Automated Assessment of Aggregate Query Imprecision for Pervasive Computing

Vasanth Rajamani and Christine Julien
Department of Electrical and Computer Engineering
University of Texas at Austin
{vasanthrajamani,c.julien}@mail.utexas.edu

Jamie Payton
Department of Computer Science
University of North Carolina at Charlotte
payton@uncc.edu

***Abstract*—Queries are widely used for acquiring data and services distributed across opportunistically formed mobile networks. However, when queries are executed in such highly dynamic settings, the returned result may not be consistent, i.e., it may not accurately reflect the state of the environment. As such, it can be difficult to reason about the meaning of a returned query result. Reasoning about imperfections in the query response becomes even more complex when in-network aggregation is employed to minimize network communication, since only a single aggregate value is returned. We explore an approach to defining the semantics of aggregate queries in terms of a qualitative description of consistency and a quantitative measure of imprecision associated with a query result. We provide a protocol that performs in-network aggregation while simultaneously generating quality assessments for the returned query result. The protocol provides a convenient mechanism to intuitively interpret the semantics associated with an aggregate query's execution in pervasive environments.**

I. Introduction

The proliferation of laptops, sensors, and other wireless devices has drastically increased the number of data providers embedded in our environment, making pervasive computing a reality. Interconnections among these devices have driven interest in the mobile ad hoc networks that support them, in which devices exchange information and services opportunistically. Despite recent strides in technology, the ability to obtain data, process it, and expose meaningful information to applications in such dynamic networks remains a major software engineering challenge.

Query abstractions make such information-rich environments more accessible to application developers by masking complex details of the network. An important class of queries in pervasive computing networks are *aggregate* queries, where the user receives a summary of the information rather than raw data values. Aggregate queries are particularly popular in these dynamic networks because they provide a basis to perform *in-network aggregation* [1], [2], [3]. In-network aggregation builds on the realization that computation is significantly cheaper than communication in terms of resource consumption. Consequently, it is prudent for individual hosts to aggregate as much of the raw data as possible, thereby reducing the communication overhead associated with collecting query results.

Though the use of queries can simplify application development, the unpredictable and possibly frequent changes in connectivity that characterize pervasive computing networks make it difficult to ensure that a query's result is *consistent*, i.e., the result completely and accurately reflects the environment during query execution. Consider a prototypical pervasive computing application in the construction domain. To intelligently perform asset and operations management, a site supervisor may want to monitor the amount of some material present on the site (whether stationary or mobile, e.g., in a delivery truck) to determine when to order more of the material. Each pallet of the material may be tagged with a device that represents the count (or weight) of the material present on that pallet. The supervisor may issue a simple aggregate query that returns the sum of all of this material across the site. While a query is executing, pallets are being moved around the site, which may cause a particular pallet's value to be counted more than once or not counted at all. This results in the total reported to the site supervisor being inconsistent with the actual total of the material on the site. Traditionally, delivering query results with strong consistency semantics is achieved through distributed locking protocols, which are ill-suited for use in dynamic networks. Most existing practical solutions, therefore, rely on "best-effort" queries, which make no guarantees about the quality of the result. When a query response represents the ground truth to an arbitrary degree, it is difficult for an application to know with certainty how to use the results, and it is virtually impossible to automate responses to uncertain results. Thus, a fundamental requirement for building applications that depend on aggregate queries is the ability to interpret the imperfections associated with retrieving data from dynamic networks.

In this paper, we formally define semantics for a basic set of aggregate queries (min, max, count, sum, and average) and demonstrate a query processing protocol that can automatically attach an intuitive indicator of the semantics achieved to the query result. We define query semantics qualitatively in terms of consistency and quantitatively in terms of numeric imprecision in the query result. Consistency semantics allow a user to reason about whether different types of environmental change impacts his query result, while numeric bounds provide a concrete measure of their impact.

Our contributions to examining the imprecision of aggregate queries are twofold. We define a conceptual model for estimating numerical bounds that define a query's imprecision and demonstrate how our framework can be used to express the semantics of aggregate queries (Sections II and III). Second, we develop a protocol that computes aggregate queries while simultaneously assessing the semantics achieved by the query execution; this assessment is attached to the result to support reasoning about the returned value (Section IV). To demonstrate feasibility, we provide a prototype implementation that we evaluate through simulation (Section V). Our work enables software designers to implement pervasive computing applications that automatically interpret the robustness of data collected in these inherently dynamic networks, which can be instrumental in facilitating the adoption of pervasive computing applications.

II. Background: Expressing Query Consistency in Dynamic Networks

Our previous work defined a new perspective on discrete query consistency in dynamic networks [4]. The result is a query processing model that can easily express mobility and time as state transitions and a set of consistency semantics expressed using the model. Below, we describe the fundamentals of our approach to modeling a dynamic network; we use this foundational model to capture and present the semantics of aggregate queries.

A. Modeling the Environment

We view a mobile ad hoc network for pervasive computing applications as a closed system of hosts; a host is represented as a tuple (ι, ν, λ), where ι is a unique identifier, ν is the host's data value, and λ is its location. The global abstract state of the network, a *configuration*, is a set of host tuples. Given a specific host $\bar{h}$ in a configuration, an *effective configuration* (E) is the projection of the configuration with respect to the set of hosts that are *reachable* from $\bar{h}$. Reachability is often defined in terms of physical network connectivity, captured by a relation that conveys the existence of a (possibly multi-hop) network path between a pair of hosts. We use a binary logical connectivity relation $\mathcal{K}$ to express the existence of a direct (one-hop) communication link between two hosts. Reachability is defined as the reflexive transitive closure $\mathcal{K}^*$. An example relation that represents a physical connectivity model with a circular, uniform communication range can be defined using host tuples' location fields:

$$(h_1, h_2) \in \mathcal{K} \Leftrightarrow |h_1 \uparrow 3 - h_2 \uparrow 3| \leq d$$

where $\uparrow 3$ refers to a tuple's third field (the host's location), and d is the communication range.

