Automated Assessment of Aggregate Query Imprecision in Dynamic Environments

Vasanth Rajamani[1], Christine Julien[1], and Jamie Payton[2]

[1] Department of Electrical and Computer Engineering
The University of Texas at Austin
{vasanthrajamani, c.julien}@mail.utexas.edu
[2] Department of Computer Science
University of North Carolina, Charlotte
payton@uncc.edu

Abstract. Queries are widely used for acquiring data distributed in opportunistically formed mobile networks. However, when queries are executed in such dynamic settings, the returned result may not be consistent, i.e., it may not accurately reflect the state of the environment. It can thus be difficult to reason about the meaning of a query's result. Reasoning about imperfections in the result becomes even more complex when in-network aggregation is employed, since only a single aggregate value is returned. We define the semantics of aggregate queries in terms of a qualitative description of consistency and a quantitative measure of imprecision. We provide a protocol that performs in-network aggregation while simultaneously generating quality assessments for the query result. The protocol enables intuitive interpretations of the semantics associated with an aggregate query's execution in a dynamic environment.

1 Introduction

The proliferation of laptops, sensors, and wireless devices has increased the number of data providers embedded in our environment. The ability to obtain data and expose meaningful information to applications in dynamic networks remains a major challenge. Queries are a popular abstraction for making information-rich environments more accessible by masking complex network details. An important class of queries are *aggregate* queries, which are particularly popular in dynamic networks because they enable *in-network aggregation* [1–3]—the observation that computation is significantly cheaper than communication in terms of resource consumption. Individual hosts, then, should aggregate as much raw data as possible to reduce the communication overhead of queries.

Though queries can simplify application development, the unpredictable connectivity changes in mobile ad hoc networks make it difficult to ensure that a query's result is *consistent*, i.e., the result completely and accurately reflects the state of the environment during query execution. Consider an application in the construction domain for intelligent asset management. A site supervisor needs 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. Each pallet is tagged with a device that represents the count (or weight) of the material present. The supervisor may issue a simple aggregate query that returns the sum of this material across the site. While the query is executing, pallets move around the site, which may cause a pallet's value to be counted more than once or not at all. In addition, the failure of any device results in the loss of all the values that were aggregated at that device. This results in an inconsistency between the reported total and 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 solutions, therefore, rely on "best-effort" queries, which make no guarantees about the quality of the result. Consequently, the query result represents the ground truth to an arbitrary degree, making it difficult for applications to know how to use the results. Thus, a fundamental requirement for applications employing aggregate queries is the ability to interpret the imperfections associated with retrieving data from dynamic networks.

To measure query imperfection, we define semantics for a basic set of aggregate queries 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. Specifically, we make the following contributions. First, we define a conceptual model for estimating numerical bounds that define a query's imprecision and demonstrate how we can express the semantics of aggregate queries (Sections 2 and 3). Second, we develop a protocol that computes aggregate queries while assessing their semantics; this assessment is attached to the result to support reasoning about the returned value (Section 4). Third, we have prototyped and evaluated a reference implementation (Section 5).

2 Background : Modeling Query Execution

Our previous work [4] defined a query processing model to express mobility as state transitions and a set of consistency semantics for simple queries. We review this model and use it to define the semantics of aggregate queries.

In our model of a mobile network, a host is a tuple (ι, ν, λ), where ι is a unique identifier, ν is a data value, and λ is its location. The global abstract state of the network, a *configuration*, is a set of host tuples. An *effective configuration* (E) is the projection of the configuration with respect to the hosts *reachable* from a specific host $\bar{h}$. Reachability is often defined in terms of network connectivity, captured by a relation that conveys the existence of a (possibly multi-hop) path between hosts. We use a binary logical connectivity relation $\mathcal{K}$ to express the existence of a direct link between two hosts. Reachability is defined as the reflexive transitive closure $\mathcal{K}^*$. An evolving network is a state transition system with a state space defined by the set of possible configurations, and transitions defined as configuration changes. Sources of configuration change include: 1) *variable*

assignment, in which a host changes its data value, and 2) *neighbor change*, in which a host's changing location impacts the network connectivity.

