PAQ: Persistent Adaptive Query Middleware for Dynamic Environments

Vasanth Rajamani[1], Christine Julien[1], Jamie Payton[2], and
Gruia-Catalin Roman[3]

[1] The University of Texas at Austin
{vasanthrajamani, c.julien}@mail.utexas.edu
[2] The University of North Carolina, Charlotte
payton@uncc.edu
[3] Washington University in Saint Louis
roman@wustl.edu

Abstract. Pervasive computing applications often entail continuous monitoring tasks, issuing persistent queries that return continuously updated views of the operational environment. We present PAQ, a middleware that supports applications' needs by approximating a persistent query as a sequence of one-time queries. PAQ introduces an integration strategy abstraction that allows composition of one-time query responses into streams representing sophisticated spatio-temporal phenomena of interest. A distinguishing feature of our middleware is the realization that the suitability of a persistent query's result is a function of the application's tolerance for accuracy weighed against the associated overhead costs. In PAQ, programmers can specify an inquiry strategy that dictates how information is gathered. Since network dynamics impact the suitability of a particular inquiry strategy, PAQ associates an introspection strategy with a persistent query, that evaluates the quality of the query's results. The result of introspection can trigger application-defined adaptation strategies that alter the nature of the query. PAQ's simple API makes developing adaptive querying systems easily realizable. We present the key abstractions, describe their implementations, and demonstrate the middleware's usefulness through application examples and evaluation.

1 Introduction

Computing and communication have undergone a dramatic change with the introduction of mobile devices and sensor networks, enabling new applications characterized by a tight embedding of computation to the environment, dynamic network topologies, and the physical distribution of application components. The ad hoc nature of such networks aligns with fluid applications that must respond to rapid and frequent changes. As such, applications are often designed to monitor changes in information or conditions in the surrounding environment. As examples, an application on a construction site may monitor for the presence of a hazardous materials leak to ensure safety conditions, and a driver's navigation

system may monitor a network of vehicles to detect traffic conditions that could impact the planned travel route.

Programming applications that monitor information across an open and rapidly changing network can be challenging. A *persistent query* is an abstraction that can simplify the development of applications that require continuous monitoring. A persistent query allows a programmer to describe the data of interest to the application without requiring him to specify network communication details. At the abstract level, a persistent query may be defined as the continuous reporting of relevant state changes in a dynamic network. However, accurate evaluation of a persistent query that continuously reports all state changes is feasible only in relatively static networks; the cost of continuous monitoring is prohibitive in the face of networks that exhibit rapid change.

To support application development using persistent queries, we introduce the Persistent Adaptive Query (PAQ) middleware. PAQ introduces strategies that approximate a persistent query using a sequence of reports generated by successive one-time queries, i.e., queries evaluated once at a given time over some portion of the network. Although query processing systems exist which execute long-lived queries in this manner [1–4], the results are typically presented to the application in a traditional static database format. In contrast, PAQ presents the results in a way that more closely simulates continuous monitoring, conveying the dynamic and streaming nature of the persistent query. Key to supporting this is a new abstraction called an *integration strategy*, which specifies how the history generated by consecutive one-time queries is transformed into a semantically precise approximation of the corresponding persistent query. Integration strategies go beyond capturing simple aggregation schemes, such as those in [1, 3], allowing the programmer to specify compositions of one-time query results that relate to spatial, temporal, and semantic properties of the collected information. For example, a developer can specify an integration strategy in which the result delivered to a construction site supervisor shows materials that were not used throughout the day (i.e., the result includes data items that remained available and unchanged throughout the execution of a persistent query).

A key insight in our work is that the suitability of an *inquiry strategy*, which controls when, how, where, and what type of one-time queries are issued, depends on the application's needs with respect to overhead and the desired degree of accuracy in the approximated persistent query result. For example, an application that requires a high degree of accuracy and can tolerate significant overhead may employ a query that floods the entire network, while an application with stricter overhead constraints may employ an inquiry strategy that randomly samples a set of network nodes. To balance these tradeoffs, PAQ allows an application developer to specify an inquiry strategy that is best suited to serve the application's needs. More important, however, is the realization that the suitability of the inquiry strategy changes as the dynamics of the network change. Therefore, PAQ provides a programming abstraction called an *introspection strategy*, which assesses properties of a persistent query's execution as well as returned results to determine its suitability. For example, an introspection strategy may use the

locations of responding hosts to determine if the query adequately covers a desired area. Based on the value of such introspection metrics, an application can use an *adaptation strategy* to dynamically adjust its inquiry strategy.

In this paper, Section 2 reviews related work on query processing and adaptation. Section 3 presents an overview of the PAQ middleware. Details on PAQ's abstractions for query execution and appear in Section 4, while Section 5 describes abstractions related to adapting query execution. Section 6 describes our prototype implementation using two application examples, and Section 7 presents a performance evaluation. Section 8 concludes.

2   Related Work and Motivation

In the sensor networks and database communities, several query processing systems provide some version of persistent queries [1–3]. Persistent queries (also called "continuous queries") are typically implemented either as 1) a continuous push of updated data from sensors to a collector with queries executed over the collected data, or 2) as a sequence of one-time queries periodically propagated over the network. The "push" approach requires maintenance of a distributed data structure, which can be costly in dynamic settings. In addition, this method often requires that a query issuer interact with a collector that is known in advance and reachable at any instant, which is often an unreasonable assumption. Therefore, we think of a persistent query as being approximated by a sequence of one-time queries issued with a given frequency from any node.

Researchers have recognized that a query's environment changes over time and that query processing should adapt [5]. The focus is typically to change the order of query operations to optimize for the dynamics. For example, Continuous Queries (CACQ) [6] relies on eddies [7] to determine the order in which tuples are processed by different operators. Similarly, SteamMon [8] adapts the query plan to accommodate arbitrary changes in the data stream. These approaches use system-defined adaptations. Alternate approaches use a model that suppresses the amount of data collected from the network. In model-driven approaches [9], a local model of the environment is constructed and used to answer queries. The model obtains data from the network only when it cannot answer a query. Adaptive filters [10] uses a model of the network to adjust the rate of updates that stream from each node in the network to a collector as part of a persistent query; the adjustment is based on acceptable tradeoffs between an application's tolerance of numerical imprecision and the current cost of sending updates. A centralized coordinator periodically adjusts the bounds of each update filter on each node to suit application needs. Such model-based approaches are not well-suited for dynamic environments because their unpredictability counteracts the temporal correlations that form the basis for the models.