We model dynamics as a state transition system in which the state space is defined by the set of possible system configurations, and transitions are defined as configuration changes. Sources of configuration change include: 1) *variable assignment*, in which a single host changes its data value, ν and 2) *neighbor change*, in which the change in a host's location impacts the network connectivity. A neighbor change occurs when a host is no longer connected to a previous neighbor or becomes connected to a new neighbor.

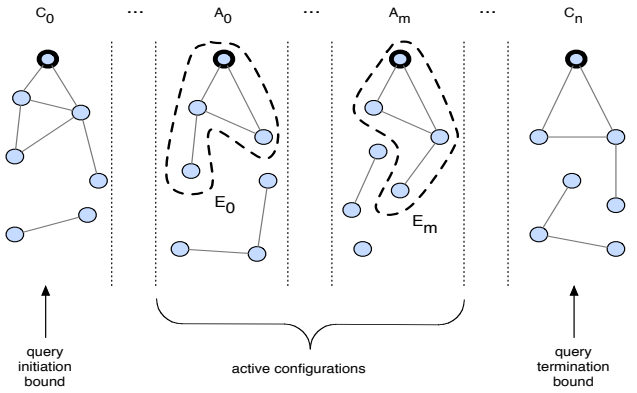

Fig. 1. Effective Active Configurations

B. Defining Discrete Queries and Results

Consider the configuration sequence in Fig. 1. A single query may span such a sequence starting with the issue of a query (the *query initiation bound*, C_0) and ending when the result is delivered (the *query termination bound*, C_n). Since there is processing delay when issuing a query to and returning results from the network, we define a query's *active configurations* as those within $\langle C_0, C_1, \ldots C_n \rangle$ during which the query actually interacted with hosts in the network. Every value contributing to a query's result must be a data element present in some active configuration. Moreover, only the query issuer's effective configurations (containing reachable hosts) can contribute to a query's result. The relevant configurations, then, are the *effective active configurations*, $\langle E_0, E_1, \ldots, E_m \rangle$, depicted in Fig. 1.

A query is a function from a sequence of effective active configurations to a set of host tuples that comprise the result. Since a configuration is simply a set of host tuples, this model lends itself to a straightforward expression of a query's result as a configuration, simplifying the expression of the consistency of those results. For example, a strongly consistent result ρ may contain values from a single effective active configuration E_i, i.e., $\rho \subset E_i$.

While this model considers the impact of environmental conditions on the achievable consistency semantics of simple discrete queries, it does not capture how in-network aggregation impacts query results. In the next section, we explore what kinds of consistency semantics can be provided for queries processed using in-network aggregation and offer an approach to present aggregate query results that convey their associated consistency semantics in an intuitive way.

III. Integrating Aggregation and Consistency

In this section, we explore the integration of approaches to in-network aggregation and the definition of consistency

semantics for queries issued in pervasive computing environments. We introduce a model for aggregate queries that applies an in-network aggregation operator and returns a bounded aggregate value as a result. These bounds convey the degree to which the provided query result reflects the ground truth (i.e., the state of the environment during query execution). The bounded aggregate is simply a triple: $[L, A, U]$, where L is a lower bound, U is an upper bound, and A is the aggregate value computed over the results available in the network during the entire query execution, i.e., A is an aggregate computed over the elements in the set $\bigcap_{i=0}^{m} E_i$.

A. Consistency Classes: Comparability

Our previous work used consistency semantics to communicate to a querier the degree to which a query result is known to match the ground truth in the network [4]. Specifically, we defined a set of semantics that lie between the common atomic (i.e., an exact representation of the ground truth) and weak (i.e., best effort) semantics. We demonstrated that, in many situations, the association of this meta-information with the query result aids an application in understanding the usefulness of the result.

We build on our previous work to associate meta-information with aggregate queries. Our consistency semantics can be divided into two classes: *comparable* and *non-comparable*. In the first class, comprised of the stronger consistency semantics, all the elements of the computed aggregate are guaranteed to have existed at the same time. In the parlance of our formal framework, all of the results returned existed in the same configuration. In the weaker class of semantics, all of the component values in the results set are guaranteed to have existed at some time during the query execution, but nothing can be said about the temporal relationships between the individual items in the result. With respect to an aggregate query, a comparable consistency semantic communicates that all of the elements that were aggregated together existed at the same time. Conversely, a non-comparable semantic only guarantees that all of the results that were aggregated existed at some time during query execution.

In our construction example, consider a query that tries to determine the truck with fewest pallets. If material is transferred from one truck to another during query execution, then the answer may not in fact point to the truck with the least amount of material. Because a configuration change occurs (transfer of material between trucks), the query may aggregate values from each truck that are not comparable, e.g., the value from one truck before the transfer and the other truck after the transfer. Conversely, if there was no configuration change, one can be assured that the answer returned is the correct one. Adding this piece of semantic information incorporates a great deal of clarity to the query response. For the rest of this section, we treat these two groups (comparable and non-comparable) in the same manner; in our evaluation, we revisit this concept to see how the classes impact the consistency of aggregation.

