulation-based experiments featuring a post-disaster route-finding scenario, where Athena is deployed in a disaster-hit region and is used by people in the region to carry out situation assessment and route-finding tasks. For simplicity, we consider a Manhattan-like map, where road segments have a grid-like layout. The EMANE-Shim network emulator [13, 57] was used to handle all data object transmissions. Below, we first discuss our simulation settings, and then proceed to talking about each set of the experiments.

For information retrieval schedule, we experimented with multiple baselines, besides our own algorithm as introduced in Section 5.3. All compared algorithms are listed as follows:

- *Comprehensive retrieval (cmp)*: As a first baseline, we include a simple algorithm where all relevant data objects for each query are considered for retrieval.

- *Selected sources (slt)*: This is one step beyond the above comprehensive retrieval baseline, where data source selection is performed to minimize the candidate set of data objects to be retrieved to cover all predicates in queries (e.g., if two cameras are overlapping on an road segment that we are interested in, we then can carry out source selection to help determine which one we use, but if two different roads are considered, we retrieve objects that cover both roads). We borrow a state of the art source selection algorithm [6] and use it in our implementation and experiments.

- *Lowest Cost Source First (lcf)*: This scheme takes the above *selected* source nodes, and sorts them according to their data object retrieval costs (i.e., data object size), prioritizing

objects with lower costs.

- *Variational Longest Validity First (lvf)*: Our scheduling algorithm as discussed in Sec. 5.3, where human labels are *not* propagated into the network for future reuse.

- *Variational Longest Validity First with Label Sharing (lvfl)*: Our scheduling algorithm, *with* human label sharing enabled. So after a human decision maker examines a piece of retrieved data object and assigns to it a label value, this label information is propagated back into the network towards the corresponding data source node. Thus any node along the path that intercepts a future request for this data object can potentially return this label value rather than (requesting and) returning the actual data object.

With these above 5 different information retrieval scheduling schemes, we carry experiments to study Athena's behavior along the following different dimensions.

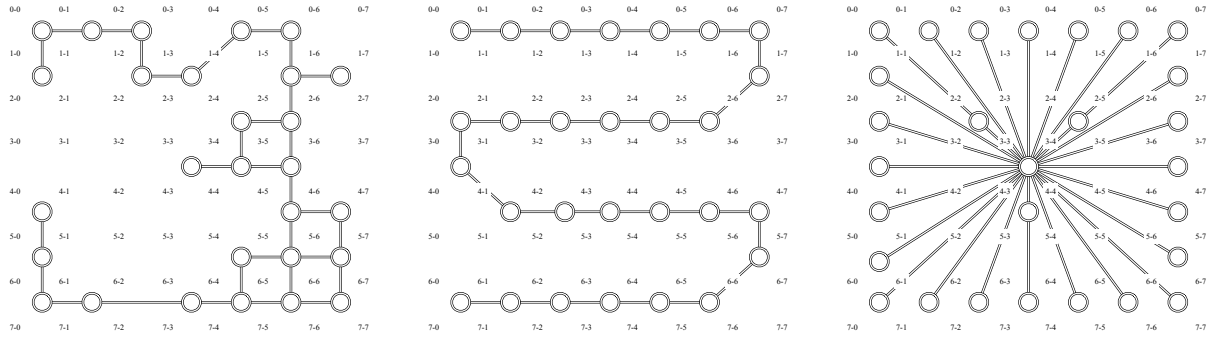
- Environment dynamics: We experiment with different levels of environment dynamics, where different portions of data objects are considered to be of fast/slow-changing nature (i.e., having short/long validity intervals).

- Query issuance pattern: We experiment with how queries are issued to nodes–more specifically, whether all queries are issued to only a few nodes or a large number of them.

- Query complexity: For each query, we experiment with varying number of candidate routes, and different number of road segments per route. These correspond to the number of courses of action that are OR'ed together, and the number of predicates AND'ed to establish viability of each course, respectively.

- Query interest distribution: We experiment with two scenarios, namely whether all inquiries are focused on a small hotspot or spanning the entire global region.

- Query locality: We experiment with how "localized" queries are. Basically, a localized query inquires about a node's immediate surrounding area, whereas a more diverse query may ask about data objects on the far end of the network.



(a) Random (b) Linear (c) Star

Figure 5.3: Example of Various Different Network Topologies Deployed on Top of Road Segment Grids

- Network topology: We generally use randomly generated network topologies for our experiments. But we also experiment with two other specific network topologies, namely linear and star shapes, to see how different patterns of network connectivity might affect system behavior.

We divide the experimental region into a Manhattan grid given by an 8×8 road segment network, with around 30 Athena nodes deployed on these segments, where each node's data can be used to examine the node's immediate surrounding segments. Example scenarios are depicted in Fig. 5.3. As shown, each x-y pair indicates a single road segment and each double edge disk corresponds to a single Athena node. As seen, for the random topology example, parts of the network are more densely connected (the lower right portions), whereas other parts are more sparse; The linear and star shape topologies are used to experiment with more extreme connectivity patterns.

