Improving Robustness of Distributed Condition Monitoring by Exploiting the Semantics of Information Change

Paul Benninghoff
Department of Computer Science and Engineering
Oregon Graduate Institute
benning@cse.ogi.edu

Sanguk Noh
Computer Science Department
University of Missouri-Rolla
nohs@umr.edu

Abstract

Monitoring complex conditions over multiple distributed, autonomous information agents can be expensive and difficult to scale. Information updates can lead to significant network traffic and processing cost, and high update rates can quickly overwhelm a system. For many applications, significant cost is incurred responding to changes at an individual agent that do not result in a change to an overriding condition. But often we can avoid much work of this sort by exploiting application semantics. In particular, we can exploit constraints on information change over time to avoid the expensive and frequent process of checking for a condition that cannot yet be satisfied. We motivate this issue and present a framework for exploiting the semantics of information change in information agents. We partition monitored objects based on a lower bound on the time until they can satisfy a complex condition, and filter updates to them accordingly. We present a simple analytic model of the savings that accrue to our methods. Besides significantly decreasing the workload and increasing the scalability of distributed condition monitoring for many applications, our techniques can appreciably improve the response time between a condition occurrence and its recognition.

Introduction

Distributed Condition Monitoring (DCM) involves tracking the state of a complex condition over multiple autonomous agents. DCM is an important component of real-time information integration. It is widely applicable in areas such as battleground scenario tracking, stock trading and portfolio management, and all manner of wireless and mobile computing (Cohen *et al.* 1994; Liu *et al.* 1998; Sistla & Wolfson 1992; 1995). DCM will continue to grow in import as network-accessible services proliferate.

Architecturally, a DCM system can be implemented as a mediator (Wiederhold 1992) that interacts with a variety of network-accessible agents. The mediator accepts monitoring requests involving groups of agents, expressed in the form of a query. The DCM problem can be cast as an instance of the view maintenance problem, which has been studied extensively by the database research community (Gupta & Mumick 1999). In this light, the request is a view, and changes at individual agents that participate in the view trigger re-evaluation. If the view has changed, the condition fires, and the change is conveyed to the requesting client. DCM can be an expensive process, however, even if incremental techniques are used. It is especially expensive in a WAN environment. Every change to a participating agent can trigger considerable processing and network traffic. High update rates can overwhelm a system and greatly degrade performance.

But a large number of DCM applications concern themselves with dynamic attributes of a relatively stable set of objects: prices of stocks, locations of moving objects, usages of pieces of capital equipment, supply volumes for products or spare parts, etc. Furthermore, attributes of interest often do not or can not change arbitrarily; they obey a set of constraints. For instance, a vehicle cannot move faster than its maximum speed, an equipment item cannot be in use for more than 8 hours in an 8 hour shift, the capacities of a maintenance shop may constrain the rate of depletion of spare parts, etc. Such semantic constraints provide a potential foothold for improving the efficiency and scalability of DCM systems.

In this paper, we describe a method for exploiting constraints on information change to reduce the work and improve the response time of distributed condition monitoring. We apply a semantic analysis, based on such constraints, to partition monitored objects by their temporal distance from a specified condition. Our approach allows us to isolate updates that are *semantically independent* of conditions of interest, and avoid work associated with such updates. We present a simple analytic model of the savings that accrue to our methods. Besides significantly decreasing the workload and increasing the scalability of distributed condition monitoring for many applications, our techniques can appreciably improve the response time between a condition occurrence and its recognition.

Partitioning Objects by "MinTTI"

Our approach applies to distributed conditions that involve tracking attributes of a set of objects whose rate of change is constrained. We are given a monitoring request expressed as a distributed query over a collection of information agents, and one or more integrity constraints that apply to attributes in the request. We assume a baseline monitoring interval, M_0, for the request. We generate a companion query that, for each object, computes a lower-bound on the time before object changes can satisfy the distributed condition of interest. If we call the original request Q, then we refer to the companion query as the *Minimum Time 'Til Interesting* with respect to Q, or $\texttt{MinTTI}_Q$. We partition the set of monitored objects into buckets based on their $\texttt{MinTTI}_Q$ values, and track changes to each bucket at the largest possible time interval that is less than the least $\texttt{MinTTI}_Q$ value for the bucket. A snapshot of this process is depicted in Figure 1.