B. Bounding Aggregates

We first provide a generic framework for expressing aggregate query results. We relax our definition of a query result (ρ) to allow for computation of imprecision bounds. In this work, instead of being a set of host tuples, our query result has two components: $\langle \mathcal{A}, Excess \rangle$, where $\mathcal{A}$ is the aggregate result given a particular aggregation operation (examples are provided in the rest of this section), and *Excess* is a set containing tuples of the form: $\langle a/d, h \rangle$, where the first component is either a or d, indicating that the tuple represents an addition or a departure, and the second component h is a host tuple. In the aggregation examples below, we do not use the a and d designations explicitly. However, these labels are necessary for computing the comparability class of a query result (discussed in Section V), and future aggregation operations may use them directly. Each element in *Excess* was present in at least one of the query's effective active configurations but missing from another:

$$e \in Excess \Rightarrow \langle \exists i, j \quad : \quad 0 \leq i, j \leq m \wedge i \neq j \\ :: \quad e \uparrow 2 \in E_i \wedge e \uparrow 2 \notin E_j \rangle^{1}$$

If a particular host's value changed multiple times or the host moved multiple times during execution, there may be multiple tuples in *Excess* for the same host. This has to be handled with care for *duplicate-sensitive* aggregation operators like sum and average that have different results when the value from a single host is used multiple times. Duplicate-sensitive aggregates must explicitly account for these duplicate items [2].

C. The Semantics of Aggregation

Every query result comprises a conservative estimate of the aggregate ($\mathcal{A}$) along with a means to measure its imprecision. For the latter, each aggregation type includes a different method for using the *Excess* set to calculate bounds. We next look at several types of aggregation and show how bounds are calculated for each to define the triple $[L, A, U]$.

1) Set Union Aggregation: In our previous work, we effectively used set union to return results to the query issuer. This union operation can be expressed as an aggregation operation over the elements in the network, where $\mathcal{A}$ contains the stable subset of query results, i.e., results from those hosts that experienced no changes during the execution of this query:

$$\mathcal{A} = \langle \text{set } h : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h \rangle$$

Excess contains all other values that were present at some point during query execution but were either added or removed (or both) during execution. We define the set of host tuples that are part of the *Excess* set as:

$$\text{E} = \langle \text{set } e : e \in Excess :: e \uparrow 2 \rangle$$

[1] In the three-part notation: $\langle \text{op } quantified\_variables : range :: expression \rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which op is applied, yielding the value of the three-part expression.

A set union query returns $[-, \mathcal{A}, \mathcal{A} \cup E]$, where $-$ indicates that there is no lower bound (as $\mathcal{A}$ is a highly conservative estimate). When no changes have occurred in the network, *Excess* is empty, and, therefore, so is E (i.e., the upper bound is the same as the estimate). In our construction example, the set $\mathcal{A}$ would consist of the values for all pallets of material on the site whose data value or network connectivity that did not change their network connectivity or data value during query execution. The upper bound, in contrast, will contain data values for pallets of materials that may have been delivered or consumed during query execution.

2) Minimum/Maximum Aggregation: The simplest aggregation types are minimum and maximum. When the aggregate query returns, $\mathcal{A}$ contains the minimum (or maximum) value calculated as part of the in-network aggregation. In our construction example, a minimum aggregation can tell the site supervisor what area of the site may be lacking a particular material. This is effectively the minimum value of the stable portion across all of the configurations during which the query executed:

$$\mathcal{A} = \langle \text{MIN } h \in E_i : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h \uparrow 2 \rangle$$

To compute the bounds on this estimate of the minimum, we need only to inspect the elements in *Excess*. If any result in this set is less than the estimated minimum ($\mathcal{A}$), it is the lower bound. That is:

$$\text{MIN}_{excess} = \langle \text{MIN } e : e \in \textit{Excess} :: (e \uparrow 2) \uparrow 2 \rangle$$

Informally, the right hand side says: take the minimum of the values that are stored in host tuples that are represented in *Excess*. A minimum aggregate query returns $[min(\text{MIN}_{excess}, \mathcal{A}), \mathcal{A}, -]$. When the *Excess* set is empty or does not contain a value less than $\mathcal{A}$, this query returns $[-, \mathcal{A}, -]$.

3) Counting Aggregation: A counting aggregate should return the number of hosts present. The counting aggregate is the first of our aggregates that is *duplicate sensitive*, and so it should attempt to avoid counting a value for the same host more than once. When the aggregate query returns, $\mathcal{A}$ contains the number of items that were present in every configuration.

$$\mathcal{A} = \langle +h : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: 1 \rangle$$

We use *Excess* to place an upper bound on the number of items that *could* have been present. We use the host's identifier to ensure that a value from the same host is not over counted.

$$\text{C}_{excess} = \langle +i : \langle \exists e :: e \in \textit{Excess} \wedge (e \uparrow 2) \uparrow 1 = i \rangle :: 1 \rangle$$

Because $\mathcal{A}$ is a conservative estimate, it is not possible to underestimate the counting aggregation. Therefore, the result returned to the querier is $[-, \mathcal{A}, \mathcal{A} + \text{C}_{excess}]$. If *Excess* is empty, the upper bound will be the same as the estimate. The site supervisor can use the conservative estimate $\mathcal{A}$, if he is interested in the number of pallets of material guaranteed to be on site. Alternatively, he can use the upper bound if he is concerned about avoiding left-over material.