Data objects range from 100 KByte to around 1 MByte, roughly corresponding to what we might expect from pictures taken by roadside cameras. The network simulator is configured with 1 Mbps node-to-node connections. Each route-finding query consists of 5 candidate routes that are computed and randomly selected from the underlying road segment network. And each node is issued 3 concurrent queries. With these generic parameter settings, we next discuss each set of the experiments, where the corresponding particular parameter settings will be specified. For collecting results, each data point is produced by repeating the particular randomized experiment 10 times. As resource constraint is our main optimization goal, we use resource consumption

(network bandwidth usage) as the main evaluation metric, unless otherwise specified in particular sets of experiments.

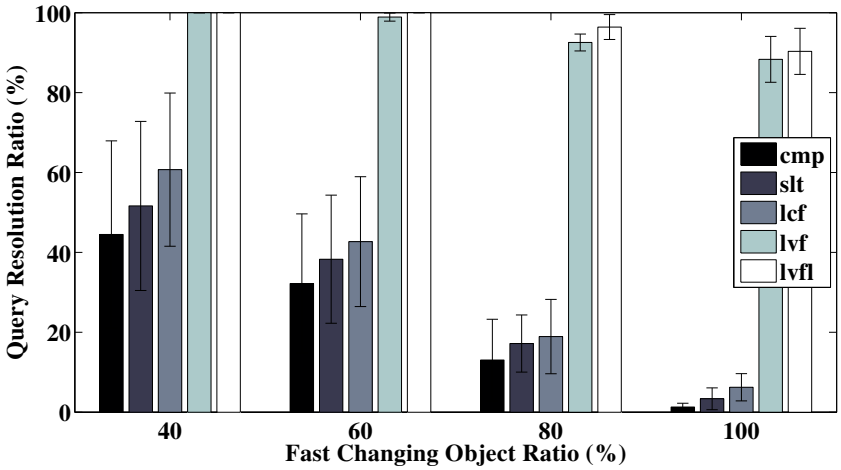


Figure 5.4: Query Resolution Ratio at Varying Levels of Environment Dynamics

First and foremost, as our Athena information management system is designed for situation assessment and decision-making under dynamic post-disaster environments, we look at how its query resolution capability is affected by different levels of environment dynamics. In our experiment, data objects generally belong to two different categories, namely slow changing and fast changing. For example, a blockage on a major highway might get cleared within hours, but a damaged bridge likely will take days/weeks or even longer to repair. In this set of experiments, we explore how different mixture of slow and fast changing objects affect the performance of each of the information retrieval schemes. The results are shown in Fig. 5.4. As seen, at all levels of environment dynamics, our data-validity aware information retrieval schemes are able to successfully resolve most, if not all, queries, whereas the baseline methods struggle even with a relatively low level of environment dynamics. This is due to their failure to take into account the data validity information when scheduling retrievals, which then leads to data expirations and refetches. This not only increases bandwidth consumption, but also prolongs query resolution process, potentially causing more data to expire.

The actual network bandwidth consumption comparisons of all schemes are shown in Fig. 5.5. We already saw from Fig. 5.4 that the various baseline schemes fall way short in terms of query resolution ratio; here we observe that they additionally consume more network bandwidth.

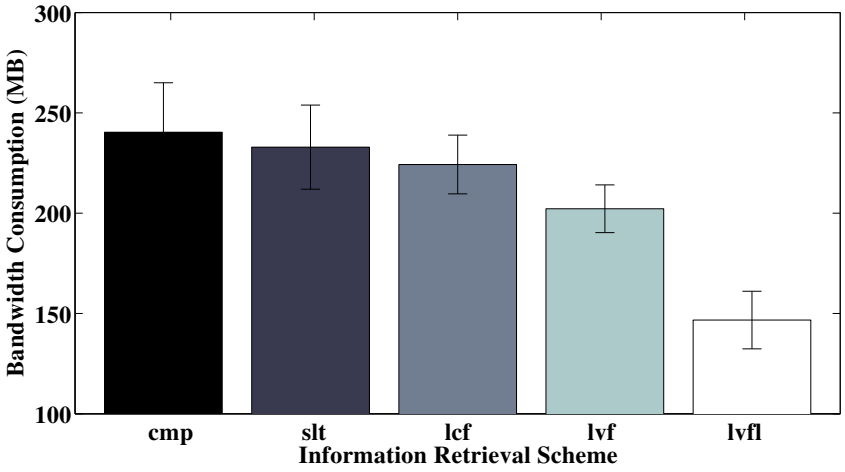


Figure 5.5: Total Network Bandwidth Consumption Comparison (with 40% Fast Changing Objects)

Comprehensive retrieval scheduling incurs the highest amount of network traffic, as it is neither careful about avoiding overlapping data object retrievals, nor does it try to follow any meaningful order when fetching data. Network bandwidth consumption marginally decreases as we include source selection (slt) and then follow a lowest-cost-first (lcf) data retrieval schedule. As mentioned before, none of these schemes take into consideration environment dynamics. Therefore, they tend to result in more information expiration and refetchings, leading to extraneous bandwidth usage. This additional usage is effectively minimized/avoided by our scheduling strategy, which leads to a considerable decrease in network bandwidth consumption. Additionally, when opportunistic label sharing (lvfl) is enabled in Athena, a more significant bandwidth saving is observed, as expected, since labels are transmitted instead of actual data objects when possible.

