VLDB Journal manuscript No.
(will be inserted by the editor)

Collaborative Joins in a Pervasive Computing Environment

Filip Perich, Anupam Joshi, Yelena Yesha, Tim Finin

Department of Computer Science & Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, Maryland 21250
e-mail: {fperic1, joshi, finin, yeyesha}@cs.umbc.edu

The date of receipt and acceptance will be inserted by the editor

Abstract Locating and obtaining context sensitive information in a mobile environment has always been a challenge. This is especially true for pervasive computing environments where in addition to limited computing and battery resources, mobile devices cannot always rely on the help of a proxy-based wired infrastructure. Rather, a collaboration among peer mobile entities is required. It allows entities to obtain data that may be otherwise inaccessible due to nature of the environment and to reduce their computation cost by reusing answers generated by other peers. We introduce the problem of data interaction among peers in ad-hoc networks and propose a collaboration query processing protocol based on the principles of Contract Nets. Once an entity identifies information it needs via the help of user's profiles, our protocol enables the device to locate data sources and obtain data matching its specific query requiring one or more data sources. We show the effectiveness of our technique through performance evaluation of entities querying for data while moving in a city-like environment.

Key words Ad-Hoc Networks, Distributed Join Processing, Peer-to-Peer Computing, Query Processing.

1 Introduction

We are becoming increasingly dependent on computers in our everyday lives. We have become accustomed to computers connected to the Internet to provide us with information at any time we want. Naturally, we expect the same ability while on the move. We would ideally like our mobile devices to provide us with the same information as their stationary counterparts, but tailored to our current context. By context, we mean a collection of information that can be used to characterize the present situation of users and their devices [16]. Such information often describes location, time, identity and the current activity.

Locating and obtaining context sensitive information in mobile environments raises two principal challenges [18, 28] - *what* and *how*? To answer the first issue, a mobile device must be able to determine what information is needed in order to satisfy user's needs. Ongoing efforts, including our own results reported elsewhere, are starting to provide good mechanisms to handle this question[28,30,12]. The second issue deals with the mechanisms a mobile device must have in order to obtain required data. Additionally, the solutions to these challenges must still account for the limited battery, computation, connectivity and storage resources of mobile devices. In this paper, we provide an answer to the second - *how* - issue.

Two very distinct approaches have been reported in the literature to answer the *how* question. One type relied on off-line pre-caching for supporting disconnected operation, pioneered in the wireless context by the Coda filesystem [24], and more recently by commercial software such as AvantGo [3]. At the other end of the spectrum were solutions exploiting traditional wireless connectivity, such as those provided by cellular networks or WLANs [19].

These solutions treat mobile devices as clients connecting to a proxy-based wired infrastructure [12,13,36]. Both approaches have their advantages and disadvantages. While precaching techniques preserve battery power, they are unable to provide answers outside the scope of their caches and are prone to stale data. The latter approach clearly provides better results; however, it does so by depleting battery and wireless resources and often by incurring monetary costs for cellular connectivity. Additionally, this approach is prone to frequent disconnections due to the environment and mobility patterns of the device and to a possible single point of failure at the server side.

The realization of ad-hoc wireless network technologies creates an interesting variation of the problem. It allows spontaneous connectivity among mobile devices. Here, mobile devices including hand helds, wearables, computers in vehicles, computers embedded in the physical infrastructure, and (nano)sensors, are equipped with short range ad-hoc networking technologies such as Bluetooth [6]. There is no longer a guarantee of infrastructure support, a crucial requirement for traditional mobile solutions. Consequently, for obtaining data, devices can not depend on the help of some centralized *big brother* server. At the same time, each device is not required to *cache it all*. This is because each device can always try to connect to others in its vicinity. Thus, mobile devices are no longer standalone entities nor simple clients. They became peers that can interchangeably function both as information consumers and providers in order to share information among themselves and to provide utility to their users [28,30]. For example, a car does not need to connect to a yellow page service or store an entire phone book directory in order to locate a gas station. It should be able to find the gas station by interacting with other objects and an electronic sign of a gas station in its vicinity.

The key aspect of this environment is the need for *collaboration* among peers. When a device in the pervasive computing scenario requires information, it can only query its immediate neighbors and other devices accessible in its vicinity. It is thus only through collaboration that a device can obtain data. Additionally, collaboration may help mobile devices to overcome their limited resources by using computed results from previous queries held by peers. This is especially true when performing joins over multiple data sources on resource limited mobile devices. A join processing is required whenever a single device cannot provide an answer to a query. Rather the device may have to *horizontally* join data streams from multiple devices holding the same type of data. More importantly, frequently the device may have to *vertically* join data streams from different devices announcing their presence.

Existing solutions for traditional mobile systems are inapplicable [18,28]. This is due to the dynamic nature of ad hoc wireless networks. The additional mobility of information providers, and not just information consumers, limits data and data source availability. More importantly, not all data sources may be concurrently available. Traditional query processing techniques, especially those performing joins, will often fail in this environment. The nature of the environment also limits the duration of any connection between two devices as well as the possibility of a reconnection with a given device to which we may have connected in the past. Lastly, the ad hoc nature of the environment can cause interaction between any two devices disregarding the heterogeneous types of data they may understand or provide [30]. Thus, the environment requires different kinds of solutions than the traditional mobile client-server model.

The networking aspect of a collaboration in pervasive computing environments has been addressed in terms of device discovery and routing protocols [2,6,22,31,32]. To the best of our knowledge very little work has been done on the part of data collaboration/sharing in the pervasive computing environments, especially in the area of processing joins over multiple data sources. Although the research on sensor networks is addressing somewhat similar questions, as we argue in Section 2 the domains differ significantly.

In this paper, we address the problem of *how* mobile devices in pervasive computing environments can locate and provide answers to user's needs with the emphasis on collaboration and especially on providing the means for performing joins in pervasive computing environments. We utilize our previous work on *what* information a user seeks based on her current context. We take a more expansive view of context and include user beliefs, desires and intentions stored in a user profile [30]. This allows a device to determine a priori what information a user will need and dynamically adapt its strategies for querying its peers and for collaborating in join processing. While our implementation builds on our specific previous work, the solution we propose is relatively agnostic – any other technique[12] could be used to provide the *what* as well.

The paper builds on our previous work in defining a profile–driven data management framework for pervasive environments where we were concerned with data representation and discovery and with profile-based data caching [28–30]. In this paper, we make the following new contributions:

- We propose a collaboration protocol based on the principles of Contract Nets [1,33]. Our Collaborative Query Processing protocol (CQP) enables a device in a pervasive computing environment to locate data sources and obtain data matching any query, regardless of whether the query is a selection or a join. The protocol extends the traditional concept of nested loop joins. It also extends our previous work on selection queries in pervasive computing environments [28]. CQP allows two or more devices to cooperate by executing any combination of select–project–join queries. The protocol accomplishes the task by subdividing queries and assigning subparts to other devices. The assignment depends on the available resources of each device. For example, CQP allows a tourist to use her handheld device to ask for the closest cheapest laundromat that is open, given her current location, time of the day and a price range. The protocol also allows the tourist to ask for the closest laundromat adjacent to a Chinese restaurant – a query requiring a join over two data *streams*.
- We design and implement a realistic experimental model for simulating a city–traffic scenario of people traveling around lower Manhattan. The model is implemented on top of MoGATU [28], which is a robust framework for profile–driven data management in pervasive computing environments. The model represents handheld/embedded computers and intersection beacons as semi–autonomous entities guided in their interactions by profiles and context [29]. The primary objective of the model is to enable each mobile device to utilize as much available information as possible in order to enhance the mobile device's functionality. It uses both static information, such as user's preferences and information about data sources, as well as dynamic information, such as the current context description. This allows each device, to gather, provide and possibly create data that will be helpful to its user in near future. Each device can randomly disseminate some of its data as well as queries; however, when answering a user's queries each mobile device relies on its local content only.
- We demonstrate the capability of CQP by implementing it in the MoGATU framework and by evaluating its performance in the city–traffic experimental model. We show how the protocol improves average success rate of satisfying user queries for each entity, and how it affects the combined computing cost and network traffic for all entities in the environment.

The remainder of the paper is structured as follows: We present related work and argue why traditional mobile solutions are not applicable in Section 2. In Section 3, we provide an overview of the MoGATU framework and define the CQP protocol. In Section 4, we present our experimental model and setup, and show the effectiveness of our technique through performance evaluation. We conclude and describe directions for future work in Section 5.