None of the above approaches to adaptive query processing provide general support for dynamically adapting a persistent query based on application-specified strategies. For example, while using numerical precision bounds as a trigger for adaptation is useful, support is still needed for expressing richer types

of adaptation triggers, such as "does the query cover an adequate area of the network", that would be useful in applications deployed in dynamic environments. We focus on providing the tools required to expose information about changes taking place in a dynamic environment and the ability to respond to them.

In general, this ability to inspect and act is called *reflection* [11, 12], and the PAQ middleware embodies our effort to systematically provide abstractions for reflection on persistent queries in dynamic networks. Consequently, we provide programming abstractions that support the construction of applications that dynamically evaluate the cost of executing a query in the current environment and adjust the query's processing according to the application's needs.

3   A Middleware for Persistent Query Processing

A persistent query should provide a reflection of the "ground truth," the actual state of the world during query execution. This is equivalent to a complete picture of all of the states of the environment that exist during the persistent query's execution. We approximate the results by modeling a persistent query as a sequence of non-overlapping *one-time queries*, or queries that appear to be issued over a single state of the environment. In this section, we introduce foundational concepts to create and control this kind of approximated persistent query. We begin by reviewing a model of one-time query execution [13] and then use the model to precisely define the PAQ perspective and its abstractions.

3.1   A Model of One-Time Query Execution

A mobile ad hoc network is a closed system of hosts, each represented as a triple (ι, ζ, ν), where ι is the host's unique identifier, ζ is its context, and ν is its data value. In a simple model, the context can be simply a host's location, but it can be extended to include a list of neighbors, routing tables, and other information. A snapshot of the global abstract state of a network, a *configuration*, C, is simply a set of these host tuples, one for every host in the network.

We capture network connectivity through a binary logical connectivity relation, $\mathcal{K}$, to express the ability of a host to communicate with a neighbor. Using the values in a host triple, one can derive physical and logical connectivity relations, e.g., if a host's context, ζ, includes the host's location, a connectivity relation can be defined based on communication range.

The environment evolves as the network topology changes, value assignments occur, and hosts exchange messages. Network evolution is modeled as a state transition system where the state space is the set of possible configurations and transitions are *configuration changes*. A single configuration change consists of a: 1) *neighbor change*: the connectivity relation, $\mathcal{K}$ changes; 2) *value change*: a single host changes its stored data value; or 3) *message exchange*: a host sends a message that is received by one or more neighboring nodes.

We assign subscripts to configurations (e.g., C_0, C_1, etc.) and use $\mathcal{K}_i$ to refer to the connectivity relation for configuration i. We define *query reachability* informally, to determine whether it was possible to deliver a one-time query to and

receive a response from some host h within the sequence of configurations [13]. A host's response to a one-time query is a copy of its host tuple. A one-time query's result (ρ), then, is a subset of a configuration: it is a collection of host tuples that constitute responses to the query. No host in the network is represented more than once in ρ, though it is possible that a host is not represented at all (e.g., because it was never reachable from the query issuer).

3.2 The PAQ Perspective

Ideally, a persistent query reflects the ground truth. An exact reflection of the ground truth is equivalent to acquiring all of the configurations $(C_0 \ldots C_j)$ of the persistent query's execution. Since providing such accuracy is feasible only in relatively static networks, we extend our model to approximate a persistent query as a sequence of nonoverlapping one-time queries. Fig. 1 provides an overview of our middleware model, described below.
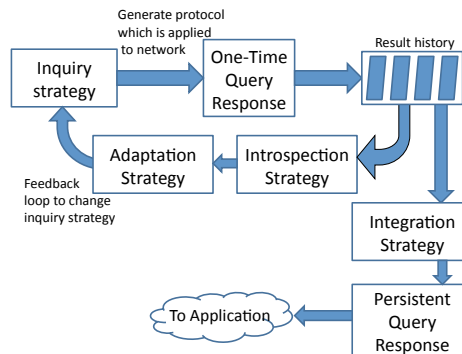

Fig. 1: A Persistent Query Framework

In evaluating a persistent query's component one-time queries, it is important to understand the behavior of an underlying query processing protocol. For example, flooding may be expensive but may achieve strongly consistent results, while randomly sampling a few nodes provides much weaker consistency, but is much less expensive. The manner in which we query the environment, the *inquiry strategy*, includes not only the one-time query protocol (called the *inquiry mode*) but also the frequency of the one-time queries.

A persistent query's result is formed from the component queries using an *integration strategy*, a function f evaluated (and reevaluated) over the sequence of one-time query results. We denote the results of the sequence of one-time queries as $\rho_0 \ldots \rho_i$, and the result of a persistent query after the results of the ith component query have been incorporated as $\pi_i = f(\rho_0 \ldots \rho_i)$. This result is still a set of host tuples, but without the constraint that the set contain only one result from any single host.

As application requirements and conditions change, applications must determine the suitability of their particular inquiry strategy. We define an *introspection strategy* also as a function over host tuples. However, in the introspection strategy, a function, d, generates not a set of host tuples but instead a value for a metric that describes the quality of that history. Based on the value of this metric, an application can specify *adaptation strategies* that govern how the inquiry strategy is changed. In the remainder of this paper, we discuss how inquiry, integration, introspection, and adaptation work together to enable applications to process expressive persistent adaptive queries over dynamic mobile networks.
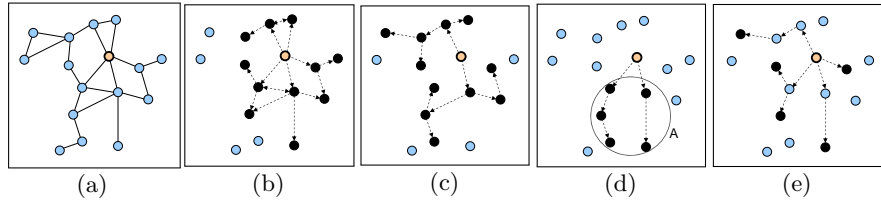
Fig. 2: Query protocols. The query issuer has a dark boundary. (a) Sample network. (b) Flooding. (c) Probabilistic protocol. Every node within the constraint (3 hops) that receives the packet retransmits it to 2 randomly selected neighbors. (d) Location based protocol that queries the nodes in region A. (e) Random protocol that queries 5 nodes.

