The VLDB Journal (2005) 14: 182–196 / Digital Object Identifier (DOI) 10.1007/s00778-004-0127-3

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, MD 21250, USA
e-mail: {filip.perich,joshi,yeyesha,finin}@umbc.edu

Edited by K. Ramamritham. Received: July 28, 2003 / Accepted: March 26, 2004
Published online: July 8, 2004 –

Abstract. We present a collaborative query processing protocol based on the principles of Contract Nets. The protocol is designed for pervasive computing environments where, in addition to operating on limited computing and battery resources, mobile devices cannot always rely on being able to access the wired infrastructure. Devices, therefore, need to collaborate with each other in order to obtain data otherwise inaccessible due to the nature of the environment. Furthermore, by intelligently using answers cached by peers, devices can reduce their computation cost. We show the effectiveness of our approach by evaluating performance of devices querying for data while moving in a citylike environment.

Keywords: Distributed join processing – Mobile ad hoc networks – Peer-to-peer computing – Pervasive computing environments – Query processing

1 Introduction

We address the problem of how mobile devices in pervasive computing environments can locate and provide information satisfying a user's needs by collaborating with peer devices. We specify a model for performing joins in pervasive computing environments that takes into account the limited battery, computation, connectivity, and storage resources of mobile devices.

Locating and obtaining context-sensitive information in a mobile environment supported by the wired infrastructure is a difficult problem. One must consider three principal challenges [18,28]: (i) *What* information will a user require? (ii) *When* will the user need it? (iii) *How* can her device obtain the required information? Pervasive computing environments only exacerbate the problem. The realization of ad hoc wireless network technologies, such as Bluetooth [6], creates an intriguing variation of the problem. These technologies allow spontaneous connectivity among mobile devices, including handheld and wearable devices, computers in vehicles, computers embedded in the physical infrastructure, and (nano)sensors. There is no longer a guarantee of infrastructure support. To obtain data, devices should no longer depend solely on the help of some fixed, centralized server [31]. The devices should also not be required to precache all required data a priori. Since mobile devices are no longer standalone entities or simple clients, they should be treated as 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 have to connect to a yellow pages 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 devices or an electronic sign of some gas station in its vicinity.

The dynamic nature of ad hoc wireless networks makes existing solutions for traditional mobile systems inapplicable [18,28]. Traditional query processing techniques, especially those performing joins, will often fail in this environment as they depend on infrastructure support. Further, because peer devices are mobile, the availability of information sources becomes spatially and temporally dependent.

Therefore, as a complement to infrastructure-based querying, a device should utilize its vicinity by collaborating with its peers. Collaborating devices will be able to obtain data otherwise inaccessible due to the nature of the environment. By querying its immediate neighbors and other devices accessible in its vicinity, a mobile device may be able to obtain the same data or data of even *higher quality* than by connecting to servers in the wired infrastructure. This is especially advantageous when a device has no access to the wired infrastructure. Additionally, by intelligently using answers cached by peers, devices will be able to reduce their computation cost. Collaboration will become especially helpful when performing joins over multiple data sources, which may be required when a single data source cannot provide the complete answer to a query. The querying device may have to *horizontally* join data streams from multiple devices holding the same type of data. The device may also have to *vertically* join data streams from different devices as they become available. We use the term *data stream* to represent both data sources according to the traditional definition and the "streaming"-data sources defined by the sensor network community [11].

We utilize our previous work on *what* information a user seeks and *when* she seeks it based on her 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. We take a more expansive view of context and include beliefs, desires, and intentions of a user stored in a user profile [30]. This allows a device to predict what information the user will need. Based on the prediction, a device dynamically adapts its strategies for querying its peer devices, caching of data, and for collaborative processing of joins. While our implementation builds on our specific previous work, the collaborative model we propose is relatively agnostic – any other technique for computing dynamic queries can be used [12].

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

- We propose a collaboration protocol based on the principles of Contract Nets [1,34]. Our collaborative query processing (CQP) protocol enables a mobile device to query its vicinity and locate peer data sources that can answer a given query. More importantly, our protocol allows the device to obtain data matching any query, irrespective of whether the query is a *selection* or a *join*. We extend the traditional concept of nested loop joins and 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. Each subquery can be assigned to a different device. The assignment is determined according to the available resources of devices present in the vicinity. 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 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 [29,28,30,31], which is a robust framework for profile-driven data management in pervasive computing environments. Our experimental model represents handheld/embedded computers and intersection beacons. We have explicitly chosen not to use any wired infrastructure in order to demonstrate that our model can operate in an ad hoc mode; however, our protocol can employ any device accessible by the querying device as a data source, including servers in the wired infrastructure. The primary objective of the model is to allow each mobile device to utilize as much available information as possible in order to enhance its functionality. The device can use static information such as user preferences and information about data sources. The device can also rely on dynamic information, such as the current context description. This allows each device to gather, provide, and possibly create data that will be helpful to the user in the near future.
- 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 the average success rate of satisfying user queries for each entity. We also show how the protocol affects the combined computing cost and network traffic for all devices in the environment.

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

2 Related work

The networking aspects of collaboration in pervasive computing environments have been well addressed [2,6,22,32,33], especially in terms of device discovery and routing protocols.

The problem of data management in wireless networks has also drawn significant attention; however, to the best of our knowledge very little work has been done on data collaboration and sharing. We are unaware of any work done in the area of processing joins over multiple data sources in a pervasive computing environment. Although research on in-network query processing and data aggregation in sensor networks addresses somewhat similar questions, we argue later in this section that the ad hoc and sensor network domains differ significantly. The existing solutions consist of two very distinct approaches to addressing problems imposed by the underlying networking technology, such as low bandwidth and high probability of disconnection.

One approach, for supporting disconnected operations, relies on offline data precaching. In wireless contexts, this was pioneered by the Coda file system [24] and more recently by commercial software such as AvantGo [3]. While precaching techniques preserve battery power and do not require connectivity, they are unable to provide answers beyond the scope of their caches and are also prone to stale data.

At the other end of the spectrum are 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 [12,13,37]. Chrysanthis et al. [26] consider disconnected operations within mobile databases by presenting a mechanism, referred to as a "view holder", that 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 that incorporate 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. These solutions rely on the support of a fixed, wired infrastructure. They place primary data on servers located within the wired infrastructure and treat mobile devices solely as clients. Using this approach all selection queries and joins are executed by servers and mobile devices

simply receive updates to their materialized views. Moreover, these solutions often incur monetary costs for cellular connectivity and are also prone to frequent disconnections due to the environment and mobility patterns. Lastly, these solutions are susceptible to a possible single point of failure at the server side.

In contrast to these approaches, our work does not require support from a fixed infrastructure. At the same time our work does not preclude the use of an infrastructure. We treat all data sources equally, irrespective of whether they are mobile devices or servers in wired networks. In related work [31], we studied the effects of decreasing the infrastructure's presence on completing transactions for mobile devices. We compared a method for committing transactions that depended on an infrastructure support with an alternative method employing an n-hop vicinity of transacting peers. We then varied the infrastructure's presence in terms of the number of available access points used for routing network traffic to the nodes in the wired network from 0 to 100%. Our results showed that even a small reduction in an infrastructure's presence drastically decreased the performance of the system. Additionally, when a mobile device requires instantaneous information, e.g., traffic updates, it may be more easily, sometimes only, accessible from other "local" mobile devices and not from a fixed node. As defined in [9], a mobile device in our work is always in nomadic mode.

Since we do not rely on infrastructure support, we must address an additional constraint not researched in the literature: information consumers are not the only mobile devices. Information sources – other peers – also change their locations. This limits data availability and causes answers to questions to become spatially and temporally dependent. Additionally, this may negatively affect execution of joins over multiple data streams because not all required data sources may be concurrently available. In traditional approaches, all stream sources must be available at the same time, which may not be the case in pervasive environments. A device performing a join must thus be able to precache *intelligently* relevant parts of the data that are currently available while searching for sources of missing data. The nature of the environment also affects the duration of any connection between a pair of devices and limits the possibility of a reconnection. Since each device moves independently, there is no guarantee that two mobile devices currently connected will ever again be able to communicate between themselves, which 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 queries. Consequently, every device must become more autonomous and adaptive in order to answer queries because it can mostly depend only on itself and its local resources [28, 30]. Essentially, the requirements to support heterogeneous devices and data render most solutions for traditional mobile networks unusable.

