Real-time Collection of Dynamic Data with Quality Guarantees

Qi Han   Nalini Venkatasubramanian
Department of Computer Science
University of California-Irvine, CA 92697-3425
{qhan,nalini}@ics.uci.edu

Abstract

In this paper, we focus on addressing the tradeoffs between timeliness, accuracy and cost for applications requiring real-time information collection in distributed real-time environments. In this scenario, information consumers require data from information sources at varying levels of accuracy and timeliness. To accommodate the diverse characteristics of information sources and varying requirements from information consumers, we use an information mediator to coordinate and facilitate communication between information sources and consumers. We develop algorithms for real-time request scheduling, request servicing and directory service maintenance, and compare our techniques with several other proposed strategies. Our studies indicate that the judicious composition of our proposed intelligent policies can improve the overall efficiency of the system. Furthermore, the proposed policies perform very well as the system scales in the number of information sources and consumer requests.

Keywords: information collection, real-time scheduling, directory service maintenance, Quality of Service, Quality of Data

1 Introduction

Recent developments in mobile computing, communications and embedded systems are likely to enable the deployment of large scale ubiquitous computing environments. We expect that real-time context information services will gain importance with the emergence of pervasive computing environments [15, 10]. Examples of applications that require real-time access to distributed information include network management, stock trading, air traffic control, security/surveillance, medical alerts and patient tracking. To facilitate such real-time applications, an information collection architecture that can seamlessly provide real-time access to dynamically changing data regardless of diverse user requirements and changing system conditions is a must.

Real-time information collection in a dynamic environment presents the system designer with interesting challenges. Firstly, information sources provide a continuous stream of data that can dynamically vary over time. This information may need to be captured and stored rapidly and accurately. Secondly, users requiring access to this dynamically changing data present variable user requirements in terms of accuracy of data and timeliness of the service. Furthermore, network and service providers would like to ensure effective utilization of underlying computation, communication and storage resources. Ideally,
applications would like to obtain accurate state information in a real-time manner with the least cost. The underlying middleware architecture must deal with the issue of balancing these competing goals of timeliness, accuracy and cost-effectiveness. Many applications are willing to tolerate information imprecision and bounded delivery latencies. Our strategy is to exploit these accuracy and latency margins to ensure that most applications receive information at the desired levels of quality and timeliness while minimizing resource consumption.

When a number of users request dynamic data at varying requirements under constantly changing system/network conditions, a mediation functionality is crucial to deliver the right data to the right user at the right time. Therefore, a key component of a real-time information collection architecture is an information mediator. Information sources communicate changes in source values to the mediator and information consumers forward their requests to the mediator. Given the data intensive nature of the system, an information repository, i.e. directory service (DS), is a must for the information mediator to function efficiently. An effective DS reflects the intensive data changes as closely as possible/necessary. Maintaining an effective DS, which reflects the data changes as closely as possible/necessary, is not a trivial task in dynamic environments. In addition to processing source updates, the system must also respond to user requests in a timely fashion. Processing source updates at the expense of user requests will mean that fewer user requests will finish on time, while delaying source updates in favor of user requests will mean that the DS will not be representative of the state of the external environment. In addition to timeliness constraints, user requests may also provide accuracy constraints. Since information in the DS may not be up to date, there is a possible mismatch between the DS accuracy and user accuracy constraints. We are faced with a cost-accuracy tradeoff since an accurate DS makes it easier to satisfy more user requests with less overhead in a more timely fashion, but could also introduce a high DS maintenance cost.

While existing and ongoing research in data management addresses the tradeoffs between timeliness and freshness in databases, as well as the accuracy/cost tradeoffs in stream based environments, the problem of supporting the tradeoffs between timeliness, accuracy and cost simultaneously in the context of information collection in highly dynamic environments has not been studied. In this paper, we develop strategies to address these tradeoffs. Specifically, this paper makes the following contributions:

• We characterize the problem of providing timeliness and accuracy cost-effectively for dynamic environments in terms of Quality-of-Service (QoS), Quality-of-Data(QoD) and Cost. We also present a metric EoS (Efficiency of the System) for system comparison;
• We propose a middleware framework for the real-time information collection process and design a family of algorithms to support QoS and QoD. We leverage well-studied real-time scheduling approaches
in the context of real-time systems, and consider a special and more popular scenario where (a) user requests arrive at a more fine-grained level (i.e., a single read operation instead of a regular transaction which consists of a sequence of read/write operation), (b) both user retrieval requests and source update requests may specify timeliness constraints, and (c) user requests can tolerate data imprecision to a certain degree. Taking all these characteristics into account, we propose a Timeliness and Accuracy Balanced Scheduling (TABS) algorithm to support timely response to both user requests and source updates while ensuring that accuracy constraints of requests are satisfied. Furthermore, we present a Minimized Cost DS maintenance algorithm (MC) that balances the distribution of source update and consumer retrieval requests to maintain a constant frequency ratio. We also argue that MC achieves an appropriate DS precision to satisfy a majority of user requests. Finally, the request servicing algorithm determines the process flow for each single request by analyzing the possibility of meeting its timeliness and accuracy constraints.

- Through extensive experiments, we study the interaction between different algorithms implemented at the mediator, explore dynamic adaptations and a judicious composition of mediation policies that address timeliness/accuracy/cost tradeoffs under varying conditions.

The rest of this paper is organized as follows. Section 2 presents a middleware architecture for real-time information collection, Section 3 formulates the accuracy driven real-time information collection problem and Section 4 addresses a solution via the algorithms for real-time request scheduling, request servicing and DS maintenance. Section 5 presents experimental results, analysis and also prototype design. Section 6 discusses related work. We conclude in Section 7 with future research directions.

2 A Middleware Framework for Real-time Information Collection

Figure 1 depicts the architectural components of a generalized real-time information collection framework. The framework is designed to operate in highly dynamic environments and the efficiency of the system depends on specific algorithms applied in each component of the framework. A typical framework consists of the following components:

- **Information Sources**: These correspond to various components in a distributed infrastructure. In other words, they are managed entities, such as servers, links, sensors or mobile/fixed hosts. They can be programmed to send updates periodically, or send updates when something abnormal occurs.

- **Information Repositories**: Given the data intensive nature of the system, an information repository is a crucial component for holding information about sources that are of interest to users. The information obtained from sources includes sensor data, network parameters (such as residual link bandwidth, end-to-end delay on links etc.), server parameters (such as CPU utilization, buffer capacity, disk bandwidth,
etc.), and mobile host parameters (such as mobile host location, connectivity, power level etc.). In this paper, we use the directory service (DS) and information repository interchangeably.

- **Information Consumers:** They are application and system level tasks that consume data collected from the information sources (stored in the directory service). For instance, resource provisioning consumes information about network and system status to perform admission control and resource allocation; traffic monitoring applications obtain data from highway sensors periodically to assist in traffic planning and routing.

- **Information Mediator:** This module connects information sources and consumers, serves as a crux of the information collection process where collection decisions are instrumented. The mediator processes reports from sources and notifications from consumers, and invokes suitable actions so that the directory service maintains information at a suitable level of accuracy to satisfy the data quality and timeliness needs of consumers. Policies implemented in the mediator must appropriately represent the information in the DS, efficiently collect the information from sources and effectively process requests from users.

Sub-components within the mediator perform the following tasks: *the scheduler* schedules requests from both consumers and sources. In other words, the scheduler prioritizes requests based on their urgency and popularity, and put them in the *consumer/source request queue*. The requests in the queues are ordered by their assigned priorities such that the first element of the queue has the highest priority; *the request servicer* determines the process flow for each admitted request; and *the DS maintainer* maintains adequate DS accuracy to serve source/consumer requests while reducing collection overhead. In Section 4, we develop customized algorithms to be implemented within each of these sub-components.

![Figure 1: A system architecture for real-time information collection](image-url)
3 Problem Formulation

In this section, we describe data and request models for highly dynamic environments, and characterize QoS (Quality of Service) and QoD (Quality of Data). These definitions are used to crystalize the goals of the system, guide the design of the algorithms, and evaluate system performance. Using the notions of QoS and QoD, we provide a formal definition of the real-time information collection problem. Currently, we assume the presence of a single logically centralized DS. We also assume that users only ask for current data, not historical data at a specified time instant.

The system consists of a number of data sources. The data obtained is reported by sources distributed in the environment. Examples of reported data include link utilization from the network management domain, stock and commodity prices from the financial trading domain, temperatures and pressures from a chemical process control domain. We specifically consider systems where state information changes rapidly generating a large amount of updates. Each source has a current instantaneous value \( V \), while its representation in the DS is a range \( R : [L, U] \) with \( L \) as the lower bound, \( U \) as the upper bound and \( L \leq V \leq U \). The range is refreshed with updates (write only) to the DS from the source, and queried (read only) from the DS by consumers. Data from certain sources are often accessed more frequently, so we define the popularity of source \( s \) (\( POP_s \)) as the ratio of the number of requests accessing \( s \) to the total number of requests.