2 Related Work

The problem of data management in wireless networks has drawn a significant degree of attention. The proposed solutions primarily address problems imposed by the underlying networking technology, such as low bandwidth and high probability of disconnection. They also address the issues related to the retrieval of location dependent information. Existing solutions often rely on the support of a fixed, wired infrastructure. These solutions place primary data on servers located within the wired infrastructure and treat mobile devices solely as clients. Using this approach, all joins are executed by servers and mobile devices mostly receive updates to their materialized views. Chrysanthis *et al* [26] consider disconnected operations within mobile databases by presenting a mechanism, referred to as a "view holder", which maintains versions of views required by a particular mobile unit. Kottkamp and Zukunft [25] present optimization techniques for query processing in mobile database systems that include location information. They present a cost model and different strategies for query optimization incorporating mobility specific factors like energy and connectivity. Demers *et al* [15] present the Bayou architecture, which is a platform of replicated, highly available, variable-consistency, mobile databases for building collaborative applications.

In contrast to these approaches, our work assumes no support from the fixed infrastructure. When a mobile device requires instantaneous information (e.g. traffic updates), it may be more easily, or only accessible, from other "local" mobile devices and not from a fixed node. A mobile device in our work is always in nomadic mode, as defined in [9]. This attribute and other characteristics of ad hoc wireless networks lead to a significantly different environment by imposing the following additional constraints on mobile data management solutions:

Information consumers are not the only mobile devices. Information sources – other peers – can change their locations as well. This limits data and data source availability. A device may obtain different answer to the same query depending on its specific location and time based on what other devices are reachable.

More importantly, not all data sources may be concurrently available. Concurrent availability drastically affects an execution of joins over multiple data streams. While in traditional approaches all sources to the streams must be available at the same time, this may not be the case here. A device performing a join must, therefore, be able to *intelligently* pre–cache relevant parts of the data which are currently accessible, while searching for the sources of the other data. Note that we use the term stream here to capture a combination of both traditional stored data sources, as well as the newer sensor based streaming data sources.

The nature of the environment also challenges the duration of any connection between two devices and limits the possibility of a reconnection. Since each device moves independently of others, there is no guarantee that two mobile devices currently connected will ever again be able to communicate between themselves. This may cause inconsistencies.

The environment also permits interaction between any two devices disregarding the heterogeneous types of data they may understand or provide. If these devices *talk* using incompatible ontologies/schemata, they will fail to answer any query that the other device may ask. Consequently, every mobile device in the pervasive computing environment must become more autonomous and adaptive in order to answer queries, since it can mostly depend on itself and its local resources only [28, 30]. The mobile device must be able to predict the types of information it may need in the future and collaborate with other devices in its vicinity while locating and obtaining data on its behalf and on the behalf of others. Essentially, these constraints render solutions for traditional mobile networks unusable.

Recently, significant research interests have been directed toward the area of sensor networks, particularly on querying continuous data streams. The Cougar [7] and Fjords [27] projects represent two such architectures. Additionally, Chandrasekaran *et al* [11] present PSoup, a system for processing continuous queries over continuous data. PSoup extends the work on Eddies [4] – a query processing mechanism, which is part of the Telegraph project [20]. Eddy enables dynamic query reordering by routing tuples between multiple operators represented as independent threads.

Although the issues of query processing in sensor networks appear somewhat similar, the environment and the objectives differ significantly. The goal of a sensor network is to collect, and optionally aggregate, fixed types of data at every node, and propagate the result to the collecting base station. Once a sensor network is stabilized, each sensor knows its peers. Each sensors simply accepts data from its peers, may aggregate the data and sends data up the stream to its parent device or a collecting base station. It is the collecting base station that uses the sensed information. Each sensor node represents a simple "slave" proxy or a provider but never a consumer. On the other hand, the pervasive computing environment is dynamic and may never stabilize. Every mobile device in our environment is consumer and interacts with others to obtain data primarily for itself. Each mobile device then only optionally becomes a proxy or a provider for some other device in its vicinity. Hence, a mobile device is always adapting the type of data it needs and the operations it must perform for itself and possibly for other devices. Consequently, data management in sensor networks and ad-hoc peer-to-peer networks require different kinds of solutions.

3 Collaborative Query Processing

The CQP protocol allows mobile devices in the pervasive computing environment to help other devices in locating and obtaining answers for queries over one or more data sources. The protocol allows the initiating mobile device to view other devices as additional sources of raw data or pre-computed answers. By reusing data made available by others, the system as a whole is able to provide data to a larger number of mobile nodes while still preserving system-wide resources. These resources, which are always limited, include battery and computing overhead. Before presenting a detailed design of the protocol, we present a brief overview of the MoGATU framework for completeness.

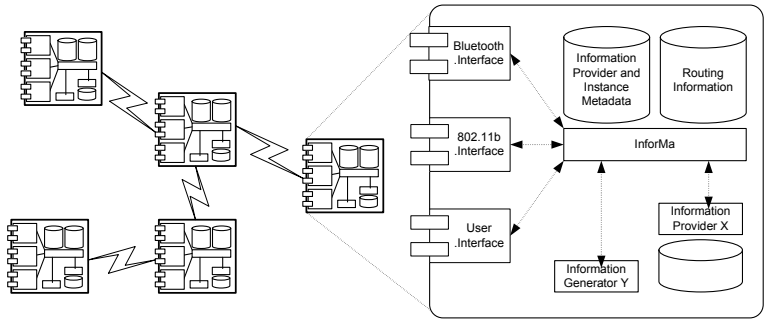


Fig. 1 MoGATU Entity and Interaction Model

3.1 MoGATU System Overview

MoGATU is a framework for handling serendipitous querying and data management in pervasive computing environments [28, 30]. The framework treats all devices in the environment as equal semi-autonomous peers. To provide uniform communications functionality, and to handle data management issues, the framework abstracts all devices in the environment in terms of Information Managers, Information Providers and Information Consumers. It also implements several communication interfaces, currently with support for Bluetooth and Ad–Hoc 802.11. Figure 1 depicts an overview of the framework, and the integration among various devices and their resources. Each device is only required to implement an Information Manager and at least one communication interface.

Information Providers represent the data sources available in pervasive environment. Every Information Provider holds a partial set of heterogeneous data, a *fragment*, available in the whole environment and is annotated in a semantic language. Given the ad-hoc nature of the environment, it may be impossible to maintain a global consistency among all Information Providers because the network remains mostly partitioned. As a result mobile nodes attempt to be at vicinity-consistent only.

Information Consumers represent entities that can query and update the data present in the environment. In the current design, the Information Consumers primarily represent human users that ask their mobile devices for context-sensitive information.

Lastly, an instance of an **Information Manager** (InforMa) must be present on every device. Information Managers are responsible for the underlying network communication and for most of the data management functions. Each Information Manager is responsible for maintaining information about peers in its vicinity. This information includes the types of devices and what types of information they can provide. Information Manager also maintains a data cache for storing instances of information obtained from other mobile devices as well as the information provided by its local Providers. Each Information Provider stores the data instance in their base forms. The Provider decomposes data instances that are defined using multiple ontologies of their respective base representations and maintains a view represented as a list of pointers to the respective base fragments. Additionally, each Information Manager may include a user profile reflecting some of the user's beliefs, desires, and intentions, a model which has been explored in multi-agent interactions [8]. This model significantly extends the work of [12] on profiles, which explicitly enumerates data and its utility. In contrast, using the BDI concept, our profiles adapt to the environment by varying both data and their utility over time and current context [29]. The Information Manager uses the profile to adapt its caching strategies [30] and to initiate a collaboration with peers in order to obtain desired information.

3.2 Data Representation

As mentioned in Section 3.1, every mobile device can hold a subset of the globally available *heterogeneous* data. Since the pervasive environment is by definition an open system, there are no restrictions or rules on what data and how the data are present. To narrow this vast space of possibilities, we have previously argued for every instance of any data in the environment to be annotated in an ontology / schema [28]. In MoGATU, ontologies are defined using the DARPA Agent Mark-up Language (DAML) [21]. This semantically rich language allows the specification

of numerous types of data and also defines relationship among classes and their properties. For example, we define ontologies for weather and traffic updates [29].

As a simple example, one can imagine each ontology to represent one tabular relational schema. For example, one can imagine all gas stations stored in one table, although the table may be horizontally fragmented over many devices in the environment. This, however, drastically reduces the power of ontologies by removing any relationship information between data instances, such as inheritance or cardinality constraints. For example, this way one cannot automatically deduce that a Chinese restaurant is an Asian cuisine restaurant without an explicit attribute stating so. Nevertheless, this way each instance of data can be perceived as a tuple that uses data from one or more ontologies / schemata.