Recently, significant research interests have been directed toward the area of sensor networks. The research particularly concentrates 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 related to query processing in sensor networks appear somewhat similar, the environment and the objectives significantly differ. The goal of a sensor network is to collect and optionally aggregate fixed types of data at every node. The nodes then propagate their results to a collecting base station. Once a sensor network is established, all sensors know their peers. 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 can be a consumer. Each device interacts with others to obtain data primarily for itself. Only optionally does the device become a proxy or a provider for some other peer in its vicinity. Hence, a mobile device is always changing 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 collaborative query processing (CQP) protocol allows the mobile devices in pervasive computing environments to help each other in locating, computing, and obtaining answers for queries involving one or more data streams. The protocol enables a querying device to view its peers as additional sources of information. By reusing data made available by other devices, the "system" is able to provide data to a larger number of individual nodes while still preserving systemwide resources. These resources, which are always limited, include battery and computing overhead. Before detailing the design of our protocol we provide a brief overview of the MoGATU framework as it constitutes the underlying base for the CQP protocol.

3.1 MoGATU system overview

MoGATU is a framework for handling serendipitous querying and data management in pervasive computing environments [28,30,31]. The framework treats all devices present in the environment as equal semiautonomous 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 specifies several communication interfaces. Figure 1 depicts the framework and the relationship among devices and their resources. Every device is required to implement an information manager and at least one communication interface.

Information providers represent the data sources available in a pervasive environment. Every information provider holds a partial set, a *fragment*, of heterogeneous data available in the whole environment. We specify our data model as a set of ontologies and express its instances in a semantic language. Each information provider stores its data in the data's *base*

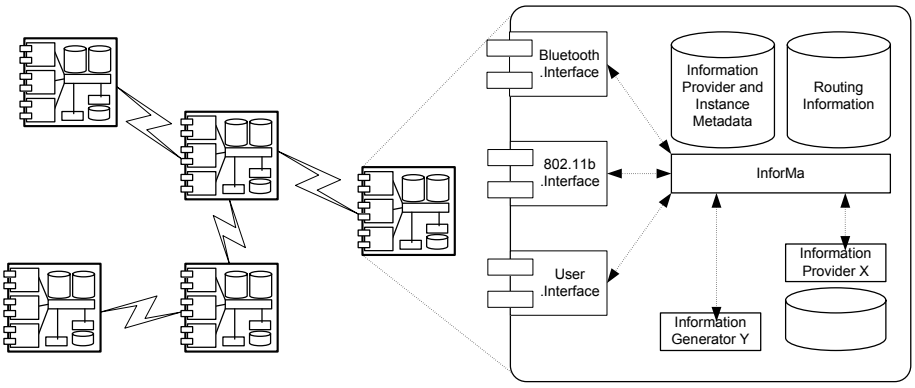


Fig. 1. MoGATU's entity and interaction model

form according to the ontology definition. Data involving one ontology are already expressed in their base forms and are stored in the format they were obtained. For data involving multiple ontologies, a provider decomposes the data into their base forms and maintains a view to link the base forms. Using this approach a view is represented as a list of pointers to the respective base fragments. It may be impossible to maintain global consistency among all information providers because the wireless ad hoc network frequently remains partitioned. As a result, mobile nodes attempt to be vicinity-consistent only.

Information consumers represent entities that query and update data that is available in the environment. In the current design, information consumers represent human users and proactive agents that actively prefetch context-sensitive information from other devices in the environment.

Lastly, an instance of an *information manager* (*InforMa*) must also be present on every device. Information managers are responsible for network communication and for most of the data management functions. Each InforMa is responsible for maintaining information about peers in its vicinity. This information includes the types of devices and information they provide. An InforMa also maintains a data cache for storing information obtained from other mobile devices and to cache information generated by its local providers. Additionally, each InforMa may include a user's profile reflecting some of the user's beliefs, desires, and intentions (BDI). This model has been explored in multiagent interactions [8] and significantly extends the work of [12] on profiles, which explicitly enumerates data and their utility. In contrast, by using the BDI concept, our profiles adapt to the environment by varying both data and their utility over time and present situations. The InforMa uses the profile to adapt its caching strategies and to initiate collaboration with peers to obtain desired information.

3.2 Data representation

Every mobile device holds a subset of the globally available *heterogeneous* data. Since the pervasive environment is, by definition, an open system, there are no restrictions or rules specifying the type and format of available data. To narrow this vast space of possibilities, we have previously argued the need for employing ontologies/schemata. In MoGATU, we define ontologies using the DARPA Agent Markup Language (DAML+OIL) [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 can define ontologies for weather and traffic updates [29].

One may argue that an ontology introduces too much of an overheard and that the use of a tabular relational schema would be sufficient. We disagree. Let us, for example, consider a case where we replace restaurant ontology by a schema defining restaurant properties in a fashion that allows the data to be stored in one table. Such a table could be horizontally fragmented over many devices in the environment, and it would allow other devices to query it. This, however, would require all devices in the environment to employ the same schema. Moreover, this would also remove any relationship information between data instances, such as inheritance or *subClassOf* constraints. For example, it would no longer be possible to deduce that the Joy Luck restaurant is an Asian cuisine restaurant without an explicit attribute stating so. Additionally, this would limit the possibility of introducing additional properties to data that already exist in the system because the cost of appending new relations to existing tables present on every device in the environment would be very high. Therefore, the use of ontologies in these environments is vital.

By using ontologies, we impose a requirement on each device to be able to parse DAML+OIL annotated information. This is not to say that all devices will, or must, understand all ontologies. Rather, we anticipate a scenario where each device has some knowledge of a set of ontologies. For new or unknown ontologies already present in the environment, a device can detect some metadata information by applying DAML+OIL rules. This will allow a device to match queries with provider advertisements without any knowledge about the particular data. This may also allow a device that understands ontology *A* to use data annotated in an unknown ontology *B* if *B* is a *subClassOf A*. For example, a device may not be able to deduce that Joy Luck is a Chinese restaurant, but it will at least know that Joy Luck is a restaurant.

In this paper, we concentrate on query processing only. We do not take into account the time necessary for reasoning over the ontology knowledge, e.g., whether a Joy Luck is a Greek restaurant. We ground each base data instance in one ontology and include a globally unique identifier. For data representing information from multiple ontologies, such as a result of a join, we assign a locally unique identifier in addition to the global identifiers of the base data.

When an information provider generates new data, it assigns the data a *lifetime* timestamp. The lifetime represents a time period after which the data should be invalidated and

purged from cache. Additionally, an InforMa assigns each data a local timestamp before placing it into its cache. We use the timestamps to calculate *recency* of cached data during query execution. In [30] we introduced a profile-based semantic cache replacing strategy. Our results showed that using context information and user preferences stored in a profile greatly improves cache performance over simple timestamp approaches. Our strategy uses a profile and context to preallocate cache space into *bins* for different types of data. For each bin, we employ a combination of timestamp, location, and user preferences to select a replacement victim when necessary. We employ the same technique in the CQP model.

3.3 Query representation

A device can either wait for the user to ask an explicit query and attempt to obtain an answer from its peer devices or it can proactively attempt to obtain data before a user asks for them. Proactive data querying is a preferred solution because it allows a device to cache data while the data are available [30]. A device employing the proactive approach has a greater possibility of providing the right data to its user than one using the reactive approach. The use of 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 – those asked by users – and (ii) implicit queries, which are *inferred* from user profiles.

The CQP model applies to both explicit and implicit queries. However, for the purpose of validating the CQP model, we only use explicit queries for measuring the *overall performance* of the framework. We restrict the devices so that they answer explicit queries using only their local resources. This way a device will not initiate an interaction with its peers when its user asks a question. On the other hand, a device will initiate the CQP model for implicit queries in order to obtain missing data from its peers. Implicit queries are excluded from the *overall success* performance because at the end only explicit queries are valuable to a user. This is because a user explicitly asks such queries, while implicit queries are inferred from a profile.