Our system includes two types of requests: source update requests and consumer requests. We first define several auxiliary parameters (periodicity, urgency, relative deadline and range precision). These parameters characterize timeliness and accuracy constraints of incoming requests. The periodicity \( PER \) of request \( r \) is defined as 0 if the request is aperiodic; otherwise it is \( p \) (\( p > 0 \)), which indicates that \( r \) arrives every \( p \) time units. To provide more flexibility to applications, we provide two parameters (urgency and relative deadline) to specify the timeliness constraints. The urgency of a request is a coarse-level qualitative description of how quickly the request must be processed; the relative deadline of a request specifies a maximum duration the request can take. Application may specify one or both of these parameters (i.e., urgency and relative deadline); when both are present, the relative deadline takes precedence. The urgency \( UR \) of request \( r \) is defined as 0, 1 or 2 respectively when the urgency is low, medium or high. The relative deadline \( RDL \) of request \( r \) is defined as 0 if there is no deadline for \( r \); otherwise, it is \( d \) (\( d > 0 \)), which indicates that \( r \) should be finished in \( d \) time units. The range precision is defined as the reciprocal of the range size, i.e., \( PREC(L, U) = \frac{1}{U-L} \). Therefore, a zero-width range contains the exact value and its precision is infinite, while an infinite-width range has no information about the exact value and its precision is zero.

Each consumer request is accompanied by a time constraint, and also a precision constraint specifying
the maximum acceptable width of the result. Since simultaneous satisfaction of both timeliness and accuracy constraints may not be feasible, we introduce the attribute Bias to indicate the preference of consumers in the event that both constraints cannot be met.

\[
Bias_{cr} = \begin{cases} 
0 & \text{no preference} \\
1 & \text{timeliness weighs more than accuracy} \\
2 & \text{accuracy weighs more than timeliness} 
\end{cases}
\]

Using the notions defined above, we define source update request and consumer request. Source update requests are write-only requests to reflect the current status of the real-world environment.

**Definition 1** A Source Update Request \( sr \) is a tuple of six elements: source \( s \), request issue time \( t \), real value \( V \), periodicity \( PER \), urgency \( UR \) and relative deadline \( RDL \). i.e., \( sr := < s, t, V, PER, UR, RDL > \).

Once a source update request is applied, the DS is updated with \( L \) and \( U \). The current value \( V \) lies at the center of the range \( R \), i.e., \( U = V + R/2 \) and \( L = V - R/2 \). To accommodate heterogeneity of source intelligence, we assume that sources are not aware of the ranges \( R \) stored in the DS; instead, the DS maintenance module is responsible for adjusting and storing the ranges.

Consumer requests are read-only queries to retrieve current values of data sources.

**Definition 2** A Consumer Request \( cr \) is defined as a tuple of the value of the desired source \( s \), request issue time \( t \), periodicity \( PER \), urgency \( UR \), relative deadline \( RDL \) and accuracy requirements \( PREC \), Bias factor, \( cr := < s, t, PER, UR, RDL, PREC, Bias >. \)

### 3.1 Characterizing QoS and QoD

QoS is a metric of how effective the system handles consumer requests, and is defined as the probability of successful consumer requests. We characterize a successful consumer request \( cr \) by timeliness(deadline) satisfaction and accuracy satisfaction. The deadline of a consumer request \( cr \) is met, if \( TT_{cr} \leq RDL_{cr} \), where \( TT_{cr} \) is the total time taken to complete \( cr \). The accuracy satisfaction is measured by (a) the answer precision \( PREC_{cr} \), i.e., \( PREC_{A} \geq PREC_{cr} \); and (b) the answer fidelity, i.e., the current source value lies within the returned range. The fidelity of answer \( A : [L, U] \) for consumer request \( cr \) is denoted by \( Fidelity(A_{cr}) \) and defined as 1 if current source value falls inside the answer range (i.e., \( L \leq V(cr.s, cr.t) \leq U \), where \( V(cr.s, cr.t) \) is the current value of source \( s \) at \( t \)), otherwise it is 0.

Typically, a successful consumer request not only finishes by the deadline but also delivers the answer at the desired accuracy. if consumer request indicates a bias towards timeliness/accuracy, we consider a consumer request to be successful if the deadline (accuracy) is met. Therefore, we define QoS as follows.
Definition 3 The QoS of the system is defined as follows.

\[
QoS = \begin{cases}
  w_1 \cdot \frac{\|\{cr_j | Bias_{cr_j} = 0, TT_{cr_j} \leq RDL_{cr_j}, Fidelity(A_{cr_j}) = 1, PREC_{A_{cr_j}} \geq PREC_{cr_j}\}\|}{\|\{cr_j | Bias_{cr_j} = 0\}\|} & \text{(for req. without bias)} \\
  + w_2 \cdot \frac{\|\{cr_j | Bias_{cr_j} = 1, TT_{cr_j} \leq RDL_{cr_j}\}\|}{\|\{cr_j | Bias_{cr_j} = 1\}\|} & \text{(for req. favoring timeliness)} \\
  + w_3 \cdot \frac{\|\{cr_j | Bias_{cr_j} = 2, Fidelity(A_{cr_j}) = 1, PREC_{A_{cr_j}} \geq PREC_{cr_j}\}\|}{\|\{cr_j | Bias_{cr_j} = 2\}\|} & \text{(for req. favoring accuracy)}
\end{cases}
\]

Where \( TT_{cr_j} \) is the total processing time of consumer request \( cr_j \), \( RDL_{cr_j} \) is the desired deadline of \( cr_j \), \( PREC_{A_{cr_j}} \) is the precision of answer of \( cr_j \) and \( PREC_{cr_j} \) is the desired precision of \( cr_j \).

Since QoS is maximized when both deadline and accuracy constraints are met when no bias is specified, we set \( w_1 \) to be greater than \( w_2 \) or \( w_3 \); in addition, the ideal case is \( QoS = 1 \), i.e., all the requests satisfy their bias (if any). In our evaluation, we set \( w_1 = 0.5 \) and \( w_2 = w_3 = 0.25 \).

QoD is a metric of how “good” the data is. The metric varies from one system to another. In our system, the QoD is measured by data accuracy. We characterize data accuracy by DS fidelity and DS validity. The DS fidelity measures the divergence between stored range in the DS and the current source value. In contrast, the DS validity with respect to consumer request compares the precision of the stored range of the data with the precision expectation of the consumer request accessing the data. A “good” or an accurate DS maintains a range that not only reflects the current source status (i.e., it is faithful), but also satisfies the precision expectation of consumers (i.e., it is valid).

The DS fidelity of source \( s \) with current value \( V \) at time \( t \) and stored DS interval \((L, U)\) is:

\[
FI_{ds}(s, t) = \begin{cases}
  1 & \text{if } L \leq V \leq U \\
  0 & \text{otherwise}
\end{cases}
\]

Therefore, the DS fidelity of source \( s \) over a certain time period \( T = [t_i, t_j] \) is defined as follows:

\[
FI_{ds}(s, T) = FI(s, [t_i, t_j]) = \frac{1}{T} \times \int_{t_i}^{t_j} FI(s, t)dt.
\]

This is equal to the fraction of time during \( T \) that \( s \) is faithful. If we assume that \( s \) is uniformly accessed during \( T \), then the probability of accessing a fresh value of \( s \) \( p_{fi}(s, T) \) is equal to the fraction of time that \( s \) is faithful [29]. Hence, we can define the aggregate DS fidelity over all sources during the entire time period \( T \) as below:

\[
AFI_{ds}(S, T) = \sum_{s_i \in S} p_{access}(s_i) \times p_{fi}(s_i, T) = \sum_{s_i \in S} p_{access}(s_i) \times FI_{ds}(s_i, T),
\]

where \( p_{access}(s_i) \) is the access ratio of \( s_i \) (the ratio of the number of consumer requesting \( s_i \) to the total number of consumer requests) and \( \sum_{s_i \in S} p_{access}(s_i) = 1 \). Note that when the system performance varies over time, focusing on a narrower time interval for \( T \) would allow applications to tune their responsiveness to such changes.
The DS validity for consumer request \( cr_i \) accessing \( s \) at time \( t \) measures if the DS precision of source \( s \) meets the \( cr_i \)'s precision expectation at \( t \). If the stored DS interval is \((L, U)\), then the DS validity is defined as follows:

\[
VA_{ds}(cr_i(s,t)) = \begin{cases} 
1 & \text{if } PREC(L, U) \geq PREC_{cr_i} \\
0 & \text{otherwise}
\end{cases}
\]

If there are \( k > 0 \) consumer requests accessing \( s \) at time \( t \), then the DS validity for source \( s \) at time \( t \)

\[
VA_{ds}(s, t) = \frac{\sum_{i=1}^{k} VA_{ds}(cr_i(s,t))}{k}
\]

Therefore, if there are \( k_s \) consumer requests accessing \( s \) over a certain time period \( T = [t_i, t_j] \), the DS validity of source \( s \) over \( T \) is defined as follows:

\[
VA_{ds}(s, T) = VA(s, [t_i, t_j]) = \frac{\sum_{u=1}^{k_s} \sum_{v=t_i}^{t_j} VA_{ds}(cr_u(s,t_v))}{k_s}
\]

This is exactly the probability of accessing a valid value of \( s \) during \( T \) \( p_{va}(s, T) \). Therefore, the aggregate DS validity can be defined as follows:

\[
AV A_{ds}(S, T) = \sum_{s_i \in S} p_{access}(s_i) \times p_{va}(s_i, T) = \sum_{s_i \in S} p_{access}(s_i) \times VA_{ds}(s_i, T),
\]

where \( p_{access}(s_i) \) is the access ratio of \( s_i \) (the ratio of the number of consumer requesting \( s_i \) to the total number of consumer requests).

As stated before, DS accuracy is the combination of DS fidelity and DS validity.