In this paper, we concentrate on query processing only and do not take into account the time necessary for reasoning over the ontology knowledge, *e.g.* whether a Chris's Gyro is a Greek restaurant. Additionally, each base data instance is grounded in only one ontology and includes a globally unique identifier – a property inherited from DAML. Data instances representing combined information from multiple ontologies, such as a result of a join, also carry a locally unique identifier for the resulting "tuple" and the original global identifiers for the data instances used to produce this combined tuple. Each Information Manager assigns a local timestamp to each new data instance before placing it into the cache. This can be used to calculate *recency* for each cached data during a query execution. Additionally, each Information Provider assigns a *lifetime* to every object it produces. The lifetime represents a time period after which the data should be invalidated and purged from cache.

3.3 Query Representation

As introduced in Section 1, one important challenge is to determine *what* data a mobile device should obtain in order to satisfy user's needs. A device can either wait for a user to ask an explicit query and then attempt to obtain the answer by contacting its peers in the vicinity. Alternatively, the device can attempt to proactively obtain data before a user asks for it. We argue that proactive data querying is a preferred solution as it allows the device to cache data while available in the hope of future use [28,30]. We have also argued that the use a profile is vital in order to allow the device to reason about its user's needs. Consequently, MoGATU distinguishes between two types of queries: (i) explicit queries and (ii) implicit queries generated from user's profiles.

We are concerned with *how* these queries can be satisfied, especially when they required multiple data streams, *i.e.* they require that a join be done over data available in the current vicinity. We use explicit queries to measure the overall performance of the framework. Explicit queries are asked by users and answered by their respective mobile devices. Devices attempt to answer these queries using their local resources only. Implicit queries, on the other hand, are *inferred* by each mobile device from a user's profile. These queries are executed in order to provide data for future explicit queries. The implicit queries are excluded from the overall success performance value.

Like the data they operate over, queries are also defined using a DAML ontology, specifically DAML-S [14], which is a standard evolving in the Semantic Web community; however, for the purposes of this paper, we abstract the query representation using a familiar *select–from–where* form. Figure 2 shows the abstract query specification form. More formally, a query represents a tuple defining a set of ontologies in use (O), selection list (σ), filtering statement (θ), and cardinality (Σ) and temporal (τ) constrains:

$$query = (O, \sigma, \theta, \Sigma, \tau) \tag{1}$$

Each query defines ontologies / schemata which are used in constructing the filtering clause and for final projection of matching data instances. The cardinality of the ontology list also specifies the degree of the query. The cardinality represents the number of joins that must be performed for obtaining an answer. The filtering clause represents a combination of boolean conjunctive and disjunctive predicates. A devices uses its cached data as well as its current geographical position and time of the day, if available, as inputs to these predicates. This allows a mobile device to place a *dynamic* query asking for the closest local gas station. It also allows a device pose a *static* query asking for a Chinese restaurant located on the W 72nd Street. Along with string and numerical comparisons, the filtering clause supports basic calculations, such as addition and multiplication. Additionally, the filtering clause supports more advanced predicates based on the ontology specification, such as a distance computation between two

```
SELECT (select_list)
FROM (ontology_list)
WHERE (conjunct_disjunct_predicate_list)
LIMIT [minCardinality, maxCardinality]
TIME neededBy
```

Fig. 2 Abstract Query Specification

geographical objects. The query also specifies the cardinality constraints on the answer and the time deadline by which a query execution should complete or terminate.

3.3.1 Explicit Query A user, or some other Information Consumer, can explicitly submit a query to the mobile device. In return the mobile device, represented by the Information Manager, attempts to answer the query by examining its local resources only. It evaluates the filtering statement θ, over any combination of data instances that are present in local fragments. These data instances must be defined using a subset of ontologies from the ontology list O. The Information Manager continues examining the data until the size of the answer is within the cardinality constraints Σ or the allowed time, τ, has expired. For selection queries over data from a single source, the Information Manager directly scans and evaluates data instances stored in one local fragment. For queries requiring joins over data from multiple sources, the Information Manager first checks whether a similar query was previously executed and if its result could be used. If no such *result view* is found or the cardinality constraint has not been satisfied, the Information Manager performs the necessary join operations over its local fragments and returns the final result.

3.3.2 Implicit Query via User Profile The Information Manager on each device may include a user profile. The profile specifies a subset of beliefs and desires that the mobile device should know [30]. The knowledge contained in the profile is initially defined using DAML. Upon loading the profile, the knowledge is converted into implicit queries including constraints describing time,location ranges, and the probability of the query being asked by the user. For example, a profile may contain preferences stating that the user likes to eat lunch between noon and 2pm. Additionally, the profile may state that the user prefers to eat at Chinese restaurants. Therefore, the Information Manager combines these two beliefs into an implicit query, which attempts to collect and maintain the location of at least one Chinese restaurant in the vicinity from 11:45am until 2pm. The details of *reasoning* over profiles to generate queries is beyond the scope of this paper, and we refer readers to [30]. Each desire and belief can be represented as a following tuple:

$$belief/desire = (O, \sigma, \theta, \Sigma, \lambda, \tau, P) \tag{2}$$

O, σ and θ define the set of ontologies in use, the selection list and the filtering statement, respectively. Σ represents the minimal cardinality of the answer that the query should return when evaluated locally. λ and τ represent the location and time when the implicit query should be actively evaluated. Finally, P represents the probability between [0..1] that the user will, in fact, ask a question that uses the data obtained by executing the implicit query. Therefore, the implicit queries are used for proactive caching of remote data instances using a semantics based approach [30] that are believed to be useful in answering future explicit queries. As stated earlier, we use explicit queries to evaluate the performance of our approach so as not to reward the system for proactively cached data which does not get used.

3.4 Collaboration Protocol

We now present the detailed design of the CQP protocol. CQP is based on the principles of Contract Nets [1,33]. Any device in the environment can initiate the protocol. Other devices in the vicinity of the originating device can, at their will, collaborate by helping in answering queries over one or more data streams. In this paper, we are not interested in studying incentive-driven mechanisms and under what conditions a device or its user is willing to allow

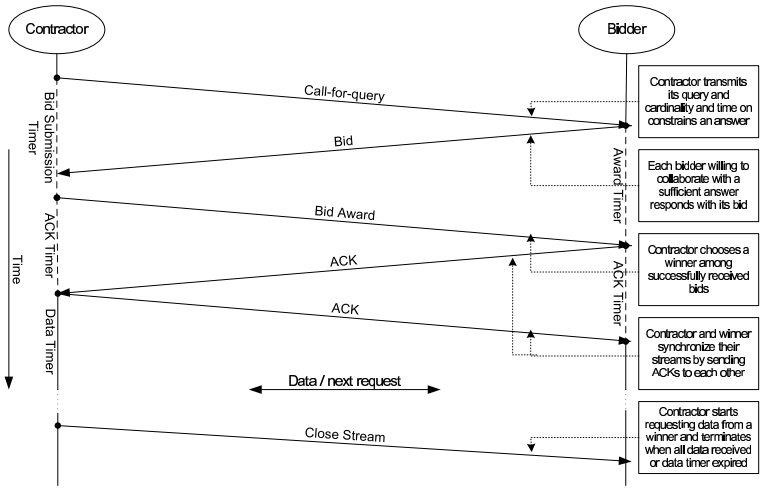


Fig. 3 Message and state flow in the Collaborative Query Processing protocol

the device to participate. We only assume that each device has a certain degree of *willingness to help* - a probability that a device responds to a query. The message and state flow of the protocol is shown in Figure 3.

We designed CQP while keeping in mind the underlying environmental characteristics. CQP attempts to address issues caused by resource limitations as well as different mobility patterns of a device and its dynamic vicinity. In order to overcome the frequent disconnections and low bandwidth, we expect each device to process as much data locally as possible before sending results back to its peers. We do so by decomposing DAML-annotated queries into selection queries over base ontologies and their possible combinations. Additionally, to overcome the network limitations we employ a Contract Net based negotiations among peers to determine which device will provide what data. For this we build upon our previous experience in service and data discovery as well as our previous work on MoGATU [2,10,28,30]. We employ both gossiping and pull-based techniques for advertising data and for querying peers. Finally, whenever possible we attempt to utilize the underlying routing protocol for routing packets between mobile peers; however, in order to overcome the uncertainty of the ad hoc environment, CQP does not need to obtain all data before it terminates as detailed in Section 3.4.6.

The resulting CQP protocol consists of a contract agreement phase, used by a device to discover peers with required data and for peers to agree on their tasks and desired outcome, the streaming phase, where peers process data locally and send results to their peers, and a termination phase, where peers independently finalize their query processing operations. We now detail each phase and their steps.