4 Two Abstractions for Persistent Query Processing

We next present PAQ abstractions that are essential to creating a persistent query result. Using these, applications can specify how to retrieve information from the network and how to combine intermediate results over time.

4.1 Inquiry Strategies

A persistent query's inquiry strategy comprises the *inquiry mode*, or the protocol used to disseminate the one-time queries, and the frequency with which one-time queries are issued, represented as a tuple $\langle \mathcal{I}, freq \rangle$. The PAQ interface is:

```
Listing 1: InquiryStrategy

public class InquiryStrategy{
  public InquiryStrategy(InquiryMode mode, int frequency);
}
```

Defining an inquiry mode effectively entails generating a routing protocol that defines how the query and its replies propagate in the network. Fig. 2 depicts a sample network and example inquiry modes. The most common type of queries in mobile networks are flooding queries and their derivatives that reduce overhead by restricting the query's scope [1, 14, 15]. Fig. 2(b) depicts a simple scoped flooding query restricted to a two hop radius around the query issuer. Several approaches explore parameterizing flooding protocols using probabilities [16–18], as shown in Fig. 2(c). Location information can direct queries to particular regions (Fig. 2(d)). Finally, a random sampling algorithm randomly selects k hosts to send the query, as depicted in Fig. 2(e). The network paths used to communicate in random sampling depend on the network's connectivity. In all these cases, significant differences between successive one-time queries can occur even if they are issued close in time using the same inquiry mode. Variance stems from randomness, network dynamics, and even environmental factors. These aspects can all influence the suitability of a particular inquiry mode to a particular persistent query.

To specify an inquiry mode in PAQ, we rely on the insight of previous work [13], which showed that inquiry modes can be described as a combination of a *forward* and a *respond* function; these functions use a host's state to determine whether the host should propagate the query and respond to it, respectively. In PAQ, we leverage these abstractions to allow developers to create new inquiry modes as a combination of forwards and responds functions.

4.2 Integration Strategies

A PAQ application can define an integration strategy, which dictates how a history of one-time query results are transformed into a persistent query result. An integration strategy's execution is managed by the PAQ middleware. As we will see, since a one-time query's result is a set of host tuples, a natural way to express integration is through the use of set operations.

In the PAQ middleware, an application developer can introduce a new integration strategy by implementing the `IntegrationStrategy` interface:

Listing 2: IntegrationStrategy

```
public interface IntegrationStrategy{
  QueryResult integrate(Vector<QueryResult> history);
}
```

In the above, `history` is the complete set of historical one-time query results. Next, we present a set of integration strategies; this set is not exhaustive, but instead demonstrates PAQ's ability to address the needs of a variety of queries.

The simplest way to get a persistent query result from a sequence of one-time queries' results is to simply return all results to the application. Such cumulative integration is useful when a persistent query is intended to generate a picture of all results available over the query's lifetime. For example, on a construction site, the supervisor may want to monitor the identities of all workers and visitors to the site. In this case, the persistent query result is: $\pi_i = \pi_{i-1} \cup \rho_i$. Cumulative integration is depicted in Fig. 3(a).
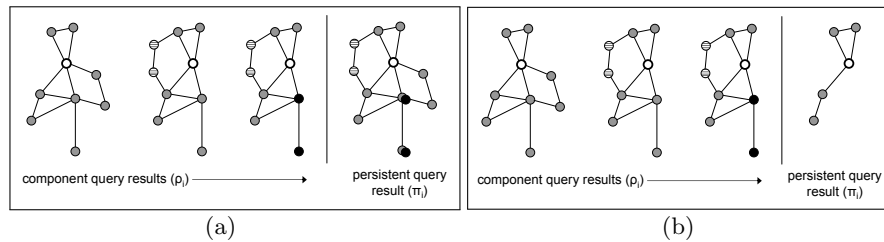

Fig. 3: Cumulative Integration (a) and Stable Integration (b). The query issuer is white. Other colors indicate data values. Between the first two queries, two nodes were added and two departed. Between the last two queries, two nodes' values changed.

A cumulative integration strategy that uses only a specified window of the history of one-time query results to construct a persistent query result can be expressed by providing an implementation for the `IntegrationStrategy` interface, and, most importantly, defining the `integrate` method[4]:

```
Listing 3: WindowedCumulativeIntegration

public QueryResult integrate(Vector<QueryResult> history){
  //omitted: define top and bottom of history window
  QueryResult temp = new QueryResult();
    for(int i = top; i>=bottom; i--){
      QueryResult nextResult = history.elementAt(i);
      Vector<HostResult> results = nextResult.getResults();
      for(int j = 0; j < results.size(); j++){
        if(!temp.getResults().contains(results.elementAt(j)))
          temp.addResult(results.elementAt(j));
      }
    }
    return temp;
  }
}
```

Cumulative integration may result in delivering an overwhelming amount of data to an application, much of which may not be required. More tailored strategies may better serve the needs of specific applications; we give examples below.