Consider the configurations in Fig. 1. A query begins with its issue (the *query initiation bound*, C_0) and ends 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, a query's *active configurations* are those within $\langle C_0, C_1, \ldots C_n \rangle$ during which the query interacted with the network. Every value contributing to a query's result must be present in some active configuration. Moreover, only reachable hosts can contribute to a query's result. The relevant configurations, then, are the *effective active configurations*, $\langle E_0, E_1, \ldots, E_m \rangle$, containing hosts reachable from the query issuer.

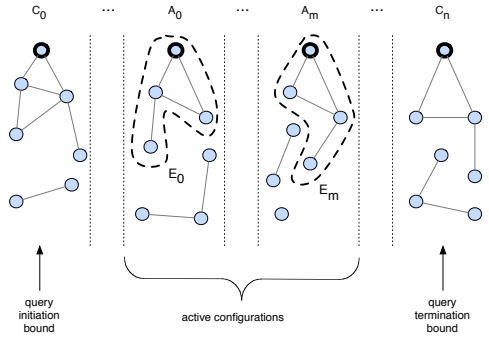

Fig. 1. Effective Active Configurations

A query is a function from a sequence of effective active configurations to a set of host tuples that constitute the result. 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. While our original model considers the impact of environmental conditions on the achievable consistency of simple discrete queries, it does not capture how in-network aggregation impacts query results. In the next section, we explore consistency semantics for queries processed using in-network aggregation and offer an approach to present aggregate query results that intuitively convey their associated consistency semantics.

3 Integrating Aggregation and Consistency

We introduce a model for aggregate queries that applies an in-network aggregation operator and returns a bounded aggregate value. The bounds convey the degree to which the query result reflects the state of the environment during query execution. The bounded aggregate is the triple $[L, A, U]$: L is a lower bound, U is an upper bound, and A is the aggregate value computed over the results available throughout query execution, i.e., A is computed over $\bigcap_{i=0}^{m} E_i$.

3.1 Consistency Classes: Comparability

In our previous work for discrete queries, we defined a set of semantics that lie between the common atomic (i.e., exact) and weak (i.e., best effort) semantics [4]. Our consistency semantics can be divided into two classes: *comparable* and *non-comparable*. In the first, stronger class, all the elements of the computed

aggregate are guaranteed to have existed at the same time, i.e., all of the aggregated results existed in the same configuration. In the weaker class, all of the aggregated values existed at some time during the query execution, but nothing can be said about the temporal relationships between the aggregated items.

In our construction example, consider a query for the truck with the fewest pallets. If material is transferred between trucks during query execution, the result may not report the truck with the fewest pallets because the query may aggregate values that are not comparable, e.g., the value of one truck before the transfer and of the newly loaded truck after the transfer. Conversely, if there was no configuration change, it is clear that the answer returned is correct. Adding this semantic information gives new clarity to the query result. In our evaluation, we revisit how consistency classes relate to the semantics of aggregation.

3.2 Numeric Bounds

We relax our definition of a query result (ρ) to allow for computation of imprecision bounds. A query result has two components: $\langle \mathcal{A}, \mathit{Excess} \rangle$, where $\mathcal{A}$ is the aggregate result, 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 5). Each element in *Excess* was present in at least one of the query's effective active configurations but missing from another:

$$e \in \mathit{Excess} \Rightarrow \langle \exists i, j : 0 \leq i, j \leq m \wedge i \neq j :: e.h \in E_i \wedge e.h \notin E_j \rangle^3$$

If a host's value or location changed multiple times during execution, there may be multiple tuples in *Excess* for the same host. This must be handled with care for *duplicate-sensitive* aggregation operators [2] like sum and average.

3.3 Determining the Semantics of Aggregate Queries

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

Set Union Aggregation. Set union aggregation can be expressed as an aggregation operation where $\mathcal{A}$ contains the stable subset of query results, i.e., results from hosts that experienced no changes during the query's execution:

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

[3] In the three-part notation: $\langle$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.

Excess contains values either added or removed (or both) during execution:

$$\mathrm{S}_{\mathrm{excess}} = \langle \mathrm{set}\ e : e \in \mathit{Excess} :: e.h \rangle$$

A set union query returns $[-, \mathcal{A}, \mathcal{A} \cup S_{excess}]$, where $-$ indicates an absent lower bound. When no changes occurred, *Excess* is empty, and the upper bound is the same as the estimate. For example, $\mathcal{A}$ consists of the values for pallets of material whose data value or connectivity did not change during execution. The upper bound will contain data values for pallets of materials that may have been delivered or consumed during the query.