4) Summation Aggregation: An aggregate summation query should represent the sum over all hosts receiving the query. $\mathcal{A}$ contains the sum of items that were present in every configuration.

$$\mathcal{A} = \langle +h : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h \uparrow 2 \rangle$$

The members of *Excess* are used to place bounds on a sum that is calculated using items that *could* have been present during query execution. In this case, the upper and lower bounds are calculated based on the worst possible scenario. We create all of the different permutations of elements of *Excess*, combine their sums with $\mathcal{A}$, and take the minimum of these (if less than $\mathcal{A}$) as the lower bound and the maximum of these (if greater than $\mathcal{A}$) as the upper bound. In calculating these permutations, we must also suppress duplicates; any potential sum should include only one value from any particular host. We first define a set of sets, $\mathcal{P}$ which contains all relevant permutations of the elements of *Excess*. This is effectively a duplicate sensitive power set of *Excess*. Formally, $\mathcal{P}$ contains all possible sets p that satisfy the following constraints:

$$|p| \neq 0$$
$$\langle \forall h : h \in p :: \langle \exists e : e \in \textit{Excess} :: e \uparrow 2 = h \rangle \rangle$$
$$\langle \forall h_1, h_2 : h_1 \in p \wedge h_2 \in p :: h_1 \uparrow 1 \neq h_2 \uparrow 1 \rangle$$

Informally, p is a legal permutation if 1) p is not empty; 2) every element in p corresponds to an element in *Excess*; and 3) no two elements in p are from the same host. Then U_{SUM} can be defined as:

$$\text{U}_{\text{SUM}} = \langle \max p :: p \in \mathcal{P} :: \langle +h : h \in p :: h \uparrow 2 \rangle + \mathcal{A} \rangle$$
$$\text{U}_{\text{SUM}} = max(\text{U}_{\text{SUM}}, \mathcal{A})$$

L_{SUM} is defined similarly, using min instead of max. A summation aggregation query returns $[\text{L}_{\text{SUM}}, \mathcal{A}, \text{U}_{\text{SUM}}]$.

5) Average Aggregation: Our final aggregation type, average aggregation, is similar to summation. As with all of the other basic aggregation types, $\mathcal{A}$ is the aggregate over all of the stable results. However, to be able to recalculate averages for bound computations, we must also keep track of how many results contribute to the aggregate average. So in this case, $\mathcal{A}$ is a tuple: $\mathcal{A} = \langle \mathcal{A}', C \rangle$, where $\mathcal{C}$ is simply a count of contributors to $\mathcal{A}'$, and the aggregate value is calculated as:

$$\mathcal{A}' = \langle \text{avg } h : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h \uparrow 2 \rangle$$

We use the elements of the *Excess* set to calculate all of the potential average values after removing duplicates. That is, we define the same set of permutations, $\mathcal{P}$ and calculate the upper bound, U_{AVG} as:

$$\text{U}_{\text{AVG}} = \left\langle \max N : N \in \mathcal{P} :: \frac{\langle +p : p \in N :: p \uparrow 2 \rangle + \mathcal{A}'}{C + |N|} \right\rangle$$
$$\text{U}_{\text{AVG}} = max(\text{U}_{\text{AVG}}, \mathcal{A})$$

L_{AVG} is defined similarly using min. Assuming $\text{L}_{\text{AVG}} < \mathcal{A}'$ and $\text{U}_{\text{AVG}} > \mathcal{A}'$, an aggregate average query returns $[\text{L}_{\text{AVG}}, \mathcal{A}', \text{U}_{\text{AVG}}]$. In our construction site example, if the

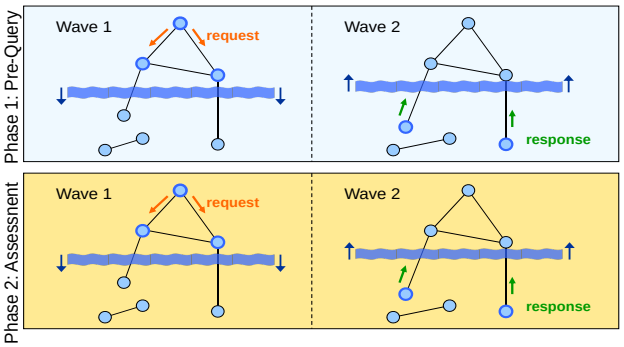

Fig. 2. Protocol Phases and Waves

amounts of available material on queried pallets varies during query execution, the site supervisor can use the range provided by the upper and lower bounds to determine how much credibility to place in his query response.

IV. Assessing Aggregation Imprecision

We next present a query execution protocol that performs in-network aggregation while continuously monitoring environmental changes during query execution to dynamically calculate error bounds on the aggregate result.

A. Protocol Overview

Query protocols that provide well-defined semantics lock data values to ensure strongly consistent results, but these protocols are often impractical in dynamic networks. Our protocol can provide different semantics under different operating conditions, while at the same time dynamically assessing the imprecision of the result. Our self-assessing protocol makes an initial examination of data values that will be accessed during aggregate computation and maintains state about the values during query processing. This state information is used to determine the consistency class and to compute bounds associated with the result.

We employ controlled floods in two phases—*Pre-Query* and *Aggregation Assessment*. The Pre-Query phase establishes the set of query participants (setting the *query initiation bound*). Additionally, it computes and caches partial aggregates at each participant. The Aggregation Assessment performs the in-network aggregation and provides a conservative aggregate result. Establishing participants in the first phase provides a reference that our protocol can use to observe changes that impact the query execution semantics, and the cached partial aggregates can be used to provide error bounds on the returned aggregate.

As shown in Fig. 2, each phase consists of two *waves*: the first disseminates a request, and the second returns the response. Each node that forwards a request waits for its children to respond before sending its own response. Participating hosts monitor changes in state that can impact the query's imprecision. Changes in data values or connectivity that occur behind the second wave of the Pre-Query and in front of the second wave of the Aggregation Assessment may impact the semantics of the aggregate query. Table I describes how such changes relate to the computation of and the consistency class achieved by the aggregate; data value changes are modeled as a departure followed by an arrival.