Stable Integration (Fig. 3(b)). A *stable integration* gives the results that have not changed during the query. A construction supervisor may want to know which materials are not commonly used and thus available for reallocation. A stable integration's persistent query result is: $\pi_i = \pi_{i-1} \cap \rho_i$. This result depends only on the result at the previous stage and the result of the current query.
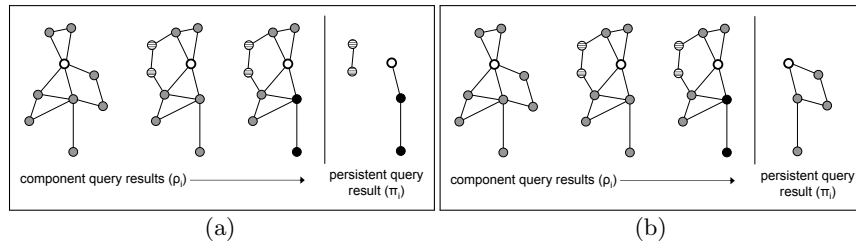

Fig. 4: Additive Integration (a) and Departure Integration (b)

Additive and Departure Integration (Fig. 4). Two additional integrations collect results that have added or departed since the start of the query. The former allows the construction supervisor to monitor materials that have been delivered, while the latter allows him to keep track of assets that have been consumed. An *additive* integration is the difference between the current result and

---

[4] We provide only this single example of an integration strategy implementation due to space constraints. Section 6 demonstrates their use, and the complete PAQ implementation can be found at http://mpc.ece.utexas.edu/AdaptiveFramework/index.html.

the first result: $\pi_i = \rho_i - \rho_0$. A *departure* integration is the difference between the first and current results: $\pi_i = \rho_0 - \rho_i$. These compare results for two instances in time; they cannot collect transient changes. More sophisticated (and therefore potentially more expensive) transient integrations can capture these semantics.

Transient Additive and Departure Integration (Fig. 5). *Transient additive* integration provides a complete view of all assets added to the site, even if they were subsequently consumed: $\pi_i = (\pi_{i-1} \cup \rho_i) - \rho_0$. *Transient departure* integration monitors results that departed, even if they returned. For example, a construction supervisor may keep track of tool usage since frequently used equipment may require maintenance. Recursively, this is: $\pi_i = \pi_{i-1} \cup (\rho_0 \cap (\rho_{i-1} - \rho_i))$. A straightforward extension would count the number of times a particular result departed, the result π_i being a set of pairs.
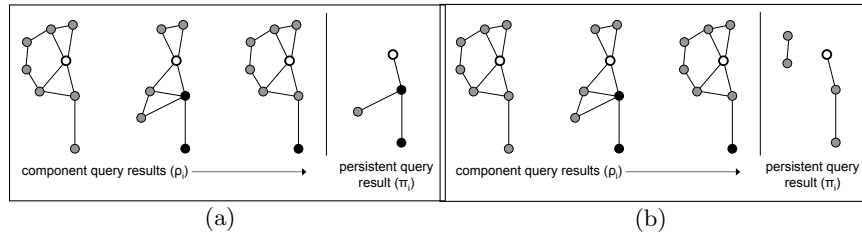

Fig. 5: Transient Additive Integration (a) and Transient Departure Integration (b)

Returns Integration (Fig. 6). A *returns* integration gives exactly those results that departed, but have since returned; this could give a construction site supervisor a picture of all of the tools used today.

The returns integration is more difficult to state in terms of previous persistent query results, but the result is directly related to a transient departure integration. A returns integration simply checks whether a departed result is present in the current query result:

$$\pi_i = \pi_{i-1} \cup \langle \mathrm{set}\, p : p \in \rho_i \wedge p \in \pi_{i-1}^{\mathrm{t\_departs}} :: p \rangle^5$$ Fig. 6: Returns Integration.

The negation of this could monitor tools that went missing during the query. This example demonstrates the power of integration; by defining fundamental integration strategies, new strategies can be defined.

5 Two Abstractions for Persistent Query Adaptation

Different inquiry and introspection strategies entail different tradeoffs. A programmer must be able to evaluate these tradeoffs in light of his application needs. We describe PAQ's two abstractions for query adaptation: introspection and adaptation. The former specifies *when* to adapt and the latter defines *how*.

---

[5] In the three-part notation: $\langle \mathrm{op}\ quantified\_variables : range :: expression \rangle$, 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. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for op, e.g., *true* if op is $\forall$.

5.1 Introspection Strategies

We define an introspection strategy as the use of information about a persistent query's execution to determine how well-suited the associated inquiry strategy is. An introspection strategy examines the persistent query's history and compares it to an idealized result. We model introspection as a function d over the component query results and the ideal result; d maps to a single numeric value that conveys the quality of the result; the range of d must be a partial ordering. The application also defines a threshold δ that is the application's tolerance for the introspection metric. Although introspection strategies may often be similar to integration strategies, the purpose is fundamentally different. While the result of integration answers an application-level question, the result of introspection measures the quality of that answer. In PAQ, an application developer can introduce a new introspection strategy through the following interface:

<div style="background:gray;color:white;padding:4px">Listing 4: IntrospectionStrategy Interface</div>

```
public interface IntrospectionStrategy{
    double introspect(Vector<QueryResult> history);
}
```

Environmental Introspection Metrics. Introspection can determine the quality of a persistent query with respect to a desired property of the *execution environment*, as captured by the context (ζ) for each host tuple. In general, this kind of distance metric can be expressed as $d = (\gamma, \mathcal{P})$, where γ is the ideal property and $\mathcal{P}$ is a function over the history of query results. The context-aware computing community has performed introspection over context data successfully in the past. For example, context-aware tour guides adapt displays according to a tourist's location and interests [19]. Consider the following example of environmental introspection in the PAQ middleware.

    Spatial Coverage Introspection. Applications may require queries to provide sensing coverage of a physical area. For example, a persistent query that monitors a chemical's dissipation across a construction site to ensure readings are within safety guidelines will have to acquire readings from across the entire site. We can determine the spatial coverage achieved by the persistent query using the location information included in the context associated with query results. We can then compare this achieved spatial coverage to the desired spatial coverage to determine whether or not the inquiry strategy is appropriate in the current operational environment. The distance metric can be expressed simply as the difference between the achieved spatial coverage region s and the ideal region (i.e., $d = |ideal - s|$).

    For example, a construction supervisor may describe the desired coverage area as a circle centered at some point on the site. For each component query, we can find the radius of the region by finding the maximum distance between a pair of points using the location information λ in the context variable ζ:

$$r = \langle \max \zeta.\lambda, \zeta'.\lambda', i : (\iota, \zeta.\lambda, \nu) \in \rho_i \ \wedge (\iota, \zeta'.\lambda', \nu) \in \rho_i :: | \ \zeta.\lambda - \zeta'.\lambda' \ | \rangle$$

We can then find the center of the circle; using the center of the spatial coverage area and the radius, we can plot the actual circular coverage area achieved by the component query. We can then determine the amount of overlap between the circle that represents the achieved spatial coverage region and the circle that represents the ideal spatial coverage region.