3.4.1 Call for Query When a mobile device concludes it needs to satisfy some of the beliefs and desires stored in the user's profile, the device converts the requirement into an implicit query [30]. Initially, the mobile device attempts to satisfy the implicit query by using only its local cache. When the device is unable to reach the desired cardinality for the answer, the device determines one or more sub-queries and asks other devices in its vicinity to answer them. The mobile devices constructs a *call–for–query* message. The message contains the sub-query, cardinality requirements and the time deadline by when the complete answer should be delivered. The message also includes the time when a winner will be announced. The mobile device sends the message to its current peers and starts its *bid–submission* timer. The mobile device is now a *contractor*.

As depicted in Figure 4, the mobile device may require help from one or more devices in its vicinity. For queries involving only one data stream, the mobile device needs to ask one peer at a time to scan its local fragment matching the particular query. For queries involving two data streams, the mobile device may again simply ask one peer to perform the entire query over its local fragments and return the results; however, the mobile device may decide to ask the peer for only one stream and join the output with its local stream. To the extreme, the contractor may even ask one peer to scan one stream, another peer to scan the second stream, and a third peer to perform the join using data from the other two peers as inputs. The CQP protocol is sufficiently abstract to allow any of these approaches.

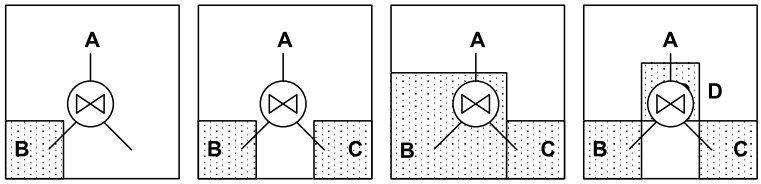


Fig. 4 Possible Query Executions over Two Streams

Unlike traditional join techniques, the CQP protocol does not require all input streams to be concurrently available. A mobile device can use the CQP protocol to cache only one input stream which is currently available. The device can then delay the join until the other stream becomes available in time and space. Therefore, a mobile device is able to better utilize the serendipitous nature of the environment. Additionally, the protocol enables chaining of multiple collaborations, allowing devices to collaboratively execute a query of almost any complexity.

3.4.2 Bid Submission Upon receipt of a *call–for–query*, a mobile device *infers* whether it should interact in the proposed collaboration. A mobile device wishing to collaborate, calculates the cardinality of an answer that it can provide for the contractor in the allowed time. If for some reason the mobile device does not believe it can satisfy the proposal, the device simply ignores the *call–for–query*. On the other hand, when the mobile device determines that it can provide a valid answer, the device returns a *bid* estimating the cardinality and time epoch that it will take to compute its answer. The responding mobile device then starts a timer awaiting a *bid–award* message and its role becomes that of a *bidder*.

3.4.3 Bid Award When a contractor's timer for bid submission expires, the contractor evaluates all successfully received bids. For multiple bids, the contractor chooses the bidder that promises to deliver the most data in the shortest possible time. The contractor creates a *bid–award* message, starts an *ack* timer, and sends the message to the bidding peer. The devices attempts to contact the peer at most n-times before discarding the current session.

3.4.4 Acknowledgment Once the bidder submits its initial response (*bid*), it awaits a *bid–award* message. If the bidder does not receive the message before its timer expires, the bidder resends its *bid* message at most n-times more before discarding the current session. If the bidder does not receive the *bid–award* message after n trials, the bidder assumes someone else was awarded the bid and discards the current session. On the other hand, when the bidder receives the message on time, the bidder, now a *winner*, sends back an *ack* message. The winner starts its *sync* timer and awaits for *ack* from the contractor. Similarly, the contractor waits at most n-times the period of the *sync* timer for receiving *ack* message from the winner before again discarding the current session.

3.4.5 Streaming Data Once the winner and contractor receive *ack* messages from each other, the winner is ready to start sending an answer in terms of data instances and the contractor is ready to accept them. The contractor continuously asks for the next data instance until it either receives *end–of–stream* or *stream–error* message or until the *stream* deadline timer expires.

The contractor is effectively treating other peers as data streams. These streams can consist of one data type but also of a combined stream whenever the source peer is sending a result of a join operation. In order to preserve the limited bandwidth, the receiving peer can further send an additional selection query to further filter the results calculated by the source peer. This way the source peer may is instructed to process its outgoing stream before actually send it out as to avoid sending out unnecessary data objects that will be dropped by the receiving device.

3.4.6 Termination As hinted above, the CQP protocol can terminate based on two counts. First, the protocol terminates when the winner has sent its entire answer, as defined by its maximal cardinality in the agreement, and the contractor has successfully received it. Otherwise, the protocol can terminate when a timer expires. A timer expires when the original agreed deadline passes. A timer can also expire when a partition occurs in the ad-hoc network due to the mobility of either the contractor, the winner or any other routing mobile device. Therefore, both contractor and winner terminate the session either once a timer has expired or after a predefined period of inactivity.

3.4.7 Error Handling In order to overcome possible errors caused by network partitions and resource limitations, our protocol operates on a best effort basis only. When a device determines it needs to obtain additional data, it initiates the CQP protocol and interacts with the winner. If the interactions fails before the contract is satisfied, the initiating device can attempt to find other source after a parametrized period of time. Additionally, we employ a caching policy for storing local results as well as data received from peers. Finally, we have chosen the initial contract agreement interaction for two reasons. First, the device is able to collect a catalog its vicinity based on the given query and select the most applicable peer for interaction. Secondly, by requiring a longer initial interaction we can overcome situations when a device is in range only for the agreement and disappears as soon as agreement is reached but execution fails.

3.4.8 Join Query over Two Streams In this environment, we have to differentiate between two kinds of join execution. One join execution involves only one device operating over two local streams and thus it can employ any join algorithm it desires. The other kind of join involves one or more network streams as shown in Figure 4. These streams represent data sent to the processing device by its peers. For this type of join processing, the device may be unable to use any advanced hashing or sorting techniques as it cannot influence the stream order or size. Therefore, for our implementation we have chosen to use the traditional nested-loop join processing algorithm. If a device is able to cache the entire network stream locally, then the device can again use any algorithm it desires. Although, speed is an important factor in traditional query processing, we do not envision it to be as important in ad hoc environments since the complexity and amount of data should be relatively smaller than for traditional database problems.

3.4.9 Example of Join Execution Among Peers To illustrate how the protocol works, let us consider a mobile device *A* with query joining two streams α and β, *e.g.* restaurants and gas stations. We should note that the device can only collect data currently available in its vicinity. As a result, the device is never guaranteed that it will ever hold a complete result to its query. Instead, the device tries to collect as much data as available. The device, therefore, attempts to first contact someone who can completely satisfy the query. If that fails the device can then ask for each data stream separately and perform the join itself or with help of another peer. More importantly, the device can ask for one stream and delay the query for the second stream for a fixed period of time. This is especially important in pervasive computing environments where the concurrent availability of all required sources is not guaranteed. For example, let us consider the presence of two other devices: First device, *B*, contains data for stream α and is available now. The second device, *C*, contains data for the stream β and will become available in 10 minutes. Using the CQP protocol, device *A* asks device *B* first to execute the complete query, and then to execute a sub–query over stream α only. Device *A* caches the result and periodically checks for a presence of someone holding stream β. Once device *C* becomes available, it is asked to collaborate in performing another select sub–query over its stream. Device *A* then uses the incoming result to immediately join the two streams α and β. This would be impossible using traditional techniques because those return either a valid answer or no data at all.

4 Performance Experiments

In this section, we show the improvement of the system performance of the MoGATU framework when using the CQP protocol. To do so, we have converted the concept of the MoGATU framework to function as a part of the GloMoSim simulator [35]. GloMoSim is a scalable simulation environment for wireless and wired networks. It is designed using a discrete event simulation capability provided by Parsec. We have implemented the MoGATU framework at the application layer with support for all features defined in the original architecture [28]. Each device includes the Information Manager, Information Providers and one Information Consumer representing a user. The Information Manager can advertise and solicit information about other mobile devices in its vicinity applying traditional epidemic/gossiping techniques used for mobile environments. It also can advertise and solicit Information Providers as well as send out *bulk* data. Most importantly, each Information Manager uses the CQP protocol for query execution. This new version of MoGATU derives implicit queries and adjust its caching strategies via profiles. We used Intel Pentium 4 1.4GHz desktops with 256MB RAM each, running Linux 2.4.18-5. The GloMoSim and the MoGATU extension are written in C.