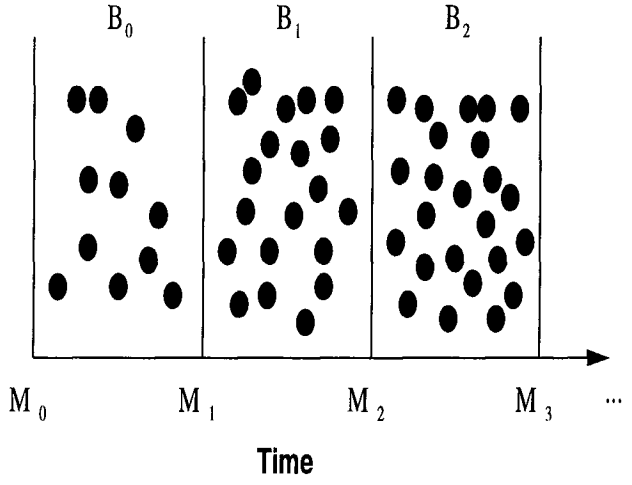

Figure 1: Partitioning objects based on Minimum Time 'Til Interesting.

A Motivational Example

Consider a (slightly simplified) example distributed condition that arises in DARPA's "Command Post of the Future" project. We have an information agent that provides sensor data on the movements of enemy units over a far-reaching battle area. Another information source provides the location of landing zones where helicopters can land to transport personnel and material in and out of the area. Other agents provide information on types of enemy units and the ranges of weapons associated with the units by type. A geocoder agent is able to compute distances between entities based on the "latlong" coordinates provided by the location sensors. A snippet of the battleground is depicted in Figure 2.

A commander in the field wants to be notified whenever an enemy unit "threatens" a landing zone, meaning whenever a unit comes within its weapon's range of the zone. This condition can be expressed as the following query, expressed in datalog notation (Ullman 1988):

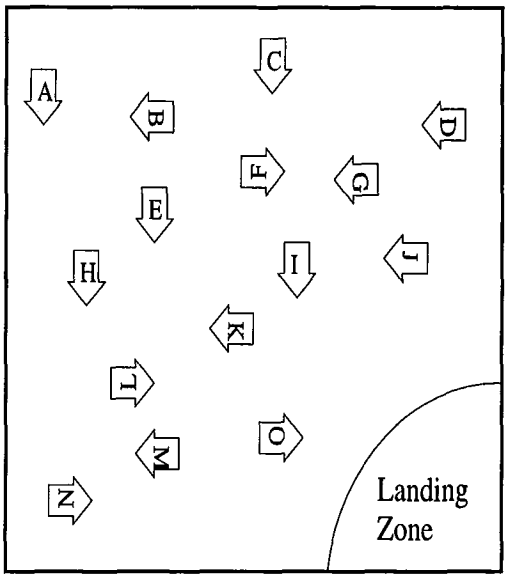

Figure 2: A battlefield scene with a landing zone and enemy units.

- Threaten(UnitID, LZ) :-
  LandingZone(LZ, Loc0) $\wedge$ EnemyUnit(UnitID) $\wedge$ UnitType(UnitID, Type) $\wedge$ WeaponRange(Type, R) $\wedge$ Position(UnitID, Loc1) $\wedge$ Distance(Loc0, Loc1, D) $\wedge$ D $\leq$ R.

Note that the key changeable attribute in this request is the location of enemy units. Adds and deletes can happen in other predicates (e.g., *LandingZone* or *EnemyUnit*), but the most frequent updates will be modifications of the location attribute in *Position*. But a unit's position cannot change arbitrarily. In particular, suppose we are given the following constraint:

- :- upd(Position(UnitID, (Loc0,Loc1)), T) $\wedge$ UnitType(UnitID, Type) $\wedge$ MaxSpeed(Type, MPH) $\wedge$ Distance(Loc0, Loc1, D) $\wedge$ D > MPH $\times$ T.

This constraint is headless, meaning it derives *false*. It is interpreted as follows: If $Position(UnitID, Loc0)$ is true at time 0, and $Position(UnitID, Loc1)$ is true at time T, then the distance between $Loc0$ and $Loc1$ cannot be greater than the product of the maximum speed of $UnitID$, MPH, and the time transpired, T. Note that this constraint incorporates information from multiple agents, and it makes use of supplemental information, namely the $MaxSpeed$ predicate, that is not part of the original request.