**Definition 4** The overall QoD or Aggregate DS Accuracy \( p_{aac}(S) \) is defined as follows:

\[
QoD = p_{aac}(S, T) = AFI_{ds}(S, T) \times AV A_{ds}(S, T)
\]

### 3.2 Problem Statement

Formally stated, given a set of \( n \) sources \( S = \{s_1, s_2, ..., s_l\} \) and an input instance (request set) \( I \), which is a collection of \( m \) incoming source update requests and \( n \) consumer requests \( I = SR \cup CR = \{sr_1, sr_2, ..., sr_m; cr_1, cr_2, ..., cr_n\} \), our objective is to maximize EoS (efficiency of the system)

\[
EoS = \frac{QoS \cdot QoD}{Cost}
\]

In other words, our objectives are to:

- maximize overall QoS indicated by the probability of successful consumer requests that meet deadline and/or accuracy requirements.
- maximize overall QoD indicated by the probability of accessing accurate data in the DS.
- minimize overall Cost involved in the process of maintaining the DS and also in serving requests.

A source update request \( sr_i \) (Definition 1) arrives with explicit indication of its urgency, periodicity and relative deadline; consumer request \( cr_j \) (Definition 2) arrives with its desired periodicity, relative
deadline, accuracy and its bias of timeliness against accuracy. The effectiveness of handling consumer requests is represented by QoS (Definition 3), the effectiveness of handling source update requests is represented by QoD (Definition 4), and the communication overhead involved in the process of serving all the requests is represented by Cost which is the average number of messages exchanged per request. The cost includes all messages exchanged for maintaining the DS and for additional probing that may service the accuracy requirements of consumer requests.

In practice, due to highly dynamic system and network conditions, unpredictable application workloads, and frequently changing information sources, the joint optimization of these three factors is very complicated. Therefore, we aim to find good heuristics that addresses the tradeoff between timeliness (QoS), accuracy (QoD) and overhead. In fact, the inter-relationship between QoS and QoD is not straightforward. One may argue that by maintaining an accurate DS (i.e., improving QoD), the number of deadline meets can be increased (i.e., improving QoS is achieved) since additional probes are avoided. A natural approach to improve QoD is to give a higher priority to source update request. However, this may result in an increase in the number of missed deadlines and consequently decreasing QoS. This complex inter-relationship illustrates the need for a scheduling mechanism that can work in concert with the DS maintenance component. Therefore, we frame the tradeoff as two sub-problems. We manipulate QoS by proposing an algorithm to schedule incoming source update requests and consumer requests, aiming to maximize their accuracy and deadline constraints, hence desired QoS is achieved. This is done under the assumption that the directory service is maintained reasonably accurate. Simultaneously, we focus on adjusting QoD by presenting an efficient directory service maintenance algorithm that works in concert with the scheduling mechanism. The DS maintenance algorithm focuses on maximizing QoD without increasing the management overhead. Combining these two algorithms, the overall EoS is expected to improve and our experiments show that the proposed approaches result in very good EoS.

4 Design of Information Mediator to Balance QoS and QoD

The information mediator schedules and processes consumer/source update requests, as well as maintains the DS. These tasks are independent and also interrelated, we therefore propose three sub-components (scheduler, request servicer and DS maintainer) for the mediator. The scheduler (where TABS is implemented), aims to provide better QoS; the DS maintainer (where MC is realized) is responsible for data freshness (QoD); and the request servicer sub-component decides the specifics of processing each dispatched request, and serves as a conduit between the scheduler and the DS maintainer. The request servicer determines whether the values stored in the DS are accurate enough for incoming re-
quests, and notifies the DS maintainer when accuracy violations occur. Based on this feedback, the DS maintainer adjusts its policy accordingly so that the DS is maintained at a reasonable accuracy level. In other words, via the request servicer, QoD is maintained by the DS maintainer to assist the scheduler in achieving better QoS. This is because QoD has a direct impact on the frequency of future source update requests and consumer requests, which affects the system load and schedulability, thus indirectly has an impact on QoS.

A good scheduler and request manager will ensure that each request will not wait for too long to get processed and that requests with deadlines will have the maximum possibility of satisfying their deadlines. An effective DS maintenance module will keep the DS entries at the suitable level of accuracy so that requests can be served directly from the DS; this in turn, will shorten the execution time. In the following sections, we present detailed techniques for scheduling (section 4.1), request servicing (section 4.2) and DS maintenance (section 4.3).

4.1 Timeliness-Accuracy Balanced Scheduling (TABS)

Given that requests can arrive from both data sources and consumers, the request scheduler must very carefully balance the requirements of consumer requests and their deadlines (QoS) against the need to keep the DS entry up-to-date (QoD). If source updates are executed with higher priority, the system may be left with no time to meet the deadlines of consumer requests. On the other hand, if consumer requests are given preference, they might read stale data. Neither of these outcomes are desirable. Missing deadlines might mean missing opportunities, operating on stale data might mean making wrong decisions. The scheduling component must therefore determine the order in which the incoming source and consumer requests are handled so as to maximize the possibility of meeting both accuracy and deadline constraints.

Solutions to address the conflict between user request timeliness and data freshness have been developed in the context of real-time databases (RTDB) [2, 17, 28]. These scheduling approaches may be generalized in our context as follows:

- **Source update request First (SF):** This approach applies the source update request when it arrives at the system, i.e., it gives all source update requests higher priority than all consumer requests.
- **Consumer request First (CF):** This approach applies source request updates only when no consumer requests are waiting.
- **Split Updates (SU):** This approach is a compromise between CF and SF and classifies data objects as being popular and unpopular. Source update requests to popular data will be applied on arrival and updates to less popular data will be applied when no consumer requests are waiting.
- **On-Demand source request updates (OD):** This approach is an extension to the CF (consumer first)
policy where consumer requests are normally given precedence over source update requests. However, when a consumer request encounters a stale object (i.e. an object for which there exists a source update that has not yet been applied), the corresponding source update is given precedence.

There are some basic issues in mapping RTDB solutions to the information collection scenario. Firstly, in the RTDB context, consumer requests correspond to transactions (a series of read/write requests), which typically take longer time to be processed. Secondly, source update requests in the RTDB context do not have deadlines associated with them, enabling them to be treated differently from real-time transactions. In our case, more fine-grained and timely interaction between the consumer and system is needed. Both source and consumer requests represent a single operation on a single data source. Furthermore, both source and consumer requests may specify timeliness requirements and a uniform mechanism for assigning a scheduling order to both types of requests is needed. We propose a Timeliness-Accuracy Balanced Scheduling (TABS) mechanism to balance timeliness and accuracy.

TABS attempts to schedule consumer and source update requests to ensure that deadlines are met and source update requests are processed rapidly enough to maintain accuracy. A suitable scheduling mechanism must address the following issues to obtain a balance between QoS and QoD: (1) Decide on an ordering of the incoming source update requests (2) Decide on a relative ordering of source update and consumer requests. We classify all incoming source update and consumer requests into four categories shown in Table 1 based on periodicity and deadline. We motivate the utility of these categories via the following example. Consider a toxic chemical detection system that continuously monitors the density of a certain toxic chemical in an area. Under normal conditions, a periodic non-deadline based query is issued to the system; interested users may issue aperiodic non-deadline based queries to check the density irregularly. When the density of the toxic chemical is above certain threshold, aperiodic deadline-based queries with explicit deadlines are issued so that a chemical threat can be quickly identified and false alarms can be avoided. Once a real threat is identified, periodic deadline-based queries may be issued to provide timely and accurate density level information to aid emergency response teams in mitigating the hazard.

<table>
<thead>
<tr>
<th>request category</th>
<th>absolute deadline $ADL_{r_i}(s,t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>periodic and deadline based</td>
<td>P-DL</td>
</tr>
<tr>
<td>periodic and non-deadline based</td>
<td>P-NODL</td>
</tr>
<tr>
<td>aperiodic and deadline-based</td>
<td>AP-DL</td>
</tr>
<tr>
<td>aperiodic and non-deadline based</td>
<td>AP-NODL</td>
</tr>
</tbody>
</table>

$t + RDL_{r_i}$

$t + PER_{r_i}$

$max(t, ADL_{ar_{k-1}}) + E_i/U_{AP}$

$max(t, ADL_{ar_{k-1}}) + E_i/U_{AP}$

Table 1: Absolute deadline assignments for incoming requests
Source update requests arrive at the system either periodically or sporadically. The objective here is to determine dynamically the update schedule which maximizes the overall QoD. If multiple source update requests to the same source exist, the most recent update will be processed first (i.e. the most recent value is the candidate for a DS update). The remaining values will be processed with lower priority (for archival purposes). We then prioritize source update requests from different sources based on the popularity (of the source), urgency and deadline of the request.

The problem of scheduling a mixed set of hard periodic tasks and soft aperiodic tasks in a dynamic environment has been widely considered when periodic tasks are executed under an EDF algorithm [12, 13, 21]. The goal of such EDF based joint scheduling algorithms has been (a) to meet all the deadlines of periodic tasks and (b) to minimize the average response time for aperiodic tasks. A thorough comparison of several scheduling techniques in terms of performance, schedulability and implementation complexity is presented in [48]. Among those studied techniques, the Total Bandwidth Server algorithm exhibits superior overall performance and low implementation complexity. It assumes that all periodic tasks have hard deadlines (i.e. their periods), and all aperiodic tasks do not have deadlines.