Please note that we intend to use this first set of experiments to give readers an overall impression of system performance comparison, and the trends shown remain for the rest of our experiment sets. Therefore, for presentation clarity and brevity, for the rest of this section, we omit the inclusion of error bars when appropriate as well as query resolution ratio plots. We instead focus on presenting more results along different dimensions showing more interesting bits.

We next take a look at how query issuance patterns affect system performance. This essentially means how many nodes of the entire network users are issuing queries to. If the number is 1 (which is less likely for realistic scenarios), this essentially is equivalent to a centralized

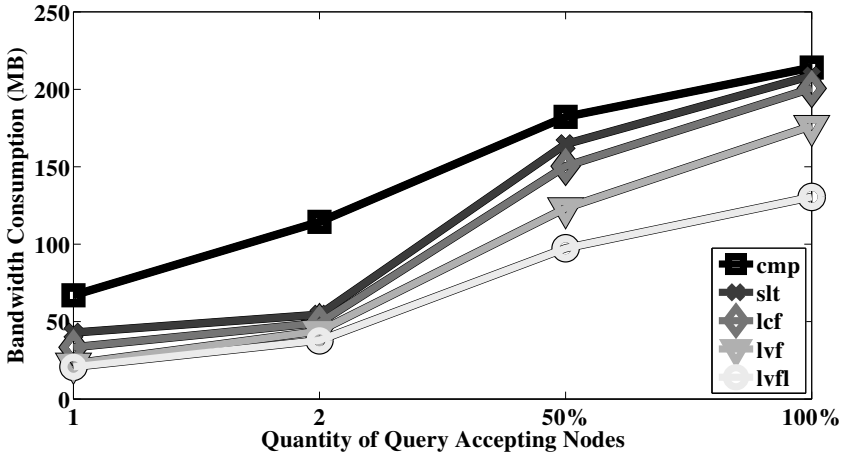


Figure 5.6: Different Query Issuance Patterns

network where all users issue all queries at a central node; under realistic settings, however, this number will be high as users all around the region would potentially need to request information at each of their own respective locations. We experiment with issuing queries to a single, a pair, half, as well as all nodes. The results are shown in Fig. 5.6. First of all, we observe that the more centralized the query issuance pattern is, the lower the network bandwidth consumption. This makes sense because, given the same number of random queries, having fewer query nodes means higher chance of cache hits for both data objects and shared labels, which then leads to lower number of transmissions of data objects from their original source nodes to requester nodes. We also observe that, as query issuance pattern shifts from centralized towards distributed, our information retrieval schemes lead to slower network traffic increase compared to baseline methods as well as consistently stable performance on query resolution ratio.

Next up, we look at how query complexity affects system performance. As each query is represented in its logical form, an OR of ANDs, we can vary the number of ANDs under the OR, as well as the number of tests that needs to be performed for each AND in our route finding scenario. This naturally corresponds to the number of candidate routes for each query, and the number of road segments for each candidate route. As shown in Fig. 5.7, increasing the number of routes per query leads to higher network bandwidth consumption for all information retrieval schemes. It is worth noting that, our semantic-aware scheduling algorithms (lvf, lvfl) again lead to slower bandwidth increase compared to other baseline methods, thanks to their ability to

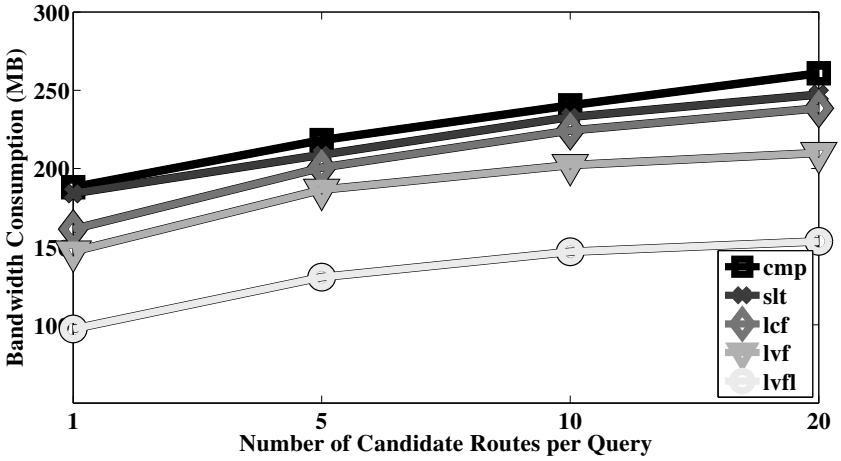


Figure 5.7: Varying Number of Candidate Routes per Query

exploit queries' internal structure and prune logical evaluations, which would otherwise lead to unnecessary network traffics.