We define queries using a DAML+OIL ontology. In particular, the queries are specified in DAML-S [14], which is a standard evolving in the Semantic Web community. For presentation in this paper, we abstract the query representation using a familiar *select–from–where* form. Figure 2 shows the abstract representation form. A query is specified by a tuple consisting of a set of used ontologies (O), selection list (σ), filtering statement (θ), cardinality (Σ), and temporal (τ) constraints:

$$query = (O, \sigma, \theta, \Sigma, \tau). \quad (1)$$

Each query defines the ontologies used for constructing the filtering clause and for final projection of the matching data instances. The set can include a specific ontology multiple times if the filtering clause consists of a join over multiple data streams represented in that ontology. The size of the ontology list, therefore, specifies the *degree* of the query. The degree 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

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

Fig. 2. Abstract query specification

device uses its cached data and context information, including current geographical position and time of day, as inputs to these predicates. This allows a mobile device to perform a *dynamic* query asking for the closest local gas station. It also allows a device to pose a *static* query asking for a Chinese restaurant located on W 72nd Street. Along with string and numeric 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 geographical objects. The cardinality constraints of the query specify the minimum and maximum size of a required answer. Lastly, the temporal constraint specifies when the query should be completed. This is used by the device to periodically query its peers, given an implicit query, when time permits and the device has not yet cached a sufficient answer.

3.3.1 Explicit query

When a user asks her mobile device a question, the device, represented by the InforMa, attempts to answer the query by examining its local resources only. The InforMa evaluates the filtering statement θ over any combination of data that are present in local fragments. The data must be defined using a subset of ontologies from the ontology list O. The InforMa 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 InforMa directly scans and evaluates data instances stored in one local fragment. For queries requiring joins over data from multiple sources, the InforMa first checks whether a cached result from a similar query could be used. If no such *result view* is found or the cardinality constraint has so far not been satisfied, the InforMa performs the necessary join operations over its local fragments and returns the final result.

3.3.2 Implicit query via user profile

The InforMa on each device may include a user's profile, which specifies a subset of beliefs and desires that the mobile device should know. The knowledge contained in the profile is initially defined using DAML+OIL. Upon loading the profile, the InforMa converts the knowledge into implicit queries. These queries include constraints describing time, location ranges, and the probability of the query being asked by the user. For example, a profile may contain a preference stating that the user likes to eat lunch between noon and 2 pm. Additionally, the profile may state that the user prefers Chinese restaurants. The InforMa combines the two beliefs into an implicit query, which collects and maintains the location

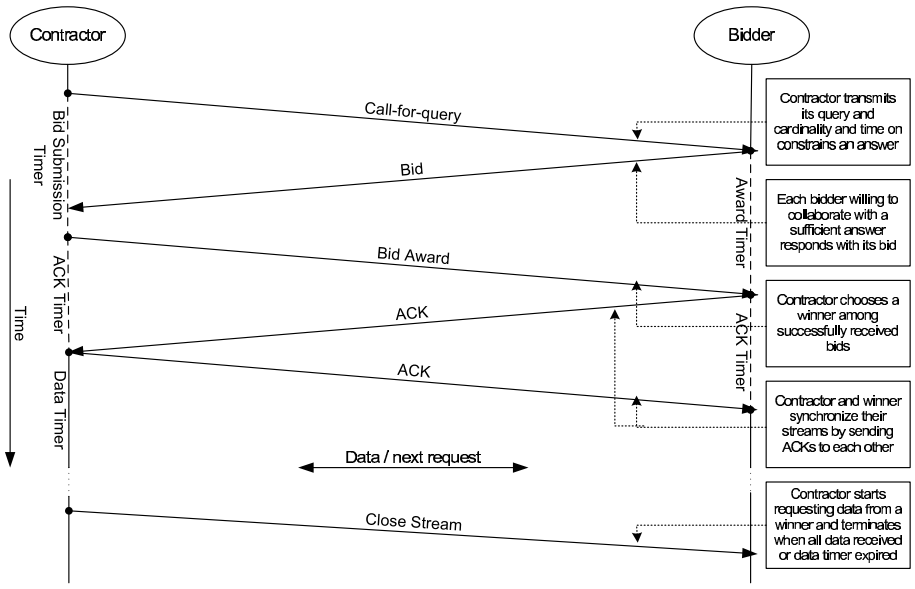


Fig. 3. Message and state flow of the collaborative query processing protocol

of at least one Chinese restaurant in the current vicinity from 11:45 am until 2 pm. The details of *reasoning* over profiles for generating queries is beyond the scope of this paper and we refer readers to [30].

Each belief or desire can be represented as the following tuple:

$$belief, desire = (O, \sigma, \theta, \Sigma, \lambda, \tau, P), \quad (2)$$

where O, σ, and θ define the set of used ontologies, 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. P denotes the probability predicting that the user will, in fact, ask a question that uses the data obtained by executing the implicit query. When evaluating the success of our framework, we consider results for explicit queries only. This is because we do not want to reward a device for proactively caching data to questions its user never asks; however, for evaluating the overhead of the CQP model we measure the computing and networking cost for processing both implicit and explicit queries.

3.4 Collaboration protocol

We now present the detailed design of the CQP protocol. We show the message and state flow of the CQP model in Fig. 3. CQP is based on the principles of Contract Nets [1,34]. 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 involving one or more data streams. In this paper, we are not interested in studying incentive-driven mechanisms and the conditions for a 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.

We designed CQP with the underlying environmental characteristics in mind. In order to overcome the frequent disconnections and low bandwidth, we expect each device to locally process as much data as possible before sending results to its peers. We do so by decomposing DAML+OIL annotated queries into selection queries over base ontologies and their combinations. To overcome the network limitations, we employ a Contract Net-based negotiation among peers for identifying devices that will provide certain data. For this we build upon our previous experience in service and data discovery [2,10] and our previous work on MoGATU. We employ both gossiping and pull-based techniques for advertising data and for querying peers. Whenever possible, we attempt to utilize the underlying routing protocol for network packets between mobile peers; however, in order to overcome the uncertainty of the ad hoc environment, CQP does not need to collect all data before terminating, as detailed in Sect. 3.4.6.

The resulting CQP protocol consists of a contract agreement phase, a streaming phase, and a termination phase. In the first phase, a device discovers peers that have the required data. The originating device creates an agreement with its peers on their tasks and the desired outcome. In the streaming phase, collaborating devices process data locally and return results to their peers. In the last step the involved devices finalize their query processing operations. Since there is no guarantee of a stable connectivity, in addition to exchanging messages, as a last resort the involved devices also employ timers. We now detail each step:

3.4.1 Call for query

When a mobile device needs to satisfy some of the beliefs and desires stored in a user's profile, the device converts the requirement into an implicit query. 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 number of cache hits, it constructs a *call-for-query* message. The message contains the query or a part of it, cardinality requirements, and the deadline for delivering the complete answer. The message also includes the time when a winner will be announced. The device sends the message to its peers up to

n-hops away and starts its *bid-submission* timer. The mobile device is now a *contractor*.

The *contracting* device is effectively flooding its *n*-hop vicinity with the *call-for-query* message. Using this method, the device is not required to maintain any information describing the type of data its peer devices are able to provide. However, if the *contracting* device has some knowledge about a specific peer that can provide an answer to the query based on past experience or from the peer's advertisement, the *contracting* device can optionally send a direct request to that specific peer.

For a selection query, the device includes the entire query in the message. This is because the device needs to simply ask one peer at a time to scan its local fragment matching the query. For queries involving multiple data streams, the device may require help from more than one device in its vicinity, as depicted in Fig. 4. The device first searches for a peer that can perform the entire query over its local fragments and return the results. Alternatively, when it does not receive a response within the *bid-submission* time, the device starts to decompose the query by vertically fragmenting it into subqueries involving fewer data streams. The device sends a new message for each subquery. When it does not receive a positive response, the device continues to decompose the query until it generates base selection queries. Each device uses the τ timer of the query, the *bid-submission* timer, and the number of cached answers to iterate through the decomposition process. We employ this heuristic, as a device cannot predict a priori what sources are currently available.