Based on this constraint, we can write the following definition for $\texttt{MinTTI}_{\texttt{Threaten}}$:

- $\texttt{MinTTI}_{\texttt{Threaten}}$(UnitID, T) :-
  LandingZone(LZ, Loc0) $\wedge$ EnemyUnit(UnitID) $\wedge$ UnitType(UnitID, Type) $\wedge$ WeaponRange(Type, R)

$\wedge$ Position(UnitID, Loc1) $\wedge$ Distance(Loc0, Loc1, D) $\wedge$ D > R $\wedge$ MaxSpeed(Type, MPH) $\wedge$ T = (D-R)/MPH.

Note that we will take the minimum T value for each $UnitID$. In general, since we are computing a lower-bound time, we always take the minimum time grouped by the object identifier for `MinTTI`. Note, further, that the definition of $\texttt{MinTTI}_{\texttt{Threaten}}$ has a high degree of commonality with the *Threaten* predicate itself. Such commonality implies that whenever *Threaten* must be computed, $\texttt{MinTTI}_{\texttt{Threaten}}$ can be computed as well at little additional cost. This pattern of commonality between a condition and its `MinTTI` query is typical.

Partitioning Pragmatics

Automatic generation of `MinTTI` is an important aspect of our research, but it is beyond the scope of this paper. In fact, for an important class of requests and constraints, we can employ a combination of semantic query optimization techniques (Chakravarthy, Grant, & Minker 1990) and other simple program transformations to automatically generate `MinTTI`. We plan to describe this process in a forthcoming paper. In the absence of automatic generation, however, it is perfectly reasonable to write `MinTTI` directly by hand. The system can then use the definition to optimize the monitoring process.

Given the definition of `MinTTI`, we compute a relation of *OID-Time* pairs that covers every object currently being monitored at a particular source. This relation provides the basis for dividing monitored objects into buckets. In general, there is always an "urgent bucket", which we monitor at the baseline interval (often as quickly as possible). An important issue is how many additional buckets we have, and how the bucket boundaries are defined. An extreme approach is to have a separate bucket for each object. The other extreme is to place all the objects in a single (urgent) bucket. We will describe an approach based on two buckets. But we believe a clustering approach, where a natural partition emerges from the data, is promising.

Another important issue is how buckets are implemented. There are a number of options, but we prescribe an agent "bucket-trigger" capability that operates (modulo several minor variations) as follows:

1. A group of object ids is associated with each bucket in the bucket-trigger.
2. A "change bucket" table is maintained for each bucket, with an associated monitoring interval.
3. As object attributes change, the change is recorded in the appropriate change bucket. If a value is already present for the given OID, it is overwritten.
4. A daemon process empties each change bucket at the appropriate interval, and passes the contents to the client of the trigger (the mediator, in our DCM architecture).
5. Bucket triggers can be created and modified atomically.

By implementing bucket triggers at the individual agent, irrelevant updates are filtered away as early as possible. The mediator never sees them.

The intuition behind our approach is straightforward. If an object is far from satisfying a condition of interest, we can ignore changes to it for a while. In our example, if we are concerned with enemy units that threaten a given landing zone, we needn't repeatedly compute the *Threaten* predicate in response to the movement of tanks, say, that are hundreds of miles away. By ignoring such changes, we avoid the expensive process of evaluating a complex, distributed condition for every change. The price of our scheme, however, is the need to do bucket maintenance. We now look more closely at the costs and benefits of our approach.

Modeling Monitoring Costs

Given a query Q that describes a complex distributed condition involving dynamic attributes of a set of N objects, S, with monitoring interval M, the *naive* approach to monitoring Q with respect to S is as follows:

1. At the source of S, at each monitoring interval, check if there is a change to S.
2. If there is a change, call it ΔS, send ΔS to the mediator.
3. At the mediator, compute ΔQ with respect to ΔS.

We assume that the cost of local monitoring (Step 1) is relatively small, and that, in any case, it doesn't vary enough in the approaches we compare to be of consequence. We will consider the cost of sending ΔS to the mediator as part of the cost of computing ΔQ, and that cost is given by the cost function of the query optimizer. We assume the statistical properties of ΔS follow those of S, with the exception of its cardinality (denoted as $|\Delta S|$). Hence the cost of computing ΔQ is a function of $|\Delta S|$, and the expected cost of the *naive* approach is given by:

$$E(Cost_{naive}) = \sum_{x=0}^{N} p(|\Delta S| = x) \times C(\Delta Q, x)/M \quad (1)$$

where $p(|\Delta S| = x)$ is the probability of $|\Delta S| = x$, and $C(\Delta Q, x)$ is the cost, given by the cost function, of the query ΔQ with respect to ΔS, with $|\Delta S| = x$. Note that the cost is expressed per unit time.