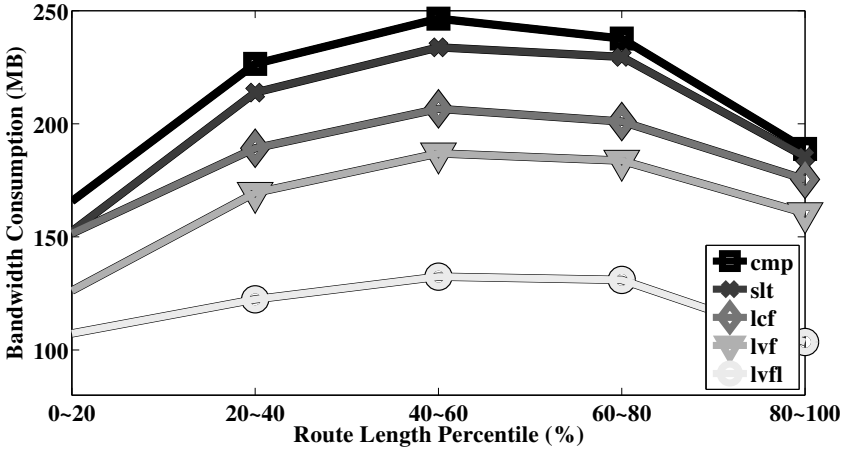


Figure 5.8: Varying Length Ranges of Queries' Intended Routes

Fig. 5.8 shows the experiment results with varying route lengths, where routes are categorized into 5 different length percentiles. As seen, the general relative comparison in terms of network bandwidth consumption remains the same as that of previous experiments. However, we notice an interesting convex shape rather than a monotonic trend (i.e., longer route lengths lead to higher network bandwidth consumption) that some might have expected. The reason lies in the fact that the higher the percentile bucket, the lower the number of route choices there are– i.e., we can easily find in the road network a large number of different short routes, but there

might be few options for extremely long ones. This lack of route choices then leads to a higher number of repeated road segments in the queries, which in turn leads to higher cache hit rates and thus fewer transmissions of data objects from their original source nodes. Therefore, this set of experiments also illustrates how queries' interest distributions affect system performance; namely, all other conditions being equal, the more concentrated the queries' interests are, the lower the system bandwidth consumption is. This is due to higher cache hit rates for data objects as well as shared labels values.

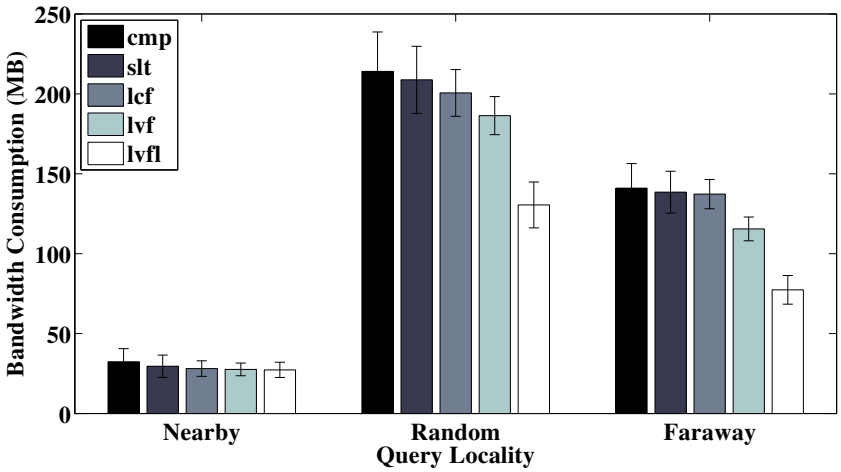


Figure 5.9: Different Levels of Query Localities

We next examine how queries' localities can affect system performance. Here, "locality" refers to how close a query's interests are from the node where it is issued. For example, a *nearby* query is likely only expressing interests in the vicinity of its issuance node, whereas a *faraway* query might inquire about data objects located at the far end of the network. Experimental results on network bandwidth consumption are shown in Fig. 5.9. We again omit reiterating the comparison between the different information retrieval scheduling schemes as it is similar to that of previous experiments. We do want to point out the non-monotonicity trend of bandwidth comparison as we move from nearby to random and then to faraway queries. First of all, when each query is only interested in its close vicinity, the query itself is often of low complexity (i.e., containing fewer candidate routes, and fewer road segments per route), and few data objects need to travel long paths to reach their requesters. Thus, the overall network bandwidth usage remains low. None of these mentioned characteristics still hold when queries' interests shift

from nearby to randomly covering the entire region, causing much higher traffic in the network. Then finally, when we further shift queries' interests to be only focusing on faraway objects, we have actually limited the object interest candidate pool, causing queries to overlap more on their relevant data object set. This too leads to higher cache hit rate for both data objects and human labels, similar to what we observed in Fig. 5.8.