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, i.e., hold, one input stream that is currently available. The device then delays the join until the other stream becomes available in time and space. This allows a mobile device to better utilize the serendipitous nature of the environment, as we show in the next section.

The CQP protocol also enables chaining of multiple collaborations. For example, a device that agreed to provide an answer to a query may be unable or choose not to perform the operation itself. Instead, the device may decompose the query and request other devices to perform the necessary suboperations. This allows resource-limited devices to collaboratively execute a query of almost any complexity.

3.4.2 Bid submission

Upon receipt of a *call for query*, a device *decides* if it should interact in the proposed collaboration based on an inference. A mobile device that wishes to collaborate calculates the size of the answer it can provide to the contractor. If for some reason the mobile device does not believe it can satisfy the proposal, the device simply ignores the *call-for-query* message. On the other hand, if the mobile device determines that it can provide a valid answer, the device returns its *bid* message including the estimated size of its answer. The responding mobile device, now a *bidder*, starts a timer awaiting a *bid-award* message.

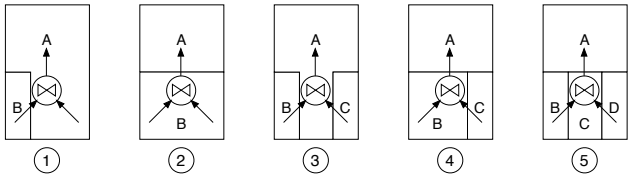


Fig. 4. Possible decompositions of a query involving two streams. In case 1, querying device *A* asks its vicinity for one input stream only since it already holds the second stream. In case 2, *A* asks its vicinity for the final join result only. In case 3, *A* asks for each stream separately in order to perform the join locally. In case 4, *A* asks *B* to process the query, but *B* needs to first obtain the second stream from some other device *C*. In case 5, *A* "delegates" the task to *C*, which asks its vicinity for the input streams instead. There are other possible combinations; however, the current CQP model employs only these five cases

3.4.3 Bid award

Because the contractor cannot predict effectively how many responses it would receive, the contractor employs a timer. The contractor waits a predefined time period for any response before evaluating them. When a contractor's timer for bid submission expires, the contractor evaluates all successfully received bids, choosing the bidder that claims to be able to deliver the most data in the shortest possible time. The contractor prefers a short streaming period because the network can quickly become disconnected. The contractor creates a *bid-award* message, starts an *ack* timer, and sends the message to the winning bidder. The contractor attempts to contact the winner at most *n* times before discarding the current session.

3.4.4 Acknowledgment

If a bidder does not receive a *bid-award* message before its timer expires, the bidder resends its *bid* message at most $n-1$ more times before discarding the current session. If the bidder does not receive the *bid-award* message after the last attempt, the bidder assumes someone else was awarded the contract. It also discards the current session. On the other hand, when the bidder receives the *bid-award* message, the bidder, now a *winner*, sends back an *ack* message. The winner starts its *sync* timer and waits for *ack* from the contractor. The contractor also waits at most *n* times the period of the *sync* timer in order to receive an *ack* message from the winner before discarding the current session.

In the CQP protocol, interacting devices employ timers because the wireless ad hoc network may not support a reliable message delivery. This could be due to frequent network partitioning or fast device mobility. Therefore, devices proactively retransmit messages in addition to using any available routing protocols for reliable message delivery.

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 a data stream. The contractor is also ready to accept

the stream. The contractor continuously asks for the next data instance until it receives either an *end-of-stream* or a *stream-error* message. Alternatively, the contractor stops receiving data when its *streaming* deadline timer expires.

The contractor is effectively treating other peers as data streams. These streams can consist not only 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 filter the results calculated by the source peer. In that case the source peer processes its outgoing stream before sending it out so as to avoid transmitting data that the contractor would find "useless".

3.4.6 Termination

The CQP protocol terminates in two ways. First, the protocol terminates when the winner has sent its entire answer and the contractor received it. The size of the answer was previously defined by its maximal cardinality in the agreement. Alternatively, the protocol terminates when a *stream-error* exception is generated. The exception can be generated at either the contractor or the winner site. The exception is generated when there are no additional data or when the streaming deadline expires. The exception is also raised when a partition occurs in the ad hoc network due to the mobility of the contractor, the winner, or some routing device. Therefore, both contractor and winner terminate the session either once all data are exchanged or after a predefined period of inactivity.

3.4.7 Error handling

To overcome possible errors caused by network partitions and resource limitations, our protocol operates on a best-effort basis only. We employ a profile-based caching policy for storing local results and data received from peers. When a device determines that it needs to obtain additional data, it initiates the CQP protocol and interacts with the winner. If the interaction fails before the contract is satisfied, the contractor attempts to find another source after a parametrized period of time. We employ the initial contract agreement phase for two reasons. First, the device is able to collect a catalog of its vicinity based on its query. Using the catalog the device can select the most applicable peers for interaction. Second, by requiring a longer initial interaction, we overcome situations where a device is in range only during the agreement phase and disappears as soon as agreement is reached, causing the execution to fail.

3.4.8 Join query over two streams

In this environment, we must differentiate between two types of join executions. One join execution involves only one device operating over two local streams. Such a device can employ any join algorithm. The other kind of join involves one or more network streams as shown in Fig. 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 an advanced hashing or sorting technique because the device cannot influence the stream order or size. For our implementation we have chosen to use a traditional nested-loop join processing algorithm. At the same time, if a device can cache the entire network streams locally, then the device can again use any alternative algorithm. Although speed is an important factor in traditional query processing, we do not envision it to be as important in ad hoc environments. We assume that the complexity and amount of data will 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*. *A* has an implicit query that involves joining two streams α and β, e.g., restaurants and gas stations. It has generated the query based on the prediction that its user is likely to ask a similar query in the near future. The device can only collect data that are 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 possible. Using CQP protocol, the device attempts to first contact someone who can completely satisfy the query. If that fails, the device decomposes the query into two *selection* subqueries, one for each stream. The device then attempts to collect data for each stream and performs the join locally or with the help of another peer. More importantly, the device does not have to ask for both streams at the same time. As it iterates through the possible decompositions of the implicit query, the device can ask for one stream and delay the query for the second stream. 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: the 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 min. Using the CQP protocol, device *A* asks device *B* first to execute the complete query and then to execute a subquery 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 subquery 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. At the same time, when device *A* caches the results from device *B*, it may hold stale data with respect to its current context. Therefore, if device *A* determines it has stale data, the device will requery its vicinity in the permitted time. This way the device is continuously collecting data for *active* implicit queries. Therefore, if a query depends on a location and time, the device will continue to proactively locate and obtain new data. We also note that it is explicit queries that are important from the standpoint of system performance.

4 Experiments

The objective of our experiments was to study the performance of the CQP protocol in ad hoc networks. We employed an experimental spatiotemporal environment representing a city scenario. The scenario closely resembled a 4-h period in lower

Manhattan in New York City. Devices in our environment represented moving objects such as pedestrians' handhelds and devices embedded in bicycles and cars. Devices also represented stationary objects such as store beacons or intersection lights. Every device implemented the MoGATU framework. Each device included an InforMa, an information consumer, and several information providers. Each InforMa could advertise and solicit information about other mobile devices in its vicinity. It accomplished that using epidemic/gossiping techniques traditionally used in mobile environments [30]. Each InforMa could also advertise or solicit information providers and send *bulk* data. Most importantly, each InforMa derived implicit queries from profiles and used the CQP protocol for answering them. Each InforMa also used the profile to adjust its caching strategies.