Semantic Introspection Metrics. The quality of a query can also be assessed based on data collected; we call this *semantic introspection*. These metrics are computed over the data values (ν) in result's host tuples. We give the implementation of the `introspect` method for a simple semantic discovery metric. Here, the introspection metric evaluates to 1 if a specified value is found in the history of query results and to 0 otherwise. We model the complete history of results as always being provided to an introspection strategy, even if the introspection only uses part of the history (in this case the most recent result). In this strategy, the variable `value` is the target value that triggers adaptation.

Listing 5: SemanticDiscoveryIntrospection

```
double introspect(Vector<QueryResult> history)
  QueryResult newResult = history.elementAt(history.size()-1)
  Vector<HostResult> results = newResult.getResults();
  HostResult nextResult = null;
  for(int i = 0; i < results.size(); i++){
    if(nextResult.getValue().equals(value)
      return 1;
  }
  return 0;
}
```

This metric could be used in a construction site supervisor's safety monitoring application to trigger adaptation from a low-overhead inquiry strategy like random sampling to a flooding inquiry with higher accuracy when a dangerous chemical reading is discovered. Semantic introspection metrics like this one, however, are not limited to evaluation over data values of direct interest to an application. Instead, these metrics can be evaluated over any data values collected for the purpose of measuring the quality of the query's execution.

Each of our integration strategies can be translated into a semantic introspection metric that quantifies the kinds of changes that occur. In general, this is captured by quantifying the difference between the query result at stage k and a windowed history of previous results. For example, we can define an introspection metric based on stable integration. The interesting part of this metric describes an evaluation over the history of results; here, $\mathcal{P}$ is defined as:

$$\mathcal{P} = |\ \langle \text{set } \nu : (\iota, \zeta, \nu) \in \rho_k :: \nu \rangle - \bigcup_{i=j}^{k-1} \langle \text{set } \nu : (\iota, \zeta, \nu) \in \rho_i :: \nu \rangle\ |$$

where k is the current stage of the persistent query, and $0 \leq j < k - 1$.

Data Change Rate Introspection. In many cases, the suitability of an inquiry strategy depends on the data dynamics. Our *data change rate* introspection measures the rate at which values change over time. For example, if the

data values in the network are relatively stable, an expensive flooding based high frequency strategy may not be necessary. Similarly, we define *additive data change rate* introspection as a running percentage of data values that have been added to the persistent query's result. The introspection metric relates to the use of the history of results to describe the achieved quality in defining $\mathcal{P}$:

$$\sum_{j=i-k+1}^{i} \frac{\mid \frac{\langle \text{set } \nu : (\iota, \zeta, \nu) \in \rho_j :: \nu \rangle - \langle \text{set } \nu : (\iota, \zeta, \nu) \in \rho_{j-1} :: \nu \rangle}{\langle \text{set } \nu : (\iota, \zeta, \nu) \in \rho_j :: \nu \rangle} \mid}{k}$$

When $k = i$, this measures the rate since the beginning of the persistent query. This introspection can be specified for departures and changes in a similar fashion. In measuring the rate of change due to newly arriving hosts, we instead sum the number of data values associated with new unique host identifiers.

5.2 Adaptation Strategies

Applications can use introspection to assess the quality of the persistent query's reflection of the environment. If the persistent query's result does not meet the application's requirements, adaptation strategies can be used to change the persistent query to achieve, for example, higher quality results or to process the persistent query at a lower cost. In general, an adaptation strategy is:

$$\langle \langle \mathcal{I}, \textit{freq} \rangle, d, \delta^{+/-}, \langle \mathcal{I}^*, \textit{freq}^* \rangle \rangle,$$

where $\langle \mathcal{I}, \textit{freq} \rangle$ is the persistent query's current inquiry strategy, d is the introspection strategy used when the persistent query uses this particular inquiry strategy, and $\delta^{+/-}$ is a threshold on the value resulting from applying d to the history of one-time query results. If the superscript on δ is $+$, the adaptation is triggered if the value of d exceeds δ; if the superscript is $-$, then the adaptation is triggered if the value of d falls below δ. The persistent query switches to the new inquiry strategy, $\langle \mathcal{I}^*, \textit{freq}^* \rangle$, when the computed value of the introspection strategy, d goes either above or below the threshold δ.

As a simple example of how an adaptation strategy could be employed, consider a persistent query using the basic flooding inquiry mode in which the component one-time queries are issued every 10 seconds. The application may associate with this persistent query an introspection strategy that changes the frequency of the one-time queries if the rate of change between the component queries grows too large. This adaptation policy would be defined as:

$$\langle \langle \mathcal{I}_{\textit{flooding}}, 10s \rangle, d_{\textit{data\_change\_rate}}, 0.05^+, \langle \mathcal{I}_{\textit{flooding}}, 5s \rangle \rangle$$

where the initial inquiry strategy $\langle \mathcal{I}_{\textit{flooding}}, 10s \rangle$ adapts to be $\langle \mathcal{I}_{\textit{flooding}}, 5s \rangle$ when the data rate of change introspection strategy indicates a greater than 5% rate of change in the data reported for successive one-time queries.

To define adaptation strategies in the PAQ middleware, an application developer instantiates an `AdaptationStrategy` that comprises a set of `AdaptationPolicy` instances. Specifically, to create an `AdaptationStrategy`, the interface presented to the developer is:

```
public class AdaptationStrategy{
  public AdaptationStrategy();
  public addAdpatationPolicy(AdaptationPolicy toAdd);
  public removeAdpatationPolicy(AdaptationPolicy toRemove);
}
```

The interface to construct an `AdaptationPolicy` is:

```
public class AdaptationPolicy{
  public AdaptationPolicy(InquiryStrategy start,
                          IntrospectionStrategy introspect,
                          double threshold, InquiryStrategy end);
}
```

More complex realizations of adaptation policies are also possible; for example, the adaptation may change not only the inquiry but also the integration strategy.

6 The PAQ Middleware: Example Applications

This section describes our Persistent Adaptive Query (PAQ) Middleware and demonstrates its use and performance through a pair of application examples.