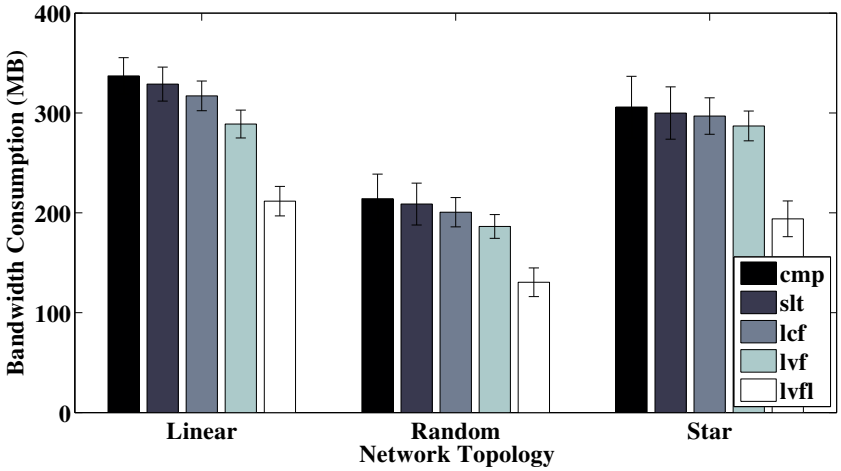


Figure 5.10: Different Network Topologies

Finally, we take a brief look at how different network topologies might affect system performance. In addition to randomly generated topologies, we also experiment with linear and star shape networks (example topologies as shown in Fig. 5.3). Results are shown in Fig. 5.10. As seen, networks where nodes are linearly connected result in significantly higher bandwidth consumption. This is understandable because of the excessively long paths data packets need to travel through to reach their destinations, which, together with the higher probability of information expiring caused by long retrieval latencies, overshadows the benefits of cache hits along the long paths. The star shape network, on the other hand, have short transmission paths. But the network bandwidth consumption is not much lower than that of linear networks, due to the existence of a bottleneck at their star centers, which lead to network congestions. This, in turn, causes more information expirations and excessive refetches. It is promising to see that in general random network topologies, Athena, with our retrieval schedule schemes, gives good performance.

Chapter 6

Related Work

Crowdsensing and social-sensing have become an important channel of data inputs, given rise by the fast growing popularity of various mobile, wearable, embedded devices, and their ubiquitous connectivities. For example, BikeNet [16] is a sensor network for bikers to share data and map regions. CarTel [30] is a mobile sensing system for automobiles, where data can be collected, processed, and visualized. Coric et al. [11] design a crowdsensing system that helps to identify legal parking spaces. MaWi [69] is an indoor localization system with improved accuracy by relying on crowdsensing for spot survey. SmartRoad [28] is a system that takes advantage of vehicular crowdsensing data to automatically detect and recognize traffic lights and stop signs. GreenGPS [19, 50] collects users' vehicle information and computes fuel-efficient routes for navigation. Various data transmission techniques have also been studied [27, 38]. Given the richness of crowdsensing systems, to help cleanup and therefore better utilize crowdsensing data, various fact-finding techniques [36, 43, 44, 59, 62, 63] have been proposed. The sensing and information management system introduced in this dissertation complements these prior studies by exploiting coverage and decision logic relations among data items when operating under resource limitations.

Liu et al. proposed a QoS-heterogeneous prioritization algorithm [40], to allow data packets with deadlines to be transmitted first in order to increase the possibility of offloading them faster. Previous efforts exist on redundancy elimination in networks including application-level [64] and packet level [5] techniques. Their work only try to eliminate redundant data, but do not consider the information carried in the data. For example, if two files have the very similar content, such as traffic speed measurements of the same street block at the same time, but different names, their work will consider these two files as different ones. However, these two files actually are

redundant in information, which will be handled efficiently by our approach.

Resource-limited scenarios (e.g., disaster monitoring, alert, and response) have been studied in the community. For example, Breadcrumb [39] is an automatic and reliable sensor network for the firefighting situations. In the SensorFly [48] project, low cost mobile sensing devices are utilized to build an indoor emergency response system. PhotoNet [56] provides a post-disaster picture collection and delivery service for situation awareness purpose. Our design and system can be used to operate on the above mentioned systems in improving communication efficiency under resource constraints.

The goal of improving communication efficiency and data/information quality is shared among several existing studies. For example, time-series prediction techniques are used to reduce communication burden without compromising user-specified accuracy requirements in wireless sensor networks [7]. Regression models are used to estimate predictability and redundancy relationships among sensors for efficient sensor retrievals [4]. MediaScope [33] is a crowdsensing system with various algorithms designed to help with timely retrievals of remote media contents (e.g. photos on participants' phones) upon requests of multiple types (nearest-neighbor, spanners, etc). Gu et al. [24] design inference-based algorithms for data extrapolation for disaster response applications. Minerva [61] and Information Funnel [60] explore data prioritization techniques based on redundancy or similarity measures for information maximization. Data aggregation techniques are also studied to reduce network transmission and improve classification tasks' accuracies [51–53]. Gregorczyk et al. [22] discuss their experiences on in-network aggregation for crowdsensing deployments. Tham et al. [54] propose the Quality of Contributed Service as a new metric for crowdsensing systems. Different from the above work, our system design exploits coverage and logic relations among data items, and can perform cost optimization under timeliness and freshness constraints.

More closely related to our work, Carlog [32], ACE [45], and Kim et al. [34, 35] have also studied optimization techniques for query requests. In particular, Carlog focuses on optimizing response latency under a vehicular sensing scenario; ACE aims at using association rules to improve energy efficiency for continuous mobile sensing; Kim et al. formulates this real-time scheduling problem under particular settings, and shows theoretical results as well as algorithmic

comparisons. Our work complements these research efforts by designing a holistic distributed information management framework under realistic settings.