Minimum/Maximum Aggregation. In this simple form of aggregation, $\mathcal{A}$ contains the minimum (or maximum) value from in-network aggregation. As an example, a minimum aggregation can tell the site supervisor what area of the site may lack a particular material:

$$\mathcal{A} = \langle \mathrm{MIN}\ h \in E_i : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h.\nu \rangle$$

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

$$\mathrm{MIN}_{excess} = \langle \mathrm{MIN}\ e : e \in \mathit{Excess} :: (e.h).\nu \rangle$$

A minimum aggregate query returns $[min(\mathrm{MIN}_{excess}, \mathcal{A}), \mathcal{A}, -]$. When *Excess* is empty or contains no value less than $\mathcal{A}$, this query returns $[-, \mathcal{A}, -]$.

Counting Aggregation. The counting aggregate is the first of our aggregates that is *duplicate sensitive*, and so it should attempt to avoid counting the same host more than once. When the 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 using the host's ID to prevent double counting.

$$\mathrm{C}_{excess} = \langle +i : \langle \exists e :: e \in \mathit{Excess} \wedge (e.h).\iota = i \rangle :: 1 \rangle$$

The result returned to the querier is $[-, \mathcal{A}, \mathcal{A} + \mathrm{C}_{excess}]$. 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.

Summation Aggregation. An aggregate summation should represent the sum over all hosts receiving the query:

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

In this case, the upper and lower bounds are calculated based on the worst possible scenario. We create all of the permutations of *Excess*, combine their sums with $\mathcal{A}$, and take the minimum (if less than $\mathcal{A}$) as the lower bound and the maximum (if greater than $\mathcal{A}$) as the upper bound. In calculating these permutations, we must also suppress duplicates. We first define a set of sets, $\mathcal{P}$, a duplicate sensitive power set of *Excess*. $\mathcal{P}$ contains all possible sets p that satisfy:

$$|p| \neq 0 \wedge \langle \forall h : h \in p :: \langle \exists e : e \in \mathit{Excess} :: e.h = h \rangle \rangle \wedge$$
$$\langle \forall h_1, h_2 : h_1 \in p \wedge h_2 \in p :: h_1.\iota \neq h_2.\iota \rangle$$

p is a legal permutation if p is not empty, every element in p corresponds to an element in *Excess*, and no two elements in p are from the same host. $\mathrm{U_{SUM}}$ is:

$$\mathrm{U_{SUM}} = \mathit{max}(\langle \max p :: p \in \mathcal{P} :: \langle + h : h \in p :: h.\nu \rangle + \mathcal{A} \rangle, \mathcal{A})$$

$\mathrm{L_{SUM}}$ is defined similarly using min. A summation query returns $[\mathrm{L_{SUM}}, \mathcal{A}, \mathrm{U_{SUM}}]$.

Average Aggregation. Average aggregation is similar to summation. However, to recalculate averages for computing bounds, 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 a count of contributors to $\mathcal{A}'$:

$$\mathcal{A}' = \langle \mathrm{avg}\ h : \langle \forall i : 0 \leq i \leq m :: h \in E_i \rangle :: h.\nu \rangle$$

We use the elements of the *Excess* set to calculate all of the potential average values after removing duplicates, using $\mathcal{P}$ as above. $\mathrm{U_{AVG}}$ is:

$$\mathrm{U_{AVG}} = \mathit{max}\left(\left\langle \max N : N \in \mathcal{P} :: \frac{\langle + p : p \in N :: p.\nu \rangle + \mathcal{A}'}{C + |N|} \right\rangle, \mathcal{A}\right)$$

$\mathrm{L_{AVG}}$ is defined similarly using min. Assuming $\mathrm{L_{AVG}} < \mathcal{A}'$ and $\mathrm{U_{AVG}} > \mathcal{A}'$, an aggregate average query returns $[\mathrm{L_{AVG}}, \mathcal{A}', \mathrm{U_{AVG}}]$. In our construction site example, if the amounts of available material on pallets varies during query execution, the site supervisor can use the range provided by the upper and lower bounds to determine how much confidence to place in the query response.

4 Assessing Aggregation Imprecision

The previous section discussed how to determine the semantics of an aggregate query with global knowledge using our model. We next present a practical query execution protocol that performs in-network aggregation and calculates error bounds on the aggregate result using nodes' local perspectives on the world.