The goal of our environment is to study the performance of the CQP protocol in ad hoc networks. For that we have decided to employ an experimental spatio-temporal environment representing a city scenario, closely resembling a four-hour period in lower Manhattan in New York City. Entities in our environment represent moving objects such as pedestrians' handhelds or devices embedded in bicycles or cars. Entities also represent stationary objects such as store beacons or intersection lights.

For our experimental environment we chose 893 nodes – 793 stationary intersection beacons and 100 users, described below. We use 793 intersection beacons as the number is equivalent to the number of intersections in the lower Manhattan model. We use these beacons as sources for information about their locations, *e.g.*, each intersection beacon has some knowledge of its vicinity. The users in the environment move around and optionally query the intersection beacons but also other users in their current vicinity. We put a restriction that only users can execute joins while beacons can provide answers to only one stream query at a time. This is to make the users work "harder" for their information.

Users in our model can represent pedestrians, people traveling using public transport and people driving in cars. For our experiments reported here, we choose the parameters of our environment (speed of movement and direction of movement) to mimic vehicular traffic, cars and their drivers. Given that this represents the higher end of the speed parameter and thus leads to frequent changes in the "vicinity" of a user, it represents the worst case scenario for our approach. This is somewhat mitigated however by the fact that a car in a city drives in a more predictable and organized fashion (along roads) than a device allowed to walk completely randomly (direction– and speed–wise) through a two-dimensional space. The use of a city traffic environment thus enables us to better examine the CQP protocol performance in realistic scenarios, and our particular choice of parameters will stress the protocol.

Within the environment, we measure the system performance in terms of the average success rate of answering explicit queries per user. We measure the average computing cost each user's device spends answering its explicit and implicit queries. We then measure and compare it to the average computing cost each device spends answering queries posed by other devices in its vicinity. Finally, we measure the network traffic incurred by each device for answering its queries and the traffic due to answering queries for its peers. For our experiments, we either vary or measure the following parameters:

To avoid confusion, in the remaining part of the paper we refer to user's devices simply as *cars* and to other devices as *beacons*. Again we would like to emphasize the fact that user's devices are not required to be just cars, they can be bicycles or handheld devices carried by pedestrians.

- *Query Success Rate* – The Average Query Success Rate represents a fraction of explicit queries that each device is able to answer using its local fragments. Initially, a device of every user has an empty cache. The device can obtain data by either accepting *bulk* advertisements from intersection beacons or by executing implicit queries that were derived from a profile.
- *Profile Accuracy* – A Profile Accuracy measure represents how closely a user profile resembles the real actions of the particular user. Completely accurate profile knows the precise explicit queries as well as the location and time the query is going to be asked. At the same time, a completely accurate profile does not imply a 100% query success rate. A car knowing a complete profile may still be unable to obtain the required data due to its limited resources and to the nature of the environment. For example, a car driving in New York City will most likely be unable to answer queries involving the current weather conditions in Washington, D.C. 0% accurate profile has no knowledge while the profiles of accuracy from 1% to 99% are synthetically constructed as permutation of the complete profile to on average be able to answer that many explicit queries. For example, 50% accurate profile should on average be able to answer 50% of explicit queries.
- *Willingness to Help* – This value represents the probability of a car responding to a *call–for–query* given that it can provide a valid answer matching the needs of the requested query. Willingness level set to 0% implies no help from other cars in the vicinity, while willingness set to 75% implies that car is willing to help three times out of four whenever it can.
- *Computing Cost* – Computing Cost represents the average amount of energy used by a user's device for performing operations while pursuing either its goals or goals of its peers. We create an abstract function converting each operation a car can perform to a value depending on the complexity of the operation and the time it takes to complete the operation. We sum the values of all operations performed by each car during the simulation into a single

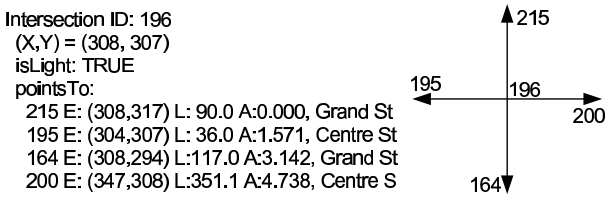


Fig. 5 Intersection Representation

scalar value. One can see this abstract value as an average number of instructions or the energy consumption per car.

– *Network Traffic Cost* – Network Traffic Cost represents the average number of packets sent and received by each car for pursuing its goals or goals of other peers. We count the number and size of packets each car sends and receives when attempting to satisfy its implicit queries. We also count the number and size of packets each car sends and receives when helping others.

We use the results to these metrics to measure the effectiveness of the CQP protocol by comparing and ultimately studying the trade-offs between:

1. Query Success Rate vs. Profile Accuracy 4.2.1
2. Computing Cost vs. Profile Accuracy 4.2.2
3. Query Success Rate vs. Willingness to Help 4.2.3
4. Computing & Network Cost vs. Willingness to Help 4.2.4
5. Query Success Rate / Computing Cost vs. Willingness to Help 4.2.5

4.1 Experimental Setup

4.1.1 Environment As mentioned above, we have created a realistic model mapping streets and their intersections south of the 72nd Street in Manhattan. We have calculated intersections between two or more streets by creating vertexes in a directed graph. Our resulting model is represented as graph with 793 intersections (vertexes). Each intersection is assigned an (x,y) coordinate. Additionally, each intersection is assigned a list of its neighboring intersections, *i.e.* endpoints of their edges. This allows us to obtain an angular direction and distance to the next intersection. Figure 5 exemplifies what information each intersection has and how it relates to the map.

4.1.2 Beacon Entity For our simulation scenario, we assigned a stationary beacon to each intersection for a total of 793 beacons. Each beacon implements an instance of MoGATU and is able to transmit up to a distance of 125m with a maximum throughput of 2Mbps. A beacon represents a computationally limited resource; however, it is able to provide a base of location-dependent data and has a vast supply of energy. For example, a beacon at the 52nd Street and 5th Avenue may contain information about restaurants, movie theaters and the latest traffic conditions within 200 meters. Although, each beacon holds a large amount of data and is willing to share it with others, it does not have the capability to perform joins over multiple streams. In other words, each beacon provides only scanning capabilities over its local resources.

4.1.3 Car Entity In the experiment, we use 100 cars, which represent the other type of devices in our environment. Cars can also transmit up to a distance of 125m and have a maximum throughput of 2Mbps. Each car can move freely according to the mobility model, which will be described below. Initially, every car is assigned a profile, which represents the beliefs and desires of its driver with a certain degree of accuracy. No car has any other data when the simulation starts. Therefore, to obtain data a car must either be close to another device in the environment, *i.e.* pass a beacon, when the other device is broadcasting *spam* data. Alternatively, a car must be able to infer an implicit query based on the profile. The car can then use the implicit query to interact with other devices in its vicinity, regardless of whether those are cars or beacons. Unlike beacons, a car represents a computationally powerful device but with a limited amount of data. When a car is willing to collaborate, it can help in processing a query over single and multiple streams.

4.1.4 Mobility Model To allow cars to move in a more organized manner than the random-way-point model provides, we have extended the work on the smooth random-way-point mobility model [5]. That authors of that model defined how a moving object can accelerate, decelerate and move in random-way-point pattern. In our model, each car drives from one intersection to another. Our model also accounts for acceleration and deceleration of an object; however, our object does not have to come to a full stop in order to change its direction. Additionally, we account for all traffic lights in the environment.

When a car starts, it is assigned its initial location, preferred speeds, acceleration properties, and parking delay and frequency. We set the turning and maximum speeds to 17.5mph and 35mph, respectively. Additionally, we fixed the minimum and maximum acceleration to -4m/s and +2.5m/s as suggested in [5]. After collecting all traces of all 100 cars, the average speed was 21.4mph, which is similar to a realistic traffic and speed patterns in a city.

In the simplest case, a car emulates a tourist driving randomly throughout the city. The car starts by accelerating to its maximum speed and tries to maintain the maximum speed while it is not too close to the next intersection. Once the car is approaching an intersection, it decides on its next move. It may decide to turn left or right and therefore decelerates to its turning speed. The car may also decide to continue going straight, if possible, and so it maintains its current speed. Alternatively, the car may notice a yellow or red light and initiate the process of coming to a complete stop. The car continues to move in a similar fashion for a random amount of time bound by the value of parking frequency. Otherwise, the car stops moving and waits for an amount of time up to the value of parking delay before continuing to drive until the end of the simulation.