We develop a Timeliness-Accuracy Balanced Scheduling (TABS) algorithm for real-time joint scheduling of source update requests and consumer requests using the TBS algorithm as a basis and assign an absolute deadline for each incoming request (see Table 1). We use the following assumptions and terminology:

- Each periodic request \( r_i \) has a constant period \( PER_i \) and a constant worst case execution time \( E_i \);
- All periodic requests \( r_i : i = 1, ..., n_p \) have deadlines specified either as \( RDL \) for P-DL or as its period \( PER \) for P-NODL;
- The arrival time of the \( l^{th} \) periodic instance is given by \( AR_i(l) = AR_i(l-1) + PER_i \);
- The absolute deadline of the \( l^{th} \) periodic instance is given by \( ADL_i(l) = AR_i(l) + RDL_i \) for P-DL or \( ADL_i(l) = AR_i(l) + PER_i \) for P-NODL;
- All aperiodic deadline based requests \( ard_j : j = 1, ..., n_{apd} \) have relative deadlines \( RDL \), so its original absolute deadline \( OADLard_j = ARard_j + RDLard_j \);
- All aperiodic non-deadline based requests \( arnd_k : k = 1, ..., n_{apnd} \) do not have deadlines;
- The worst case execution time of each aperiodic request \( E \) is known at its arrival time.

**Assigning Absolute Deadlines for Aperiodic Requests:** We define the utilization factor for periodic requests as \( U_P = \sum_{i=1}^{n_p} \frac{E_i}{\min(RDL_i, PER_i)} \), and the capacity of the total bandwidth server as \( U_{AP} \). To improve the response time of aperiodic requests, we assign a possible earlier deadline to each aperiodic request by applying the TBS algorithm [48]. Therefore, each time an aperiodic request enters the system, we optimistically assign it a deadline assuming that the total bandwidth of the server (\( U_{AP} \))
can be allocated to that request immediately. When the $k^{th}$ aperiodic request arrives at time $t = AR_k$, it receives a deadline

$$ADL_k = \max(AR_k, ADL_{k-1}) + \frac{E_k}{U_{AP}},$$

where $E_k$ is the worst-case execution time of the request. By definition $ADL_0 = 0$. If the request is aperiodic deadline based, we compare the assigned absolute deadline $ADL_k$ with its original absolute deadline $OADL_k$. If $ADL_k > OADL_k$, then the request is rejected. Otherwise, the request is inserted into the ready queue of the system and scheduled by EDF [32], as any other periodic instance or aperiodic request already present in the system. Intuitively, the assignment of the deadlines is such that in each interval of time the ratio allocated by EDF to the aperiodic requests never exceeds the server utilization $U_{AP}$, that is, the processor utilization of the aperiodic requests is at most $U_{AP}$ [49].

**Operational Flow of The TABS Algorithm:** Based on the discussion above, for each incoming request, we set an absolute deadline by which the request must be completed(Table 1). Requests are ordered in their corresponding queues (source update and consumer request queue) by their absolute deadlines. When two requests have the same deadline, we calculate priority values to break the tie. The assigned priority value reflects the popularity of the requested source ($POP_s$) and urgency of the request ($UR_{ri}$) and is calculated as: $PR_{ri(s,t)} = W_{pop} \cdot POP_s + W_{ur} \cdot UR_{ri}$. A request with a higher priority value will be assigned and dispatched earlier than a request with a lower priority value.

When a new request (either from consumer or source) arrives, the scheduler inserts it into the corresponding queue so as to preserve deadline ordering and additionally preserve priority ordering for multiple requests with the same deadline. When a dispatched request finishes processing (at the request servicing module), the scheduler decides which request to process by comparing the deadlines in both source update request and consumer request queues: the request with the earliest absolute deadline or the highest priority is the next one to be dispatched to the request servicing module. Figure 2 shows the outline of the scheduling algorithm.

**Lemma (TABS Schedulability):** Given a set of $n_p$ periodic requests with processor utilization $U_P$, and a TB server with processor utilization $U_{AP}$, the whole set of requests is schedulable if $U_P + U_{AP} \leq 1$.

**Proof:** Suppose there is an overflow at time $t$. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point $t'$ on, only instances of requests (periodic or aperiodic) ready at $t'$ or later and having deadlines less than or equal to $t$ are run. Let $E$ be the total execution time demanded by these instances. Since there is an overflow at time $t$, we must have $t - t' < E$. Let $E_{AP}$ be the total execution time required by aperiodic requests arrived at $t$ or later and processed with deadlines less than or equal to $t'$, then $E_{AP} \leq (t' - t)U_{AP}$ (Lemma 2 in [48]). We also
TABS()
/* thread 1: to manage both queues */
for each incoming request r
{  PR_r = W_{pop} \ast POP_s + W_{ur} \ast UR_r;
  switch (request category)
  {  case P-DL: /* Periodic with deadline*/
    ADL_r = t + RDL_r; break;
    case P-NODL: /*Periodic w/o deadline*/
    ADL_r = t + PER_r; break;
    case AP-DL: /* Aperiodic with deadline*/
    case AP-NODL: /*Aperiodic w/o deadline*/
    ADL_r = \max(t, ADL_{ar_{j-1}}) + \frac{E_i}{U_{AP}};
    break;
  }
  call insert(r);
}
/* thread 2: to dispatch requests in ready queues */
while (true)
{  /* select the request with the earliest deadline or the highest priority*/
  r^* = \{r | ADL_r \leq ADL_{sr_i}, 1 \leq i \leq m \text{ and } ADL_r \leq ADL_{cr_j}, 1 \leq j \leq n\};
  call RequestServicing(r^*);
}

insert(r)
{  switch (request type)
  {  case (source update request sr):
    /* insert into source update req. queue */
    /* so as to preserve EDF order */
    SR = SR \cup \{sr\} such that
    ADL_{sr_i} < ADL_{sr_{i+1}} or
    PR_{sr_i} \geq PR_{sr_{i+1}}, i = 1, \cdots, m - 1;
    break;
    case (consumer request cr):
    /* insert into consumer request queue */
    /* so as to preserve EDF order */
    CR = CR \cup \{cr\} such that
    ADL_{cr_j} < ADL_{cr_{j+1}} or
    PR_{cr_j} \geq PR_{cr_{j+1}}, j = 1, \cdots, n - 1;
    break;
  }
}

Figure 2: The TABS algorithm

know that
\[
E \leq \sum_{i=1}^{n_p} \frac{t - t'}{\min\{RDL_i, PER_i\}} E_i + E_{AP}
\leq \sum_{i=1}^{n_p} \frac{t - t'}{\min\{RDL_i, PER_i\}} E_i + (t - t')U_{AP}
\leq (t - t')(U_P + U_{AP}).
\]

It follows that $U_P + U_{AP} > 1$, a contradiction. \hfill \Box

4.2 Time-sensitive Request Servicing

The request servicer accepts a source update or consumer request selected by the scheduler and determines the specifics of how individual requests will be processed in the system so as to satisfy the timeliness/accuracy/cost tradeoff. To relieve the DS access contention and speed up request processing, we introduce a small local cache in the mediator. Assume that we have a two-level store consisting of a local cache (LC) of size $k$ and the DS of size $n$. Initially, the LC is filled with the bounds of the first $k$ requested sources. If the source requested by a consumer is in the LC (a hit) and the accuracy con-
straints are met, then the cached value is returned to the consumer. This reduces DS access contention and reduces the time required to process the incoming request. If a cache miss occurs, the DS must be contacted to retrieve the current value/range and the newly requested value replaces the least recently used (LRU) entry in the cache. We now discuss the details of how source update and consumer requests are dealt with in the request servicing module.

**Handling Source Update Requests:** We divide source update requests into two types: Type 1 requests are source update requests reporting current status caused either by a periodic update or an abnormal conditions report. Type 2 requests are consumer-initiated source update requests that improve the accuracy of answers to consumer requests. Type 2 source update requests inherit $DL$ and $UR$ from the triggering consumer requests. When the request servicer accepts a source update request, it forwards the request to the DS maintenance module that ultimately determines whether the DS should be updated. Note that any DS updates will invalidate corresponding LC entries.

**Handling Consumer Requests:** Handling consumer requests is more complicated than handling source update requests since consumer requests may have accuracy and timeliness requirements. When the request servicer accepts a consumer request, more careful analysis is needed to determine whether both accuracy and timeliness requirements can be met (ideally). If only the preferred requirement (as indicated by $Bias$) can be met, the request servicing module has to choose among several processing options. Figure 3 illustrates the delays incurred during different operations.

We break down the request processing time into time spent in each step. Let $T_{sched}$ be the scheduler...