Next, we provide a detailed description of our self-assessing aggregate query execution protocol. Since flooding an entire network can be prohibitively expensive, we constrain the flood using the query's logical connectivity relation $\mathcal{K}$. This is similar in fashion to other constrained flooding approaches [5], [6]. We assume each host can detect connection and disconnection of its neighbors. Finally, we assume reliable message delivery [7], [8].

B. Self-Assessing Aggregation

Fig. 3 shows the protocol's state variables for node A; each query has a duplicate set. The protocol's behavior is defined using I/O Automata notation [9]. We show the behaviors of a single host, A, indicated by the subscript A on each behavior. Each *action* (e.g., $ParticipationRequestReceived_A(r)$ in Fig. 4) has an effect guarded by a precondition. Actions without preconditions are *input actions* triggered by another host. Each action executes in a single atomic step. We abuse notation slightly by using, for example, "send *ParticipationRequest*(r) to *Neighbors*" to indicate a sequence of actions that triggers *ParticipationRequestReceived* on each neighbor.

id	– A's unique host identifier
neighbors	– A's logically connected neighbors (given $\mathcal{K}$)
membership	– boolean, indicates A is in the query
monitoring	– boolean, indicates A is preparing result
parent	– A's parent in the tree
replies-waiting	– neighbors still to respond
participants	– A's participating descendants
op	– query's aggregation operator
data-val	– A's local data
estimated-val	– estimate of applying *op* on A's subtree
actual-set	– data values of A and A's neighbors
actual-val	– conservative aggregate value computed
count	– number of nodes in subtree rooted at A
child-yield(x)	– contributions of child x to the aggregate

Fig. 3. State Variables for Protocol

1) Pre-Query Phase: Conceptually, the Pre-Query establishes a baseline that can be used to determine how well the query's aggregate result compares to the "ground truth." Pre-Query establishes the set of participants by creating a spanning tree rooted at the query issuer. The query issuer initiates the phase by sending a *ParticipationRequest* message. Fig. 4 depicts a host's behavior upon receiving such a request. Generally, each host forwards the request to its neighbors, waiting for their responses before replying.[2] This wave continues until

[2]If a node receives more than one participation request for a query (possibly along different communication paths), the node cancels the duplicate request and notifies the sender, removing the node from the sender's subtree. This action is omitted for brevity.

Changes	Analysis	Impact
No changes	all possible results are included in aggregate	comparable consistency; no bounds
Departing Participants	not all participants contributed; aggregate is computed from results that existed at the same time	comparable consistency; bounds include values of departed nodes
Departing and Arriving Participants	not all participants contributed; aggregate is computed from results that did not necessarily co-exist	non-comparable consistency; bounds include departed and added values, prevent double-counting

TABLE I
IMPACT OF DYNAMICS

a participation request reaches a host at the network boundary (defined by $\mathcal{K}$). A boundary host caches its current data value as an estimated aggregate result. All nodes also compute the number of nodes currently in their subtree; for a boundary node, this *count* is one. The boundary node then initiates the second wave of the Pre-Query by packaging the estimated aggregate and the counter into a *ParticipationReply* message that it sends to its parent.

```
ParticipationRequestReceived_A(r)
 Effect:
    if ¬membership then
        membership := true
        parent := r.sender
        op := r.op
        if (neighbors − r.sender) ≠ ∅ then
            for each B ∈ (neighbors − r.sender)
                send ParticipationRequest(r) to B
            replies-waiting := neighbors − r.sender
        else
            estimated-val := data-val
            count := 1
            send ParticipationReply to parent
    else
        send CancelParticipationRequest to r.sender
```

Fig. 4. *ParticipationRequestReceived* Action

When a host receives a *ParticipationReply* (Fig. 5), the reporting child is considered to be committed to the query. Any future changes related to this child impact the quality of the returned aggregate result. On receiving this message, a host locally stores the child's estimated aggregate and count values. After all its children have reported, it computes the partial aggregate for its subtree (*estimated-val*), and forwards it to its parent. Locally cached values from children can be later used to assess a departed child's contribution to the imprecision bounds.

The Pre-Query ends when the querier has collected *ParticipationReply* messages from all of its children. The estimates of aggregates established in Pre-Query allow each node to capture a local "snapshot" of the environment and aid in calculating the aggregate query result's imprecision.

2) Aggregation Assessment Phase: Once the Pre-Query completes, the query issuer initiates the Aggregation Assessment phase. As before, a parent waits for responses from all of its children before sending its own reply, and boundary hosts initiate the second wave of the phase. Only results from nodes that were present in both phases of the protocol and

```
ParticipationReplyReceived_A(r)
 Effect:
    replies-waiting := replies-waiting−r.sender
    participants := participants ∪ {r.participants}
    child-yield(r.id) := (r.estimated-val, r.count)
    if replies-waiting = ∅ then
        child-yield(id) := (data-val, 1)
        (estimated-val, count) := op(child-yield(∗))
        if r.requester ≠ id
            send ParticipationReply to parent
        else
            send Query to neighbors
```

Fig. 5. The *ParticipationReplyReceived* Action

without any value change can contribute to the final aggregate response. This value for each subtree is stored in *actual-val* at the root of each subtree and propagated up the tree to the query issuer. The *actual-val* (i.e., $\mathcal{A}$ in our framework) returned to the user reflects a conservative aggregation of values that were present during query execution, as defined by the consistency class associated with the query's execution. Once the query issuer has received *QueryReply* messages from all of its children, it prepares the result. The protocol dynamically assesses and tags a query result with bounds indicating the aggregate query's imprecision and the achieved consistency class. In this work, we focus more on how interpretation of changes can be used to compute the imprecision bounds.