For our *bucketing* approach the expected cost is the sum of the expected costs of each bucket. For L buckets, the cost is given by:

$$E(Cost_{buckets}) = \sum_{i=0}^{L-1} E(Cost_{B_i}) \quad (2)$$

B_0 is distinguished as the *urgent bucket*. The processing related to this bucket is similar to that of the

naive approach, except we have the additional process of bucket maintenance to worry about. That is, a change to an object in B_0 may result in a change to Q, and it may also result in a change to the $\texttt{MinTTI}_Q$ value associated with that object, which in turn may result in a bucket change for that object. The steps associated with B_0 are as follows:

1. At the source of S, at each monitoring interval, check if there is a change to the members of S in B_0.
2. If there is such a change, call it ΔB_0, send ΔB_0 to the mediator.
3. At the mediator, compute ΔQ and perform bucket maintenance with respect to ΔB_0.

Assume that we use a partitioning strategy where the dividing points between buckets, once established, are fixed. Then the cost of bucket maintenance is dominated by the cost of computing a new $\texttt{MinTTI}_Q$ for each object in ΔB_0.[1] By a similar argument to that of the naive case above, the expected cost of the urgent bucket is given by:

$$E(Cost_{B_0}) = \sum_{x=0}^{N_0} p(|\Delta B_0| = x) \times C(\Delta Q + \Delta\texttt{MinTTI}_Q, x)/M_0 \quad (3)$$

where $C(\Delta Q + \Delta\texttt{MinTTI}_Q, x)$ is the cost, given by the cost function, of executing the two queries ΔQ and $\Delta\texttt{MinTTI}_Q$ with respect to ΔB_0, with $|\Delta B_0| = x$. Note that M_0, the monitoring interval for the urgent bucket, is the user-defined monitoring interval, which is the same M that applies to the naive case (Equation 1). N_0 is the number of objects in B_0, which is less than N.

Now consider the non-urgent buckets, $B_i, i > 0$. Recall that the $\texttt{MinTTI}_Q$ value for any object in a non-urgent bucket is greater than the monitoring interval for that bucket, M_i. Thus, by definition, a tuple in ΔB_i, $i > 0$, cannot result in a change to Q. Therefore the computation of ΔQ is not needed for the non-urgent buckets; only bucket maintenance is required. The expected cost for each non-urgent bucket, then, is given by:

$$E(Cost_{B_i}) = \sum_{x=0}^{N_i} p(|\Delta B_i| = x) \times C(\Delta\texttt{MinTTI}_Q, x)/M_i \quad \text{for} \quad i = 1, \cdots, L-1 \quad (4)$$

[1]This is conservative in that bucket maintenance may be done more efficiently in some cases. It is generous in that we disregard the cost of notifying the remote source of bucket changes.

Cardinalities of Deltas

The equations we have derived thus far depend on the distribution of the cardinalities of the deltas, $|\Delta S|$ and $|\Delta B_i|$. To model these distributions, assume that changes to objects in S are independent, and that the probability of an object changing in a given time interval is independent of and identical to the probability of it changing in any other time interval of equal length.[2] This implies that the number of changed objects for a given set of objects within a given monitoring interval, and thus the cardinalities of the deltas, follows a binomial distribution. Suppose, further, that the probability that a given object will change in the smallest monitoring interval, M, is θ. The distributions for $|\Delta S|$ and $|\Delta B_0|$ are given by:

$$p(|\Delta S| = x) = b(x; N, \theta) \quad (5)$$

$$p(|\Delta B_0| = x) = b(x; N_0, \theta) \quad (6)$$

where $b(x; n, \theta) = \binom{n}{x} \theta^x (1-\theta)^{n-x}$ is the x^{th} binomial coefficient.