---

**Figure 3:** Processing delays for request servicing

**Figure 4:** The request servicing algorithm
processing time; $T_{rmgr2ds}$ be the data request message transfer time from the request servicer to the directory service; $T_{rmgr2src}$ be the message transfer time for the source probe from the request servicer to the source; $T_{ds2rmgr}$ be sum of the directory service access time and the message transfer time from the DS to the request servicer; and $T_{src2sched}$ be the source update request transfer time from the source to the scheduler. Each consumer request follows the same path 1→2→3 prior to reaching the request servicer. Subsequent to that, there are three possible paths/ways to serve the consumer request: (1) PATH_A (3→1 in Figure 3): in the case of an LC hit, the value may be directly obtained from the cache; (2) PATH_B (3→4→3→1): in the case of an LC miss, we obtain the desired value (range) from directory service; (3) PATH_C (3→4→3→5→2→3→1&6): if accurate data is needed based on policy, we can probe the source to obtain current exact value, return a response to the consumer, and concurrently issuing a DS update (if necessary). Therefore,

$$T_{PATH_A} = 0,$$

$$T_{PATH_B} = T_{rmgr2ds} + T_{ds2rmgr},$$

$$T_{PATH_C} = T_{rmgr2ds} + T_{ds2rmgr} + T_{rmgr2src} + T_{src2sched} + T_{sched}.$$ 

Each sub-latency $T_i$ is bounded by the best case and the worst case latency $[T_i(L), T_i(H)]$, so the total latency is bounded. i.e., if $T_i(L) \leq T_i \leq T_i(H)$ for any $i$, then $\Sigma T_i(L) \leq \Sigma T_i \leq \Sigma T_i(H)$. Given the estimated latency bounds and the timeliness/accuracy constraints of a consumer request $cr$, we can roughly determine if the deadline can be met as follows:

- if $DL_{cr} \leq \Sigma T_i(L)$, then it is not possible to satisfy the timeliness bound;
- if $\Sigma T_i(L) < DL_{cr} \leq \Sigma T_i(H)$, then it is uncertain if the deadline will be met or not. The consumer specified Bias factor plays a role in determining the path of the request in this case.
- if $DL_{cr} > \Sigma T_i(H)$, then the deadline can be met.

The general flow of handling consumer requests is shown in Figure 4. A consumer request $cr$ containing necessary parameters (as defined in Definition 2) is received by request servicer. We decompose the processing of $cr$ into a mandatory part followed by an optional part. As part of the mandatory processing, the current value is obtained from LC and/or the DS. If the obtained value satisfies the accuracy constraints of $cr$, it is returned as the result. Otherwise, we must determine whether probing the source is necessary, i.e, whether the optional PATH_C needs to be taken or not. Analysis of latency involved in PATH_C indicates that PATH_C will improve data accuracy but introduce a larger delay. Hence, providing the best service (accurate answer and timely response) for each incoming request may not lead to the best overall system performance. In contrast, by sacrificing the quality of some individual consumer requests, the overall QoS and QoD may be enhanced. In order to address this tradeoff,
we propose local and global optimization policies that decide if $PATH_C$ will be taken.

**Local/Global Policies for Path Selection:** Our objective is to support both timeliness and accuracy and tailor the result to the consumers' preference if both constraints cannot be satisfied at the same time. We describe the local and global optima policies in more details below.

- **Local Optima (LO):** This policy maximizes the accuracy while ensuring that the deadline is met, i.e., if the deadline will not be violated, $PATH_C$ will be taken even if the precision from the LC or the DS is good enough. More specifically, there are the following three cases.

  - if $DL_{cr} \leq T_{PATH_C}(L)$, the value stored in the DS is returned as the answer;

  - if $T_{PATH_C}(L) < DL_{cr} \leq T_{PATH_C}(H)$, the deadline may be violated, whether or not $PATH_C$ is taken is dependent upon $Bias_{cr}$. If $Bias_{cr} = 0$ or $Bias_{cr} = 2$, i.e., the consumer has no preference or prefers to ensure accuracy, then $PATH_C$ will be taken and the obtained current value is returned as the answer; otherwise if $Bias_{cr} == 1$, i.e., the consumer prefers to ensure timeliness, then $PATH_C$ will not be taken and the value stored in the DS is returned as the answer.

  - if $DL_{cr} > T_{PATH_C}(H)$, then $PATH_C$ will be taken, i.e the source will be probed regardless of $PREC_{ds}$ or $PREC_{lc}$, so the obtained current value is returned as the answer.

- **Global Optima:** From the overall system performance perspective, global optimization can focus on ensuring either timeliness or accuracy.

  - Maximize the deadline-meet ratio ($G_{dl}$): This policy maximizes the number of deadlines met by returning current value in the DS without probing the sources, thereby decreasing the processing time for each request.

  - Maximize the accuracy-meet ratio ($G_{ac}$): This policy maximizes the accuracy of the result by always probing the sources for the current exact value regardless of whether it will violate the deadline or not.

### 4.3 Cost-based Directory Service Maintenance

The directory service maintenance module is responsible for keeping the directory service accurate enough so that most of the consumer requests can be served directly by consulting the directory service without probing the sources. As a result, communication overhead and delay are both reduced. Furthermore, as stated in Section 3, the directory service maintains a range instead of an instantaneous value for each source.
Different applications are interested in the source values at different granularity, so it is desirable to sample the sources at a very high frequency initially so that the dynamics of the underlying sources can be captured. The goal of the DS maintenance algorithm is to find an efficient way to adjust collection parameters (sampling frequency, range size) so that desired information accuracy is maintained while minimizing the communication overhead. i.e., the algorithm should minimize the information collection cost while still maintaining reasonable information precision that can satisfy user requirements. We summarize existing approaches to address the cost and accuracy tradeoff inherent in the information collection problem.

• **Instantaneous Snapshot Based Information Collection (SS):** In this policy [50], information about the desired parameters (e.g. residue capacity of network nodes and server nodes) is based on an absolute value obtained from a periodic snapshot. During each sampling period, probing is initiated to gather the current information from the managed entities (e.g. router nodes); and the information repository (e.g. the directory service) is updated with the collected values. The challenge here is to determine a sampling period so that accurate DS is maintained without incurring very high overhead. Our studies indicate that for a variety of traffic conditions, a sampling period of 10 seconds yields the best overall cost-performance tradeoff.

• **Static Interval Based Information Collection (SI):** In this policy [3], we define a fixed interval $B$ which is used to partition the capacity of the collected information into a fixed number (say $n$) of equal size classes: $(0, B), (B, 2B), (2B, 3B), ..., ((n-2)B, (n-1)B)$. The classes are represented by corresponding indices $0, 1, 2, ..., (n-1)$. A probe is initiated at each sampling interval to obtain current information from the managed entities. If the obtained value is out of the range indicated by the current index, the repository is updated with another index, otherwise no update is needed.

• **Dynamic Range Based Information Collection:** In this policy, the information repository holds the monitored parameter using a range with an upper bound $U$ and a lower bound $L$; the range may be modified dynamically based on the sampled information. Several dynamic range based strategies have been studied to address the cost-accuracy tradeoff, and they can be further classified by whether or not sampling is used. Two sampling based approaches have been studied in the context of network management applications. In these two approaches, a stable value initiates range tightening, thereby enhancing accuracy; a fluctuating value enlarges the range. The first approach uses statistical analysis techniques based on time series [19] to derive a range and a sampling rate such that the deviation between the predicted and observed values remains in the range with a given confidence level. The second approach is a simpler throttle-based approach (TR) [22], where ranges are increased or decreased exponentially using a pre-specified throttle factor. In our prior studies, we have shown that the simpler
TR approach works well for network management. A non-sampling cost-driven policy proposed in [36] tightens the range when incoming consumer requests do not satisfy accuracy constraints and relaxes the range when source updates indicate a range violation.

The Minimized Cost DS Maintenance Algorithm (MC)

In this section, we propose the Minimized Cost DS Maintenance Algorithm (MC), a dynamic range based approach with restricted sampling. The MC algorithm is implemented in the DS maintenance module that balances the cost/accuracy/timeliness tradeoffs. Our algorithm can cater to a variety of sources (e.g., sensors with very limited resources and intelligence, or network intermediate components like routers with sufficient resources). The information sources, in our case, are assumed to be simple. The sources have the basic intelligence to respond to sampling requests and can be programmed to report status periodically. The basic approach is to tailor the collection process to account for consumer accuracy requirements using a cost-based approach that accounts for both value and consumer initiated source updates.

The basic MC algorithm (See Figure 5A) executes as follows. As illustrated in Figure 1, the request management module forwards source update requests to the DS maintenance module. The source updates may be a result of periodic probing, an abnormal condition report update (source value falls outside predetermined threshold) or a consumer-initiated source update (to improve result accuracy for the consumer). Any of these updates provide an opportunity for the range to be adjusted. The objective in selecting a good range size is to avoid the need for future updates, since we want to minimize the communication cost. To avoid consumer-initiated updates, the range should be as small as possible. On the other hand, to avoid value-initiated updates, the range should be large enough to more accurately reflect the source value changes.

We use a cost-driven process to determine when and how the DS range should be altered. The cost analysis follows an approach similar to [36]. The cost incurred during a consumer-initiated update is denoted by \( C_{cu} \). A value-initiated update incurs cost \( C_{vu} \) and occurs whenever the exact value at a source exceeds its range at the DS. Let \( P_{vu} \) and \( P_{cu} \) represent the probability that a value- and consumer-initiated update will occur at each time step. Then the expected cost rate per time step is

\[
C = P_{cu} \cdot C_{cu} + P_{vu} \cdot C_{vu}.
\]