Alternatively, some cars emulate taxi drivers. The car follows the same speed and turning principles as in the previous case; however, instead of making semi-random decisions at each intersection, the car pre-computes its route, *i.e.* a shortest path to the destination, and follows the route to the final destination. This emulates an actual taxi driver being asked by a passenger to drive to a particular location in the city. Once the car reaches the destination, it waits for a fixed amount of time before servicing the next passenger.

For our environment, we computed mobility traces for 50 taxi drivers and 50 mobility traces for tourist drivers. We used the resulting traces as inputs to GloMoSim for the 100 mobile nodes emulating cars in our environment.

We have deliberately chosen tourists in cars rather than tourists walking as examples for this paper because it represents a "bad" scenario for us. Given the speed of motion of cars, it is possible that by the time a data source is discovered and selected by the protocol, it would have moved out of range. Our experiments therefore report results under stress conditions. These experiments conducted under more favorable conditions that assume people rather than cars will clearly lead to better results. We also did not choose to use bicyclists because they simply represent cars driving at slower speeds.

4.1.5 Data, Profiles and Queries For our experiments, we have defined 10 distinct ontologies. Each ontology defines five attributes and their respective interrelationships. For example, one ontology depicts an abstraction of a restaurant database, while other ontology is an abstraction of traffic conditions measured and used by both cars and intersection beacons. We have then created 4096 data instances of each ontology, and distributed those between the intersection beacons. For ontologies using (x,y) coordinates, *e.g.* a restaurant location, we place the data to beacons in that location. Thus a beacon knows only about restaurants in its immediate vicinity. For ontologies not following (x,y) coordinates, we distributed the data randomly. Some ontologies included the concept of time, and thus data availability depended on both location and the time of the day. For example, a traffic jam warning information is available only when the jam occurs.

We have constructed 36 distinct queries for each car, each containing between 1 and 16 conjunctive boolean predicates. One half of queries were static in the sense that the filtering clause for these queries was fixed and did not depend on the location and time a query is asked, *e.g.* "Where is the National Art Gallery?". Other queries were dynamic in the sense that their filtering clause depended upon the car's location and the time of the day, *e.g.* "Where is the closest Chinese Restaurant?". Out of the 36 queries, 18 queries involved one data stream, 12 queries required joins over two data streams and 6 queries required the joining of three data streams. We note that intersections can help in answering queries involving only one stream as specified in Section 4.1.2. In addition to specifying a query, a car is also provided with the query prerequisite information describing the time, location, frequency and the probability that the query will be asked. Therefore, the total number of queries placed during the entire simulation depended on mobility pattern of each car and on satisfying the query prerequisites.

Additionally, we constructed profiles for each car with a varying accuracy from 0% to 100% by steps of 20%. The 0% accurate profile contained no information. A completely accurate profile represented a knowledge enabling a car to compute implicit queries equivalent to the 36 queries each car could ask. The other four cases held a permuted subset of the complete profile, such that the data collected using the implicit queries could answer on average only that many explicit queries. We synthetically computed the incomplete profiles, such that given all *global databases*, the number of entries returned by executing the union of explicit and implicit queries divided by the required cardinality of answers was *close* to the desired accuracy rate. Note, however, that even the precise knowledge of the user explicit queries is insufficient because no car knows its precise location at a future point in time when a dynamic query is posed. Additionally, since queries were constructed randomly and only a limited amount of data existed, it was often the case that no valid answer could be delivered or no such answer existed. On the other hand, this does not limit the accuracy of our experiments. No car in a real environment has a complete global knowledge of all data. Consequently, the car cannot always determine whether a query can yield a result with the desired cardinality.

4.2 Computation Performance

This Section presents the empirical results of 100 cars driving in 5000 x 9000m environment that closely resembles lower Manhattan. The cars drive through the environment for a period of 4 hours. The cars can interact with other cars and with 793 stationary intersection beacons. For that the cars initiate the CQP protocol to execute queries involving one, two or three streams. We refer to queries involving one stream as *simple queries*, while queries involving more than one stream are referred to as *complex queries*.

4.2.1 Query Success Rate vs. Profile Accuracy In the first experiment, we measure how profile accuracy improves the average query success rate. We vary the profile accuracy of each car from 0% to 100% with an incremental step of 20%. For each step, we calculate the average success rate of answering all explicit queries per car. We also calculate the average success rate of answering explicit queries involving only one, two or three streams. We obtain the rate by giving a partial credit to all cars. If n represents the number of queries asked by a car, α represents the cardinality of the answer obtained for query i and γ represents the desired cardinality for query i, then a car has a total partial credit, from 0 to n:

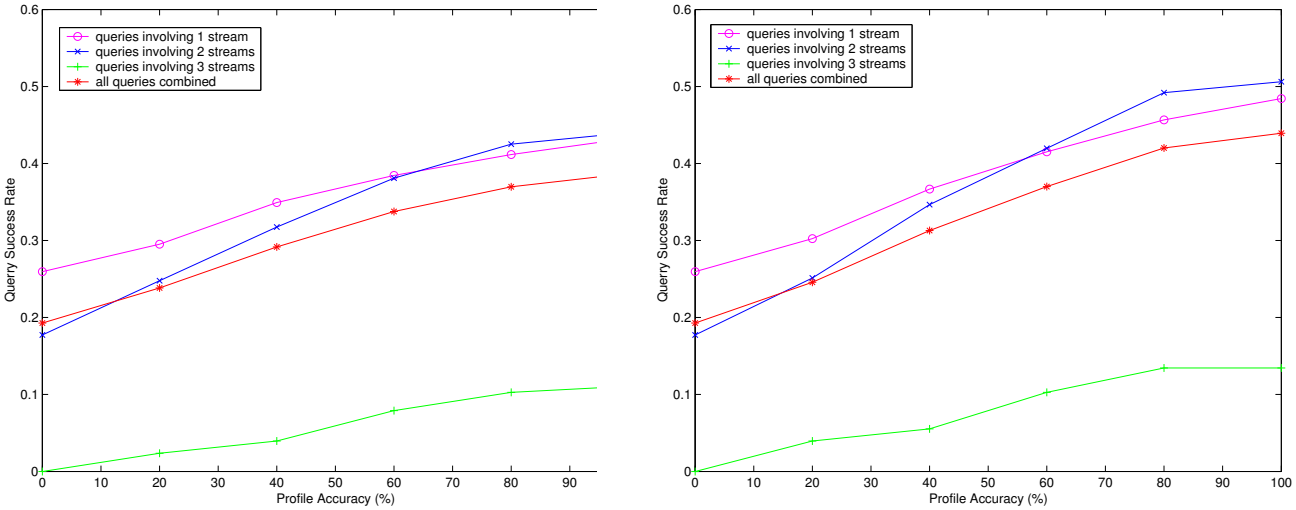
$$\frac{1}{n}\sum_{n}\frac{min(\alpha,\gamma)}{\gamma} \tag{3}$$

We look at two different scenarios: In the first case, no car is willing to help with answering queries proposed by peers. In the second case, each car has a willingness–to–help level set to 75%. Consequently, in this case each car responds, on average, to three out of four *call–for–query* messages by sending its matching *bid* when the car has a valid answer. Figures 7 A & B plot the obtained results for willingness levels equal to 0% and 75%, respectively.

The results for 0% accurate profile in the first scenario define our base–line. These results specify the average query success rate a car can obtain using only intersection beacons in its vicinity. Each car could answer its explicit

Duration	4 hours
Space (x,y)	5000 x 9000m
Tx Range	125m
Tx Throughput	2Mbps
Data	10 ontologies, each 4096 entries
Entities	100 cars (C); 793 beacons (B)
Profile Accuracy	C: any; B: none
Will to Help	C: any; B: 0/join,100%/select
Initial Distribution	C: no data; B: local data

Fig. 6 Experiment Parameters



(a) Willingness to Help = 0%

(b) Willingness to Help = 75%

Fig. 7 Query Success Rate vs. Profile Accuracy

queries using only the *bulk* data advertised by peers, *i.e.* intersection beacons. The base–line overall query success rate was 0.193. The base-line query success rate for explicit queries involving one, two and three streams were 0.260, 0.177 and 0, respectively.

As expected, a more accurate profile yields a higher average query success rate. In the first case, complete profile accuracy improved the overall success rate from original 0.193 to 0.387. This is a 100% improvement to results obtained using 0% accurate profiles. In the second case, a complete profile accuracy together with 75% change of help from other cars improved the overall success rate from 0.193 to 0.439. That is a 127% improvement.