For the non-urgent buckets, $B_i, i > 0, M_i > M$, and thus $\theta_i > \theta$. Assume that $M_i = k_i \times M$ for some integer k_i. Then the probability of x changes to a single object within M_i is $b(x; k_i, \theta)$, and θ_i, the probability of some change to a single object within M_i, is $1 - b(0; k_i, \theta) = 1 - (1-\theta)^{k_i}$. Thus the distribution of $|\Delta B_i|$ is given by:

$$p(|\Delta B_i| = x) = b(x; N_i, 1-(1-\theta)^{k_i}) \quad \text{for} \quad i = 1, \cdots, (L-1) \quad (7)$$

Notice that if we monitored the objects in B_i at interval M, as we would in the naive case, we would expect to process $\theta k_i N_i$ updates to these objects every M_i time period. With bucketing we expect to process $(1-(1-\theta)^{k_i})N_i$ updates in the same period. That is, the expected volume of *Meaningless Updates Discarded (MUD)* for B_i is given by:

$$E(MUD_{B_i}) = \theta k_i N_i - (1-(1-\theta)^{k_i})N_i \quad (8)$$

MUD, and the avoidance of related processing costs, is the source of the savings produced by the bucketing approach. When these savings exceed the overheads introduced by bucket maintenance, our approach is beneficial.

Performance Assessment: 2-Bucket Case

We now consider the simplest possible implementation of our methods: two buckets ($L = 2$), B_0 and B_1, with a fixed boundary between them, M_1. Assume, without

[2]This assumption is imperfect, but not damaging. Trying to model complex inter-update correlations is difficult at best, and we postulate that our methods tend to work better where such correlations exist.

loss of generality, that $M = M_0 = 1$. Then $M_1 = k_1$. For simplicity of analysis, we will also assume simple linear forms for the query cost functions as follows:[3]

$$\begin{aligned} C(\Delta Q, x) &= Ax \\ C(\Delta \texttt{MinTTI}_Q, x) &= Bx \\ C(\Delta Q + \Delta \texttt{MinTTI}_Q, x) &= (A+B)x \end{aligned} \quad (9)$$

Note that Equation 9 represents a worst case cost for computing ΔQ and $\Delta \texttt{MinTTI}_Q$ together. But if Q and $\texttt{MinTTI}_Q$ are highly similar, as will often be the case, we can do much better. Multiple-query optimization techniques (Sellis 1988) may enable us to compute $\Delta \texttt{MinTTI}_Q$ for very little additional cost, in which case:

$$C(\Delta Q + \Delta \texttt{MinTTI}_Q, x) \approx Ax \quad (10)$$

In any event, the cost of the *naive* monitoring implementation is given by:

$$\begin{aligned} E(Cost_{naive}) &= \sum_{x=0}^{N} b(x; N, \theta) \times C(\Delta Q, x) \\ &= \theta \times N \times A \end{aligned} \quad (11)$$

We now consider how a choice of M_1 can be made to minimize the cost of the 2-bucket approach, and compare this cost to the naive approach. Suppose we have a stable uniform distribution of $\texttt{MinTTI}_Q$ values for the set of objects, S, over the range $[0..R]$. Given M_1, $N_0 = (M_1/R) \times N$ and $N_1 = (1 - M_1/R) \times N$. The cost of the 2-bucket approach in this case is given by:

$$\begin{aligned} &E(Cost_{2-buckets}) = \\ &\sum_{x=0}^{(M_1/R)N} b(x; (M_1/R)N, \theta) \times C(\Delta Q + \Delta \texttt{MinTTI}_Q, x) + \\ &\sum_{x=0}^{(1-M_1/R)N} b(x; (1 - M_1/R)N, 1 - (1-\theta)^{(1-M_1/R)N}) \times \\ &\quad C(\Delta \texttt{MinTTI}_Q, x)/M_1 \\ &= ((M_1/R) \times N \times \theta \times (A+B)) + \\ &\quad ((1 - M_1/R) \times N \times (1 - (1-\theta)^{(1-M_1/R)N}) \times B)/M_1 \end{aligned}$$

As a final simplification, for the sake of exposition, assume the probability that an object in B_1 will change within M_1 is 1. That is, we will round $\theta_1 = 1 - (1 - \theta)^{(1-M_1/R)N}$ to 1 in the above equation. Note that while θ_1 approaches 1 as θ goes to 1 and as M_1 gets large, this is an overestimate of the costs associated with B_1. But it allows us to simplify the above equation to:

$$\begin{aligned} E(Cost_{2-buckets}) &= ((M_1/R) \times N \times \theta \times (A+B)) + \\ &\quad ((1 - M_1/R) \times N \times B)/M_1 \end{aligned} \quad (12)$$