Previous work justifies that \( P_{vu} = \frac{K_1}{R^2} \) and \( P_{cu} = K_2 \cdot R \) [36], where \( K_1 \) and \( K_2 \) are model parameters that depend on the characteristics of source updates and consumer requests. Therefore, \( C = K_2 \cdot R \cdot C_{cu} + \frac{K_1}{R^2} \cdot C_{vu} \). By setting the derivative of this equation to be zero, we can get the optimal range \( R \) which introduces the minimum cost; at the same time, we observe that the following condition must hold \( \rho \cdot P_{vu} = P_{cu} \), where \( \rho = 2 \cdot \frac{C_{vu}}{C_{cu}} \).
MC(request sr):
{
    switch (sr.type)
    {
    case Type1:
        if (sr.V < L or sr.V > U)
        {
            /∗update DS based on the source changes;*/
            with probability min{ρ, 1}:
                call adjustR();
        }
    apply Lazy Sampling;
    break;
    case Type2:
        with probability min{1/ρ, 1}:
            set \( R_{\text{new}} = R_{\text{old}}/(1 + \alpha) \);
            \( L = sr.V - \frac{R_{\text{new}}}{2} \);
            \( U = sr.V + \frac{R_{\text{new}}}{2} \);
            if (s ∈ LC and the DS is updated)
                update the range for s in the LC;
            break;
    }
}

adjustR()
{
    get the median of current monitoring window \( V_{med} \);
    compute slope of current monitoring window \( m_w \);
    compute slope of previous monitoring window \( m_{w-1} \);
    if \( (m_w \geq TH_m) \)
        switch \( (m_w) \)
        {
        case \( m_w > m_{w-1} + \varepsilon \):
            R_{\text{new}} = R_{\text{old}} * (1 + \alpha); break;
        case \( m_{w-1} - \varepsilon \leq m_w \leq m_{w-1} + \varepsilon \):
            R_{\text{new}} = R_{\text{old}}; break;
        case \( m_w < m_{w-1} - \varepsilon \):
            R_{\text{new}} = R_{\text{old}}/(1 + \alpha); break;
        }
    } else { /*m_w < TH_m */
        switch \( (m_w) \)
        {
        case \( m_w > m_{w-1} + \varepsilon \):
            R_{\text{new}} = R_{\text{old}}; break;
        case \( m_{w-1} - \varepsilon \leq m_w \leq m_{w-1} + \varepsilon \):
            R_{\text{new}} = R_{\text{old}}; break;
        case \( m_w < m_{w-1} - \varepsilon \):
            R_{\text{new}} = R_{\text{old}}/(1 + \alpha); break;
        }
    }
    \( L = V_{med} - \frac{R_{\text{new}}}{2} \); \( U = V_{med} + \frac{R_{\text{new}}}{2} \);
}

A. The flow of the MC algorithm

B. The specifics of value-initiated updates

Figure 5: The MC algorithm

In order to maintain this condition, it is not necessary to adjust the range on every update. We
now determine conditions under which range updates happen. In cases where \( \rho < 1 \), it is desirable
for value initiated updates to be more likely than consumer-initiated updates. Thus, the range is
decreased on every consumer initiated update but only adjusted with probability \( \rho \) on value-initiated
updates. Conversely, in cases where \( \rho > 1 \), the range is adjusted on every value initiated update but
only decreased with probability \( \frac{1}{\rho} \) on consumer-initiated updates.

Having determined when to update the range, we now proceed to determine how to modify the
range. Let \( R_{\text{new}} \) represent the new range and \( R_{\text{old}} \) the current range in the DS. Let \( \alpha \) be the adaptation
parameter. A consumer-initiated update is applied by setting \( R_{\text{new}} = \frac{R_{\text{old}}}{1+\alpha} \) with probability \( \min\{\frac{1}{\rho}, 1\} \).
A value-initiated update is applied with with probability \( \min\{\rho, 1\} \). The process of determining the
new range is however more complicated than with consumer initiated updates.

A Curve Fitting Approach for Value-Initiated Updates: In some earlier approaches [36, 22],
consumer-initiated updates always initiated range reduction, value-initiated updates always triggered range relaxation. However, we observe that range relaxation for all value-initiated updates does not truly reflect the value changes. The fact that a source value falls outside the current range does not always imply that the range is too small and needs to be increased; it can also occur when the source value gradually shifts towards a new set of values. In our approach to value-initiated range adjustment, the general idea is to compare the approximate trend of source value changes using the following steps (Figure 5B illustrates the details). Whether the range should be expanded or tightened is dependent upon (a) the absolute slope of the trend of source value changes during the current monitoring period and (b) the difference between the current slope and previous one. To find the slope of the trend of source value changes, we use a Least Squares curve fitting method to find the most suitable line to approximate source value changes. The slope of the obtained line represents the rate of change of source values. If the values in a sliding window \( w \) of size \( N \) are \( V_1, V_2, \ldots, V_N \), then the slope \( m_w = \frac{n \sum xy - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2} \), where \( \Sigma \ldots \) stands for \( \Sigma_{i=1}^{N} \ldots i \). We compare \( m_w \) with a specified threshold \( TH_m \) to decide whether the source change is significant enough. The comparison between \( m_w \) and \( m_{w-1} \) is made against a pre-defined \( \varepsilon \).

When the current change exceeds the threshold, we adjust the range as follows:
- the range size is increased to accommodate more changes if the current change \( m_w \) is bigger than the previous change \( m_{w-1} \);
- the range size remains the same if \( m_w \) is similar to \( m_{w-1} \);
- the range size is decreased if the change is smaller than the previous one.

When the current change is not significant enough, the range size is decreased if the previous change \( m_{w-1} \) is larger than the current change \( m_w \), otherwise the range size remains the same. When the range is tightened, \( R_{new} = R_{old}/(1 + \alpha) \); when the range is relaxed, \( R_{new} = R_{old} \cdot (1 + \alpha) \). No matter how the range is changed, we let the median value \( V_{med} \) in current sliding window fall at the center of the range, i.e., \( U = V_{med} + \frac{R_{new}}{2} \) and \( L = V_{med} - \frac{R_{new}}{2} \).

**Lazy Sampling to reduce probing cost:** To reduce the cost incurred by periodic probing, we apply a lazy sampling strategy. The sampling frequency can be reduced in two ways (a) If the number of source update requests \( N_{sr} \) that do not cause the DS updates in a given period is more than a predetermined value \( TH_{sr} \), we can infer that the current approximation of the source value is reasonable (i.e., the source value does not change very dramatically), therefore we can reduce the sampling frequency; (b)if the range is relaxed to exceed a certain value (e.g., \( 1/4 \) of the maxima \( R_{max} \) as indicated in our experiments), it is likely that the range is large enough to accommodate reasonable changes in the source value, hence we can reduce the sampling frequency. In both cases, we decrease the sampling frequency, i.e. \( SF_{new} = SF_{old}/(1 + \beta) \).
Integrating TABS and MC: The integration of our proposed algorithms for request scheduling (TABS), servicing and DS maintenance (MC) maintains a good balance between QoS, QoD and Cost. TABS provides a joint scheduling of both source update requests and consumer requests, thus ensuring that one type of requests are not delayed because extensive resources are allocated to the other type. The DS maintenance algorithm keeps track of the changes in the source values and system conditions, adjusts DS representations to reflect these changes, and makes sure that the DS closely reflects the real world without incurring too much maintenance overhead. The well-maintained DS provides TABS with convincing information to make its scheduling decisions. TABS, in turn, provides MC with valuable feedback about whether each incoming request meets its requirement. These information exchanges are done via request servicer. Overall, the combination of our proposed approaches performs very well under varying system conditions, which will be shown in the following section.

5 Performance Evaluation

The objective of the simulation is to study in detail the performance of the system by comparing our algorithms with existing algorithms proposed by other previous work. Table 2 illustrates the policies we evaluated for scheduling, request servicing and DS maintenance.

<table>
<thead>
<tr>
<th>Scheduling Policies</th>
<th>Request Management Policies</th>
<th>DS Maintenance Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABS (Timeliness-Accuracy Balanced)</td>
<td>LO (Local Optima)</td>
<td>MC (Minimized Cost based)</td>
</tr>
<tr>
<td>FCFS (First Come First Serve)</td>
<td>$G_{dl}$ (Global opt. timeliness)</td>
<td>SS (System Snapshot based)</td>
</tr>
<tr>
<td>CF (Consumer request First)</td>
<td>$G_{ac}$ (Global opt. accuracy)</td>
<td>SI (Static Interval based)</td>
</tr>
<tr>
<td>SF (Source update request First)</td>
<td></td>
<td>TR (Throttle based)</td>
</tr>
<tr>
<td>SU (Split Update)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OD (On-Demand update)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Scheduling, request servicing and DS maintenance policies studied

While we studied all the 72 combinations of the policies, we only illustrate the key results here. As described in the problem formulation, our objective is to maximize the EoS of the overall system. Our performance study consists of the following specific experiments:

(a) Evaluation of all the possible policy combination in terms of the overall EoS;
(b) Evaluation of system heterogeneity: in a general architecture, source capabilities can vary significantly. For instance, some sources can only respond to a request, while others have certain computing capability. Also, different applications exhibit varying deadline requirements from the information collection process. We study the system heterogeneity under these variations.
(c) Evaluation of relative merit of adding intelligence into each module of the mediator and the benefit of intelligent mediator policies as the system scales.
5.1 The Simulation Environment

The simulation environment consists of five components as in the framework: source, consumer, mediator, information repository and queues. The simulation settings for the information source and the two types of incoming requests (source update request and consumer request) are described as follows.

There are $n = 200$ sources in the system. Each source holds one exact numeric value, the local cache in the request servicing module holds $k \leq n$ interval approximations to exact source values, and the DS holds all the interval approximations. Source values $V_s$ are picked randomly and uniformly in the range $[-150, 150]$; source values are changed periodically (initially set to be every 100 milli-second): some of the sources change their values very slightly from $\pm 0.5$ to $\pm 1.5$, while others change more dramatically from $\pm 5$ to $\pm 15$.

For each source, a source update request is sent out regularly and the period is uniformly distributed in the range $(100\,ms, 50s)$. The arrivals of aperiodic source update requests are dependent upon their source value changes: we randomly pick sources that send out urgent source update requests when their values reach certain thresholds. For requests that have timeliness requirements, deadlines are uniformly distributed in the range $(500\,ms, 1\,min)$. All sources have an equal probability of generating a source update request.