Lastly, boolean predicate evaluation optimization has been studied in the theory community as well. It is generally referred to as the PAOTR (probabilistic AND-OR tree resolution) problem. Greiner et al. [23] have given analyses on various subproblems of PAOTR and given corresponding theoretical results. Luby et al. [42] have proposed a set of tools for analyzing the truth probability of AND-OR structures. Several heuristic-based algorithms and their performance comparisons have also been given by Casanova et al. [9] for the general NP-hard PAOTR problem. Inspired by these more theoretically oriented studies, our Athena system is designed to operate under much more complex and realistic settings, and handle concurrent requests, deadline requirements, freshness constraints, and distributed processing.

Chapter 7

Conclusion & Discussion

In this dissertation, we have explored information management under various system resource and environment constraints to enable efficient data collection and dissemination and facilitate effective situation assessment and decision makings. Several algorithms are proposed to cope with information redundancy, query deadlines, information freshness constraints, network system limitations, etc. A holistic fully distributed system, Athena, is designed and implemented to realize our vision of efficient and effective decision-centric information management. We evaluated our system through a comprehensive series of simulated experiments under realistic settings. Results show that Athena is able to efficiently manage information gathering and delivery in support of situation assessment and decision-making in the face of various dynamic environment and systems constraints.

In conducting, completing, and foreseeing possible future directions of the work reported in this dissertation, we do identify interesting topics that could potentially enhance the capability and improve the performance of our Athena system. We will give a brief discussion of several such points as follows.

Resource consumption has been the core of the optimization objective of the studies reported in this dissertation—more specifically, the minimizing of the *aggregated* resource consumption across the entire networked system. It would be interesting to explore additional optimization modes besides aggregation: for example, on a battery operated sensor network, in addition to minimize the total energy draw (i.e., to minimize the sum), we might also want to prevent any single node from premature battery drain (i.e., to maximize the minimum), as a way to increase the longevity of the system's operation life, as, perhaps, a single critical node's dying could greatly affect the rest of the system.

One other aspect of Athena that can be improved upon is to have more accurate estimation of data object delivery latencies. Currently a rudimentary method is used for this estimation, as it is quite challenging to make more accurate predictions due to the distributed nature, and the complex data and label caching and sharing techniques currently employed by Athena, as ways to reduce resource consumptions. We imagine more advanced modeling techniques and/or efficient logging and bookkeeping mechanisms can possibly to developed to improve this latency estimation accuracy, which definitely will lead to more efficient planning and resource handling.

On the network side, it would also be quite interesting to explore network dynamics and mobilities, where nodes could potentially be constantly joining, leaving, and moving around during the task phase, which might not be uncommon under some specific post-disaster scenarios. Our current Athena system has been designed to operate well on top of more stable and static network topologies; it would be interesting to explore how we can specifically account for potential network dynamics and mobilities to still guarantee efficient and effective system operations.

As also mentioned in the Athena system architecture, human labels for data objects are shared and propagated back into the network, to be potentially used by future requests of the same data objects for similar decision goals. As subjective human factors are now introduced into the otherwise objective information pool, constructs like trust and/or preference relations can be explored to enable better guidance of information flows. This of course can potentially also be extended to the data objects as all sensor nodes (machine or human) are not equal, which might lead to preference differentiations for their generated data objects as well.

Throughout our study and discussion, our algorithms and system are only passively reacting to the queries issued by the human users—It would be interesting to explore what we might call *Anticipatory Information Management*, which could account for not only objective historical physical data, but also additional information that can potentially enable preemptive processing to facilitate even more intelligent task predictions and information deliveries. For example, certain tasks might be carried out under predefined work-flows, which we might be able to take advantage of to help pruning unnecessary search spaces and predict future action sets; whats more, we might even be able to tap into humans' cognitive decision process models to design more advanced information prefetching mechanisms before the questions are even asked.

Finally, we would like to close by arguing that, from a philosophical point of view, the principle as embarked upon in this dissertation can potentially be applied far beyond the sensor-networks domain, or even the confine of Computer Science. Decision-making and the trade-off balance in evidence gathering and evaluation are key for a wide span of fields, ranging from Agriculture to Finance, from Engineering to Justice and Law. We hope the work reported in this dissertation, besides its immediate utilities, can inspire and capture the imagination of colleagues and researchers from closely related communities as well as afar, and are eager for the excitements coming forth.

References

[1] Google Galaxy Nexus. `http://www.google.com/nexus`, 2012.

[2] CCNx Prototype Software. `http://www.ccnx.org`, 2013.

[3] User guide of t-drive data. `http://research.microsoft.com\newline/pubs/152883/User\_guide\_T-drive.pdf`, 2013.

[4] C. C. Aggarwal, A. Bar-Noy, and S. Shamoun. On sensor selection in linked information networks. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2011.

[5] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 219–230. ACM, 2008.

[6] A. Bar-Noy, M. P. Johnson, N. Naghibolhosseini, D. Rawitz, and S. Shamoun. The price of incorrectly aggregating coverage values in sensor selection. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2015.

[7] Y.-A. L. Borgne, S. Santini, and G. Bontempi. Adaptive model selection for time series prediction in wireless sensor networks. *Signal Processing*, 87(12):3010 – 3020, 2007.

[8] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, 1997.

[9] H. Casanova, L. Lim, Y. Robert, F. Vivien, and D. Zaidouni. Cost-optimal execution of boolean query trees with shared streams. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2014.