Finally, $E(Cost_{2-buckets})$ is minimized by choosing:

$$M_1 = \sqrt{BR/(A+B)\theta} \quad (13)$$

Table 1 and Table 2 show analytical results for a range of values of R and θ in the pessimistic model and the optimistic model, respectively. Here we assume that there are 10,000 objects to be monitored ($N = 10,000$). We show cost comparisons for the pessimistic assumption that $C(\Delta Q + \Delta \texttt{MinTTI}_Q, x) = (A+B)x = 2Ax$, and the optimistic assumption that $C(\Delta Q + \Delta \texttt{MinTTI}_Q, x) \approx Ax$. In this setting, the objects are monitored over the values, R, ranging from 100 to 10,000. The probability that a given object will change in a specific monitoring interval, θ, varies between 0.01 and 0.25. We compute M_1, C_{naive}, and $C_{buckets}$ using Equation 13, 11, and 12, respectively. In each table, the Gain is the difference between the cost of the bucketing approach and that of the naive approach.

Table 1: Cost comparisons for the pessimistic model

R	θ	M_1	C_{naive}	$C_{buckets}$	Gain
100	0.01	71	100A	182.8A	-82.8A
100	0.05	32	500A	532.5A	-32.5A
100	0.25	14	2500A	1314.3A	1185.7A
1,000	0.01	224	100A	79.4A	20.6A
1,000	0.05	100	500A	190.0A	310.0A
1,000	0.25	45	2500A	437.2A	2062.8A
10,000	0.01	707	100A	27.3A	72.7A
10,000	0.05	316	500A	62.2A	437.8A
10,000	0.25	141	2500A	140.4A	2359.6A

Table 2: Cost comparisons for the optimistic model

R	θ	M_1	C_{naive}	$C_{buckets}$	Gain
100	0.01	100	100A	100.0A	0
100	0.05	45	500A	347.2A	152.8A
100	0.25	20	2500A	900.0A	1600.0A
1,000	0.01	316	100A	53.2A	46.8A
1,000	0.05	141	500A	131.4A	368.6A
1,000	0.25	63	2500A	306.2A	2193.8A
10,000	0.01	1000	100A	19.0A	81.0A
10,000	0.05	447	500A	43.7A	456.3A
10,000	0.25	200	2500A	99.0A	2401.0A

In the pessimistic model, when $R = 100$ and $\theta = .01$ and .05, our bucketing approach is worse than the naive approach since the cost of bucket maintenance

[3] We choose not to use the $AX + Y$ form since it does not capture the important notion, in our setting, that $X = 0$ implies $C = 0$.

outweighs the (relatively small) savings due to MUD. In all other cases, however, under our assumptions, our method provides a more efficient monitoring capability.

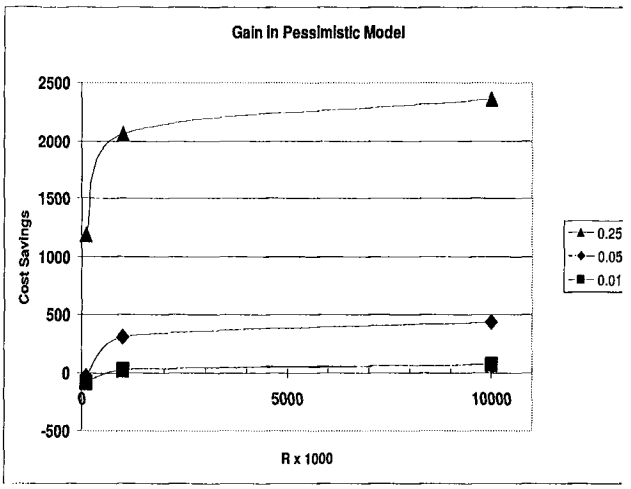

Figure 3: The pessimistic cost savings.