One periodic consumer request is issued for each source and its period is uniformly distributed in the range $(100\,ms, 50s)$. Aperiodic consumer request arrival is modeled as a Poisson process with arrival rate $10/sec$ and with inter-arrival times being exponentially distributed. Deadlines associated with consumer requests are uniformly distributed in the range $[500\,ms, 1\,min]$. It is randomly decided whether the aperiodic consumer request is deadline or non-deadline based. Furthermore, each source has an equal probability of being requested by consumer requests. The urgency of each consumer request is randomly chosen to be 0, 1 or 2. Each request is accompanied by an accuracy constraint $\text{PREC}_{cr}$ specifying the maximum acceptable width of the result. The accuracy constraint are generated based on parameters $\text{PREC}_{avg} = 20$ (average accuracy constraint) and $\text{PREC}_{var} = 1$ (accuracy constraint variation): they are sampled from a uniform distribution between $\text{PREC}_{min} = \text{PREC}_{avg} \cdot (1 - \text{PREC}_{var})$ and $\text{PREC}_{max} = \text{PREC}_{avg} \cdot (1 + \text{PREC}_{var})$.

5.2 Experimental Results

Figure 6 shows the system efficiency classified by the six scheduling algorithms: TABS, FCFS, CF, SF, SU, OD. Overall, we observe that the four combinations of policies (TABS+$G_{ac}$+SS, TABS+$G_{ac}$+SI, SF+$G_{ac}$+MC, TABS+$LO$+MC) result in good EoS. These four combinations use TABS or SF as the scheduling policy. This is because TABS keeps a good balance between source update requests and
consumer requests, thus rendering reasonably good QoS and QoD; on the other hand, SF gives higher priority to source update requests, thus keeping the directory service up-to-date and high QoD, which indirectly improves the QoS. In addition, three of the four best combinations use $G_{ac}$ as the request servicing policy. This is because in policy $G_{ac}$, sources are probed for each consumer request to improve QoD regardless of how well the directory service is maintained. As long as the DS maintenance policy does not introduce significant overhead (SS, SI, MC), the policy combination should perform well. However, when $LO$ is used for servicing consumer requests, whether the DS is well maintained is of great importance. Since MC keeps the DS reasonably accurate while minimizing the maintenance overhead, the combination of (TABS+$LO$+MC) results in a good EoS.

![Figure 6: Overall performance of mediator policies](image)

This figure illustrates the performance of the six request scheduling policies under different combinations of request servicing policies and DS maintenance policies.

In addition, Figure 6 shows two remarkable phenomena: (1) low EoS of policy combinations using $G_{dl}$ as the request servicing policy \(^1\); and (2) low EoS of policy combinations using TR as the DS maintenance policy. To further identify the reasons of the low EoS using $G_{dl}$, Figure 7 shows the three factors (QoS, QoD and Cost) of EoS for the combination of TABS, MC and the three different consumer request servicing polices ($LO$, $G_{dl}$, $G_{ac}$). We can see that QoS of using $G_{dl}$ is much lower than using $LO$ and $G_{ac}$. As mentioned before, $G_{dl}$ returns the values in the DS for all the consumer requests with the

\(^1\)If most requests in the system have small deadlines, $G_{dl}$ may show better performance; however, in our system, the deadlines of consumer requests vary in a wide range.
aim of shortening each request processing time, thus increasing the deadline meet ratio. However, the deadline meet ratio is not the only factor affecting the system performance, the other factor of accuracy meet ratio is greatly decreased under $G_{dl}$. In addition, we also observe that the QoD of using $G_{dl}$ is also lower than using LO and $G_{ac}$. $G_{dl}$ does not invoke source probings, which means that no consumer requests trigger any DS updates, thus limiting the accuracy of the DS representation and lower QoD. Therefore, even though $G_{dl}$ seemingly saves some overhead by not probing sources, the overall system efficiency is still very low. We analyze low EoS caused by TR through Figure 8. Studies of QoS, QoD and Cost show that the low EoS is caused by the extremely low QoD. This indicates that the crude way of adjusting range size does not perform well when the changes in information sources are very dramatic and do not follow any particular pattern\(^2\).

**System Heterogeneity Evaluation:** In our basic performance evaluation, we assume all the network and server information. Network and server status has been shown to be “self-similar”, which make it easier for TR to adapt to.

\(^2\)Our previous work [22] showed that TR works best under different traffic patterns and application workloads when it is used to collect network and server information.
sources are passive, i.e., they can only respond to probes. In the real world, some sources may have the capability of reporting significant changes in their own values. This requires basic processing and memory capabilities at source; we designate such sources to be smart sources. This additional intelligence relieves the mediator from the burden of probing, however, resources may be wasted when no consumers are interested in the collected values. We discuss how the four best combinations of policies adapt to source heterogeneity for the following cases: (a) all-passive-sources: all of the sources are passive; (b) mixed-sources: a mixed configuration with smart and passive sources (50% of each in the base case); (c) all-smart-sources: all of the sources are smart. From Figure 9A, we observe that with an increasing number of smart sources, the performance of the combination (TABS+LO+MC) consistently improves.

When all of the sources are intelligent, it outperforms the the other combinations. Obviously, with increased processing capability at the sources, the communication overhead (i.e., cost) is reduced with little reduction in QoD, hence the improvement in EoS.

A. source heterogeneity

B. request heterogeneity

Figure 9: Effect of source/request heterogeneity on the performance of mediator policies

Figure 9B demonstrates system performance as the deadlines of the consumer requests vary. We classify three ranges of possible deadlines: (a) dl-small: the deadlines are uniformly distributed from 500 ms to 1 second; (b) dl-mixed: the deadlines are uniformly distributed from 500 ms to 1 minute; (c) dl-large: the deadlines are uniformly distributed from 50 second to 1 minute. Figure 9B illustrates that when all the deadlines are very small, none of the combinations exhibit very high EoS, since the deadlines are so small that even just getting data from the DS cannot meet the deadlines. As the deadlines get bigger, EoS increases greatly, especially note that (TABS+LO+MC) exhibits the highest EoS, since it schedules the consumer requests and source update requests in a balanced way. Furthermore, (TABS+LO+MC) probes the sources only when necessary (i.e., when the DS does not have accurate data and the deadline will not be violated) and also maintains the DS at an adequate
level of accuracy while minimizing the cost involved. As the deadlines grow even bigger, SF+$G_{ac}$+MC outperforms the other combinations. While larger deadlines can be satisfied by either SF or TABS, SF assigns higher priorities to source update requests which renders much higher QoD, thus leading to higher EoS.

**System Scalability Evaluation:** We show the benefits of adding intelligence to each of the three components of the mediator. FCFS, $G_{dl}$ and SS are the simplest policies respectively for scheduling, request servicing and DS maintenance. We gradually enhance the intelligence of the system by enhancing the scheduler (TABS+$G_{dl}$+SS), followed by the request servicer (TABS+$LO$+SS) and finally arrive at the most intelligent set of policies by enhancing the DS maintainer (TABS+$LO$+MC). Figure 10A shows the benefits of adding more intelligence to each component gradually and also the changes of the benefits as the number of sources increases. As expected, the EOS increases as more intelligence is added to each component. Merely adding intelligence to scheduler (i.e., replacing FCFS with TABS) shows marginal benefit, but combining this with an intelligent request servicer (i.e., replacing $G_{dl}$ with $LO$) improved the EOS significantly. We can see from this that while a scheduling algorithm decides when to serve which request, request servicer is the one which decides how to process each dispatched request, i.e., a good scheduler ensures fairness among the requests and makes sure that no requests wait for too long to be dispatched, an effective request servicer ensures efficiency of request servicing and makes sure that no requests take longer than necessary. In addition, adding intelligence to DS maintainer (i.e., replacing SS with MC) decreases the overhead involved in maintaining the DS, thus further increasing the EoS. Note however, as the number of the sources in the system increases, the EoS decreases since the system is busy handling more source update requests. We observe that the enhanced mechanisms remain useful as the number of sources increases.

![Scalability of the policies](image)

Figure 10: Scalability of the policies
Figure 10B shows the change of the benefits as the number of consumer requests increases. We observe that as the number of consumer requests in the system increases, the EoS decreases slowly. Also as expected, the EoS increases as more intelligence is added to each component. This trend remains the same as the system is receiving more consumer requests, however there is an exception when (TABS+LO+SS) works better than (TABS+LO+MC) (the system receives 350 to 400 consumer requests in our case here). This implies that there is an intermediate region in which SS works marginally better than MC (TABS as the scheduling policy and LO as the request servicing policy). This is because when there are few requests in the system, maintaining approximate information (as MC) is sufficient; when there are a large number of incoming requests, maintaining accurate information (as SS) consumes a significant amount of energy, hence decreasing EoS; when the number of incoming requests lie in a certain range, it is desirable to have more accurate information (using SS) to ensure EoS. Overall, the most intelligent policy combination (TABS+LO+MC) works very well as the system scales.

Performance Summary: Our performance studies indicate that the policies combination of (TABS+LO+MC) exhibits higher system efficiency as compared to the simple policies of FCFS, G_{dl} and SS. This illustrates the advantage of enhancing the three main components of the system with intelligent strategies. We also find that as the system scales in terms of the number of sources or the number of consumer requests, this combination continues to perform well. In addition, the four best policy combinations under generic scenario are very robust to the source heterogeneity and consumer request deadline variation.