6.1 Monitoring Hazardous Conditions

We first explore an application for an instrumented construction site that would allow monitoring recording, and reacting to the presence of a dangerous volatile organic compound (VOC). If a VOC leak occurs, the area of the incident may spread as liquid chemicals spill and as airborne droplets are released. A site supervisor wants emergency response crews to have as much and as accurate information about the incident as possible to facilitate containment and response.

The application issues a persistent query over sensors scattered across the construction site. If there is not a high risk for or evidence of a leak, the application should be conservative in its use of network resources to perform background monitoring of hazardous materials. So the query initially uses random sampling with a low frequency to sample over the entire construction site. If the query detects a dangerous concentration, the supervisor requires additional information to determine if the reading is anomalous or indicative of an actual leak. Therefore, this query will use introspection for detection of a particular value (a high concentration) and will adapt the query to a much more frequent flood of the entire site to attempt to corroborate the initial detection. To summarize:

- **Inquiry strategy:** random sampling with a low probability (e.g., $k = 0.5$), low frequency (e.g., 5 seconds)
- **Integration strategy:** windowed cumulative integration, to acquire all concentrations that were sampled over the last 20 seconds

– **Introspection strategy:** semantic discovery of any dangerous reading
– **Adaptation strategy:** upon detection of a value over the threshold, change the approach to flood the network with high frequency

In the PAQ middleware, defining this persistent query requires instantiating each of the strategies, and creating and starting the persistent query.

```
private void startQuery(){
  myInquiry = new Inquiry(new RandomSampling(0.5), 5000);
  myIntegration = new WindowedCumulativeIntgration(4);
  myIntrospection =
    new SemanticDiscoveryIntrospection(new Integer(thresh));
  PersistentQuery initialQuery =
    new PersistentQuery(myInquiry, myIntegration,
                        myIntrospection, initialAdaptation);
}
```

The initial adaptation (`initialAdaptation`) takes the persistent query from this query strategy to its second phase. In this second phase, the application is alerted to a potential hazardous leak and begins a more expensive but robust detection to corroborate the initial detection. This new query is:

– **Inquiry strategy:** flooding with a high frequency (e.g., 0.5 seconds)
– **Integration strategy:** cumulative integration, to acquire all concentrations sampled since adapting the persistent query
– **Introspection strategy:** semantic additive change rate to measure the rate of discovery of corroborating detections
– **Adaptation strategy:** if more than 10% of the sensors are newly detecting a leak, localize the persistent query around the area of detection

```
private void adaptQuery(){
  ...
  myInquiry = new Inquiry(new Flooding(), 500);
  myIntegration = new CumulativeIntegration();
  myIntrospection =
    new SemanticAdditiveIntrospection(new Integer(thresh));
  PersistentQuery initialQuery =
    new PersistentQuery(myInquiry, myIntegration,
                        myIntrospection, leakAdaptation);
}
```

This second adaptation policy (`leakAdaptation`) changes the query strategy from this second phase to target the persistent query exactly around the area of the detected leak. This allows the network's resources and the application's attention to be focused on exactly the desired sensing area. The persistent query continues to monitor this area, adapting the size and scope of the target area as the detected values change. This phase of the query continues until the danger dissipates, when the query returns to the original random sampling:

- **Inquiry strategy:** location-based flooding, focused around the leak
- **Integration strategy:** windowed cumulative, to acquire the readings over the threshold in the last 60 seconds
- **Introspection strategy:** spatial coverage, to ensure the area around the leak is well-covered
- **Adaptation strategy:** adjust the flooding area if the coverage is poor; return to random sampling if the leak disappears

Listing 10: Location Query

```
private void adaptQuery(){
  ...
  myInquiry = new Inquiry(new LocBased(mid.x, mid.y, radius), 500);
  myIntegration = new WindowedCumulativeIntegration(6);
  myIntrospection =
    new SpatialCoverageIntrospection(mid.x, mid.y, radius);
  PersistentQuery initialQuery =
    new PersistentQuery(myInquiry, myIntegration,
                        myIntrospection, locationAdaptation);
}
```

Here, `mid` is the location at the center of the detected leak and `radius` is the sensed spread of the leak. This application and the PAQ middleware are available for download[6]. Fig. 7 shows a sequence of screenshots that demonstrate the three phases described above.
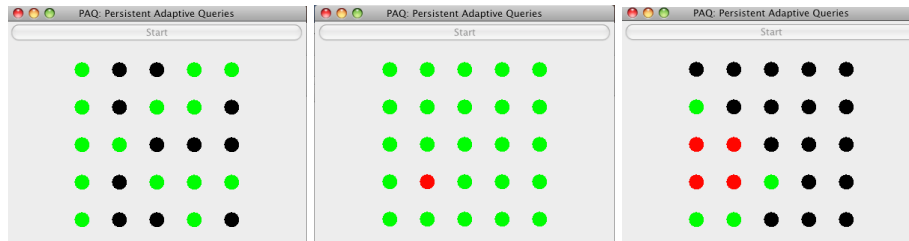

Fig. 7: A sample execution for a network of 25 nodes. The values of black nodes are unknown. The values of green nodes are below the threshold. The values of red nodes are above the threshold. The left shows a snapshot in the middle of the first phase of the persistent query; the middle shows a snapshot in the middle of the second phase; the right shows a snapshot in the middle of the third phase.

6.2 Road Traffic Monitoring

Most cities have troublesome spots that have periodic episodes of jammed traffic. We next use the PAQ middleware to construct an application for intelligent traffic monitoring. The application monitors an area of interest and informs the application if there is a backlog of vehicles. We imagine a scenario where both vehicles and stationary objects are equipped with networked sensors.