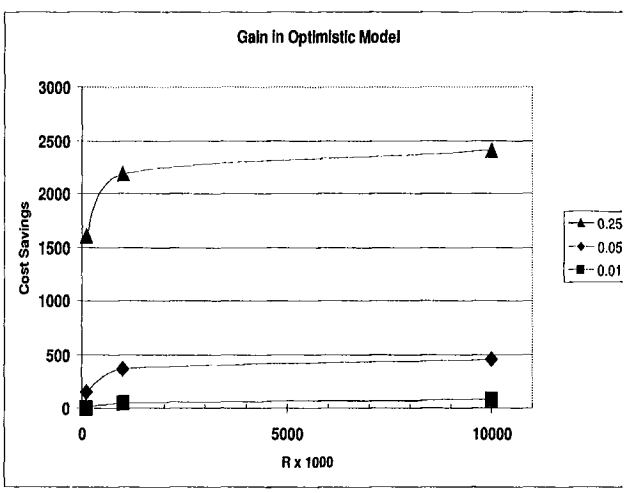

Figure 4: The optimistic cost savings.

Figures 3 and 4 depict the pessimistic and optimistic cost savings for a range of R values with θ fixed at .01, .05, and .25. In both figures, as θ increases, the cost savings of the bucketing approach increases as well. As object changes become more frequent, more MUD is generated by our methods, which translates to more savings.

Note that in general we may, at best, only be able to estimate (or guess) at θ. But simple calculations based on the above show that even if we are wrong in our guesses, and our choice of M_1 is imperfect, we can still do better than the naive approach.

Conclusions

We have presented a framework for exploiting constraints on information change in Distributed Condition Monitoring. For a large class of applications, our techniques can greatly reduce the update rate seen by a DCM system, and thus improve the scalability and response time of such systems. These improvements should translate into greater robustness in the face of update bursts (for example) as well.

There are several clear areas for near-term expansion of these ideas. Requiring the manual specification of `MinTTI` for each request in a DCM is burdensome and error-prone. We are currently investigating techniques for automatically generating `MinTTI`. The issue of how bucket boundaries are defined is important as well. We intend to investigate a range of approaches for doing this, including clustering techniques.

Interesting future avenues to pursue include trying to broaden the range of constraints we can work with. Often absolute constraints do not exist, but probabilistic ones do. A stock currently priced at $20, for example, is highly unlikely to drop to $2 in a matter of hours (recent events notwithstanding). How can constraints that apply (merely) with high probability be integrated into our scheme? What are the tradeoffs between efficiency and scalability in applying these techniques, and accuracy or consistency (a familiar theme in Internet-based systems)? Can reasoning about the intent of monitored objects be incorporated into our constraint reasoning as well? Also, "time" can be more general than clock time. Constraints might be based on some necessary path of changes, required intermediate steps, or some discrete number of update events.

In the Internet age, scalability and robustness are a constant challenge in computing. We believe semantics provide a key source of traction in meeting this challenge. The ideas in this paper represent a first step toward tapping this source in DCM systems. Significant work in view maintenance has sought to determine when updates are syntactically independent of views (Levy & Sagiv 1993); the work we present here is the first attempt, to our knowledge, to isolate updates that are *semantically* independent of views.

Acknowledgments

We thank Dr. Dave Maier and Dr. Phil Cohen for support and feedback on this work. We gratefully acknowledge the support of the DARPA CoABS Program (contract F30602-98-2-0098, AO G352) for the research presented in this paper.

References

Chakravarthy, U. S.; Grant, J.; and Minker, J. 1990. Logic-based approach to semantic query optimization. *IEEE Transactions on Database Systems* 15(2):162–207.

Cohen, P. R.; Cheyer, A.; Wang, M.; and Baeg, S. C. 1994. An open agent architecture. In *AAAI Spring Symposium*, 1–8. AAAI Press/MIT Press.

Gupta, A., and Mumick, I. S., eds. 1999. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press.

Levy, A. Y., and Sagiv, Y. 1993. Queries independent of updates. In Agrawal, R.; Baker, S.; and Bell, D. A., eds., *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, 171–181. Morgan Kaufmann.

Liu, L.; Pu, C.; Tang, W.; Buttler, D.; Biggs, J.; Benninghoff, P.; Han, W.; and Yu, F. 1998. Cq: A personalized update monitoring toolkit. In *Proceedings of the ACM SIGMOD.*

Sellis, T. K. 1988. Multiple-query optimization. *IEEE Transactions on Database Systems* 13(1):23–52.

Sistla, P., and Wolfson, O. 1992. Triggers on database histories. *Data Engineering.*

Sistla, P., and Wolfson, O. 1995. Temporal conditions and integrity constraints in active database systems. In *Proceedings of International Conference on Management of Data.* ACM-SIGMOD.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2.* Computer Science Press.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *IEEE Computer* 25(3):38–49.