6 Related Work

In this section, we compare our work to related research in temporal and real-time databases, data caching, data streaming and event-driven/real-time middleware.

Temporal and Real-time Databases: The concept of time has been studied within the database community in the context of temporal databases and real-time databases. Temporal database systems [51] are designed for applications that require past, present and/or future data values. Representation of time-varying data in traditional databases results in significant storage overhead. Research in temporal databases characterizes the semantics of temporal data, and provides support for modeling, efficient storage and querying of time-varying information. In real-time databases [45], transactions have deadlines or timing constraints. In order to ensure timely access to data, much of the effort has been focused on developing high performance transaction scheduling algorithms [1] and concurrency control algorithms [27]. Typically, performance is measured by the ability to reduce transaction tardiness.

With the emergence of real-time monitoring applications such as process control and surveillance, the
ability to model temporal dimension of the real world and to respond within time constraints to changes in the real world is essential. This motivates the need to incorporate both capabilities (temporal and real-time) [41, 16, 46] to ensure temporal data consistency while providing time-constrained transaction processing. Several projects address the tradeoff between transaction timeliness and data freshness. The STRIP (Stanford Real-Time Information Processor) project [2] considers the scheduling of data updates to preserve transaction timeliness without sacrificing data timeliness in the context of memory resident databases. Similarly, ARCS [17] attempts to achieve equilibrium with regard to the conjoint processing of frequently arriving state updates and time constrained transactions. In addition, ARCS considers the satisfaction of temporal consistency by explicitly supporting historical views (i.e., many versions need to be stored). The QMF [28, 52] project focuses on providing transaction timeliness guarantees by dynamically adapting the update policies. Such a QoS sensitive approach balances the deadline miss ratio with data freshness. QMF applies a feedback control real-time scheduling policy called FC-UM to control the miss ratio without under-utilizing the CPU in the presence of unpredictable workloads.

An alternate approach to support transaction timeliness is to sacrifice the correctness of answers to database queries. Such an imprecise computation approach [14] relies on making available results that are of poorer but acceptable quality on a timely basis when results of the desired quality cannot be produced in time. An incoming task is decomposed into a mandatory part (characterized by minimum execution time) and an optional part. When a task is terminated normally, the error in the result produced by it is zero. When a task terminates prematurely, the result produced by it is acceptable as long as the mandatory part is executed, i.e., the duration of its execution is equal to or longer than its minimum execution time. Note that in the context of imprecise computation, the timeliness requirements only come from transactions, no data updates are involved.

Our work differs from the above mentioned prior research in several ways. Firstly, a regular transaction is a sequence of database operations, whereas the incoming requests in our case (consumer requests or source update requests) are special transactions with only one single operation. Real-time transaction processing techniques focus on optimized execution of the group of operations to meet transaction deadlines; in our case, the deadlines are on the fine-grained operations themselves. This implies that we must explore relative tradeoffs between the individual fine-grained operations, and traditional real-time transaction processing techniques will not suffice. Secondly, in aforementioned research, transactions have specific deadlines while data update requests do not. In our system, both consumer requests and data update requests may have specific deadlines. This requires an integrated approach to handle both consumer requests (transactions) and source update requests. Thirdly, data is represented approximately in our system in order to reduce communication overhead for maintaining the repository. The
approximation also implies that source update requests can have a direct impact on the data accuracy, which can indirectly affect timeliness. Our objective is to maintain a separation of concerns and isolate tasks within the real-time path flow (scheduling, request servicing) from tasks that maintain accuracy and minimize cost (DS maintenance). At the same time, we would like to allow adequate interaction between these components to balance the cost/accuracy/time tradeoffs.

**Data Caching:** Approximate data caching approaches [26, 36, 22] have been proposed to address the tradeoff between the accuracy of cached data and the overhead involved in maintaining the cache; these approaches differ in approximate representations of data and cache refreshing strategies. One approximation strategy is to minimize the overall divergence between source objects and cached copies by selectively refreshing a subset of sources [38]. In our work, we address time constraints in addition to precision requirements that are imposed on user queries, this further complicates the cache (i.e., repository) maintenance problem. The tradeoff between cost and accuracy has also been addressed in the context of aggregate queries [37]. To provide reasonable quality of results for aggregate queries over point data while minimizing query processing time, a progressive algorithm based on a multi-resolution tree structure is proposed in [31]. Our work considers system performance in terms of overall deadline satisfaction ratio, rather than for each individual query. Therefore, a scheduling mechanism is needed.

**Data Streaming:** Researchers have been re-investigating traditional data management and processing techniques in the presence of multiple continuous time-varying data streams. Data streaming is now a vibrant research field for both traditional Internet environments (e.g., OpenCQ [43, 33], Niagara [11], Telegraph [35], STREAM [4], Aurora [7]) and emerging ad-hoc sensor networks (e.g., COUGAR [6], Quasar [30]). Researchers in this area have addressed a whole gamut of issues such as system architectures, concepts/semantics of continuous queries, QoS specifications for fresh information delivery, system scalability etc. Our work targets a similar context where data is fast changing and update intensive, but we provide support for collecting and accessing the data in real-time. In addition, we take advantage of application’s tolerance towards data imprecision and address the three factors request timeliness, data accuracy and maintenance overhead.

**Event-driven/Real-time Middleware:** Similar concepts such as information source, information consumer and information mediator in our real-time information collection architecture have been proposed in event based middleware such as TAO [40, 47] and COBEA(CORBA-based Event Architecture) [34, 5, 42], and publisher/subscriber service architectures such as SIENA [8, 9], CMU Pub/Sub [44] and IBM’s Gryphon [39]. The architecture we proposed in this paper is different from the existing event based middleware in several ways. Firstly, unlike the event channel in publisher/subscriber architectures, our mediator component supports specification of QoS constraints (coarse and fine-grained timeliness
requirements, and accuracy constraints) from both publishers (sources) and subscribers (consumers). The mediator also supports both periodic and aperiodic requests. Furthermore, the mediator component incorporates a sophisticated repository management sub-component, which maintains the repository at a certain accuracy level so that desired timeliness and accuracy constraints from consumers can be achieved. Languages and methodologies for event composition, which are inherent in event-based middleware, complement our effort and can be incorporated as an additional layer/service in the mediator component.

7 Conclusions

A key concern of our work is to exploit the accuracy and latency margins to ensure that most applications receive information at the desired levels of accuracy and timeliness while minimizing the consumption of resources (storage, network bandwidth etc.) in distributed environments. In this paper we have shown, using the proposed real-time information collection architecture, how an application’s tolerance of information imprecision can be taken into account to improve system timeliness. We have also indicated that tradeoff between timeliness and accuracy can be addressed by scheduling requests from both sources and consumers in a uniform manner (i.e., not favoring either type). At the same time, we observe that, a well-maintained DS (in concert with the scheduling algorithm) is essential for balanced QoS and QoD under different system load and input characteristics. In addition, the general middleware framework and proposed metrics to capture timeliness, accuracy and cost tradeoffs forms the basis for future research in the direction of bringing together real-time systems and data management middleware.

We are in the process of building a prototype system to study the performance of the proposed framework in a realistic distributed environment for heterogeneous requests and information sources. Initial studies indicate that the latency incurred by message exchange among different components in the distributed system is non-negligible as compared to the processing time of each request in the mediator. This implies that a real-time communication protocol will help improve system performance. Preliminary results also show that updating an existing attribute value in the DS incurs significantly more overhead than adding a new attribute or retrieving a value. This observation confirms that approximate representation helps in improving query responsiveness, since it shortens query processing time via reducing the likelihood of DS updates. For more detailed description and discussion about the design of the prototype and preliminary results, interested readers may refer to [25].

In general, the dynamic nature of applications executing under varying system/network conditions and application workload imply that collection policies must be dynamic and customizable. Our prior work [18, 20, 19, 23, 24] has studied a family of information collection policies together with several
QoS-based resource provisioning techniques in the context of network management and multimedia applications. We observe that policy composition plays an important role in determining system performance under varying application workload and system conditions. The AutoSeC (Automatic Service Composition) project [22] at UC-Irvine is driven by these observations. AutoSeC is an integrated middleware framework that can dynamically select an appropriate combination of services based on current system conditions and user requirements; the current version supports composition of policies for information collection and QoS-based resource provisioning. We are enhancing the AutoSeC tool to support real-time information services by integrating techniques proposed in this paper.

The timeliness/accuracy/cost tradeoffs addressed in this paper bear remarkable relevance in emerging applications in distributed sensor environments. Sensor devices promise to revolutionize interaction with the physical world by allowing continuous monitoring and reaction to natural processes, hence real-time communication is crucial. A major concern in sensor networks is energy, therefore, the cost is better measured by energy consumption. Sensing itself also introduces uncertainty since it captures the underlying process only as accurately as sensing technology allows. Hence, information in such environments is approximate at best, lending our techniques to support approximate information representation useful. However, sensor networks are subject to higher fault rates than traditional networks: connectivity between nodes can be lost due to environmental noise and obstacles; nodes may die due to power depletion, environmental changes or malicious destruction. Characterization of QoS and QoD for sensor applications will require further work that takes into account resource-limited nature of sensor nodes, imprecision of sensing data and the high fault rate of sensor networks and .

The eventual goal of our work is to develop effective tools for system management in highly dynamic distributed environments. Middleware techniques for achieving the competing goals of timeliness, accuracy and cost-effectiveness such as those described in this paper are key to delivering the right information to the right person at the right time in highly dynamic distributed environments.

References