3) Handling Dynamics: If a host detects that one of its children departs after the participants are established but before the node has replied in the Aggregation Assessment phase, then the *actual-val* returned will not be computed using all values that existed during query execution. Consider a minimum aggregate query. If a node with the smallest value departs the network during the query, the computed aggregate result will not reflect the smallest value that existed in the network. We must therefore include the departed value in the bounds associated with the computed aggregate. To do so, the parent node calculates its estimated value for the departed node's subtree using the value stored in *child-yield* and sends this value up the tree in an *EstimateReply* message. Similarly, an inconsistency arising due to a node adding itself to the network should be reflected in the bounds of the aggregate result. When a node detects such an addition, it sends a new *EstimateRequest* message to the added node. On receiving such a message, a node creates an *EstimateReply* containing its data value and unique node identifier that is forwarded to the query issuer. We model a data value change as a

$NeighborAdded_A(B)$
Precondition:
$connected(A, B) \wedge B \notin neighbors$
Effect:
$neighbors := neighbors \cup \{B\}$
if *membership* **then**
if $\neg monitoring \wedge (replies\text{-}waiting \neq \emptyset)$ **then**
send $ParticipationRequest$ to B
$replies\text{-}waiting := replies\text{-}waiting \cup \{B\}$
else
send $EstimateRequest$ to B

$NeighborDeparted_A(B)$
Precondition:
$\neg connected(A, B) \wedge B \in neighbors$
Effect:
$neighbors := neighbors - \{B\}$
if *membership* **then**
if $B = parent$ **then**
[reset state]
else if $\neg monitoring \wedge (replies\text{-}waiting \neq \emptyset)$ **then**
$replies\text{-}waiting := replies\text{-}waiting - \{B\}$
else if $\neg monitoring$ **then**
$participants := participants - \{B\}$
send $EstimateReply(B)$ to $parent$
else
$replies\text{-}waiting := replies\text{-}waiting - \{B\}$
send $EstimateReply(B)$ to $parent$

Fig. 6. Actions for Neighbor Changes

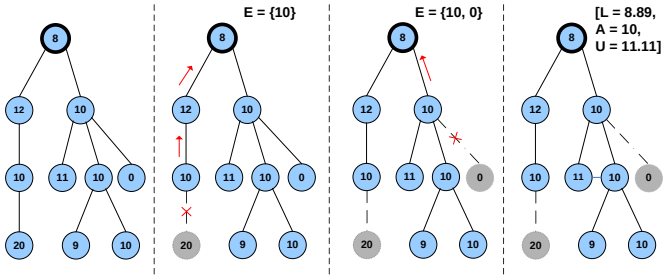

Fig. 7. Example Average Query

node departure followed a node addition. An estimate for the departed node and the newly updated added node will reach the query issuer as described above. Since the *EstimateReply* includes the unique node identifier of the value contributer, we can perform post-processing at the query issuer to account for duplicate values when necessary. These actions are shown in Fig. 6.

Fig. 7 illustrates network dynamics for an aggregate query that computes an average value. In the first three scenarios, the nodes are participating in the Aggregation Assessment phase but have not yet sent their replies. The first graph shows the set of established participants. The second graph illustrates that a node with a value of 20 departs. The neighboring node detects the departure and uses locally cached values to compute an *EstimateReply* sent to the query issuer. In the third graph, a node with a value of 0 departs, and a similar process ensues. The final graph depicts the final computation of the protocol. The values that were present and unchanged throughout the query's execution are used to perform in-network aggregation, which results in the average $\mathcal{A} = 10$.

Computation of bounds on the aggregate result is performed using the *EstimateReply* messages (arrows in Fig. 7) which carry the "excess" values to the querying host. Specifically, we calculate all potential average values after removing duplicate contributions. In this example the lowest possible average includes the node with value 0 that departed, yielding an average of 8.89. The highest possible average is 11.11; this is the average yielded by the configuration that included the departed node with value 20 but not the departed node with value 0. Therefore, the numerical result for this average aggregate query would be [8.89, 10, 11.11]. In addition, because the query issuer has recorded *EstimateReply* messages that indicate node departures but no additions, the query achieved *comparable* consistency, and the estimated bounds were computed using comparable values, i.e., values that existed at the same time.

In general, the result returned to the query issuer includes the actual aggregate result, bounds on the result, and an assessment of the result's consistency semantics. Application developers can use the protocol in different network settings and receive different query replies and their associated semantics and bounds. This enables users to intuitively reason about the uncertainty associated with their query responses.

V. Evaluation

We have prototyped our protocol using OMNeT++ and its mobility framework [10], [11]. Our protocol executes a query, establishes its consistency, and provides bounds on the response. We evaluate how well our reference protocol accomplishes this task in different situations.[3]

We executed our protocol in a 1500m x 1500m rectangular area with 50 nodes (a network of moderate density and good connectivity). The nodes move according to the random mobility model, in which each node is initially placed randomly in the space, chooses a random destination, and moves in the direction of the destination at a given speed. Once a node reaches the destination, it repeats the process. We used the 802.11 MAC protocol. When possible, 95% confidence intervals are shown on the graphs.