4.1 Protocol Overview

Query protocols that lock data values to ensure strongly consistent results are impractical in dynamic networks. Our protocol provides different semantics under different conditions, while dynamically assessing the result's imprecision. Our self-assessing protocol makes an initial examination of data values accessed during aggregate computation and maintains state about the values during query processing to determine the consistency class and compute the result's bounds.

We employ two controlled floods—*Pre-Query* and *Aggregation Assessment*. The Pre-Query establishes the query participants. 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 we use to observe changes that impact the query execution semantics, and the cached partial aggregates help provide error bounds on the returned aggregate.

As shown in Fig. 2, each phase consists of two *waves*: one to disseminate a request; one to return the response. Each node waits for its children to respond before responding itself. Participating hosts monitor changes that can impact the query's imprecision; these include 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. If no changes occur, the query has comparable consistency and requires no bounds since it is exact. If the changes include only departing participants, the aggregate is computed from results that existed at the same time, and the query has comparable consistency. The bounds include the values of the departed nodes. Finally, if the query encountered both departing and arriving participants, the aggregate is computed from results that did not necessarily co-exist, resulting in non-comparable consistency, and the bounds account for both departed and added values.

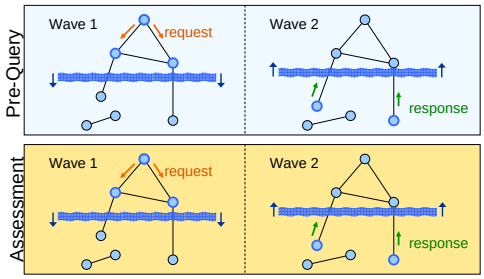

Fig. 2. Protocol Phases and Waves

Flooding an entire network can be expensive, so we constrain the flood using the query's logical connectivity relation $\mathcal{K}$; this is similar to other constrained flooding approaches [5, 6].

4.2 Self-Assessing Aggregation

We next detail our protocol. Fig. 3 shows the state for a single query. We use I/O Automata [7] to show the behaviors of a host, A. Each *action* has an effect guarded by a precondition and executes in a single atomic step. Actions without preconditions are *input actions* triggered by another host. 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 for Node A

Pre-Query Phase. The Pre-Query establishes a core set of participants used to determine how well the query's result compares to the "ground truth." 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.[4] This wave continues until 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 impact the quality of the returned aggregate result. On receiving this message, a host locally stores the child's estimated aggregate and count values. In Fig. 5, *op(child-yield(*))* refers to performing operation *op* on the entire child-list contained in *child-yield*. After all its children have reported, it computes the partial aggregate for its subtree (*estimated-val*), and forwards it to its parent.

```
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

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 the cached values aid in calculating the aggregate query result's imprecision later.

Aggregation Assessment Phase. Once the Pre-Query completes, the query issuer initiates Aggregation Assessment. As before, a parent waits for

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

responses from all of its children before sending its own reply, and boundary hosts initiate the second wave. Only results from nodes present in both phases and without any value change contribute to the final aggregate. This value for each subtree is stored in *actual-val* at the root of each subtree and propagated to the query issuer. The *actual-val* (i.e., $\mathcal{A}$ in our framework) returned to the user reflects a conservative aggregation, as defined by the consistency class. Once the query issuer has received *QueryReply* messages from all of its children, it prepares the result. The protocol dynamically assesses and tags a result with bounds indicating the query's imprecision and the achieved consistency class.

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, then the *actual-val* returned will not be computed using all values that existed during query execution. Consider a minimum aggregate. If a node with the smallest value departs the network during the query, the computed aggregate will not reflect the smallest value that existed in the network. We must therefore include the departed value in the query result's bounds. To do so, the parent calculates its estimated value for the departed node's subtree using the value stored in *child-yield* and sends this value to the root in an *EstimateReply* message. Similarly, an inconsistency arising due to a node adding itself to the network should be reflected in the aggregate's bounds. When a node detects such an addition, it sends a new *EstimateRequest* message to the added node, which creates and sends an *EstimateReply* containing its data value and unique node identifier. We model a data value change as a node departure followed a node addition. Since the *EstimateReply* includes the unique node identifier, we can perform post-processing

```
NeighborAdded_A(B)
  Precondition:
      connected(A, B) ∧ B ∉ neighbors
  Effect:
      neighbors := neighbors ∪ {B}
      if membership then
          if ¬monitoring ∧ (replies-waiting ≠ ∅) then
              send ParticipationRequest to B
              replies-waiting := replies-waiting ∪{B}
          else
              send EstimateRequest to B