At the same time, the average success rate never reached 1. This implies that no car, on average, was able to answer all explicit queries. The best outcome was in the second case, which yielded only 0.506 for queries joining two streams. The best overall average success rate was then 0.439, again in the second scenario. These results are due to the fact that some car asked queries that could not be answered by anyone in its current and/or past vicinity. The queries were not answered either because no answer existed in the environment or because no device with the answer was ever in the vicinity of the querying car. In addition, an important factor was the speed of each car in the environment. On average, the cars traveled at 21.4mph. Consequently, some interactions could not been completed because the two interacting peers moved away too soon.

4.2.2 Computing Cost vs. Profile Accuracy In the next experiment, we measured how much work a car has to perform for itself. We again vary the profile accuracy of each car from 0% to 100% with an incremental step of 20%. For each step, we then calculate the Computing Cost by measuring the operations and time it required to execute the operations. We consider only those operations that a car performs while executing its implicit queries. We do not include the cost inflicted upon cars caused by other devices requesting help. Similarly to the previous experiment, we look at two different scenarios. The first scenario sets the willingness to help to 0%, and the other scenario sets willingness to help to 75%. Figure 8 shows the results.

As expected, the computing cost quickly increases with more accurate profile. With a 0% accuracy level of a profile, no car derives any implicit query. Therefore, a car incurs no computing cost with a 0% accurate profile. For other accuracy levels, the cost increases by almost a factor of two from one accuracy level to the next. This is due to the fact that each car has more implicit queries it needs to answer, half of which cannot be answered as suggested in Section 4.2.1. This can also be validated from the fact that the costs from the two scenarios, 0% and 75% willingness

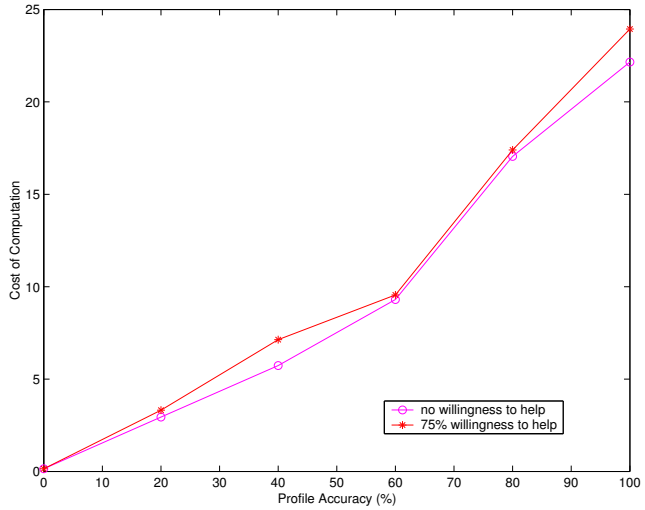


Fig. 8 Computing Cost vs. Profile Accuracy

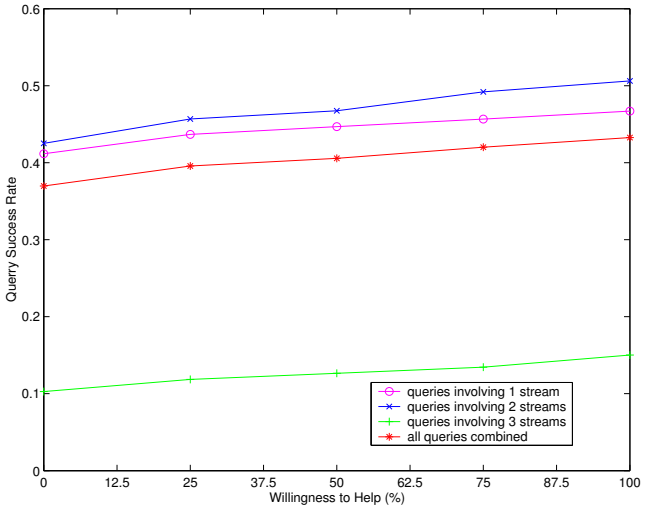


Fig. 9 Query Success Rate vs. Willingness to Help

to help, closely resemble each other. Even though cars are more willing to help in the second case, they still cannot provide answers to the *other* half of queries.

4.2.3 Query Success Rate vs. Willingness to Help After establishing the experimental base line in Section 4.2.1, we measure how the different willingness to help levels improve the average query success rates. We set the profile accuracy to 80%. This means that each car has an almost complete knowledge about the types, time and location of queries its user is likely to ask. We now vary the willingness to help level, *i.e.* car degree of collaboration, from 0% to 100% with incremental steps of 25%. To one extreme, no car is willing to help others at all, while in the other extreme the car is always trying to help whenever it is able to. We use the same approach as used in the first experiment to calculate the partial credit for each car, and to plot the results in Figure 9.

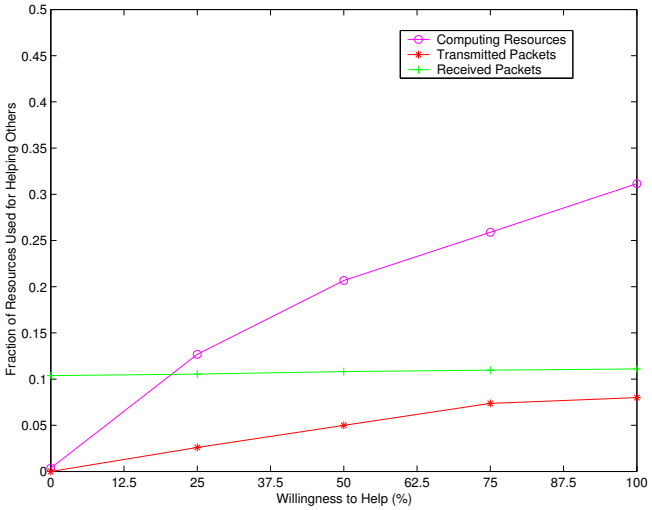


Fig. 10 Cost Helping Others vs. Willingness to Help

The measured results show that the CQP protocol improves the average query success rate for each car. The total query success rate is 0.37 when no car is willing to help. The total query success rate is 0.433 when all cars are willing to help. This is a 13.5% improvement. We are, however, more interested in the performance of explicit queries over two and three data streams. The success rates for queries over two and three streams are 0.425 and 0.103 respectively when no car is willing to help . The help of all cars yields success rates 0.506 and 0.15 respectively. For the queries over three data streams, the rate of improvement is 45.6%. Therefore, the CQP protocol significantly improves the execution of joins.

4.2.4 Computing & Network Cost vs. Will to Help Although, every car, on average, improves its query success rate, the cost of the improvement is not without some cost. In exchange for better success rates, each car pays in terms of its computing cost and additional network traffic. In this experiment, we compare how much of the total resources was, on average, allocated to helping others. We do so by collecting and calculating the fraction of total cost (computing and traffic) that a car used to help other peers. Again, we set the profile accuracy level to 80% and vary the willingness to help level from 0% to 100% with an incremental step of 25%. We collect the total amount of computing costs used by each car as well as the amount of computing done on behalf of others. The latter cost includes evaluation of *call–for–query* messages, submission of *bids*, performing selection and join operations for peers, for sending data to peers, and for collaboration-related timer processing. We also collect the total amount of data send and received by each car and differentiate among traffic inflicted while helping peers and traffic for answering the car's own implicit queries. We display the values in Figure 10. We note that the vertical axis defines a fraction of resources used for helping others, and the displayed range is only between 0 and 0.5.

The results show that the cost for helping others increases both in terms of computing resources and network traffic; however, the computing costs expended for helping others is at most one third of the total resource consumption and the traffic cost is at most 11%. Interestingly, the fraction of received data vs. the total amount of received data does not significantly change. The fraction is equivalent to 10.4% when nobody is willing to help and 11.1% when everyone is willing. This is due to the fact that as a car receives and accepts more *call–for–query* messages causing many request for subsequent data instances matching the query, other cars experience the same pattern. Therefore, both the number of data received when helping others and the amount of data received in total increase proportionally.