A. Using Aggregate Imprecision: An Application Scenario

We first demonstrate how an application might apply our protocol using our previous construction asset management application example. We define an example scenario in which materials are delivered to the site over a period of time and then consumed. Devices attached to the palettes of material measure the amount of available material, and the palettes may move around the site as they are posistioned for use. Data for this scenario is generated by initially assigning nodes data values based on a Gaussian distribution with a mean and variance of 0 and 20 respectively. The value increases in steps of 10 for 50 seconds (representing material delivery);

[3]The source code and settings used are available at http://mpc.ece.utexas.edu/AggregationConsistency/index.html

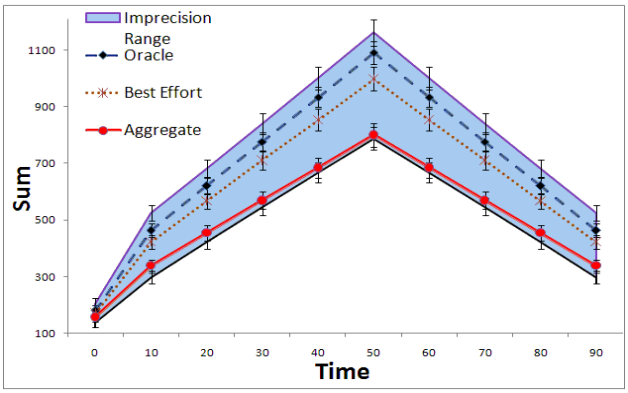

Fig. 8. An Application Example

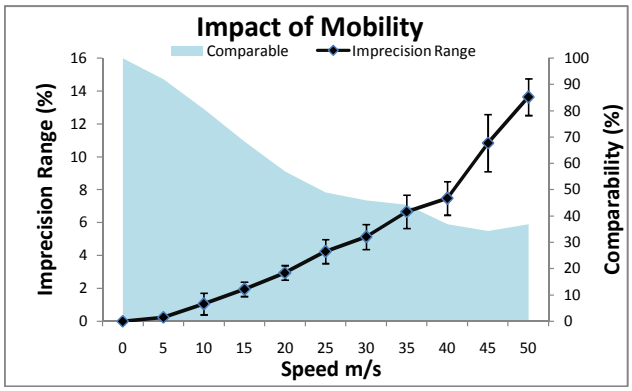

Fig. 9. Imprecision Change with Dynamics

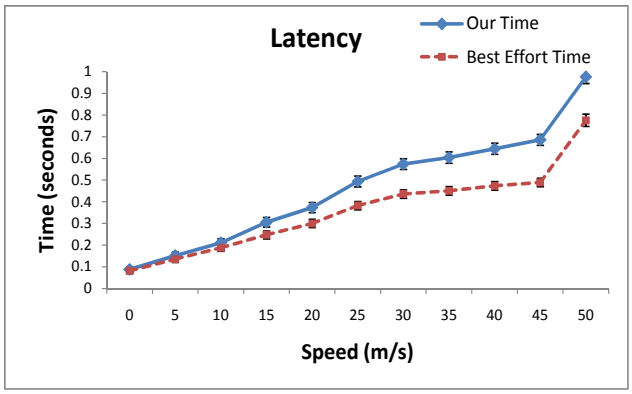

Fig. 10. Protocol Overhead

then reduces to its original value in steps of 10. We model a dynamic scenario where all devices are mobile and move at 20 m/s. A node selected to be the query issuer requests the sum of the amount of material across the site every 10 seconds. Fig. 8 plots three lines: our protocol's conservative estimate of the aggregate value ($\mathcal{A}$); the *actual* sum of the material amounts at each time slot (the "oracle"); and the summation value that would be calculated by an existing state-of-the-art best-effort querying technique (e.g., [2]). The shaded region contains values between our upper and lower bounds, and it represents the imprecision range associated with our result. The relationships between the lines are of more interest than the data itself.

The plot confirms that best effort solutions often differ (sometimes significantly) from the truth. Our approach provides both a stated consistency semantic (here, the results are *non-comparable*) as well as bounds of imprecision in the reported aggregate. While our aggregate result is conservative and may also differ from the oracle, the range gives the user an indication of the space of all possible answers. The behavior exhibited here is true for all other operators, and these graphs are omitted for brevity. The upper and lower bounds typically encapsulated the oracle value. In relatively static networks, the aggregate value ($\mathcal{A}$) returned is close to the oracle, and the distance between the upper and lower bounds is small. In highly dynamic networks, the aggregate value tends to be closer to the lower bound, and the imprecision range is wider. An exception to this generalization is in very dense networks, where the oracle tends to be higher than our upper bound due to the significant numbers of packets dropped because of contention in the wireless medium.

B. Impact of Mobility

While the previous results demonstrate our protocol's capabilities, we next show how our imprecision measures change with changing network conditions. We evaluated all five of our aggregation operators but use the average operator to elucidate the impact of mobility. The shaded region in Fig. 9 represents the percentage of query responses that were *comparable* as node speed increased. When the network is static, all the query results are *comparable* because the configuration remains the same throughout query execution. However, as mobility increases, dynamics render the results increasingly *non-comparable*. Fig. 9 also shows the impact of mobility on the aggregate imprecision. The line shows the size of the imprecision range obtained by executing our protocol[4]. As mobility increases, the imprecision range widens. Using a best effort protocol can produce responses that vary significantly from the ground truth; we found that the difference between the oracle and best effort responses vary from 0 to 27%. This shows that, in a highly dynamic environment, the query responses of current protocols can be a poor reflection of the ground truth. By explicitly exposing the degree of uncertainty, we allow applications to reason about their received results effectively.