[10] N. Chand, R. C. Joshi, and M. Misra. Efficient cooperative caching in ad hoc networks. In *International Conference on Communication Systems Software and Middleware (COMSWARE)*, 2006.

[11] V. Coric and M. Gruteser. Crowdsensing maps of on-street parking spaces. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2013.

[12] R. Cristescu, B. Beferull-Lozano, M. Vetterli, and R. Wattenhofer. Network correlated data gathering with explicit communication: Np-completeness and algorithms. *IEEE/ACM Transactions on Networking*, 14(1):41–54, 2006.

[13] W. Dron, A. Leung, J. Hancock, M. Aguirre, Thapa, and R. Walsh. Core shim design document. In *NS-CTA Technical Report*, 2014.

[14] W. Dron, A. Leung, M. Uddin, S. Wang, T. Abdelzaher, R. Govindan, and J. Hancock. Information-maximizing caching in ad hoc networks with named data networking. In *IEEE Network Science Workshop (NSW)*, 2013.

[15] W. Dron, M. Uddin, S. Wang, T. Abdelzaher, A. Leung, A. Iyengar, R. Govindan, and J. Hancock. Caching for non-independent content: Improving information gathering in constrained networks. In *International Conference for Military Communications (MILCOM)*, 2013.

[16] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell. Bikenet: A mobile sensing system for cyclist experience mapping. *ACM Transactions on Sensor Networks (TOSN)*, 6(1):6, 2009.

[17] ElasticSearch. `https://www.elastic.co`, 2015.

[18] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[19] R. Ganti, N. Pham, H. Ahmadi, S. Nangia, and T. Abdelzaher. Greengps: A participatory sensing fuel-efficient maps application. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 151–164. ACM, 2010.

[20] Google Maps Geocoding API. `https://developers.google.com/maps/documentation/geocoding/`, 2015.

[21] Gosmore. `http://wiki.openstreetmap.org/wiki/Gosmore`, 2015.

[22] M. Gregorczyk, T. Pazurkiewicz, and K. Iwanicki. On decentralized in-network aggregation in real-world scenarios with crowd mobility. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2014.

[23] R. Greiner, R. Hayward, M. Jankowska, and M. Molloy. Finding optimal satisficing strategies for and-or trees. *Artificial Intelligence*, 170(1):19–58, 2006.

[24] S. Gu, C. Pan, H. Liu, S. Li, S. Hu, L. Su, S. Wang, D. Wang, T. Amin, R. Govindan, et al. Data extrapolation in social sensing for disaster response. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2014.

[25] H. Gupta, V. Navda, S. Das, and V. Chowdhary. Efficient gathering of correlated data in sensor networks. *ACM Transactions on Sensor Networks (ToSN)*, 4(1):4, 2008.

[26] S. Hu, S. Li, S. Yao, L. Su, R. Govindan, R. Hobbs, and T. Abdelzaher. On exploiting logical dependencies for minimizing additive cost metrics in resource-limited crowdsensing. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2015.

[27] S. Hu, H. Liu, L. Su, H. Wang, T. F. Abdelzaher, P. Hui, W. Zheng, Z. Xie, J. Stankovic, et al. Towards automatic phone-to-phone communication for vehicular networking applications. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1752–1760. IEEE, 2014.

[28] S. Hu, L. Su, H. Liu, H. Wang, and T. F. Abdelzaher. Smartroad: a crowd-sourced traffic regulator detection and identification system. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.

[29] S. Hu, S. Yao, H. Jin, Y. Zhao, Y. Hu, X. Liu, N. Naghibolhosseini, S. Li, A. Kapoor, W. Dron, et al. Data acquisition for real-time decision-making under freshness constraints. In *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[30] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden. Cartel: A distributed mobile sensor computing system. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

[31] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking named content. In *International conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[32] Y. Jiang, H. Qiu, M. McCartney, W. G. J. Halfond, F. Bai, D. Grimm, and R. Govindan. Carlog: A platform for flexible and efficient automotive sensing. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2014.

[33] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan. Mediascope: Selective on-demand media retrieval from mobile devices. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.

[34] J.-E. Kim, T. Abdelzaher, L. Sha, A. Bar-Noy, and R. Hobbs. Sporadic decision-centric data scheduling with normally-off sensors. In *IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[35] J.-E. Kim, T. Abdelzaher, L. Sha, A. Bar-Noy, R. Hobbs, and W. Dron. On maximizing quality of information for the internet of things: A real-time scheduling perspective. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016.

[36] Q. Li, Y. Li, J. Gao, B. Zhao, W. Fan, and J. Han. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *SIGMOD*, 2014.

[37] C. Liu, K. Wu, and J. Pei. An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(7):1010–1023, 2007.

[38] H. Liu, S. Hu, W. Zheng, Z. Xie, S. Wang, P. Hui, and T. Abdelzaher. Efficient 3g budget utilization in mobile participatory sensing applications. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1411–1419. IEEE, 2013.