We used GloMoSim [36] to simulate the environment. GloMoSim is a scalable simulation environment for wireless and wired networks. It is designed to use a discrete event simulation capability provided by Parsec. GloMoSim defines several mobility models, such as a random-waypoint algorithm, and allows users to define additional models in order to emulate network partitioning in time and space. GloMoSim simulates the Open Systems Interconnect layered model defined by the International Standards Organization. The simulator also defines various radio models and radio propagation algorithms. Most importantly, GloMoSim computes radio noise generated by devices in the vicinity and pseudorandomly introduces additional radio noise in order to delay or drop packets. To perform the simulation, we used Intel Pentium IV 1.4-GHz desktops with 256 MB RAM, running Linux 2.4.18-5. The GloMoSim and the MoGATU extension were written in C.

Users in our model could represent pedestrians, people traveling using public transport, and people driving in automobiles. We chose parameters of our environment (speed of movement and direction of movement) to mimic vehicular traffic, cars, and their drivers. This represented the higher end of the speed parameter and led to frequent changes in the "vicinity" of a user. This also represented the worst-case scenario for our approach; however, it was somewhat mitigated by the fact that an automobile drives in a more predictable and organized fashion in the city (along roads) than a device allowed to walk completely randomly (direction- and speed-wise) through two-dimensional space. The use of a city traffic environment thus enabled us to better examine the CQP protocol performance in realistic scenarios, and our particular choice of parameters stressed the protocol.

Within the environment we measured the system performance in terms of the average success rate of answering explicit queries per user. We measured the average computing cost each user's device spent answering its explicit and implicit queries, and we measured and compared the cost to the average computing cost each device spent answering queries posed by other devices in its vicinity. Finally, we measured the network traffic incurred by each device when answering its queries and the traffic due to answering queries of its peers. For our experiments, we varied and measured the following parameters:

- *Query success rate*: The average query success rate represented a fraction of explicit queries that each device was able to answer using its local fragments. Initially, a device of every user had an empty cache and thus had no "remote" data. The device could obtain additional data by either accepting *bulk* advertisements from intersection beacons or by executing implicit queries that were derived from a profile.
- *Profile accuracy*: This represented how closely a user profile resembled the real actions of the particular user. A completely accurate profile knew precise explicit queries. An accurate profile also knew the location and time a specific query was going to be performed. At the same time, a completely accurate profile did not imply a 100% query success rate. A car knowing a complete profile might 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 was unable to answer queries involving the current weather conditions in Los Angeles. A 0% accurate profile had no knowledge, while the profiles with accuracy from 1 to 99% synthetically changed the complete profile to, on average, be able to answer that percentage of explicit queries. For example, 50% accurate profile could, on average, answer 50% of explicit queries.
- *Willingness to help*: This value represented the probability of a car responding to a *call for query* given that it could provide a valid answer. A willingness level set to 0% implied no help from other cars in the vicinity. A willingness set to 75% implied that a car was willing to help three times out of four whenever it could.
- *Computing cost*: Computing cost represented 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 created an abstract function that converted each operation a car performed to a value. The function considered the complexity of the operation and the time it took to complete the operation. We summed the values of all operations performed by each car during the simulation into a single scalar value. This value represented an average number of instructions or the energy consumption per car.
- *Network traffic cost*: This represented the average number of packets sent and received by each car. We counted the number and size of packets each car sent and received when attempting to satisfy its implicit queries. We also counted the number and size of packets each car sent and received when helping others.

We used the results of these metrics to measure the implementation performance of the CQP protocol, that is, we measured and studied the effects of:

1. Profile accuracy vs. Query success rate (Sect. 4.2)
2. Profile accuracy vs. Computing cost (Sect. 4.3)
3. Willingness to help vs. Query success rate (Sect. 4.4)
4. Willingness to help vs. Computing and network cost (Sect. 4.5)
5. Willingness to help vs. Query success rate/computing cost (Sect. 4.6)

To avoid confusion, in the remaining part of the paper we refer to user devices simply as *cars* and to stationary intersection devices as *beacons*. We would like to emphasize the fact that user devices are not necessarily just cars. They can be bicycles or handheld devices carried by pedestrians.

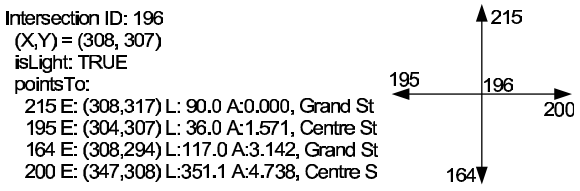


Fig. 5. Sample intersection representation used for the experimental model

4.1 Experimental setup

4.1.1 Environment

We created a realistic model that mapped streets and their intersections south of 72nd Street in Manhattan. We computed intersections between two or more streets by creating vertices in a directed graph. Our resulting model represented a graph with 793 intersections (vertices) in a 5000×9000 m space. Each intersection was assigned an (x, y) coordinate. Additionally, each intersection was given a list of its neighboring intersections. This allowed us to obtain an angular direction and distance between each pair of intersections. Figure 5 exemplifies what information each intersection had and how it related to the original map.

4.1.2 Beacon entity

We assigned a stationary beacon to each intersection, which resulted in a total of 793 beacons. We used these beacons as sources for information about their locations, i.e., each beacon had some knowledge about its vicinity. Each beacon implemented an instance of MoGATU and was able to transmit up to a distance of 125 m with a maximum throughput of 2 Mbps. A beacon represented a computationally limited resource but was able to provide a base of location-dependent data and had an unlimited supply of energy. For example, a beacon at 52nd Street and 5th Avenue contained information about restaurants, movie theaters, and the latest traffic conditions within 200 m. Although each beacon held a large amount of data and was willing to share them with others, it did not have the capability to perform joins over multiple streams. Beacons could provide answers to a selection query involving one stream only, i.e., a *scan*. We chose this scenario to make users (cars) work "harder" for their information.

4.1.3 Car entity

We used 100 cars, which represented the other type of devices in our environment. Cars also transmitted up to a distance of 125 m and had a maximum throughput of 2 Mbps. Each car moved freely according to the mobility model, which we describe below. Initially, every car was assigned a profile, which represented the beliefs and desires of its driver with a certain degree of accuracy. To obtain additional data, a car had to be close to another device in the environment when the other device was broadcasting *bulk* data, i.e., pass a beacon. Alternatively, a car had to infer an implicit query based on its profile and use that to interact with other devices in its vicinity, irrespective of whether those were cars or beacons. Unlike beacons, a car represented a computationally powerful device. When a car was willing to collaborate, it could help in processing a query over a single or multiple streams.

4.1.4 Mobility model

To allow cars to move in a more organized manner than using the random-way-point model, we extended the work on the smooth random-way-point mobility model [5]. This model defined parameters allowing a moving object to accelerate, decelerate, and move in a random-way-point pattern. In our model we restricted each car to drive from one intersection to another. Our model also accounted for acceleration and deceleration of an object; however, our objects did not have to come to a full stop in order to change their directions. Additionally, we considered all traffic lights in the environment to ensure that cars obeyed the local driving laws.

When a car started moving, it had an initial location, preferred speeds, acceleration properties, maximum time period for parking, and parking frequency. We fixed the turning and maximum speeds at 17.5 mph (28.16 km/h) and 35 mph (56.32 km/h), respectively. Additionally, we fixed the minimum and maximum acceleration to -4 m/s and $+2.5$ m/s. These values were suggested in [5]. After collecting all traces for 100 cars, the average speed was 21.4 mph (34.44 km/h), which is similar to a realistic traffic and speed patterns in a city.

In the simplest case, a car was driven by a tourist randomly throughout the city. The car started by accelerating to its maximum speed. It maintained the maximum speed while it was not *too* close to the next intersection. Once a car started approaching an intersection, it decided on its next move. It could decide to turn left or right and thus would have decelerated to its turning speed. The car could also decide to continue going straight, if possible. In that case the car maintained its current speed. Alternatively, the car could notice a yellow or red light and so would start braking in order to come to a complete stop. The car continued to move in a similar fashion for a random amount of time bound by the value of a parking frequency. At that time, the car stopped moving and waited for a pseudorandom time up to the predefined maximum parking period. The car then continued to drive in a similar fashion for the rest of the simulation.