---

[6] http://mpc.ece.utexas.edu/AdaptiveFramework/index.html

The application issues a persistent query over sensors that have been scattered across the an area of interest to monitor location information of the vehicles in the area. Initially, the query probes only the area of interest to conserve the battery of the hosts in the area. If the application detects several stationary vehicles in the monitored area for an extended period of time, it can be deduced that a traffic jam has indeed occurred. Therefore, this query will use an introspection strategy that checks whether a certain number of hosts are present in several continuous query results. Once, the traffic jam has been detected, the application performs a random sample the entire network to detect alternative routes to the application. This phase can be summarized as:

– **Inquiry strategy:** location-based random sampling, focused on a known traffic trouble spot, low frequency query (e.g., 20 seconds)
– **Integration strategy:** None
– **Introspection strategy:** windowed stable integration to measure whether the same cars remain in the query area in multiple consecutive queries
– **Adaptation strategy:** upon detection of a value over the threshold indicating stranded cars, change the approach to random sampling of the entire network with higher frequency

In the PAQ middleware, defining this persistent query requires instantiating each of the strategies, and creating and starting the persistent query:

Listing 11: Initial Query

```
private void startQuery(){
  myInquiry = new Inquiry(new LocBased(mid.x, mid.y, radius), 20000);
  myIntegration = null;
  myIntrospection =
    new WindowedStableIntrospection(3, new Integer(thresh));
  PersistentQuery initialQuery =
    new PersistentQuery(myInquiry, myIntegration,
                        myIntrospection, trafficjamAdaptation);
}
```

This adaptation (`trafficjamAdaptation`) is invoked if the initial query notices an unacceptable number of stranded cars. It takes the persistent query from this query strategy to its second phase in which the application issues randomly distributed queries to find alternate routes to re-route the stalled traffic:

– **Inquiry strategy:** random sampling with a relatively low probability (e.g., $k = 0.5$) but a higher frequency (e.g., 5 seconds)
– **Integration strategy:** cumulative integration, to observe all elements of all of the roadways and get a picture of where to reroute the stranded cars
– **Introspection strategy:** counting introspection, to ensure that this particular component query has been issued a given number of times (e.g., 5)
– **Adaptation strategy:** when the query has sent alternate routes to the jammed cars (not part of the persistent query), return to monitoring the initial area to ensure that the traffic is clearing.

Aside from its use of `CountingIntrospection`, this query is similar to the initial query in the first application; we omit its listing for brevity.

This second query's adaptation policy changes the query strategy to once again target the original location to see if the rerouting instructions are causing it to clear of traffic. This phase continues unless the re-route did not succeed, and cars cleared from this location are returning:

- **Inquiry strategy:** location-based flooding, focused on the original location
- **Integration strategy:** windowed cumulative integration, to acquire the readings over the threshold (i.e., returning a number of cars indicative of a traffic jam) in the last 60 seconds
- **Introspection strategy:** windowed returns introspection, to see if cars that were instructed to reroute are in fact returning to the trouble spot
- **Adaptation strategy:** if the process has failed to clear the traffic, repeat by resampling for rerouting possibilities

Listing 12: Location Based Query

```
private void adaptQuery(){
  ...
  myInquiry = new Inquiry(new LocBased(mid.x, mid.y, radius), 5000);
  myIntegration = new WindowedCumulativeIntegration(10);
  myIntrospection =
    new WindowedReturnsIntrospection(5, new Integer(thresh));
  PersistentQuery locationQuery =
    new PersistentQuery(myInquiry, myIntegration,
                        myIntrospection, trafficjamAdaptation);
}
```

This query employs `WindowedReturnsIntrospection`, which checks for vehicles that return to the trouble spot. If many of the same vehicles return, (`trafficjamAdaptation`) is used again, and the process repeats. An application may also define a more complex persistent query that becomes increasingly aggressive about finding alternate routes as it cycles through the process.

7 Evaluating the PAQ Middleware

In addition to the middleware implementation, we also prototyped PAQ using OMNeT++ [20, 21]. We executed the described applications, modeling the network space as a 1000m × 900m rectangular area. We used the 802.11 MAC protocol.[7] The included graphs show 95% confidence intervals.

We used the first example application to explore PAQ's performance in the absence of mobility. We implemented two different types of adaptation: "moderate" and "aggressive." Given a leak detection, the "aggressive" version immediately starts issuing flooding queries at a high frequency. The "moderate"

---

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

adaptation increases the query rate more slowly (i.e., by increasing the query rate by one every five seconds). We compare the behavior of these two adaptive approaches with two standard query styles: "flooding," a query that floods all of the time, and "sampling," a query that always samples half of the reachable hosts.[8] Fig. 8 highlights the value of employing adaptation to lowering overhead.
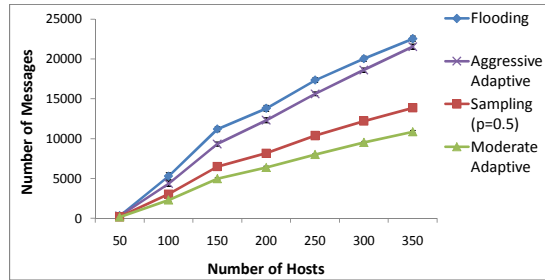

Fig. 8: Number of packets exchanged

Allowing applications to adapt leads to substantially lower overhead, especially as network density increases. The moderate version has lower overhead than randomly sampling. At first glance, the overhead of the aggressive approach is similar to flooding; however, the aggressive version issues three times as many queries because it adapts the query rate after leak detection based on the application's requirements. Even the moderate version issues 30% more queries in certain time periods but still transmits far fewer messages.
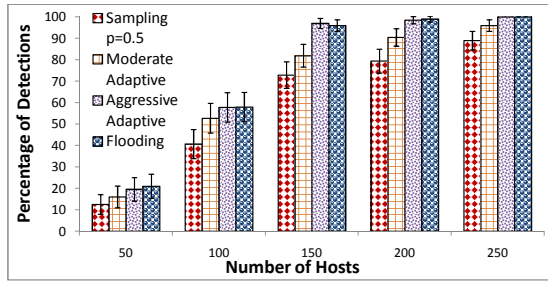

Fig. 9: Percentage of Detections

These performance benefits may come at a cost to accuracy. Since the nodes are initially randomly sampled, it is possible for the leak to go undetected if relevant sensors are not sampled. Fig. 9 shows the percentage of times the leak is successfully detected for the different versions. In sparse networks, none of the versions are very effective because the leak is often in an unreachable network partition. The moderate approach does not flag a leak unless two values pass the semantic introspection test while the aggressive version requires only one detected leak value before switching to a flooding based inquiry. Consequently, the moderate version misses more leaks. Both the aggressive and flooding approaches produce a highly accurate picture. A similar effect can be observed when the leak detection latency is compared. A flooding strategy detects the leak slightly faster than the aggressive strategy which in turn is slightly faster than the moderate strategy. This graph is omitted for brevity.