NeighborDeparted_A(B)
  Precondition:
      ¬connected(A, B) ∧ B ∈ neighbors
  Effect:
      neighbors := neighbors − {B}
      if membership then
          if B = parent then
              [reset state]
          else if ¬monitoring ∧ (replies-waiting ≠ ∅) then
              replies-waiting := replies-waiting −{B}
          else if ¬monitoring then
              participants := participants − {B}
              send EstimateReply(B) to parent
          else
              replies-waiting := replies-waiting−{B}
              send EstimateReply(B) to parent
```

Fig. 6. Actions for Neighbor Changes

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 average query. In the first three scenarios, the nodes are in the Aggregation Assessment phase but have not yet sent replies. The first graph shows the query's participants. In the second graph, a node with a value of 20 departs. The neighboring node detects the departure and uses locally cached values to compute and send an *EstimateReply*. In the third graph, a node with a value of 0 departs. The final graph depicts the final computation. The values present and unchanged are used to perform in-network aggregation, which results in the average $\mathcal{A} = 10$.

Bounds on the aggregate result are computed using the *EstimateReply* messages (arrows in Fig. 7) which carry the "excess" values to the querying host. In this example, we calculate all potential average values after removing duplicate contributions. 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 comes from the configuration that included the departed node with value 20 but not the departed node with value 0. Therefore, the numerical result for this 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. Thus, the bounds were computed using comparable values, i.e., values that existed at the same time.

In general, the query result includes the 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 query responses.

5 Evaluation

We have prototyped our protocol using the OMNeT++ simulator and its mobility framework [8, 9]. Our protocol executes a query, establishes its consistency,

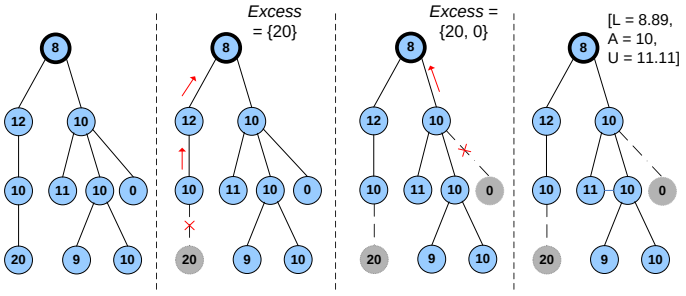

Fig. 7. Example Average Query

and provides bounds on the response. [5] We executed our protocol in a 1000m x 900m 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.

5.1 Using Aggregate Imprecision: An Application Scenario

We first demonstrate how an application might apply our protocol using a construction asset management example. In our scenario, 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 positioned for use. Initial data values are generated using a Gaussian distribution ($\mu = 0$, $\sigma = 20$); a value increases in steps of 10 for 50 seconds (representing material delivery) 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 (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.

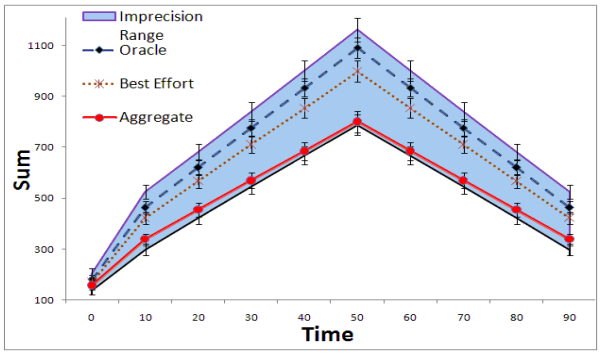

Fig. 8. An Application Example

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 imprecision bounds. While our aggregate result is conservative and may differ from the oracle, the range gives an indication of the space of all possible answers. The behavior exhibited here is true for all other operators; 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 is in very dense networks, where the

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

oracle tends to be higher than our upper bound due to the significant numbers of packets dropped because of contention in the wireless medium.

5.2 Impact of Mobility

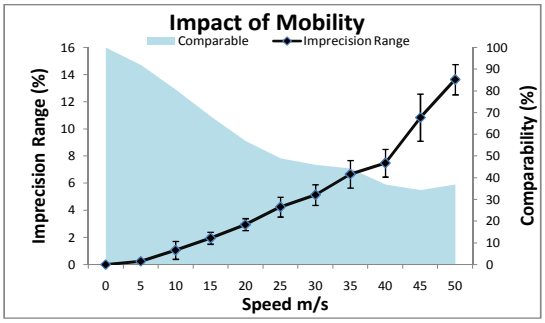

Fig. 9. Imprecision Change with Dynamics

We now 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 during query execution. However, as mobility increases, the results are increasingly *non-comparable*. Fig. 9 also shows the impact of mobility on aggregate imprecision. The line shows the size of the imprecision range obtained by executing our protocol[6]. As mobility increases, the imprecision range widens. Using a best effort protocol can produce responses that vary significantly from the ground truth; the difference between the oracle and best effort responses varied from 0 to 20%. The imprecision range is lower than Fig. 8 because mobility alone (not data changes) contributes to the uncertainty here. 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 query results.