In a more complex case, cars were driven by taxi drivers. They followed the same speed and turning principles, as did tourists' cars; however, instead of making semirandom decisions at each intersection, they precomputed their routes. The routes were computed as the shortest paths to the destination from current positions. The taxis followed the routes to the final destinations. This simulated an actual taxi driver asked by her passenger to drive to a particular location in the city. Once the car reached the destination, it waited for a fixed amount of time before serving the next passenger.

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

We placed tourists in automobiles rather than as pedestrians because the mobility afforded by an automobile is more

Duration	4 h
Space (x,y)	5000 x 9000 m
Tx range	125 m
Tx throughput	2 Mbps
Data	10 ontologies, each 4096 entries
Devices	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. Experimental parameters

difficult from a data management perspective. Given the speed of cars, it is possible that, by the time a data source is discovered and selected by the protocol, it has already moved out of range. Our experiments therefore report results under stress conditions. These experiments conducted under more favorable conditions, which assume people rather than cars, would clearly lead to better results. We also did not choose to use bicyclists because they would represent yet another *type* of car but driving at slower speeds.

4.1.5 Data, profiles, and queries

Ten distinct ontologies were defined for our experiments. Each ontology defined five attributes and their respective relationships. For example, one ontology depicted an abstraction of a restaurant database. Another ontology was an abstraction of traffic conditions measured and used by both cars and intersection beacons. We created 4096 data instances of each ontology and distributed the data among intersection beacons. For ontologies using (x, y) coordinates, e.g., a restaurant location, we placed these data at specific beacons in that location. This ensured that a beacon knew 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 day. For example, traffic jam warning information was available only when a jam occurred.

We constructed 36 distinct explicit queries for each car. Each query contained between 1 and 16 conjunctive Boolean predicates. One half of the queries were static, i.e., the filtering clause for these queries was fixed. The clause did not depend on the location and time a query was posed (e.g., "Where is the National Art Gallery?"). Other queries were dynamic, i.e., their filtering clause depended on the car's current location and the time of day (e.g., "Where is the closest Chinese restaurant?"). Out of the 36 queries, 18 involved one data stream, 12 required joins over two data streams, and 6 required three data streams. We refer to queries involving one stream as *simple queries*, while queries involving more than one stream are referred to as *complex queries*. We note that intersection beacons could provide answers to simple queries only as specified in Sect. 4.1.2. In addition to specifying a query, a car also had the query prerequisite information describing the time, location, frequency, and probability that each query would be asked.

Additionally, we constructed profiles for each car with a varying accuracy from 0 to 100% with a step of 20%. The 0% accurate profile contained no information. A completely accurate profile represented 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 so 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 was insufficient. This was because no car knew its precise location at a future point in time when a dynamic query was posed.

The profiles and explicit queries were constructed randomly in order to limit any correlation bias among different cars. Additionally, since explicit 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 did not limit the accuracy of our experiments. No car in a real environment would have complete global knowledge of all data. Consequently, the car could not always determine whether a query could yield a result with the desired cardinality of an answer.

4.2 Profile accuracy vs. query success rate

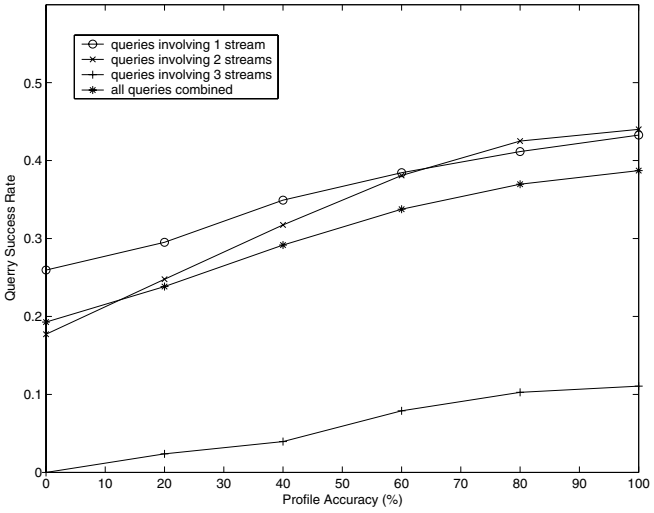
In the first experiment, we measured how profile accuracy improved the average query success rate. We varied the profile accuracy of each car from 0 to 100% with an incremental step of 20%. For each step, we calculated the average success rate of answering all explicit queries per car. We also calculated the average success rate of answering explicit queries involving only one, two, or three streams. We obtained each result by assigning a partial credit to all cars. If *n* represented the number of queries asked by a car, α represented the cardinality of the answer obtained for a query *i* and γ denoted the desired cardinality for *i*, then a car had a total partial credit, from 0 to *n*, as:

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

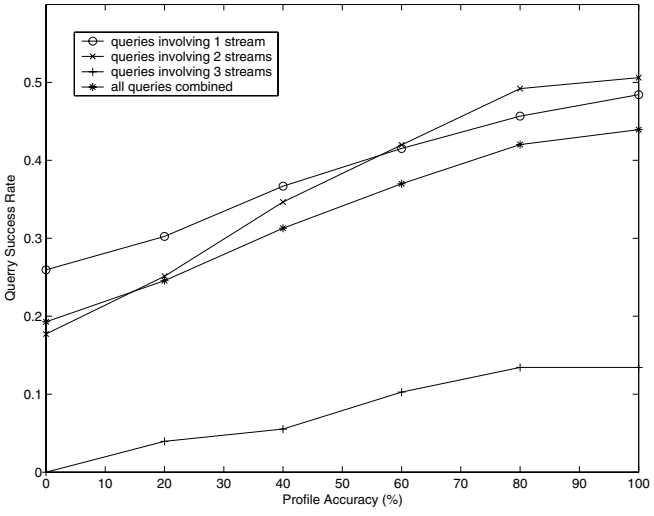
We looked at two different scenarios: in the first case no car was willing to help answer queries posed by peers. In the second case each car had the willingness-to-help level set to 75%. This ensured that when a car had a valid answer it responded to, on average, three out of four *call-for-query* messages only. Figures 7a and b plot the collected results for willingness levels equal to 0% and 75%, respectively.

The results for 0% accurate profile in the first scenario defined our baseline. These results represented the average query success rate a car could obtain using only intersection beacons in its vicinity. Each car could answer its explicit queries using only *bulk* data advertised by peers, i.e., intersection beacons. The overall baseline query success rate was 0.193. The baseline query success rate for explicit queries involving one, two, and three streams was 0.260, 0.177, and 0, respectively.

As expected, a more accurate profile yielded a higher average query success rate than a less accurate profile. In the first case, complete profile accuracy improved the overall success rate from the original 0.193 to 0.387. This was a 100%



a



b

Fig. 7a,b. Effects of profile accuracy on average query success rate. **a** Willingness to help = 0%. **b** Willingness to help = 75%

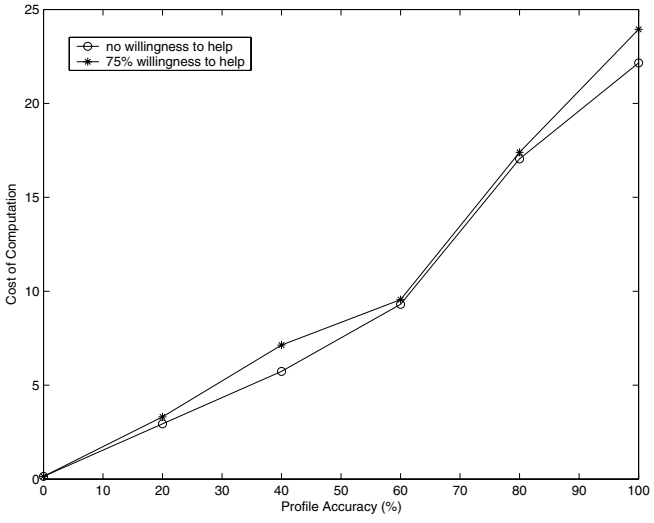


Fig. 8. Effects of profile accuracy on average computing cost

improvement in results obtained using 0% accurate profiles. In the second case, complete profile accuracy together with a 75% probability of help from other cars improved the overall success rate from 0.193 to 0.439 – a 127% improvement.