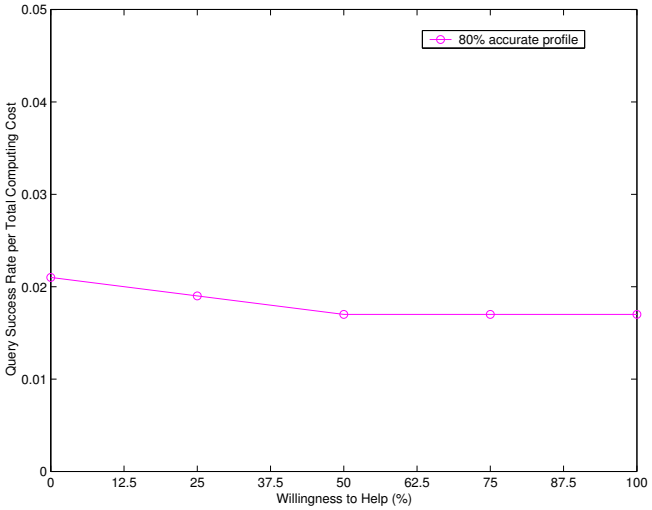


Fig. 11 Query Success Rate per Computing Resources vs. Willingness to Help

4.2.5 Success Rate / Computing Cost vs. Will to Help In this last experiment, we measure how different willingness to help levels affect the ratio of utility to cost. In our experiments, utility is represented by the average query success rate per each car. We then represent the cost value in terms of the total computing cost per car. Again, we set the profile accuracy level to 80% and vary the willingness to help level from 0% to 100% with an incremental step of 25%. We collect the average success rate for answering explicit queries and the average computing cost. We the show the results in Figure 11.

The utility to cost ratio remains constant. The ratio value for no willingness to help is higher, 0.021, because no car ever responded to a *call–for–collaboration*. In this case each car spent the least amount of resources. Moreover, each querying car in this case utilized resources of local intersection beacons only. In the other cases where cars are willing to help, we see that the ratio value stabilizes at 0.017. This is due to the fact that even though each car executes more computations, and thus has a larger computing cost, each car is also able to improve its overall query success rate. Therefore, the increasing computing cost is justified by the improving rate of query success rate.

5 Conclusions and Future Work

We have presented the design and implementation of the Collaborative Query Processing protocol. This novel protocol enables devices in pervasive computing environments to locate data sources and obtain data matching any query. In the simplest case, the query can be a selection scan over one stream; however, the query can also be a join of multiple streams. The protocol combines the traditional concept of nested–loop joins and our previous work on selection queries in pervasive computing environments together with the principles of Contract Nets. The features of the protocol enable any device, regardless its limited computing, memory and battery resources, to collaborate with other peers in its vicinity in order to obtain an answer for its explicit queries. We have shown how the protocol improves the overall system performance defined by the number of successfully answered explicit queries. We have also shown how the protocol affects the combined computing cost and the network traffic inflicted on each mobile device.

In this paper, we have not addressed the issues concerning privacy and incentive for devices to participate in information exchange. Although these are valid issues, our focus in this paper was first to enable mobile devices to exchange data in pervasive computing environments. Colleagues in our group are developing policy based security and privacy mechanisms for such environments[34, 23]. Similarly, we have not addressed the problem of ontology / schema translation. This is an important objective for our future work since any two devices understanding different

(incompatible) types of ontology can meet in this serendipitous environment. A simple solution is to ignore devices one does not understand; however, that is clearly not an efficient solution as it limits the amount of data available in the environment. Therefore, a presence of an ontology translator will most likely become paramount. There has been some work on this problem but the proposed solutions are still in their preliminary stages and often require a vast computing and memory resources [17]. Lastly, we have not addressed the notion of *update* transactions, though we have done some initial work on e-commerce type transactions [1] in these environments.

6 Acknowledgments

This work was supported in part by NSF awards IIS 9875433, 0070802 and 0209001, and DARPA contract F30602-00-2-0591. The authors also thank Sasikanth Avancha and Jeffrey Undercoffer for their help in writing this paper.

References

1. S. Avancha, P. D'souza, F. Perich, A. Joshi, and Y. Yesha, *P2P M-Commerce in Pervasive Environments*, ACM SIGEcom Exchanges, 2003.
2. S. Avancha, A. Joshi, and T. Finin, *Enhanced Service Discovery in Bluetooth*, IEEE Computer (2002).
3. AvantGo, `http://avantgo.com/`.
4. R Avnur and J. Hellerstein, *Eddies: Continuously Adaptive Query Processing*, SIGMOD, 2000.
5. C. Bettstetter, *Smooth is Better than Sharp: A Random Mobility Model for Simulation of Wireless Networks*, MSWiM'01, 2001.
6. Bluetooth SIG, *Specification*, `http://bluetooth.com/`.
7. P. Bonnet, J. Gehrke, and P. Seshadri, *Towards Sensor Database Systems*, MDM, 2001.
8. M. Bratmann, *Intentions, Plans, and Practical Reason*, Harvard University Press, 1987.
9. O. Bukhres, S. Morton, P. Zhang, E. Vanderdijs, C. Crawley, J. Platt, and M. Mossman, *A Proposed Mobile Architecture for Distributed Database Environment*, Tech. report, Indiana University, Purdue University, 1997.
10. D. Chakraborty, F. Perich, S. Avancha, and A. Joshi, *DReggie: Semantic Service Discovery for M-Commerce Applications*, Workshop on Reliable and Secure Applications in Mobile Environment, SRDS, October 2001.
11. S. Chandrasekaran and M. Franklin, *Streaming Queries over Streaming Data*, VLDB02, 2002.
12. M. Cherniak, E. Galvez, D. Brooks, M. Franklin, and S. Zdonik, *Profile Driven Data Management*, VLDB, 2002.
13. P. Chrysanthis and E. Pitoura, *Mobile and Wireless Database Access for Pervasive Computing*, ICDE, 2000.
14. DAML Services Coalition, *DAML-S: Semantic Markup For Web Services*, `http://daml.org/services/`.
15. A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch, *The bayou architecture: Support for data sharing among mobile users*, IEEE Workshop on Mobile Computing Systems & Applications, 1994.
16. A. Dey, G. Abowd, and D. Salber, *A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications*, Context-Aware Computing, Human-Computer Interaction Journal (2001).
17. D. Dou, D. McDermott, and P. Qi, *Ontology Translation and Translating Ontologies on the Semantic Web*, WWW, 2003.
18. M. Franklin, *Challenges in ubiquitous data management*, Informatics, 2001.
19. IEEE 802.11 Working Group, *Ad-hoc 802.11*, `http://ieee802org/11`.
20. J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K Hildrum, S. Madden, V. Raman, and M. Shah, *Adaptive Query Processing: Technology in Evolution*, IEEE Data Engineering Bulletin, 2000.
21. J. Hendler, *DARPA Agent Markup Language*, `http://daml.org/`.
22. D. Johnson and D. Maltz, *Dynamic source routing in ad hoc wireless networks*, Mobile Computing, 1996.
23. Lalana Kagal, Tim Finin, and Anupam Joshi, *Trust-based security in pervasive computing environments*, IEEE Computer **34** (2001), no. 12, 154–157.
24. J. J. Kistler and M. Satyanarayanan, *Disconnected operation in the coda file system*, Thirteenth ACM Symposium on Operating Systems Principles (Asilomar Conference Center, Pacific Grove, U.S.), ACM Press, 1991, pp. 213–225.
25. H. Kottkamp and O. Zukunft, *Location-aware query processing in mobile database systems*, Selected Areas in Cryptography, 1998.
26. S. Lauzac and P. Chrysanthis, *Utilizing versions of views within a mobile environment*, Conference on Computing and Information, 1998.
27. S. Madden and M. Franklin, *Fjording the Stream: An Architecture for Queries over Streaming Sensor Data*, ICDE, 2002.

28. F. Perich, S. Avancha, D. Chakraborty, A. Joshi, and Y. Yesha, *Profile Driven Data Management for Pervasive Environments*, DEXA, 2002.
29. F. Perich, S. Avancha, A. Joshi, Y. Yesha, and K. Joshi, *On Data Management in Pervasive Computing Environments*, Tech. report, UMBC CSEE, 2002.
30. F. Perich, A. Joshi, T. Finin, and Y. Yesha, *On Data Management in Pervasive Computing Environments*, TKDE (2003), accepted for publication.
31. C. Perkins and P. Bhagwat, *Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers*, ACM SIGCOMM Communications Architectures, Protocols and Applications, 1994.
32. C. Perkins and E. Royer, *Ad hoc on-demand distance vector routing*, IEEE Mobile Computing Systems and Applications, 1999.
33. R. Smith, *Readings in distributed artificial intelligence*, ch. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, 1988.
34. J. Undercoffer, F. Perich, A. Cedilnik, L. Kagal, and A. Joshi, *A secure infrastructure for service discovery and access in pervasive computing*, ACM MONET: Special Issue on Security in Mobile Computing Environments **8** (2003), no. 2, 113–125.
35. X. Zeng, R. Bagrodia, and M. Gerla, *GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks*, Workshop on Parallel and Distributed Simulation, 1998.
36. Y. Zhang and O.Wolfson, *Satelitte-Based Information Services*, ACM MONET (2002).