5.3 Protocol Performance

We measured our protocol's performance in terms of overhead (the number of bytes transmitted to evaluate a query) and latency (the time to process a query). Our protocol effectively runs the best effort protocol twice, so the overhead is about double that of the best effort protocol. In addition, our overhead increases slightly with mobility due to sending additional information to calculate the bounds. (This graph is omitted for brevity.) Fig. **??** shows that our protocol does not incur significant delays. 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 environments.

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

6 Related Work

Consistency has been expressed in terms of precise metrics defining numerical error, order error, and services [10]. The authors explore 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 [11]. 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 [12]; essentially, a result 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 which allows us to provide a range of semantics. More recent work exposes inconsistency in query results through network imprecision [13]. 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 dynamic pervasive computing networks. In addition, we combine a measure of consistency with a measure of imprecision providing a more complete way to convey query semantics.

Researchers in sensor networks have explored a model-driven approach to query processing [14, 15]. Each node constructs a local model of the data in the network and estimates the error in the model. If the estimated error is acceptable, a node conserves energy by querying the local model. Another popular approach provides approximate answers that trade accuracy for energy efficiency [16]. We calculate the error on demand at query time since pro-actively maintaining local models can be expensive in mobile environments.

7 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 and quality 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 result.

Acknowledgements

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

References

1. Krishnamachari, B., Estrin, D., Wicker, S.B.: The impact of data aggregation in wireless sensor networks. In: Proc. of ICDCS. (2002) 575–578
2. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: A tiny aggregation service for ad-hoc sensor networks. ACM SIGOPS **36**(SI) (2002) 131–146
3. Manjhi, A., Nath, S., Gibbons, P.: Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In: Proc. of SIGMOD. (2005) 287–298
4. Payton, J., Julien, C., Roman, G.C.: Automatic consistency assessment for query results in dynamic environments. In: Proc. of ESEC/FSE. (2007) 245–254
5. Kabadayı S., Julien, C.: A local data abstraction and communication paradign for pervasive computing. In: Proc. of PerCom. (2007) 57–66
6. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proc. of ICSE. (2002) 363–373
7. Lynch, N., Tuttle, M.: An introduction to I/O automata. CWI-Quarterly **2**(3) (1989) 219–246
8. Loebbers, M., Willkomm, D., Koepke, A.: The Mobility Framework for OMNeT++ Web Page. `http://mobility-fw.sourceforge.net`
9. Vargas, A.: OMNeT++ Web Page. `http://www.omnetpp.org`
10. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. on Computer Systems **20**(3) (August 2002) 239–282
11. Singla, A., Ramachandran, U., Hodgins, J.: Temporal notions of synchronization and consistency in Beehive. In: Proc. of SPAA. (1997) 211–220
12. Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The price of validity in dynamic networks. In: Proc. of ACM SIGMOD. (June 2004) 515–526
13. Jain, N., Kit, D., Mahajan, D., Yalagandula, P., Dahlin, M., Zhang, Y.: Network imprecision: A new consistency metric for scalable monitoring. In: Proc. of OSDI. (2008) 87–102
14. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., Hong, W.: Model-driven data acquisition in sensor networks. In: Proc. of VLDB. (2004)
15. Muttreja, A., Raghunathan, A., Ravi, S., Jha, N.: Active learning drive data acquisition for sensor networks. In: Proc. of ISCC. (2006)
16. Considine, J., Li, F., Kollios, G., Byers, J.: Approximate aggregation techniques for sensor databases. In: Proc. of ICDE. (2004) 449–460