The average success rate never reached 1. This implies that no car, on average, was able to answer all of the 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 0.439, again in the second scenario. These results were due to the fact that some car asked queries that could not be answered by anyone in its current and/or past vicinity. 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 21.4 mph (34.44 km h). Consequently, some interactions could not be completed because the two interacting peers moved away too soon.

4.3 Profile accuracy vs. computing cost

In the next experiment we measured the amount of effort a car expended on answering its user's queries. We again varied the profile accuracy of each car from 0 to 100% with an incremental step of 20%. For each step we calculated the computing cost by measuring the operations and time that was required for a car to execute the operations. We considered only those operations that a car performed while executing its implicit queries. We did not include the cost caused by other devices requesting help. Like the previous experiment, we looked at two different scenarios. The first scenario set the willingness to help to 0%, while the other scenario set willingness to help to 75%. Figure 8 shows the results.

As expected, the computing cost quickly increased with more accurate profile. With a 0% accuracy level of a profile, no car derived any implicit query. Therefore, a car incurred no computing cost with a 0% accurate profile. For other accuracy levels, the cost increased by almost a factor of two from one accuracy step to the next. This was due to the fact that each car had more implicit queries to answer, half of which could not be answered, as suggested in Sect. 4.2. This could also be validated from the fact that the costs from the two scenarios, 0% and 75% willingness to help, closely resemble each other. Even though cars were willing to help more often in the second case, they still could not provide answers to the *other* half of queries.

4.4 Willingness to help vs. query success rate

After establishing the experimental baseline in Sect. 4.2, we measured how the different willingness-to-help levels improved the average query success rates. We set the profile accuracy to 80%. This meant that each car had almost complete knowledge about the types, time, and location of queries its user was likely to ask. We varied the willingness-to-help level, i.e., car degree of collaboration, from 0 to 100% with incremental steps of 25%. At one extreme, no car was willing to help, while at the other extreme each car was always trying to

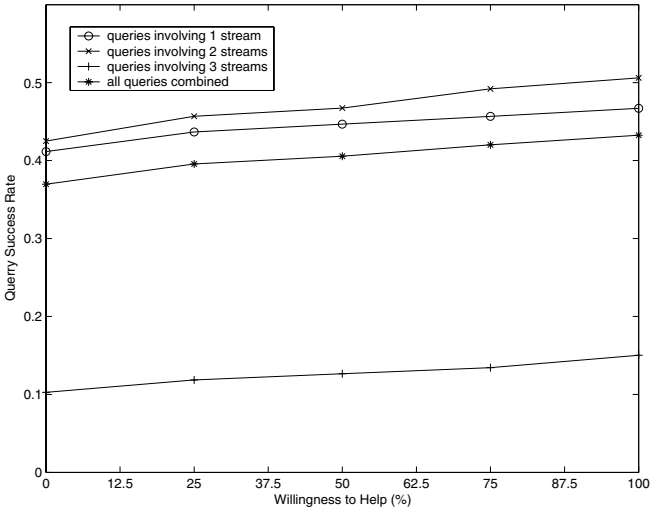


Fig. 9. Effects of willingness to help on average query success rate

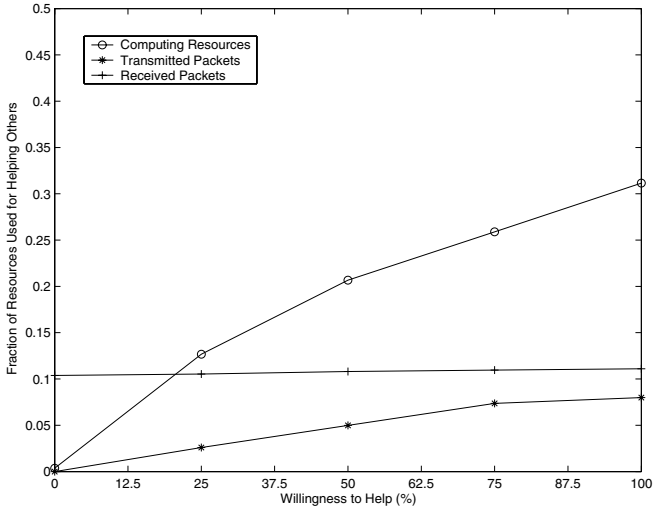


Fig. 10. Effects of willingness to help on average resource cost for helping peers

help whenever it could. We used the same approach as in the first experiment to calculate a partial credit for each car and to plot the results in Fig. 9.

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

Interestingly, the results for queries involving two streams always outperformed queries involving a single stream. This was caused by the smaller number of executed queries requiring two streams and by the fact that many of these queries could reuse locally cached data. Each car asked on average 18 single stream questions, 12 involving two streams and 6 involving three streams. These questions were generated pseudorandomly, but it turned out that some questions involving two streams could always reuse results from other questions. Since the profile accuracy was 80% for this experiment, each car was able to better predict needed data and adapt its caching and querying policies accordingly. Moreover, when defining the queries we required more "hits", i.e., larger answer cardinality, for queries involving a single stream than for complex queries. We gave partial credit to queries returning at least some answer. This, however, was not enough for simple queries to outperform queries involving two streams when profile accuracy was above 60%, as is apparent from results in Sect. 4.2.

4.5 Willingness to help vs. computing and network cost

Although on average every car improved its query success rate, the improvement was not without a cost. In exchange for better success rates, each car paid in terms of its computing cost and additional network traffic. In this experiment, we compared how much of the total resources was, on average, allocated to helping others. We did this by collecting and calculating the fraction of total cost (computing and traffic) that a car used to help other peers. We again set the profile accuracy level to 80% and varied the willingness-to-help level from 0 to 100% with an incremental step of 25%. We collected the total amount of computing costs used by each car and the amount of computing done on behalf of others. The latter cost included 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 collected the total amount of data sent and received by each car. We differentiated among traffic inflicted while helping peers and traffic due to answering the car's own implicit queries. We display the values in Fig. 10. We note that the vertical axis defines a fraction of resources used for helping others, and the displayed range is between 0 and 0.5 only.

The results show that the cost for helping others increased both in terms of computing resources and network traffic; however, the computing costs due to helping others was at most one third of the total resource consumption and the traffic cost was 11% only. Interestingly, the fraction of received data vs. the total amount of received data did not change significantly. The fraction was equivalent to 10.4% when nobody was willing to help and 11.1% when everyone was willing. This was due to the fact that cars received and accepted more *call-for-query* messages. In turn, this caused more requests for subsequent data during the streaming phase; however, other cars experienced a similar pattern. Therefore, both the number of data received when helping others and the total amount of received data increased proportionally.

4.6 Willingness to help vs. query success rate/computing cost

In this experiment, we measured how different levels of willingness to help affected the ratio of utility to cost. In our experiments, a utility was represented by the average query success

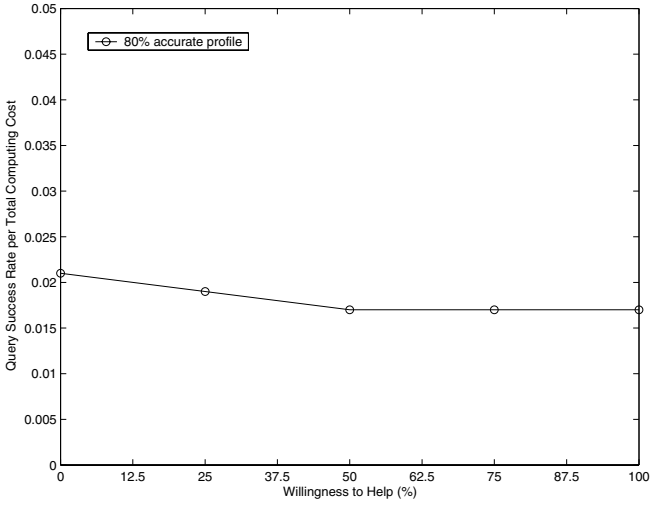


Fig. 11. Effects of willingness to help on average query success rate per computing resources

rate of each car. We then represented the cost value in terms of the total computing cost per car. Again, we set the profile accuracy level to 80% and varied the willingness-to-help level from 0 to 100% with an incremental step of 25%. We collected the average success rate for answering explicit queries and the average computing cost. We show the results in Fig. 11.