C. Protocol Performance

Finally, we measured our protocol's performance with respect to overhead (the number of bytes transmitted to evaluate a query) and latency (the time to process a query). With respect to overhead, our protocol effectively runs the best effort protocol twice, so the overhead is about double that

[4]The line in Fig. 9 plots the difference between the upper and lower bounds normalized by the answer returned by a best effort protocol.

of the best effort protocol. In addition, our overhead increases slightly with mobility due to the additional information sent with the aggregate value and used to calculate the bounds. This graph is omitted for brevity. Fig. 10 shows that our protocol does not incur significant delays in reporting the semantics and imprecision bounds. Although our protocol is clearly more expensive than current best-effort techniques, it provides significantly more information to enable applications to effectively reason about results of aggregate queries in dynamic pervasive computing environments.

VI. Related Work

Data consistency has been explored in several areas such as databases, file systems and memory management, and consistency has been expressed in terms of precise metrics defining numerical error, order error, and services [12]. The authors explore elements in the design space between strong consistency and no consistency for data access in replicated file systems. In contrast, we focus on aggregation of (non-replicated) data items and provide an accuracy range for a given query result. Similarly, *completeness* describes the probability that a node's data will be included in a query result [13]. This has been applied only to a distributed shared memory system with no concern for mobility.

Recent work has explored the impact of in-network aggregation on consistency, defining the "single site validity" principle, in which a query result appears to be equivalent to an atomic execution from the query issuer's perspective [14]. In essence, a query response is valid if every host that was connected to the querier during the querying interval contributed. In a complementary manner, we categorize contributions from nodes depending on the type of environmental change (host addition or deletion) which allows us to provide a range of semantics. More recent work exposes inconsistency in query results through network imprecision [15]. This work, like ours, provides a network monitoring approach that reports the network imprecision with the query result. Information indicating network imprecision includes the number of reachable nodes and the number of potentially over-counted nodes. Both these approaches are designed for static networks. Since the impact of dynamics is significant, these architectures are not feasible in mobile the ad hoc networks that commonly support pervasive computing. In addition, we combine a measure of consistency with a measure of imprecision providing a more complete, yet intuitive, way to convey query semantics.

Researchers in sensor networks have explored a model-driven approach to query processing for the purposes of energy conservation [16], [17]. Each node constructs a local model of the data in the network and estimates the error in the model. If the estimated error of the local model is acceptable, a node conserves energy by querying the local model. Another popular approach in the sensor network community is to provide approximate answers that trade accuracy for energy efficiency [18]. We calculate the error on demand at query time because pro-actively maintaining local models can be very expensive in mobile environments.

VII. Conclusions

In this paper, we presented an approach to defining semantics for aggregate queries issued in dynamic pervasive computing networks. Our approach combines a qualitative measure of consistency and a quantitative measure of imprecision to provide a more intuitive way of communicating the meaning of a query's aggregate result. To make this approach more accessible to developers of query-based applications, we developed an automated process for query execution that simultaneously assesses the aggregate query's semantics while performing in-network aggregation, and returns the assessment along with the aggregate query result.

Acknowledgements

This research was funded, in part, by NSF Grants # CNS-0620245 and OCI-0636299. The authors would also like to thank EDGE. The conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] B. Krishnamachari, D. Estrin, and S. B. Wicker, "The impact of data aggregation in wireless sensor networks," in *Proc. of ICDCS*, 2002, pp. 575–578.

[2] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS*, vol. 36, no. SI, pp. 131–146, 2002.

[3] A. Manjhi, S. Nath, and P. Gibbons, "Tributaries and deltas: Efficient and robust aggregation in sensor network streams," in *Proc. of SIGMOD*, 2005, pp. 287–298.

[4] J. Payton, C. Julien, and G.-C. Roman, "Automatic consistency assessment for query results in dynamic environments," in *Proc. of ESEC/FSE*, 2007, pp. 245–254.

[5] S. Kabadayı and C. Julien, "A local data abstraction and communication paradigm for pervasive computing," in *Proc. of PerCom*, 2007, pp. 57–66.

[6] G.-C. Roman, C. Julien, and Q. Huang, "Network abstractions for context-aware mobile computing," in *Proc. of ICSE*, 2002, pp. 363–373.

[7] W. Si and C. Li, "RMAC: A reliable multicast MAC protocol for wireless ad hoc networks," in *Proc. of ICPP*, Aug. 2004, pp. 494–501.

[8] B. Vellambi, R. Subramanian, F. Fekri, and M. Ammar, "Reliable and efficient message delivery in delay tolerant networks using rateless codes," in *Proc. of MpbiOpp*, June 2007, pp. 91–98.

[9] N. Lynch and M. Tuttle, "An introduction to I/O automata," *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.

[10] M. Loebbers, D. Willkomm, and A. Koepke, "The Mobility Framework for OMNeT++ Web Page," http://mobility-fw.sourceforge.net.

[11] A. Vargas, "OMNeT++ Web Page," http://www.omnetpp.org.

[12] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. on Computer Systems*, vol. 20, no. 3, pp. 239–282, August 2002.

[13] A. Singla, U. Ramachandran, and J. Hodgins, "Temporal notions of synchronization and consistency in Beehive," in *Proc. of SPAA*, 1997, pp. 211–220.

[14] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani, "The price of validity in dynamic networks," in *Proc. of ACM SIGMOD*, June 2004, pp. 515–526.

[15] N. Jain, D. Kit, D. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang, "Known unknowns in large-scale system monitoring," http://www.cs.utexas.edu/users/dahlin/papers/known-unknowns-oct-draft.pdf, October 2007.

[16] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *Proc. of VLDB*, 2004.

[17] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha, "Active learning drive data acquisition for sensor networks," in *Proc. of ISCC*, 2006.

[18] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *Proc. of ICDE*, 2004, pp. 449–460.