[39] H. Liu, J. Li, Z. Xie, S. Lin, K. Whitehouse, J. A. Stankovic, and D. Siu. Automatic and robust breadcrumb system deployment for indoor firefighter applications. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[40] H. Liu, A. Srinivasan, K. Whitehouse, and J. Stankovic. Mélange: Supporting heterogeneous qos requirements in delay tolerant sensor networks. In *Networked Sensing Systems (INSS), 2010 Seventh International Conference on*, pages 93–96. IEEE, 2010.

[41] J. Liu, M. Adler, D. Towsley, and C. Zhang. On optimal communication cost for gathering correlated data through wireless sensor networks. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[42] M. G. Luby, M. Mitzenmacher, and M. A. Shokrollahi. Analysis of random processes via and-or tree evaluation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998.

[43] C. Meng, W. Jiang, Y. Li, J. Gao, L. Su, H. Ding, and Y. Cheng. Truth discovery on crowd sensing of correlated entities. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[44] C. Miao, W. Jiang, L. Su, Y. Li, S. Guo, Z. Qin, H. Xiao, J. Gao, and K. Ren. Cloud-enabled privacy-preserving truth discovery in crowd sensing systems. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[45] S. Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[46] OpenStreetMap. `http://www.openstreetmap.org`, 2015.

[47] S. Pattem, B. Krishnamachari, and R. Govindan. The impact of spatial correlation on routing with compression in wireless sensor networks. *ACM Transactions on Sensor Networks (ToSN)*, 4(4):24, 2008.

[48] A. Purohit, Z. Sun, F. Mokaya, and P. Zhang. Sensorfly: Controlled-mobile sensing platform for indoor emergency response applications. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2011.

[49] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan. Medusa: A programming framework for crowd-sensing applications. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[50] F. Saremi, O. Fatemieh, H. Ahmadi, H. Wang, T. Abdelzaher, R. Ganti, H. Liu, S. Hu, S. Li, and L. Su. Experiences with greengps — fuel-efficient navigation using participatory sensing. *IEEE Transactions on Mobile Computing (TMC)*, PP(99):1–1, 2015.

[51] L. Su, J. Gao, Y. Yang, T. F. Abdelzaher, B. Ding, and J. Han. Hierarchical aggregate classification with limited supervision for data reduction in wireless sensor networks. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[52] L. Su, S. Hu, S. Li, F. Liang, J. Gao, T. F. Abdelzaher, and J. Han. Quality of information based data selection and transmission in wireless sensor networks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 327–338, 2012.

[53] L. Su, Q. Li, S. Hu, S. Wang, J. Gao, H. Liu, T. F. Abdelzaher, J. Han, X. Liu, Y. Gao, et al. Generalized decision aggregation in distributed sensing systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[54] C. Tham and T. Luo. Quality of contributed service and market equilibrium for participatory sensing. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2013.

[55] M. Uddin, H. Wang, F. Saremi, G. Qi, T. Abdelzaher, and T. Huang. Photonet: A similarity-aware picture delivery service for situation awareness. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 317–326. IEEE, 2011.

[56] M. Y. S. Uddin, H. Wang, F. Saremi, G.-J. Qi, T. Abdelzaher, and T. Huang. Photonet: a similarity-aware picture delivery service for situation awareness. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[57] US Naval Research Lab Networks and Communication Systems Branch. Extendable mobile ad-hoc network emulator (emane). `http://www.nrl.navy.mil/itd/ncs/products/emane`, 2016.

[58] M. C. Vuran, Ö. B. Akan, and I. F. Akyildiz. Spatio-temporal correlation: theory and applications for wireless sensor networks. *Computer Networks*, 45(3):245–259, 2004.

[59] D. Wang, L. Kaplan, H. Le, and T. Abdelzaher. On truth discovery in social sensing: A maximum likelihood estimation approach. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2012.

[60] S. Wang, T. Abdelzaher, S. Gajendran, A. Herga, S. Kulkarni, S. Li, H. Liu, C. Suresh, A. Sreenath, H. Wang, et al. The information funnel: Exploiting named data for information-maximizing data collection. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2014.

[61] S. Wang, S. Hu, S. Li, H. Liu, M. Y. S. Uddin, and T. Abdelzaher. Minerva: Information-centric programming for social sensing. In *IEEE International Conference on Computing Communication and Networks (ICCCN)*, 2013.

[62] S. Wang, L. Su, S. Li, S. Hu, T. Amin, H. Wang, S. Yao, L. Kaplan, and T. Abdelzaher. Scalable social sensing of interdependent phenomena. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.

[63] S. Wang, D. Wang, L. Su, L. Kaplan, and T. F. Abdelzaher. Towards cyber-physical systems in social spaces: The data reliability challenge. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 74–85. IEEE, 2014.

[64] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 16–31. ACM, 1999.

[65] X. Xu, J. Luo, and Q. Zhang. Delay tolerant event collection in sensor networks with mobile sink. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2010.

[66] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2011.

[67] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 99–108. ACM, 2010.

[68] K. Yuen, B. Liang, and B. Li. A distributed framework for correlated data gathering in sensor networks. *IEEE Transactions on Vehicular Technology*, 57(1):578–593, 2008.

[69] C. Zhang, J. Luo, and J. Wu. A dual-sensor enabled indoor localization system with crowdsensing spot survey. In *IEEE International Conference on Distributed Computing in Sensor Networks (DCoSS)*, 2014.