The utility to cost ratio remained constant. The ratio value for no willingness to help was 0.021. The value was higher than for other willingness-to-help levels 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 were willing to help, we saw that the ratio value stabilized at 0.017. This was due to the fact that, even though each car executed more computations, and thus had a higher computing cost, each car was also able to improve its overall query success rate. Therefore, the increasing computing cost was well justified by the improving rate of query success rate.

5 Conclusions and future work

We have presented the design and implementation of the CQP protocol that enables devices in pervasive computing environments to locate data sources and obtain data matching their queries. A query can represent a selection scan over one data stream. It can also consist of a join over multiple streams. The protocol combines the traditional concept of nested-loop joins, our previous work on *selection* queries in pervasive computing environments, and the principles of Contract Nets. The features of the protocol enable any device, irrespective of its limited computing, memory, and battery resources, to collaborate with other peers in order to obtain an answer for its queries. We have shown that the protocol improves the overall system performance in terms of 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.

We have not addressed the issues concerning privacy and incentive for devices participating in information exchange. Although these are valid issues, our focus in this paper was primarily on enabling mobile devices to exchange data in pervasive computing environments. Colleagues in our group are developing policy-based security and privacy mechanisms for such environments [23,35]. We also 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. As a simple solution a device may ignore peers it does not understand; however, that is clearly an inefficient solution because it limits the amount of data available in the environment. The presence of an ontology translator will likely become paramount. There has been some work on this problem, but the proposed solutions are still in their preliminary stages and require vast computing and memory resources not available to mobile devices [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.

Acknowledgements. 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 Jeffrey Undercoffer and Sasikanth Avancha for their help.

References

1. Avancha S, D'Souza P, Perich F, Joshi A, Yesha Y (2003) P2P M-commerce in pervasive environments. ACM SIGEcom Exchanges 3(4):1–9
2. Avancha S, Joshi A, Finin T (2002) Enhanced service discovery in Bluetooth. IEEE Comput 35(6):96–99
3. AvantGo. http://avantgo.com/
4. Avnur R, Hellerstein J (2000) Eddies: continuously adaptive query processing. In: Proceedings of ACM SIGMOD
5. Bettstetter C (2001) Smooth is better than sharp: a random mobility model for simulation of wireless networks. In: Proceedings of MSWiM'01
6. Bluetooth SIG. Specification. http://bluetooth.com/
7. Bonnet P, Gehrke J, Seshadri P (2001) Towards sensor database systems. In: Proceedings of MDM
8. Bratmann M (1987) Intentions, plans, and practical reason. Harvard University Press, Cambridge, MA
9. Bukhres O, Morton S, Zhang P, Vanderdijs E, Crawley C, Platt J, Mossman M (1997) A proposed mobile architecture for a distributed database environment. Technical report TR-CIS-596-05, Indiana University-Purdue University Indianapolis
10. Chakraborty D, Perich F, Avancha S, Joshi A (2001) DReggie: semantic service discovery for M-commerce applications. In: Proceedings of the workshop on reliable and secure applications in a mobile environment, SRDS
11. Chandrasekaran S, Franklin M (2002) Streaming queries over streaming data. In: Proceedings of VLDB02
12. Cherniak M, Galvez E, Brooks D, Franklin M, Zdonik S (2002) Profile driven data management. In: Proceedings of VLDB
13. Chrysanthis P, Pitoura E (2000) Mobile and wireless database access for pervasive computing. In: Proceedings of ICDE
14. DAML Services Coalition (2003) DAML-S: semantic markup for Web services. http://www.daml.org/services/

15. Demers AJ, Petersen K, Spreitzer MJ, Terry DB, Theimer MM, Welch BB (1994) The Bayou Architecture: support for data sharing among mobile users. In: Proceedings of the IEEE workshop on mobile computing systems and applications
16. Dey A, Abowd G, Salber D (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Context Aware Comput Hum-Comput Interact J 16:97–166
17. Dou D, McDermott D, Qi P (2003) Ontology translation on the Semantic Web. In: Ontologies, databases and applications of semantics, pp 952–969
18. Franklin M (2001) Challenges in ubiquitous data management. In: Informatics – 10 Years Back. 10 Years Ahead. Springer, Berlin Heidelberg New York, pp 24–33
19. IW Group, Ad-hoc 802.11. http://ieee802.org/11
20. Hellerstein J, Franklin M, Chandrasekaran S, Deshpande A, Hildrum K, Madden S, Raman V, Shah M (2000) Adaptive query processing: technology in evolution. IEEE Data Eng Bull 23(2):7–18
21. Hendler J (2001) DARPA Agent Markup Language. http://www.daml.org/
22. Johnson D, Maltz D (1996) Dynamic source routing in ad hoc wireless networks. In: Imielinski T, Korth HF (eds) Mobile computing. Kluwer, Boston, pp 153–181
23. Kagal L, Finin T, Joshi A (2001) Trust-based security in pervasive computing environments. IEEE Comput 34(12):154–157
24. Kistler JJ, Satyanarayanan M (1991) Disconnected operation in the Coda file system. In: Proceedings of the 13th ACM symposium on operating systems principles, Asilomar Conference Center, Pacific Grove, CA. ACM Press, New York, pp 213–225
25. Kottkamp H, Zukunft O (1998) Location-aware query processing in mobile database systems. In: Proceedings of the ACM symposium on applied computing Atlanta, GA. ACM Press, New York, pp 416–423
26. Lauzac S, Chrysanthis P (1998) Utilizing versions of views within a mobile environment. In: Proceedings of the conference on computing and information
27. Madden S, Franklin M (2002) Fjording the stream: an architecture for queries over streaming sensor data. In: Proceedings of ICDE
28. Perich F, Avancha S, Chakraborty D, Joshi A, Yesha Y (2002) Profile driven data management for pervasive environments. In: Proceedings of the 13th international conference on database and expert systems applications (DEXA 2002), Aix en Provence, France
29. Perich F, Avancha S, Joshi A, Yesha Y, Joshi K (2002) On data management in pervasive computing environments. Technical report, University of Maryland Baltimore County, CSEE
30. Perich F, Joshi A, Finin T, Yesha Y, Joshi K (2004) On data management in pervasive computing environments. IEEE Trans Knowl Data Eng 16(5):621–634
31. Perich F, Joshi A, Yesha Y, Finin T (2003) Neighborhood-consistent transaction management for pervasive computing environments. In: Proceedings of the 14th international conference on database and expert systems applications (DEXA 2003), Prague, Czech Republic
32. Perkins C, Bhagwat P (1994) Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In: Proceedings of ACM SIGCOMM Communications Architectures, Protocols and Applications, October 1994, pp 234–244
33. Perkins C, Royer E (1999) Ad hoc on-demand distance vector routing. In: Proceedings of the 2nd IEEE workshop on mobile computing systems and applications, New Orleans, LA, February 1999, pp 90–100
34. Smith R (1988) The Contract Net protocol: high-level communication and control in a distributed problem solver. In: Bond AH, Gasser L (eds) Readings in distributed artificial intelligence. Morgan Kauffman, San Francisco, pp 357–366
35. Undercoffer J, Perich F, Cedilnik A, Kagal L, Joshi A (2003) A secure infrastructure for service discovery and access in pervasive computing. Mobile Netw Appl 8(2):113–125
36. Zeng X, Bagrodia R, Gerla M (1998) GloMoSim: a library for parallel simulation of large-scale wireless networks. In: Proceedings of the workshop on parallel and distributed simulation
37. Zhang Y, Wolfson O (2002) Satellite-based information services. Mobile Netw Appl 7(1):7–8