Using our second application scenario (the traffic monitoring example), we have also evaluated the impact of mobility on PAQ's query processing capabilities. In our experiment, the query is issued from a central station at one end of the 1000m × 900m field. The simulation consists of a well connected network

---

[8] While we experimented with different values of k for the random sampling approach, $k = 0.5$ provides representative results.

with 150 hosts. One-half of these hosts are on vehicles while the remaining one-half half represent sensors on stationary objects (e.g., traffic signals). A location is deemed to be "jammed" if more than five vehicles are stationary at a region of interest over three consecutive queries. In this experiment, we varied the speed at which hosts are moving from very slow (5mph) to reasonably fast (45 mph).

Fig. 10 shows rates of traffic jam detections. Regardless of the strategy used, some traffic jams go undetected due to the network dynamics. This is corroborated by the consistent downward slope for all of the approaches with increasing mobility. However, in addition to maintaining its lower overhead, the adaptive strategy achieves
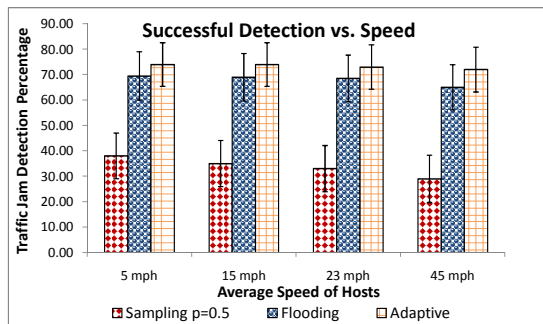

Fig. 10: Percentage of Detections

a slightly *better* traffic jam detection success rate than flooding. This is because the flooding approach generates many more messages in the network, resulting in more messages dropped due to collision. Our adaptive strategy targets only a particular area of the network, thereby reducing the number of messages in the network. Consequently, fewer messages are dropped leading to better detection.

From these results it can be seen that having a flexible mechanism for adaptation is clearly beneficial to application developers. The PAQ middleware provides software developers a flexible and simple API that allows domain experts to specify their design choices in a powerful way, allowing applications to explicitly consider performance tradeoffs in developing their query uses.

8   Conclusions

In this paper, we introduced the PAQ middleware to help programmers quickly construct applications that use adaptive persistent queries in dynamic networks. We highlighted the different abstractions in PAQ that support simple specification of policies used to govern adaptive applications. Integration allows programmers to create application-specific methods of composing and interpreting approximate one-time query results into a result that resembles streaming data. Inquiry strategies elegantly dictate how data should be gathered from a network. PAQ's introspection strategy abstraction provides programmers with the power to specify arbitrary methods of assessing the quality of achieved results as the persistent query executes; this assessment can be used to trigger adaptation of the query. Our evaluation of PAQ through the implementation of two adaptive applications indicates that our approach is feasible and can support adaptivity, and can potentially reduce persistent query costs.

References

1. Intanagonwiwat, C., Govindan, R., Estrin, D., Heideman, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE Trans. on Net. **11**(1) (Feb. 2003) 2–16

2. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: The design of an acquisitional query processor for sensor networks. In: Proc. of ACM SIGMOD. (2003) 491–502

3. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: A tiny aggregation service for ad-hoc sensor networks. In: Proc. of OSDI. (Dec. 2002)

4. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proc. of CIDR. (2003)

5. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Found. Trends databases **1**(1) (2007) 1–140

6. Madden, S., Shah, M., Hellerstein, J., Raman, V.: Continuously adaptive continuous queries over streams. In: Proc. of ACM SIGMOD. (2002)

7. Avnur, R., Hellerstein, J.: Eddies: Continuously adaptive query processing. In: Proc. of ACM SIGMOD. (2000)

8. Babu, S., Widom, J.: Streamon: an adaptive engine for stream query processing. In: Proc. of ACM SIGMOD. (2004) 931–932

9. Deshpande, A., Guestrin, C., Madden, S., Hellersetin, J., Hong, W.: Model-driven data acquisition in sensor networks. In: Proc. of VLDB. (2004)

10. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: Proc. of ACM SIGMOD. (2003)

11. Capra, L., Blair, G.S., Mascolo, C., Emmerich, W., Grace, P.: Exploiting reflection in mobile computing middleware. ACM Mobile Comput. and Comm. Review **6**(4) (Oct. 2002) 34–44

12. Chan, A., Chuang, S.N.: Mobipads: a reflective middleware for context-aware mobile computing. IEEE Trans. Soft. Eng. **29**(12) (Dec. 2003) 1072–1085

13. Rajamani, V., Julien, C., Payton, J., Roman, G.C.: Inquiry and introspection for non-deterministic queries in mobile networks. In: Proc. of FASE. (Mar. 2009) 401–416

14. Johnson, D.B., Maltz, D.A., Broch, J.: Dsr: The dynamic source routing protocol for multi-hop wireless ad hoc networks. Ad Hoc Networking **1** (2001) 139–172

15. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proc. of ICSE. (2002) 363–373

16. Haas, Z., Halpern, J., Li, L.: Gossip-based ad hoc routing. IEEE Trans. on Networking **14**(3) (June 2006) 479–491

17. Kyasanur, P., Choudhury, R., Gupta, I.: Smart gossip: An adaptive gossip-based broadcasting service for sensor networks. In: Proc. of MASS. (October 2006)

18. Ni, S.Y., Tseng, Y.C., Chen, Y.S., Sheu, J.P.: The broadcast storm problem in a mobile ad hoc network. In: Proc. of MobiCom. (1999) 151–162

19. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proc. of MobiCom, ACM Press (2000) 20–31

20. Loebbers, M., Willkomm, D., Koepke, A.: The Mobility Framework for OMNeT++ Web Page. `http://mobility-fw.sourceforge.net`

21. Vargas, A.: OMNeT++ Web Page. `http://www.omnetpp.org`